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/TNSM.2021.3132491, IEEE Transactions on Network and Service Management

DRSIR: A Deep Reinforcement Learning Approach for Routing in Software-Defined Networking

Daniela M. Casas-Velasco, Oscar Mauricio Caicedo Rendon, and Nelson L. S. da Fonseca

***Abstract*—Traditional routing protocols employ limited information to make routing decisions, which leads to slow adaptation to traffic variability and restricted support to the quality of service requirements of applications. To address these shortcomings, in previous work, we proposed RSIR, a routing solution based on Reinforcement Learning (RL) in Software-Defined Networking (SDN). However, RL-based solutions usually suffer an increase in time during the learning process when dealing with large action and state spaces. This paper introduces a different routing approach, called Deep Reinforcement Learning and Software-Defined Networking Intelligent Routing (DRSIR). DRSIR defines a routing algorithm based on Deep RL (DRL) in SDN that overcomes the limitations of RL-based solutions. DRSIR considers path-state metrics to produce proactive, efficient, and intelligent routing that adapts to dynamic traffic changes. DRSIR was evaluated by emulation using real and synthetic traffic matrices. The results show that this solution outperforms the routing algorithms based on Dijkstra's algorithm and RSIR in relation to stretch, packet loss, and delay. Moreover, the results obtained demonstrate that DRSIR provides a practical and feasible solution for routing in SDN.**

***Index Terms*—Deep Reinforcement Learning, Routing, Software-Defined Networking**

I. INTRODUCTION

ROUTING is the network function that determines the end-to-end path between a source and a destination node. Traditional routing protocols usually make decisions based on shortest path calculation with limited additional information, although this leads to slow adaptation to traffic variability and restricts its support for meeting Quality of Service (QoS) requirements. Moreover, the continuous growth of the Internet and the diversity of applications running on it have led to significant challenges affecting the efficiency of decisions based on limited information. On the other hand, the combination of Software-Defined Networking (SDN) and Machine Learning (ML) can help overcome such shortcomings. SDN provides opportunities for the improvement of networks in relation to programmability, global view, logically centralized control, and decoupling of network control and packet forwarding. ML techniques provide intelligence for SDN, enabling it to learn autonomously to make optimal routing decisions adaptable to traffic variations.

Some papers [1]–[6] have shown to improve network routing by leveraging SDN capabilities, but they do not exploit network information intelligently. Other proposals [7]–[12] have used supervised ML techniques to optimize existing routing strategies, but the training of these algorithms is based on labeled datasets obtained from the operation of traditional routing protocols, which demands high computational complexity and makes the routing decisions dependent on limited information.

The solutions in [13]–[17] have employed Reinforcement Learning (RL) to optimize the selection of routing algorithms. Compared with supervised ML techniques, RL learns by trial and error in interaction with the environment, and, thus, does not depend on labeled datasets. Moreover, optimization targets (*e.g.,* throughput and delay) can be easily adjusted by the definition of reward functions. In previous work [18], we proposed an approach called Reinforcement Learning and Software-Defined Networking for Intelligent Routing (RSIR), which employs link-state metrics for routing in SDN. Results showed that RSIR outperforms Dijkstra-based routing. Nonetheless, RL-based solutions usually lead to a significant increase in the learning process when handling large action and state spaces, requiring the RL agent to make several interactions with the environment to converge towards reliable estimations [19]. Various papers [20]–[30] have explored the use of Deep Reinforcement Learning (DRL) techniques to cope with the limitations of RL-based routing solutions. However, these papers focus on optimizing delay and neglect other performance metrics, such as loss and throughput. Moreover, the use of link metrics to build a path can be restrictive, since decisions based on local (link) information disregard the state of other links on the path to the destination node.

In this paper, we take a further step towards the goal of an efficient and intelligent routing scheme in SDN by introducing a novel approach, called **D**eep **R**einforcement Learning and **S**oftware-Defined Networking for **I**ntelligent **R**outing (DRSIR), which combines Deep Learning (DL) and RL to overcome the shortcomings of RL-based routing solutions. DRSIR introduces a model-free DRL-based algorithm that uses path-state metrics and the global view and control offered by SDN to compute and install optimal routes proactively in forwarding devices, thus allowing adaption to dynamic traffic changes without prior knowledge of the underlying network. Using path-state metrics enables the reduction of knowledge abstraction needed by the routing agent since this approach directly explores different path options instead of link state information. The DRSIR algorithm calculates optimal routes using Target and Online Neural Networks (NNs) which allows the DRL agent to reduce the error in estimations based on path information. Moreover, the DRL agent uses Experience Replay Memory to accelerate the learning process. In summary, the contribution of this paper is a novel routing algorithm that employs path-state metrics, Deep Q-Learning,

D. M. Casas-Velasco, and N. L. S. da Fonseca are with the Institute of Computing, University of Campinas, Brazil. e-mail: danielac@lrc.ic.unicamp.br and nfonseca@ic.unicamp.br.

O. M. Caicedo is with the Department of Telematics, Universidad del Cauca, Popayán, Colombia. e-mail: omcaicedo@unicauca.edu.co.

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidade Estadual de Campinas. Downloaded on March 16,2022 at 22:17:31 UTC from IEEE Xplore. Restrictions apply.

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/TNSM.2021.3132491, IEEE Transactions on Network and Service Management

Experience Memory Replay, Target Neural Networks, and Online Neural Networks to define optimal routes in SDN. The results show that DRSIR outperforms RSIR and four other variations of Dijkstra's algorithm in relation to stretch, link throughput, delay, and packet loss produced. Therefore, DRSIR is a promising solution for cognitive routing in SDN.

The remainder of this paper is organized as follows. Section II presents a brief background for the relevant concepts. Sections III and IV detail our approach by introducing the DRL-based routing architecture, the DRSIR agent, and the routing algorithm. Section V presents the DRSIR prototype and the results of its evaluation. Section VI compares the DRSIR to existing approaches. Section VII presents conclusions and suggestions for future work.

II. Background

In RL models, a Markov Decision Process (MDP) formulation defines a set of states and actions (S, A), and a reward model, hereafter, called environment. In RL, a learning entity (an agent) interacts with the environment by iterating over states and selecting actions to be made. In the interaction, the agent receives a reward value for each action performed [31]; this expresses its impact on the system performance. It aims at learning a policy that optimizes the cumulative reward. Thus, the agent learns by trial and error, observing the reward obtained by the different actions performed. The learning of an optimal policy π involves the estimation of a value function (*e.g.,* action-value function), which indicates the goodness of an action A_t performed when the MDP is in a state S_t under policy π [32].

RL algorithms can be either model-based, or model-free [33]. For model-based algorithms, the agent employs a model of the environment (*i.e.,* a function that predicts state transitions and rewards); this model defines how agents should operate in finding a solution. Such algorithms require a ground-truth model of the environment that is usually not available. On the other hand, model-free algorithms do not need such a model of the environment to find an optimal policy; they learn to estimate the expected reward for different actions either by learning a value function approximation from which the policy is inferred or by directly learning the policy function that maps states to actions. RL algorithms can also be classified as either off-policy or on-policy [31]. Off-policy algorithms are algorithm that do not follow the same policy for sampling (*i.e.,* selecting an action for the current state) and updating the value function, but rather choose the action corresponding to the best reward of actions available for updating the value function. In on-policy algorithms, the same policies are used for both updating and acting.

One well-known RL algorithm is Q-learning [34], which is model-free and off-policy. The action-values are represented by a state transition table (Q-table), where each state-action pair (S_t, A_t) has an entry called Q-value $Q(S_t, A_t)$. To update the Q-value, Q-learning uses Equation 1.

$$Q_{t+1}(S_t, A_t) = Q_t(S_t, A_t) + \alpha \cdot \left[R_t + \gamma \cdot \min_{\text{A}} Q_t(S_{t+1}, A) - Q_t(S_t, A_t) \right] \quad (1)$$

where:
- R_t is the reward retrieved by the RL agent in time t when executing an action A_t in a given state S_t.
- $\alpha \in [0, 1]$ is the learning rate that determines the weight of the newly gained information in relation to what was previously available.
- $\gamma \in [0, 1]$ is the discount factor that determines the importance of future rewards.
- The expression in square brackets is the difference between the target Q-value computed by $R_t + \min_{\text{A}} Q_t(S_{t+1}, A)$ and the current Q-value $Q_t(S_t, A_t)$ for a pair of state-action (S_t, A_t). When $\alpha = 0$, the RL agent does not learn from the latest (S_t, A_t) pair. When $\alpha = 1$ the RL agent retains the learned information by considering the immediate reward R_t from the pair (S_t, A_t). Moreover, a value of $\gamma = 0$ enables the RL agent to consider only the current reward R_t, while a factor $\gamma = 1$ makes it possible for the RL agent to consider future rewards.

In an attempt to reduce the very long learning process in RL for state and action spaces, Deep Learning (DL) has been employed, giving rise to Machine Learning models known as DRL, which makes possible solutions to problems intractable to RL [35]. DRL uses function approximators, such as Feed-Forward Neural Networks and Recurrent Neural Networks, to extract knowledge from the states visited. NNs can learn from complex data structures and build computational models by using just one or only a few additional layers of processing.

Deep Q-Network (DQN) is a DRL technique developed as an extension of Q-learning (a.k.a. Deep Q-learning). DQN usually employs a single NN to estimate Q-values (*i.e.,* $Q(S_t, A_t)$) and Target Q-values (*i.e.,* $Q^{+}(S_t, A_t) = R_t + \gamma \cdot \min_{\text{A}} Q(S_{t+1}, A_t)$); this leads to highly correlated estimations. The DQN agent takes the action A_t that has the best estimated Q-value by the NN for the state S_t, it receives a reward R_t for the action taken and observes the new state S_{t+1}. This state is then used by the NN to compute the Q-value. Some DQN implementations employ two NNs (Online NN and Target NN) to learn the optimal action for each learning step as well as achieve learning stability [36]. We refer the reader to work performed in [37] to a complete description of DRL techniques.

III. DRL-based Routing Architecture

In this section, we introduce a DRL-based routing architecture. As illustrated in Figure 1, the architecture is composed of the *Topology Module*, the *Monitoring Module*, the *Processing Module*, the *Routing Module*, and the *Installation Module*. These modules i) gather raw network data (*e.g.,* port statistics) from the forwarding devices located in the *Topology Module* (1 in the figure), ii) process the raw data to compute path-state metrics (2), iii) explore and learn information about the state of a path (path-state) to compute optimal paths for all source-destination nodes in the network (3), iv) retrieve routing path information (4); and v) execute installation of flows in the forwarding devices according to the paths computed (5).

The *Topology Module* represents the set of forwarding devices (*i.e.,* switches) in the underlying network. This module responds to query requests with statistical information

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidade Estadual de Campinas. Downloaded on March 16,2022 at 22:17:31 UTC from IEEE Xplore. Restrictions apply.

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/TNSM.2021.3132491, IEEE Transactions on Network and Service Management

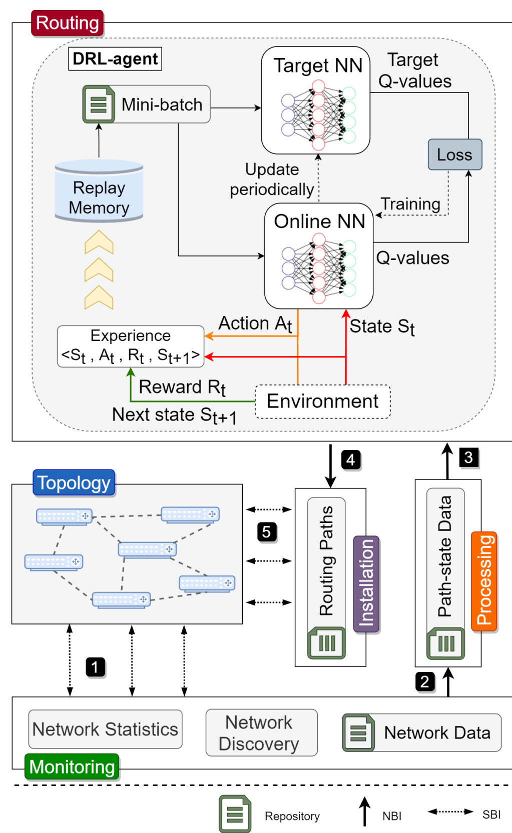


Fig. 1: DRL-based Routing Architecture

gathered at the ports of the switches during running time. The *Monitor Module* gathers this statistical information and produces a topology map by employing the *Network Discovery* and *Network Statistics* functions. The *Network Discovery* function exchanges messages with the *Topology Module* containing feature information. With the information received, the *Network Discovery* function maps switch ports to the ports of its neighboring switches and furnishes the topology as a set of tuples (*i.e.,* switch id, port id, neighbor switch id, neighbor port id) in the *Network Data Repository*. The *Monitor Module* detects topology changes related to nodes and links by periodically gathering topology information at each Change Detection Time (t_{chad}). The *Network Statistics* function is responsible for gathering statistical information by exchanging *state messages* with each switch at every monitoring time (t_{mon}). The reply messages from the *Topology Module* are stored in the *Network Data Repository* and provide an updated global view of the network.

The *Processing* Module retrieves raw data from the *Network Data Repository*, and then computes metrics that characterize the state of a path (path-state metrics), which are stored in the *Path-state Data Repository* as a set of entries containing information about the source, destination, and a tuple of metrics. The path-state metrics serve as input features for the *Routing Module*, which employs DRL to learn the network behavior and create an optimal routing plan. The *Routing Module* hosts a DRL agent using DQN (as described in subsection IV-A). The DRL agent learns and calculates paths by running the DRL-based routing algorithm (as described in subsection IV-B) during a Learning Time (t_{lear}). The *Routing Module* passes the routing plan on to the *Installation Module* and saves it in the *Routing Paths Repository*. Each entry in such a repository is a tuple representing the source, destination, and selected path.

The *Installation Module* retrieves path information from the *Routing Paths Repository* and installs/updates flow rules in the switches. The *Installation module* proactively populates the flow tables of the switches (*i.e.,* prior to all traffic matches), which takes place in a time interval that depends on the number of forwarding tables and the number of flow entries to update, called Installation Time (t_{inst}). The path-state metrics used by the DRL agent of DRSIR are periodically computed and updated in the monitoring interval of the *Network Statistics* function (t_{mon}). The paths computation and installation tasks are periodically executed, in this interval, using the updated values of the metrics that characterize the path-state.

The DRL-based Routing Architecture is based on the concept of SDN. In particular, the *Topology Module* represents the Data Plane, the *Monitoring* and *Installing Modules* work in the Control Plane as controller applications, the *Processing Module* corresponds to the Management Plane, and the *Routing Module* represents a Knowledge Plane added on the top of the SDN Planes that relies on the knowledge gathered about the network and the DRL cognitive capabilities.

IV. DRSIR

DRSIR is a DRL-based routing approach for SDN. It employs the architecture presented in Figure 1, which uses metrics describing the states of the paths to explore, learn, and exploit potential paths for all the source-destination pairs. The metrics used are path bandwidth, path delay, and path packet loss ratio. In this section, we provide details about the DRL-based agent and the DSIR routing algorithm.

A. DRSIR Agent

Figure 1 depicts the DQN agent hosted in the *Routing Module*, which also hosts the environment for training the agent on the basis of the Action and State Spaces and the path-state information provided by the *Processing Module*. The reward value R_t is computed for action A_t at state S_t, allowing the agent to find the next state S_t to be visited. The Reward function gives the path cost quantified by path-state metrics. The DQN agent learns the policy that selects actions which will minimize the reward function. The DRSIR agent learns to avoid high delay and loss ratios and prioritize paths with large available bandwidth for making intelligent routing decisions. The DQN agent includes an Online NN, a Target NN, and a *Replay Memory* database [36] to minimize the reward, learn fast, and enhance stability in learning.

Next, we give details about the state and action spaces, the reward function, the optimal policy, the Replay Memory, the Online and Target NNs, and the exploration method used by the DRSIR DQN agent.

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidade Estadual de Campinas. Downloaded on March 16,2022 at 22:17:31 UTC from IEEE Xplore. Restrictions apply.

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/TNSM.2021.3132491, IEEE Transactions on Network and Service Management

1) State Space (S): The State Space is the set of states that the DQN agent can observe. Each state corresponds to a source-destination pair of nodes that communicates over the network. Given a network topology with N nodes, the number of source-destination pairs is given by the k-permutations $P(N, j)$ of N, where $j = 2$. Therefore, the size of the State Space is $|S| = |P(N, j)| = N!/(N-2)!$. The *Network Discovery* function builds a map of the topology of the network stored in the *Topology Module*, and the DQN agent builds the State Space from the topology map.

2) Action Space (A): The Action Space is the set of actions that can be taken on the states in the State Space. Each action $A_t \in [0, ..., k]$ corresponds to the selection of a specific end-to-end path $p_i \in [p_0, ..., p_k]$ for a given state S_t. Therefore, the DQN agent can select one path among a list of k candidate paths that connect a source and a destination node. DRSIR defines the k shortest paths for each source-destination pair, computed by the *Processing Module*.

3) Reward Function: A Reward value is computed on the basis of path-state metrics and gives the cost of a potential path in the Action Space. DRSIR uses the *Monitor* and *Processing Modules* located at the SDN controller to calculate the cost of all potential paths in the Action Space. First, the *Processing Module* computes the instantaneous throughput bwa_{link}, delay d_{link} and packet loss ratio l_{link} of all links, considering the number of packets that passed through the switch port connected to the link. At each port, the SDN controller periodically samples the number of bytes transmitted and received. By comparing the retrieved values at two instants, it is possible to compute the instantaneous throughput. After the sending of a *request-stats* to the *Topology Module* at time t_1, a reply message arrives containing the number of received bytes, b_{t_1}. Then, after a period p, another *request-stats* message is sent at time t_2, and the number of bytes received b_{t_2} is retrieved from its reply message. The expression $bwu_{link} = [(b_{t_2} - b_{t_1})/p]$ gives the instantaneous throughput, where p is the duration of the sampling interval. The available link bandwidth bwa_{link} is thus computed as the difference between the link capacity cap_{link} and the instantaneous throughput of the link bwu_{link}, $bwa_{link} = cap_{link} - bwu_{link}$. The expression $l_{link} = (btx_{t_1} - brx_{t_2})/btx_{t_1}$ gives the instantaneous loss ratio, where the values of the transmitted bytes btx_{t_1} and received bytes brx_{t_2} at the times t_1 and t_2 are retrieved from the reply messages that have arrived after a *request-stats* message.

The computation of the instantaneous delay value follows the method described in [38], which uses the Link Layer Discovery Protocol (LLDP) [39] and OpenFlow messages [40]. The *Processing Module* sends an LLDP message via the SDN controller c_0; it goes through the path c_0-s_i-s_j-c_0, with s_i and s_j being the switches connected by the link (s_i, s_j). The time elapsed between transmission and reception of the LLDP message is the difference between the timestamp values ($d_{lldp_{cij}}$). The time taken by the message to go from c_0 to the s_i port (c_0-s_i) is estimated as half the time elapsed between the transmission and reception of the OpenFlow *echo_request* and *echo_reply* messages sent by c_0 to s_i. A similar procedure is used to estimate the time elapsed as the message goes from s_j to c_0. The expression $d_{si-sj} = d_{lldp_{cij}} - d_{c_0-s_i} - d_{c_0-s_j}$ gives the instantaneous delay in the link (s_i, s_j). For a given path $P \in A$, the *Processing Module* uses the bwa_{link}, d_{link} and l_{link} of each link $i \in P$, to compute the bwa_{path}, delay d_{path} and packet loss ratio l_{path} employing Equations 2, 3, and 4.

$$bwa_{path} = \min_{\mathrm{i \in P}}(bwa_{link_i}) \tag{2}$$

$$d_{path} = \sum_{i \in P} d_{link_i} \tag{3}$$

$$l_{path} = 1 - \prod_{i \in P}(1 - l_{link_i}) \tag{4}$$

The Reward Function is inversely proportional to the path available bandwidth bwa_{path} and directly proportional to the path delay d_{path} and the path packet loss ratio l_{path}. To avoid one of the path-state metrics having more influence than the others in the learning process, we normalize the metrics values using the Min-Max technique [41], which rescales the range of the values of the metrics to a range with values in an arbitrary interval $[a, b]$. The Reward Function is defined in Equation 5, where $\hat{bwa_p}$, $\hat{d_p}$, $\hat{l_p}$ are the normalized values of the path available bandwidth, delay, and loss ratio, respectively. The values β_1, β_2 and, $\beta_3 \in [0, 1]$ are tunable parameters useful in providing a weight value for a metric in the reward calculation. These weight values are used to give importance to specific metrics related to QoS requirements in the reward computation.

$$\hat{R} = \beta_1 \cdot \frac{1}{\hat{bwa_p}} + \beta_2 \cdot \hat{d_p} + \beta_3 \cdot \hat{l_p} \tag{5}$$

4) Target and Online Neural Networks: The DQN agent employs two NNs, the Online and Target Networks. The Online NN estimates the Q-values of the current state S_t (*i.e.,* $Q(s_t, A_t)$), while the Target NN outputs the Q-values of the next state S_{t+1} (see Equation 6). The Online NN is trained at each learning step to decrease the loss function (Equation 7). At the start of the learning process, the weights of the Target NN and those of the Online NN are the same. The values of these weights are temporarily frozen to enhance learning stability. During the training phase, the weights of the Target NN are periodically updated to match the Online NN after a pre-determined number of learning steps.

$$Q^{+}(S_t, A_t) = R_t + \gamma \cdot \min_{\mathrm{A}} Q(S_{t+1}, A_t) \tag{6}$$

$$Loss = (Q^{+}(S_t, A_t) - Q(S_t, A_t))^2 \tag{7}$$

In DRSIR, the Online and Target NNs have the same structure, including an input layer, one or more hidden layers, and an output layer. For each state in the State Space, the DQN agent encodes each source and destination pair as a state. The input layer has one neuron that receives the state as the input of the NN. The output layer has k neurons, *i.e.,* one neuron for each of the k actions in the Action Space A. Each neuron in the output layer estimates a Q_i-value associated with action $a_i \in A$. The number of hidden layers is defined by testing in Section V-D.

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidade Estadual de Campinas. Downloaded on March 16,2022 at 22:17:31 UTC from IEEE Xplore. Restrictions apply.

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/TNSM.2021.3132491, IEEE Transactions on Network and Service Management

5) Replay Memory: The DQN agent stores past decisions (experiences) in a dataset with entries of the form S_t, A_t, R_t, S_{t+1} in the *Replay Memory*, which allows sampling batches of experiences and training offline on previously observed data. We use this dataset to reduce the number of interactions needed by the DQN agent to learn [36]; minibatches can be sampled for this purpose, thus reducing the variance in learning updates [42].

6) Exploration Method: DRSIR uses the Decay ε-greedy exploration method [43]. This method employs a tunable parameter, $\varepsilon \in [0, 1]$, to determine if the agent should exploit with a probability $pr = \varepsilon$ or explore with a probability $pr = 1 - \varepsilon$. The ε value is usually set to start at a maximum value ε_{max} and linearly decrease at a decay rate $decr$ throughout the learning process (steps) until reaching a minimum value ε_{min}, according to the expression $\varepsilon = \varepsilon_{max} - (steps \times decr)$. This decrease allows the DQN agent to go from an exploratory policy at the start of the learning process to a more exploitative one as the learning process progresses [34]. The DQN agent follows Equation 8 to select the next action in a specific state. For each learning step, a random value $x \in [0, 1]$ is generated. If $x < \varepsilon$, the agent exploits. Otherwise, the agent explores.

$$\mathcal{A} = \begin{cases} \min_{\text{A}} Q_t(S_t, A), & if \quad x < \varepsilon \\ random\, action, & otherwise \end{cases} \tag{8}$$

B. DQN-based Routing Algorithm

DRSIR introduces a routing algorithm that implements a learning process to find the best paths for all the node pairs in the network. Algorithm 1 receives the following parameters as input: i) the number of learning episodes n, ii) the maximum value of the ε parameter, iii) the decay rate $decr$, the number of steps prior to starting the training employing the dataset stored in the *Replay Memory* (replay start size rss), iv) the frequency of steps in which the Target NN is updated (target update frequency tup), v) the list of the "k" paths per state k_{paths}, and vi) information on the state of the paths. The output is the set of best-rewarding routing paths for all pairs of nodes in the network. The path is formed by the state-action pairs with the lowest values in the Q-table.

Algorithm 1 initializes the Online and Target NNs with weights θ and $\hat{\theta}$, respectively, as well as the *Replay Memory* (Line 1). The routing algorithm goes through episodes according to the steps in the loop from Line 2 to Line 21, starting in an initial random state S_t. For each episode, the DQN agent goes through a finite number of steps in an inner loop and reaches the next state S_{t+1}, the ending state (*i.e.,* state after going through m-steps) (Lines 4 to 20). Therefore, a learning episode comprises a sequence of steps that corresponds to the states between an initial state and a final state. Each step consists of selecting and performing an action, changing the state, and receiving a reward.

The inner loop (Line 4 to Line 20) is executed as follows. First, the DQN agent updates the value of the ε parameter by considering the decay rate $decr$ (Line 5). It selects an action from the Action Space (selects A_t for S_t), which is the selection of a path from the k candidate paths for the

Algorithm 1: DRSIR Deep Q-learning routing process

```
Input :
    Number of learning episodes: n
    Exploration and exploitation parameter: ε_max
    Decay rate: decr
    Replay start size: rss
    Target update frecuency: tup
    List of "k" paths per state: k_paths
    Network path-state
Output: Set with the best routing path for all pairs of nodes in the
    network
1  Initialize Online NN with weights θ, Target NNs with weights θ̂ and
     Replay Memory
2  for episode ← 1 to n do
3      The agent gets the initial state S_t // A random pair of
         nodes (state);
4      while next state S_{t+1} is not a final state do
5          Update ε = ε_max − (steps × decr);
6          Select A_t for S_t by using ε-greedy exploration and
             exploitation method with Equation 8;
7          R_t ← R(S_t, A_t) // Agent gets the reward
             calculated from network path-state;
8          The agent gets the new state S_{t+1};
9          Save experience exp_t = (S_t, A_t, S_{t+1}, R_t) into Replay
             Memory;
10         if steps > rss then // Start training
11             Sample random mini-batch of experiences from
                 Replay Memory;
12             Estimate Q(S_t, A_t) with Online NN;
13             Calculate Q⁺(S_t, A_t) = R_t + γ · min_A Q(S_{t+1}, A_t)
                 with Target NN;
14             Minimize loss (Equation 7) with gradient descent and
                 backpropagation and update θ of Online NN;
15             if steps % tup == 0 then // Every tup steps
16                 Update θ̂ of the Target NN and biases with θ of
                     the Online NN;
17             end
18             S_t ← S_{t+1} // Move to the new state;
19         end
20     end
21 end
22 Use final θ to retrieve the path from k_paths that corresponds to the
     action with the lowest Q-value for each state;
23 Store the set of paths for all pair of nodes in the network into the
     Routes Data Repository
```

current state by using the ε-greedy method (Line 6). Then, the DQN agent obtains a reward R_t computed in the environment by using Equation 5 (Line 7). The DQN agent also gets the next state S_{t+1} (the next pair source-destination) (Line 8). Finally, the experience $exp_t = (S_t, A_t, S_{t+1}, R_t)$ is saved in the *Replay Memory* (Line 9).

The DQN agent stores the experiences in the *Replay Memory*, and verifies the number of steps needed to start the training of the Online NN, rss steps, (Line 10). When rss is reached, the DQN agent randomly takes a mini-batch from the *Replay Memory* to train the Online NN (Line 11). Training using experiences allows the agent to learn after fewer interactions than does an RL agent. Then, the DQN agent estimates $Q(S_t, A_t)$ by using the Online NN (Line 12), and calculates the associated target values $Q^+(S_t, A_t)$ by using the Target NN (Line 13). After that, the gradient descent and backpropagation algorithms are used to adjust the weights and biases of the Online NN and minimize the loss calculated by Equation 7 (Line 14). After the execution of tup steps and the storage of a certain number of experiences, the DQN

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidade Estadual de Campinas. Downloaded on March 16,2022 at 22:17:31 UTC from IEEE Xplore. Restrictions apply.

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/TNSM.2021.3132491, IEEE Transactions on Network and Service Management

agent updates the weights and biases $\hat{\theta}$ of Target NN with the weights and biases θ of the Online NN (Line 15 to Line 17). The DQN agent then moves to the next state (Line 18). Finally, after the DQN agent makes the transition, it uses the final θ value to retrieve the path from k_{paths} that corresponds to the action with the lowest Q-value for each state (Line 22). Once the DQN agent finds the best path for all pairs of source-destination nodes, it stores them in the *Routes Data Repository* (Line 23). The *Installation Module* retrieves these best paths and installs them in the routing tables of the switches.

The worst-case complexity of Algorithm 1 is derived as follows. DQN relies on the use of RL and the generalization given by the NNs. In the worst case, the RL agent visits all the states in the State and Action spaces, which would imply a complexity dependent on the size of the spaces. In DRSIR, the State Space size is $O(N^2)$, where N is the number of source-destination pairs in the network. The size of the Action Space is limited to the k potential paths associated with each state. The complexity for RL is, then, $O(kN^2)$. Since k is a constant, the worst-case complexity is $O(N^2)$.

V. Evaluation

This section presents the evaluation of DRSIR. Subsection V-A depicts the test environment. Subsections V-B and V-C show the performance metrics and traffic generation procedure, respectively. Subsection V-D presents the set up of the learning parameters, and Subsection V-E discusses the results.

A. Test Environment and Prototype

The DRSIR was evaluated for several topologies; in this paper, the results are shown for 23-node and 48-node topologies. The 23-node topology mirrors the GÉANT topology [44], an European data network, which, in 2004, had 37 links, with 50% of them having 10 Gbps, 40% having 2.5 Gbps, and 1% having 155 Mbps. We scaled these capacities to 100 Mbps, 25 Mbps, and 1.55 Mbps due to limitations imposed by the resources available in the machines running the Mininet emulator. The 48-node topology was generated using the Barabasi-Alberth algorithm [45]; which has 60 links with equal capacities of 100 Mbps. For both topologies, each switch had a host that forwarded and received traffic. The Data Plane was developed using Mininet 2.2.2 [46] with Open vSwitches 2.3.1 and Python scripts running on an Ubuntu Server 14.04 Virtual Machine (VM) with 8 GB RAM.

We used the Application Program Interface (API) of the Ryu controller [47] to develop the *Monitoring* and *Installation Modules* with Openflow 1.3 [48] as the Southbound Interface (SBI) to communicate with the *Topology Module*. We developed the *Processing Module* and the DQN agent by using Python 3.5 with Tensorflow 2.2.0 [49], Numpy 1.16.4 [50], and the Pandas 0.22 library [51]. We used Comma-Separated Values (CSV) and JavaScript Object Notation (JSON) files to store information in the Repositories. Every module in the DRL-based architecture ran on an Ubuntu 16.04 VM with a Core i5-4690 processor and 10 GB RAM. The VMs used for this prototype were hosted on an Ubuntu Desktop 16.04 with an Intel Core i5-4690 and 16 GB RAM. These VMs communicated using the Transmission Control Protocol (TCP). The DRSIR prototype, as well as all test scripts, are available in [52].

B. Performance Metrics

Table II shows the most common performance metrics used to evaluate the routing proposals: the link throughput, link loss ratio, and link delay. Moreover, we evaluated the stretch of the paths that compares the length of a path to the theoretical shortest path [53]. We developed an application on the SDN controller which computed the shortest path by implementing the Dijkstra algorithm with equal edge weights. We computed the performance metrics as explained in Subsection IV-A.

C. Traffic Generation

In the emulation experiments, we developed scripts for generating traffic using the *iperf3* tool. The scripts generated User Datagram Protocol (UDP) traffic for clients and servers on the hosts, allowing the setting of transmission rates specified by the traffic matrices. The scripts did not use the pacing option, thus generating the burstiest possible traffic pattern since packets are generated at the beginning of every interval used to verify the conformance of the generated traffic with the specified mean transmission rate[1]. The coefficient of variance of the number of bytes arrived in a time scale of 1ms varied from 6.32 to 10.38 in the experiments reported in this paper. Sixteen publicly available intra-domain traffic matrices with elements containing values collected from the GÉANT topology were used for the simulation of the 23-node topology. They were periodically collected from all the edge links within a 15 minutes interval during four months. The volume of bytes transmitted was an aggregation on the basis of source-destination pairs. The GÉANT's traffic matrices are fully described in [54].

For the 48-node topology, fourteen traffic matrices were generated via the Modulated Gravity Model implemented by the TMGen tool [55]. TMGen is a python library able to generate spatial, temporal, and spatio-temporal traffic matrices. The generated traffic matrices are arrays containing traffic volume between origin-destination pairs for different time epochs and modulated by a simple sinusoidal with a prescribed mean, and peak-to-mean ratio [56]. The matrices were generated for different hours of the day where peak hours have high traffic intensity (from 7:00h to 14:00h) regarding other hours of the day and can be found in [52].

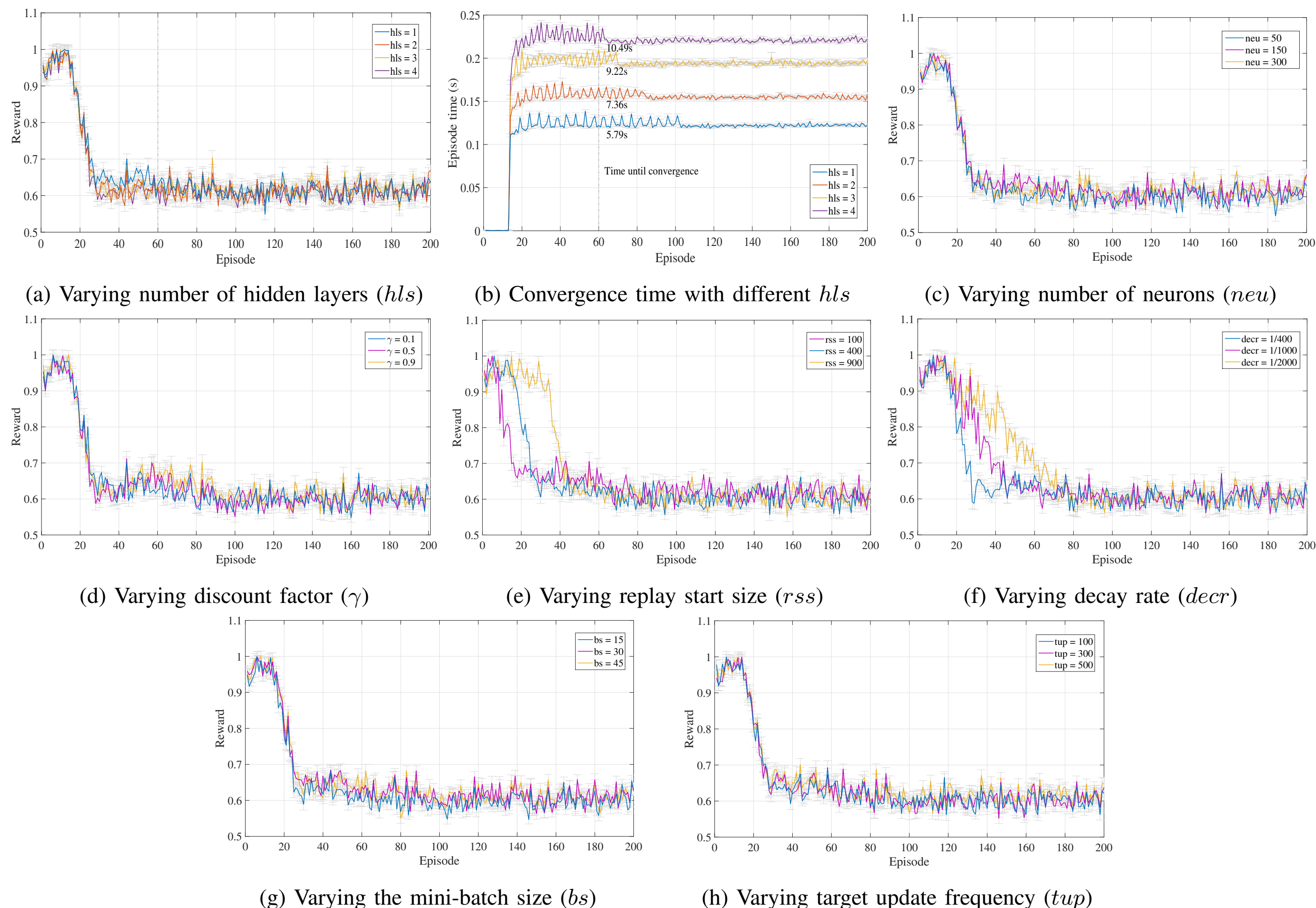
D. Learning Parameters Setup

The widely used Adaptive moment estimation (Adam) optimizer [57] was employed by the DQN agent. The Adam optimizer improves the gradient descent algorithm allowing quick convergence in the training of deep networks. Moreover, the Adam optimizer is also straightforward to implement and requires little memory [58]. The Glorot uniform initializer [59] for weights initialization and the ReLu activation function [60] were employed by the DQN agent. These techniques have been

[1] https://www.qacafe.com/resources/testing-microburst-effects-using-iperf/

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidade Estadual de Campinas. Downloaded on March 16,2022 at 22:17:31 UTC from IEEE Xplore. Restrictions apply.

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/TNSM.2021.3132491, IEEE Transactions on Network and Service Management



(a) Varying number of hidden layers (hls)

(b) Convergence time with different hls

(c) Varying number of neurons (neu)

(d) Varying discount factor (γ)

(e) Varying replay start size (rss)

(f) Varying decay rate ($decr$)

(g) Varying the mini-batch size (bs)

(h) Varying target update frequency (tup)

Fig. 2: Reward with respect to different learning parameters (number of neurons, γ, rss, $decr$, bs, and tup)

shown to maintain similar variance values during the learning process in all NN layers.

In the training and evaluation experiments, we did not consider different QoS requirements. Therefore, the tunable parameters β_1, β_2 and, β_3 in the reward function in Equation 5 were set to 1, giving the same relevance to the optimization of the three metrics that influence the reward computation. Furthermore, we set $k = 20$ candidate paths (*i.e.,* actions in the Action Space) that connect a source to a destination node, and the monitoring interval to $t_{mon} = 10s$.

For setting the learning parameters in the training of the DQN agent, the convergence of a minimized reward in episodic training was used as a parameter for comparison. Several preliminary tests were conducted to set the learning parameter values. Figure 2 shows the observed reward as a function of the following learning parameters: number of hidden layers hls in conjunction with convergence time, number of neurons in hidden layer neu, discount factor γ, number of steps to the start of training the Online NN (replay start size rss), decay rate $decr$ at which the DQN agent explores and exploits, the size of the mini-batch used to train the NNs, and the number of steps indicating the updating frequency of the Target NN (target update frequency tup). For the sake of brevity, we show only three or four results for each parameter.

We ran Algorithm 1 to observe the reward returned in a single episodic training composed of 200 episodes with 30 steps per episode for each test, to identify the algorithm convergence to a minimized reward value as a function of the learning parameters. Figure 2a shows that after roughly the 60th episode, the reward no longer decreased significantly as a function of the number of hidden layers. Figure 2b shows the impact on the convergence time for using one or more hidden layers until the 60th episode. With $hls = 1$, the produced reward is similar to that with other values of hls after convergence. However, it took only $5.79s$ to achieve these reward values, while with $hls = 2, 3$ and 4, the convergence time was $1.6s, 3.46s$ and $4, 73s$ longer. Thus, the Target and Online NNs were defined with a single hidden layer, which proved to be sufficient to approximate any function correctly [61]. Figure 2c shows that a hidden layer with 50 neurons obtained a slightly lower reward than hidden layers with 150 or 300 neurons. Moreover, a larger number of neurons required more extended training periods.

Figure 2d depicts the reward as a function of γ. For $\gamma = 0.1$, the DQN agent slightly decreased the reward, but the value was still close to those given by γ equals to 0.5 and 0.9. In Figure 2e, the rss parameter indicated the number of steps before starting to exploit the knowledge obtained from

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidade Estadual de Campinas. Downloaded on March 16,2022 at 22:17:31 UTC from IEEE Xplore. Restrictions apply.

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/TNSM.2021.3132491, IEEE Transactions on Network and Service Management

experience when training the NNs, *e.g.,* the value $rss = 100$ means that the training of the NNs started after 100 steps. Results show that a value of $rss = 400$ quickly caused the agent to start minimizing the reward, reaching a slightly lower reward than when rss was 100 or 900. For the exploitation and exploration method, ε_{max} was set to 1 to ensure the maximum probability of exploration when the learning process started. Figure 2f shows that with low $decr$ values such as $1/1000$ and $1/2000$, the DQN agent required more episodes to converge to a minimized reward than when using a high value such as $1/400$. Figure 2g shows that a mini-batch $bs = 15$ produces a slight improvement in convergence than did values of $bs = 30$ and 45. The parameter tup indicates the number of steps after which the weights and biases of the Target NN are updated. Figure 2h shows a slight improvement when $tup = 100$ rather than 300 or 500.

Table I summarizes the parameter values used in the evaluation of DRSIR. The chosen values for the learning parameters led to the minimization of the reward for the network topologies employed in the evaluation (see Figure 2).

TABLE I: Parameter Values used in the Evaluation

DRSIR	
Parameter	**Value**
Hidden layers (hls)	1
Nuerons (neu)	50
Discount factor (γ)	0.1
Replay start size (rss)	400 steps
Decay rate ($decr$)	$1/400$
Batch size (bs)	15
Target update frequency (tup)	100
Reward function weights ($\beta_1, \beta_2, \beta_3$)	1
Number of actions in Action Space (k)	20

E. Performance Analysis

1) Baselines: The DRSIR DQN-based routing is compared to that given by the RSIR algorithm [18]. Since RSIR uses link-state metrics (*i.e.,* $\text{RSIR}_{\text{links}}$), we have modified the RSIR algorithm to use path-state metrics (*i.e.,* $\text{RSIR}_{\text{paths}}$), so that a fair comparison with the DRSIR algorithm can be made. Figure 3 shows the performance of $\text{RSIR}_{\text{links}}$ and $\text{RSIR}_{\text{paths}}$ throughout the day for the 23-node topology. Figure 3a suggests that $\text{RSIR}_{\text{links}}$ produces shorter paths (on average 7%) than does $\text{RSIR}_{\text{paths}}$. Figures 3b and 3c show that $\text{RSIR}_{\text{links}}$ produces greater mean delay (on average 6%) and mean loss ratio (on average 27%) than does $\text{RSIR}_{\text{paths}}$. Figure 3d shows the mean link throughput, showing that $\text{RSIR}_{\text{links}}$ produces a slightly broader distribution of traffic across the network (< 4%) than does $\text{RSIR}_{\text{paths}}$. Nevertheless, the mean loss and mean delay values show that the paths chosen by $\text{RSIR}_{\text{paths}}$ are less congested. These results suggest that $\text{RSIR}_{\text{paths}}$ overperforms $\text{RSIR}_{\text{links}}$. Moreover, the results corroborate that the use of path-state metrics reduces knowledge abstractions needed by the routing agent since the agent can directly explore different path options rather without being limited to link state information. From now on, RSIR will be used to designate $\text{RSIR}_{\text{paths}}$.

The DRSIR algorithm was also compared to different variations of Dijkstra's algorithm. Although Dijkstra's algorithm is broadly deployed in real networks, routing decisions on scheduling flows on paths neither consider the impact of such scheduling on other flows nor accounts for the impact of past decisions. The Dijkstra's algorithm variations employed instantaneous delay ($\text{Dijkstra}_{\text{delay}}$), instantaneous loss ($\text{Dijkstra}_{\text{loss}}$), and available link bandwidth ($\text{Dijkstra}_{\text{bw}}$) as edge weights. Moreover, DRSIR was compared to a Dijkstra-based routing algorithm with edge weight values defined by Equation 9, which considers all the previously mentioned link metrics, designated ($\text{Dijkstra}_{\text{comp}}$). In an attempt to conduct a fair comparison, all the routing variations of the Dijkstra's algorithm (developed and executed as applications on the SDN controller) and the RSIR algorithm were subject to the same traffic scenarios applied to the DRSIR algorithm in the evaluated scenarios.

$$weight = \frac{1}{bwa_{link}} + d_{link} + l_{link} \tag{9}$$

Next, the results of DRSIR algorithm are compared to those of RSIR and the variations of the Dijkstra's algorithm. These are averaged hourly; the figures also show the traffic generated per hour.

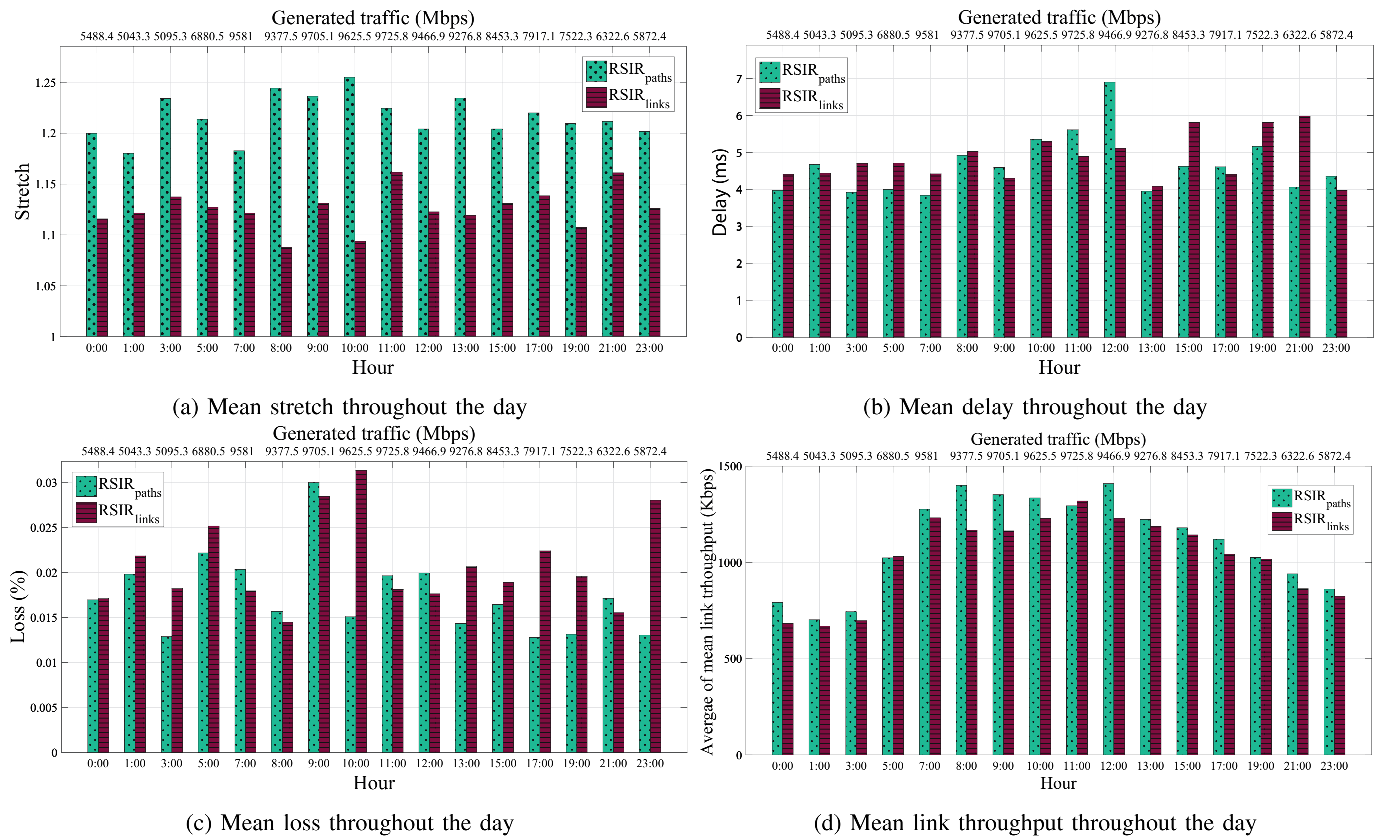
2) Results for 23-node topology: Figure 4 shows the stretch, link delay, loss ratio, and link throughput produced by DRSIR, RSIR, and the variations of the Dijkstra's algorithm for the 23-node topology. Figure 4a shows the values of the mean stretch averaged over all paths. The paths chosen by the DRSIR present a smaller stretch value than those produced by the $\text{Dijkstra}_{\text{delay}}$, $\text{Dijkstra}_{\text{loss}}$, and $\text{Dijkstra}_{\text{comp}}$. In particular, we can see that DRSIR selects a larger number of shorter paths than do the three other algorithms; with stretch values 5%, 10%, and 5% lower than those obtained by the $\text{Dijkstra}_{\text{delay}}$, $\text{Dijkstra}_{\text{loss}}$, and $\text{Dijkstra}_{\text{comp}}$ algorithms, respectively. Moreover, the paths produced by the DRSIR are shorter (up to 8%) than those produced by RSIR. The results also show that the DRSIR indicates paths with stretch values slightly higher (6%) than those resulting from $\text{Dijkstra}_{\text{bw}}$.

Figure 4b presents the mean link delay resulted from the use of DRSIR, RSIR, and the variations of the Dijkstra's algorithm. The mean link delay of the DRSIR algorithm are, on average, 14%, 43%, 59%, 25%, and 15%, at most, 44%, 73%, 87%, 44%, and 40% lower than those given by RSIR, $\text{Dijkstra}_{\text{delay}}$, $\text{Dijkstra}_{\text{loss}}$, $\text{Dijkstra}_{\text{bw}}$, and $\text{Dijkstra}_{\text{comp}}$, respectively. The mean link delay results show that the DRSIR algorithm tends to choose less congested paths than do the RSIR and Dijkstra's algorithms; therefore, the delays obtained using DRSIR are shorter than those given by RSIR and the four variations of the Dijkstra's algorithms.

Figure 4c reveals that the mean losses resulting from the use of DRSIR are less than those from RSIR, $\text{Dijkstra}_{\text{delay}}$, $\text{Dijkstra}_{\text{loss}}$, $\text{Dijkstra}_{\text{bw}}$, and $\text{Dijkstra}_{\text{comp}}$. In particular, the values produced by the DRSIR algorithm are, on average, 51% and, at most, 73% lower than those obtained by the variations of the Dijkstra's algorithm, and on average 17% and, at most, 55% lower than that produced by the RSIR algorithm. Results show that the Dijkstra's algorithm variations usually select

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidade Estadual de Campinas. Downloaded on March 16,2022 at 22:17:31 UTC from IEEE Xplore. Restrictions apply.

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/TNSM.2021.3132491, IEEE Transactions on Network and Service Management



(a) Mean stretch throughout the day

(b) Mean delay throughout the day

(c) Mean loss throughout the day

(d) Mean link throughput throughout the day

Fig. 3: Performance metrics comparison of $\text{RSIR}_{\text{paths}}$ and $\text{RSIR}_{\text{links}}$ in 23-node topology.

longer routes and more frequently use low-capacity links, causing traffic concentration and congestion in these links.

Figure 4d shows the mean link throughput throughout the day. The values produced by DRSIR are on average 20%, 18%, 33%, 18% and 30%, and at most, 31%, 30%, 41%, 29% and 42%, lower than those produced by the RSIR, $\text{Dijkstra}_{\text{delay}}$, $\text{Dijkstra}_{\text{loss}}$, $\text{Dijkstra}_{\text{bw}}$ and $\text{Dijkstra}_{\text{comp}}$ algorithms, respectively. The mean link throughput obtained by DRSIR is the lowest, and the mean delay and loss values are also lower (in most of the cases) because DRSIR uses a greater number of less utilized paths than do RSIR, $\text{Dijkstra}_{\text{delay}}$, $\text{Dijkstra}_{\text{loss}}$, $\text{Dijkstra}_{\text{bw}}$, and $\text{Dijkstra}_{\text{comp}}$.

The results for the 23-node topology show the superior performance of DRSIR compared to those of the variations of the Dijkstra's algorithm due to the direct exploration of a variety of different paths and the generalization provided by the Target and Online NNs to the DQN agent in learning optimal routes.

3) Results for a 48-node topology: Figure 5 presents the results for a 48-node topology. Figure 5a shows the mean stretch computed for all the paths found by DRSIR, RSIR, and the variations of the Dijkstra's algorithm. The results also show that DRSIR finds a larger number of shorter paths than do the $\text{Dijkstra}_{\text{delay}}$, $\text{Dijkstra}_{\text{loss}}$, and $\text{Dijkstra}_{\text{comp}}$ algorithms for this topology with stretch values 23%, 7%, and 17% smaller than those obtained by the $\text{Dijkstra}_{\text{delay}}$, $\text{Dijkstra}_{\text{loss}}$, and $\text{Dijkstra}_{\text{comp}}$ algorithms, respectively. The DRSIR algorithm finds paths with stretch values slightly higher ($< 1\%$) than those produced by $\text{Dijkstra}_{\text{bw}}$. Moreover, these paths are shorter (9% and at most 14%) than those furnished by the RSIR algorithm.

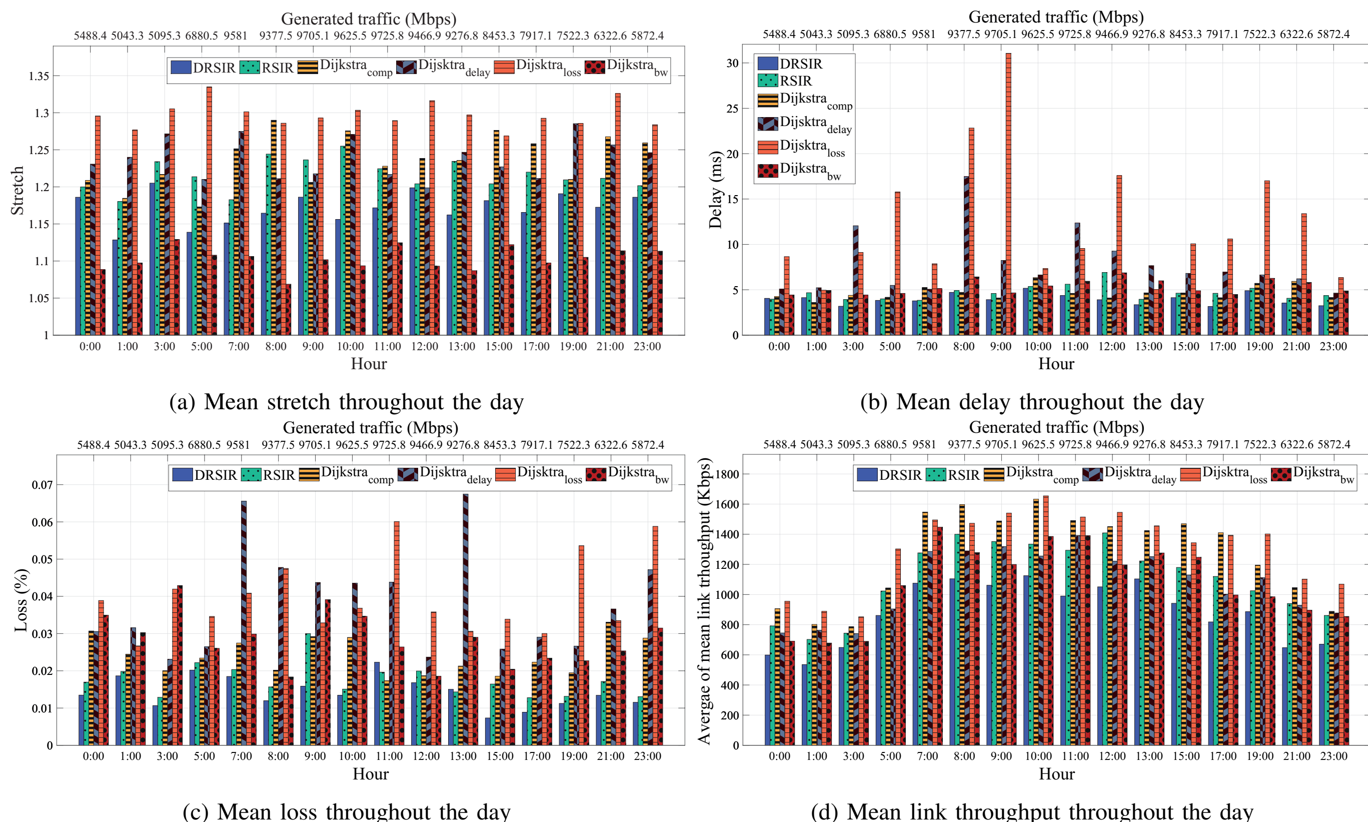
Figures 5b and 5c present the mean delay and mean loss ratio given by the DRSIR, RSIR, and the four variations of the Dijkstra's algorithm throughout the day. These results show that the DRSIR algorithm produces lower mean delay and mean loss ratio values than those produced by the RSIR algorithm and the four variations of the Dijkstra's algorithm. The mean delay produced by DRSIR are, on average, 24%, 57%, 36%, 33%, and 54% lower than the mean delay values produced by the RSIR, $\text{Dijkstra}_{\text{delay}}$, $\text{Dijkstra}_{\text{loss}}$, $\text{Dijkstra}_{\text{bw}}$, and $\text{Dijkstra}_{\text{comp}}$ algorithms, respectively. The mean loss ratio values observed with the DRSIR algorithm are on average 36% and at most 57% lower than those produced by the four variations of the Dijkstra's algorithm, and on average 10% and at most 26% lower than those obtained with the use of the RSIR algorithm.

Figure 5d presents the values of mean link throughput throughout the day. The results show that the link throughput produced by the DRSIR algorithm is at most 13%, 36%, 34%, 25%, and 40% lower than those produced by the RSIR, $\text{Dijkstra}_{\text{delay}}$, $\text{Dijkstra}_{\text{loss}}$, $\text{Dijkstra}_{\text{bw}}$, and $\text{Dijkstra}_{\text{comp}}$ algorithms, respectively. These results are due to the fact that the paths chosen by DRSIR include less utilized links than those chosen by the RSIR and variations of Dijkstra's algorithm, giving lower mean delay and loss.

The results for the 48-node topology clearly show the

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidade Estadual de Campinas. Downloaded on March 16,2022 at 22:17:31 UTC from IEEE Xplore. Restrictions apply.

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/TNSM.2021.3132491, IEEE Transactions on Network and Service Management



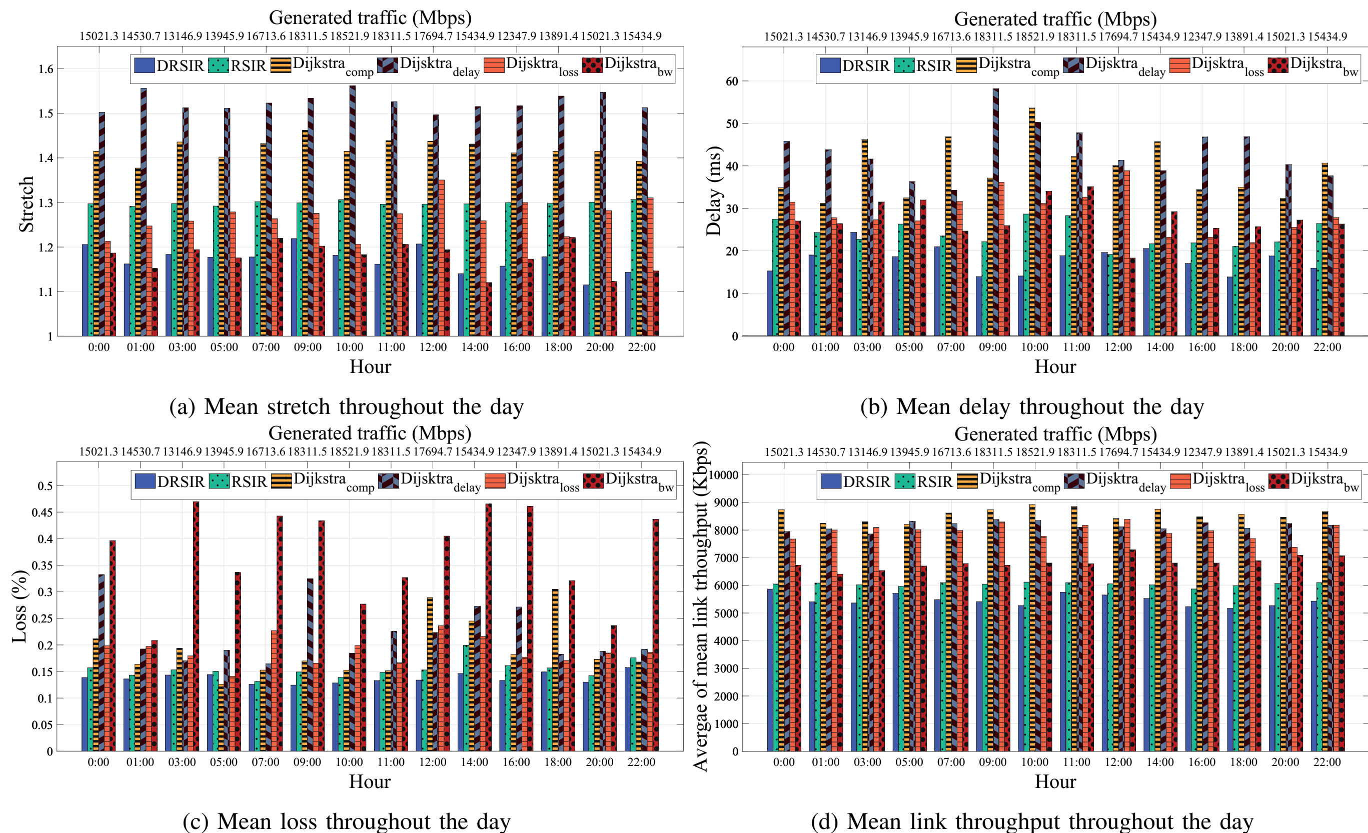
(a) Mean stretch throughout the day

(b) Mean delay throughout the day

(c) Mean loss throughout the day

(d) Mean link throughput throughout the day

Fig. 4: Performance metrics resulting from the DRSIR for 23-node topology



(a) Mean stretch throughout the day

(b) Mean delay throughout the day

(c) Mean loss throughout the day

(d) Mean link throughput throughout the day

Fig. 5: Performance metrics resulting from the DRSIR for 48-node topology

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidade Estadual de Campinas. Downloaded on March 16,2022 at 22:17:31 UTC from IEEE Xplore. Restrictions apply.

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/TNSM.2021.3132491, IEEE Transactions on Network and Service Management

advantages of DRSIR provided by the generalization furnished by NNs in the learning process of the DQN agent as well as the adoption of the routing approach based on metrics defined at the path level.

4) Topology Change Analysis: When a topology change occurs, the DRSIR algorithm detects that change, calculates new routes, and installs them. For DRSIR, the setup time t_{chad} is $1s$. The DQN agent computes all routes for the 23-node and 48-node topologies in $t_{lear} = 5.3s$ and $7,2s$ (the average execution times of Algorithm 1 for each topology), respectively. The *Installation Module* then spends on average $t_{inst} = 1.6s$ to update the flow entries. Thus, the DRSIR algorithm takes on average $t_{chad} + t_{lear} + t_{inst} = 7.9s$ and $9.8s$ to handle a change in the 23-node and 48-node topologies, respectively. In comparison, the Routing Information Protocol (RIP) would typically take $30s$ to handle such a topological change. The DRSIR algorithm also requires a shorter response time due to the employment of a centralized controller with a global view of the network, the generalization provided by the NNs, and the adoption of a network-state routing approach using metrics at the path-level. The OSPF reaction to topological changes when considering a minimum hello-interval = $1s$ and an spf-delay = $1s$ is approximately $9s$ ($5s$ + the average execution time of the Dijkstra's algorithm), which is slightly lower than the time required by the DRSIR algorithm. The RL agent of RSIR (using path-state metrics) computes all routes for the 23 and 48-node topologies in $2.8s$ and $4.7s$, respectively; thus, the RSIR algorithm takes on average $5.4s$ and $7.3s$ to handle change in the topologies. Therefore, the DRSIR algorithm obtained a t_{lear} greater than that achieved by the RSIR algorithm because the DQN agent spends more time training two NNs; however, better performance metrics more than offset this cost.

5) Overhead: In SDN, the Data Plane and Control Plane continuously interact by exchanging OpenFlow and LLDP messages to conduct topology discovery, statistics collection, and flows installation. DRSIR and RSIR use the information commonly gathered by the Control Plane during that interaction to avoid traffic overhead (extra message exchange) in the computation of the paths.

DRSIR and RSIR were also compared in relation to the consumption of CPU and storage memory of the DQN and Q-learning agents in an attempt to analyze the trade-off between overhead and performance. Figure 6 shows the results of the evaluation for the 48-node topology, disclosing that the agents do not exhaust the resources of the *Routing Module*. Indeed, they only consume 29% and 32% of the CPU capacity, and 279 and 287 MBytes of storage memory. Therefore, we can state that the DRSIR and RSIR agents are efficient in relation to these two criteria with a constant consumption of RAM of approximately 2.5%.

VI. Comparative Analysis

Table II briefly summarizes the work related to DRSIR, showing the type of control routing employed, the RL/DRL type of learning approach employed, the RL/DRL action adopted, and the metrics evaluated. The work in [13]–[17] [62] explored RL techniques, such as Q-learning, in SDN for the choice of the proper routing protocol within the environment state, the selection of per-flow path in multi-path routing and the use of the next-hop node for building a routing path. In the following paragraphs, we compare the routing solution of DSIR to those proposed in [20]–[30] [63], which employed DRL techniques for this routing in SDN. This comparison involves the use of the traditional routing protocols, learning approach and technique used, type of control routing, type of action employed, and implementation complexity. Moreover, we outline the directions for enabling multi-path routing in SDN with DRSIR.

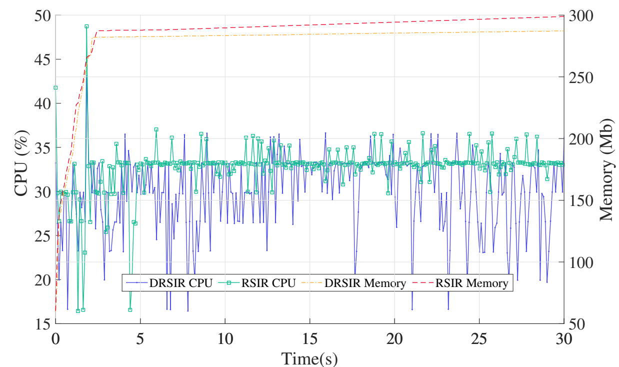


Fig. 6: CPU and Memory used by the agents of DRSIR and RSIR

Dependence on Traditional Routing Protocols. Although DRL does not require labeled datasets for agents training, the solutions in [20], [21], [23], [28] did depend on traditional routing protocols for computing paths since their agents did not learn to build or select the best path. In these papers, the agents employed are commonly those that learn to optimize link weight values and use traditional routing protocols (*e.g.,* the Dijkstra algorithm) to compute paths based on optimized weights. The work in [22], [24]–[27], [29], [30] and the DRSIR algorithm learn to select the next hop in a path or the end-to-end path for each pair of nodes.

Learning Approach. The DRSIR and the papers [20]–[30] [63] employ model-free and off-policy algorithms by learning either a value function or a policy. Specially, the work in [22], [24], [26], [30] [63], and DRSIR employed DQN, which is proper for routing problems where the MDP defines a discrete action space. The works in [20], [21], [23], [25], [27]–[29] employed actor-critic algorithms [34] such as the Deep Deterministic Policy Gradient (DDPG) and Proximal Policy Optimization (PPO) algorithms which are useful for continuous action spaces (*e.g.,* selecting link weights values for each link in a network). Unlike the DQN, the output of the DDPG algorithm is not discretized as a set of fixed actions but rather defined to return a real number/vector. Continuous actions are more challenging to learn than are discrete actions, and discretizing them can improve performance notably [64].

Routing Control. DRSIR and the works in [20]–[26], [28], [29] [63] leverage the global network view offered by SDN to deploy centralized routing strategies that use either the

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidade Estadual de Campinas. Downloaded on March 16,2022 at 22:17:31 UTC from IEEE Xplore. Restrictions apply.

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/TNSM.2021.3132491, IEEE Transactions on Network and Service Management

TABLE II: Related Work

Paper	Description	Control	Learning approach	Action	Performance metrics
[13]	QAR: QoS-aware adaptive routing with multi-layer hierarchies intended to minimize signaling delay in large SDNs	C	RL (Q-learning)	Next-hop	Delivery delay, packet loss, and hop count
[15]	CRE: a logically centralized Cognitive Routing Engine, based on Random NNs with RL to find optimal overlay paths.	C	RL (Random NN)	Next-hop	Delay
[14]	Link costs calculation for path computation with OSPF for distributed routing in SDN	D	RL (Q-learning)	Link weight values	Delay and jitter
[16]	An approach called SDCoR for choosing the proper traditional routing algorithm regarding changes in the Internet of Vehicles (IoV) environment	D	RL (Q-learning)	Routing protocol	Delivery delay, delivery ratio
[17]	A routing protocol for Wireless Sensor Networks (WSN) based on distributed RL that learns the path to the sink node	D	RL (Q-learning)	Next-hop	Node lifetime and energy-efficiency
[62]	QR-SDN: a classical tabular reinforcement learning approach that directly represents the routing paths of individual flows in its state-action space	C	RL (Q-learning)	Per flow path selection	Path delay
[18]	RSIR: An RL-based approach for intelligent routing in SDN based on link-state metrics	C	RL (Q-learning)	Next-hop	Link stretch, delay, loss and throughput
[20]	A learning agent based on DRL that optimizes routing to minimize network delay	C	DRL (DDPG)	Link weight values	Network delay
[21]	A DRL-based mechanism for achieving routing optimization called DROM in SDN	C	DRL (DDPG)	Link weight values	Delay
[22]	NetworkAI: A network architecture using network monitoring technologies and artificial intelligence for generating control policies	C	DRL (DQN)	Not reported	Delay
[23]	A DRL-based agent with convolutional neural networks in the context of KDN designed to enhance the performance of QoS-aware routing	C	DRL (DDPG)	Link weight values	Delay, loss and qualified flows
[24]	Two DQN-based algorithms designed to reduce the network congestion probability using a short transmission path	C	DRL (DQN)	Next-hop	Throughput
[25]	Tide: A time-relevant DRL network control architecture for optimizing routing	C	DRL (DDPG)	Link weight values	Delay
[26]	A DRL-based routing algorithm for managing multiple service requests of crowd distribution in smart city sectors	C	DRL (DQN)	Next-hop	Service access delay, successful access rate, network usage
[27]	DRL-THSA: A Two-hop state-aware routing strategy based on DRL for LEO satellite networks	D	DRL (DQN)	Next-hop	Delay, throughput, drop rate
[28]	A DRL-based scheme to enable intelligent Service Function Chaining (SFC) routing decision-making in dynamic network conditions	C	DRL (PPO)	Link weight values	Path delay and link usage
[29]	DQSP: A DRL-based QoS-aware secure routing approach for SDN-IoT	C	DRL (DDPG)	Next-hop	Packet delivery rate, attacked node use probability
[30]	A deep multi-agent DRL approach for packet routing to avoid network congestion	D	DRL (DQN)	Next-hop	Path delay and link usage
[63]	Routing planning for SDN considering a multi-path scheme with DRL	C	DRL (DDQN)	Next-hop	Loss ratio
DRSIR	A DRL-based solution for intelligent routing in SDN based on path-state metrics	C	DRL (DQN)	Path selection	Link stretch, delay, loss and throughput

C: Centralized and D: Distributed

Knowledge Plane or the Control Plane to compute paths and properly install flows in the tables of the forwarding devices. In contrast, the solutions introduced in [27] and [30] deploy distributed routing strategies so that routing nodes turn themselves into learning entities that makes local routing decisions based on information learned from the environment. These routing solutions can generate signaling overhead in the Data Plane, which can contribute to network congestion.

Type of Action. The Action Spaces used in the papers reviewed are commonly link and hop-based. Specifically, we identified two types of actions for the routing solutions: setting up the link weight values of a topology representing the underlying network [20], [21], [23], [25], [28], [63], and selecting the next-hop node when building a routing path [24], [26], [27], [29], [30]. Both kinds of action can limit the solutions for the use of external routing algorithms such as the Dijkstra's algorithm. Furthermore, the performance of an end-to-end path performance can only be satisfied when all hops are appropriately chosen. The performance can drop significantly when any hop is wrongly selected by a link/hop-based routing scheme. Unlike what is found in other papers, the DRSIR algorithm optimizes routing decisions by employing path-state metrics as features in the learning process, meaning that the DRSIR agent directly uses knowledge of the paths instead of extracting information from the link-state to choose the next-hop node or update link weight values.

Multi-path routing. The work in [62] introduced a multi-path routing solution based on Q-learning for SDN, where the State and Action Spaces of the RL agent are defined to learn a per-flow path selection based on network latency. However, the practicability of this solution is limited since the Q-table size grows exponentially with the number of network nodes, leading to slow learning in large environments. DRL, which accelerates learning when the State and Action Spaces are large, can address this limitation. The work in [63] presented a solution that employed DQN for multi-path routing in SDN. This approach does not consider network state metrics, thus hindering the traffic-aware paths computation. The DQN agent of the DRSIR algorithm employs path-state metrics to learn to select a unique path for routing traffic between each pair of nodes. In order to introduce multi-path routing in DRSIR, the Action Space must be modified for every action to allow a set of k candidate paths for each pair of nodes.

Complexity Analysis. The work in [22], [25] reports a worst-case complexity of $O(N^2)$ and $O(N^3)$, respectively, where N is the number of nodes in the network. The work in [24], [30] introduces the complexity as a function of the number of floating-point calculations carried out by the NN

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidade Estadual de Campinas. Downloaded on March 16,2022 at 22:17:31 UTC from IEEE Xplore. Restrictions apply.

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/TNSM.2021.3132491, IEEE Transactions on Network and Service Management

used in the proposed DRL solutions, but this calculation considers only the case studies in the papers and cannot be generalized. The work in [20], [21], [23], [28] considers states as defined by the number nodes in the network, traffic matrices or statistics vectors, with the actions defined by the link weight values; the complexity is thus dominated by the number of source-destination nodes, which in the worst-case can be $O(N^2)$. On the other hand, the work in [26], [27], [63] defines as many states as is the number of nodes, bounded by $O(N)$, and actions are defined as the next-hop node, which in the worst-case of a fully-connected network is $O(N^2)$. Therefore, the total worst-case complexity is $O(N^3)$. In DRSIR, the DQN agent learns from a reduced set of experiences by generalizing from the past decisions. The greatest advantage of employing Online and Target NNs in the DQN agent is that this makes possible the avoidance of visiting all the state-action pairs to converge to an optimal policy. The computational complexity for training NNs is bounded by their architecture and the input samples size (*i.e.,* states). Since the architecture of the Online and Target NNs is fixed (*i.e.,* the number of hidden layers, the input size, and the output size), the maximum number of iterations for learning is $O(1)$. Therefore the complexity of training the NN depends only on the total number of states, and will be $O(N^2)$. Thus, the worst-case complexity of Algorithm 1 is $O(N^2)$. We refer the reader to [65] for the derivation of the complexity of training NNs.

VII. Conclusions

In this paper, we have introduced DRSIR, an approach that employs DRL for routing in SDN. The DRSIR uses Deep Q-learning with Experience Replay Memory, Target NN, and Online NN. Deep Q-learning offers a model-free algorithm that makes DRSIR find the appropriate routing policy without making any assumptions about the dynamics of the environment. DRSIR learns while transitioning over states and actions without prior knowledge about the underlying network. The DRL-based routing algorithm of the DRSIR produces routes for every pair of source-destination nodes in a network. It prioritizes paths with large available bandwidth and low delay and loss as the best routing option based on path-state metrics. The results showed that most of the time, the DRSIR indicates shorter and less congested paths than those indicated by RSIR and the four versions of the Dijkstra's algorithm. As a result, the mean delay and loss produced by DRSIR are lower than those given by the RSIR algorithm and the variations of the Dijkstra's algorithm. It is noteworthy that the DRSIR does not require previously labeled data representing the system under consideration for training the DRL-based agent. The acquisition of labeled training datasets would lead to high computational complexity and make the routing solutions dependent on traditional routing protocols to build such datasets. The results presented in this paper encourage the use of DRSIR as a feasible and practical strategy to carry out routing in SDN.

For future work, we intend to extend DRSIR to support multi-path routing, as well as exploring multi-level DRL schemes to deal with centralized control challenges such as scalability, including large-scale networks with distinct topologies. We also plan to include flow type information into the Action and State Spaces to enable QoS-aware path computation as well as improve the set up of the learning parameters to make the agent self-configurable. Moreover, the evaluation of DRSIR using different weights in the reward function for distinct types of traffic and under less bursty traffic than that employed in our evaluation would also be an interesting study. Another interesting future work would be the comparison of routing decisions based on machine learning with those given by traditional optimal routing based on robust optimization.

Acknowledgment

The authors would like to thank CAPES, and the Sao Paulo Research Foundation (FAPESP) for grants #19/03268-0 and 2015/24494-8.

References

[1] A. Rego, S. Sendra, J. M. Jimenez, and J. Lloret, "Ospf routing protocol performance in software defined networks," in *IEEE SDS*, 2017, pp. 131–136.

[2] D. Gopi, S. Cheng, and R. Huck, "Comparative analysis of sdn and conventional networks using routing protocols," in *IEEE CITS*, 2017, pp. 108–112.

[3] J. Park, J. Hwang, and K. Yeom, "Nsaf: An approach for ensuring application-aware routing based on network qos of applications in sdn," *Mobile Information Systems*, 2019.

[4] A. Shirmarz and A. Ghaffari, "An adaptive greedy flow routing algorithm for performance improvement in software-defined network," *International Journal of Numerical Modelling: Electronic Networks, Devices and Fields*, vol. 33, no. 1, 2020.

[5] Y.-C. Wang and S.-Y. You, "An efficient route management framework for load balance and overhead reduction in sdn-based data center networks," *IEEE TNSM*, vol. 15, no. 4, pp. 1422–1434, 2018.

[6] C.-C. Chuang, Y.-J. Yu, and A.-C. Pang, "Flow-aware routing and forwarding for sdn scalability in wireless data centers," *IEEE TNSM*, vol. 15, no. 4, pp. 1676–1691, 2018.

[7] N. Kato, Z. M. Fadlullah, B. Mao, F. Tang, O. Akashi, T. Inoue, and K. Mizutani, "The deep learning vision for heterogeneous network traffic control: Proposal, challenges, and future perspective," *IEEE Wireless Communications*, vol. 24, no. 3, pp. 146–153, 2016.

[8] Z. M. Fadlullah, F. Tang, B. Mao, N. Kato, O. Akashi, T. Inoue, and K. Mizutani, "State-of-the-art deep learning: Evolving machine intelligence toward tomorrow's intelligent network traffic control systems," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2432–2455, 2017.

[9] S. Chaudhary and R. Johari, "Oruml: Optimized routing in wireless networks using machine learning," *IJCS*, 2020.

[10] A. Serhani, N. Naja, and A. Jamali, "Aq-routing: mobility-, stability-aware adaptive routing protocol for data routing in manet–iot systems," *Cluster Computing*, vol. 23, no. 1, pp. 13–27, 2020.

[11] Y. Liu, J. Zhang, W. Li, Q. Wu, and P. Li, "Load balancing oriented predictive routing algorithm for data center networks," *Future Internet*, vol. 13, no. 2, p. 54, 2021.

[12] F. Alhaidari, S. H. Almotiri, M. A. K. Mohammed Al Ghamdi, A. Rehman, S. Abbas, K. M. Khan, and A. ur Rahman, "Intelligent software-defined network for cognitive routing optimization using deep extreme learning machine approach," *Computers, Materials & Continua*, vol. 67, no. 1, pp. 1269–1285, 2021.

[13] S.-C. Lin, I. F. Akyildiz, P. Wang, and M. Luo, "Qos-aware adaptive routing in multi-layer hierarchical software defined networks: a reinforcement learning approach," in *IEEE SCC*, 2016, pp. 25–33.

[14] S. Sendra, A. Rego, J. Lloret, J. M. Jimenez, and O. Romero, "Including artificial intelligence in a routing protocol using software defined networks," in *IEEE ICC Workshops*, 2017, pp. 670–674.

[15] P. Liu, "The random neural network and its learning process in cognitive packet networks," in *IEEE ICNC*, 2013, pp. 95–100.

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidade Estadual de Campinas. Downloaded on March 16,2022 at 22:17:31 UTC from IEEE Xplore. Restrictions apply.

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/TNSM.2021.3132491, IEEE Transactions on Network and Service Management

[16] C. Wang, L. Zhang, Z. Li, and C. Jiang, "Sdcor: Software defined cognitive routing for internet of vehicles," *IEEE Internet of Things Journal*, vol. 5, no. 5, pp. 3513–3520, 2018.

[17] S. Bouzid, Y. Serrestou, K. Raoof, and M. Omri, "Efficient routing protocol for wireless sensor network based on reinforcement learning," in *IEEE ATSIP*, 2020, pp. 1–5.

[18] D. M. Casas-Velasco, O. M. C. Rendon, and N. L. S. da Fonseca, "Intelligent routing based on reinforcement learning for software-defined networking," *IEEE TNSM*, vol. 18, no. 1, pp. 870–881, 2021.

[19] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Barath, "A brief survey of deep reinforcement learning," *IEEE Signal Processing Magazine*, vol. 34, pp. 26–38, 2017.

[20] G. Stampa, M. Arias, D. Sanchez-Charles, V. Muntés-Mulero, and A. Cabellos, "A deep-reinforcement learning approach for software-defined networking routing optimization," *CoRR*, 2017.

[21] C. Yu, J. Lan, Z. Guo, and Y. Hu, "Drom: Optimizing the routing in software-defined networks with deep reinforcement learning," *IEEE Access*, vol. 6, pp. 64 533–64 539, 2018.

[22] H. Yao, T. Mai, X. Xu, P. Zhang, M. Li, and Y. Liu, "Networkai: An intelligent network architecture for self-learning control strategies in software defined networks," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4319–4327, 2018.

[23] T. A. Q. Pham, Y. Hadjadj-Aoul, and A. Outtagarts, "Deep reinforcement learning based qos-aware routing in knowledge-defined networking," in *Springer Qshine*, 2018, pp. 14–26.

[24] R. Ding, Y. Xu, F. Gao, X. Shen, and W. Wu, "Deep reinforcement learning for router selection in network with heavy traffic," *IEEE Access*, vol. 7, pp. 37 109–37 120, 2019.

[25] P. Sun, Y. Hu, J. Lan, L. Tian, and M. Chen, "Tide: Time-relevant deep reinforcement learning for routing optimization," *Future Generation Computer Systems*, vol. 99, pp. 401–409, 2019.

[26] L. Zhao, J. Wang, J. Liu, and N. Kato, "Routing for crowd management in smart cities: A deep reinforcement learning perspective," *IEEE Communications Magazine*, vol. 57, no. 4, pp. 88–93, 2019.

[27] C. Wang, H. Wang, and W. Wang, "A two-hops state-aware routing strategy based on deep reinforcement learning for leo satellite networks," *Electronics*, vol. 8, no. 9, p. 920, 2019.

[28] Z. Ning, N. Wang, and R. Tafazolli, "Deep reinforcement learning for nfv-based service function chaining in multi-service networks : Invited paper," in *IEEE HPSR*, 2020, pp. 1–6.

[29] X. Guo, H. Lin, Z. Li, and M. Peng, "Deep-reinforcement-learning-based qos-aware secure routing for sdn-iot," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 6242–6251, 2019.

[30] R. Ding, Y. Yang, J. Liu, H. Li, and F. Gao, "Packet routing against network congestion: A deep multi-agent reinforcement learning approach," in *IEEE ICNC*, 2020, pp. 932–937.

[31] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim, "Applications of deep reinforcement learning in communications and networking: A survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3133–3174, 2019.

[32] Z. Mammeri, "Reinforcement learning based routing in networks: Review and classification of approaches," *IEEE Access*, vol. 7, pp. 55 916–55 950, 2019.

[33] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *Foundations and Trends in Machine Learning*, vol. 11, no. 3-4, pp. 219–354, 2018.

[34] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction-second edition*. Cambridge, Massachusetts, 2018, vol. 1.

[35] I. J. Okonkwo and I. D. Emmanuel, "Comparative study of eigrp and ospf protocols based on network convergence," *IJACSA*, vol. 11, no. 6, 2020.

[36] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[37] N. C. Luong *et al.*, "Applications of deep reinforcement learning in communications and networking: A survey," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 4, pp. 3133–3174, 2019.

[38] L. Liao and V. C. Leung, "Lldp based link latency monitoring in software defined networks," in *IEEE CNSM*, 2016, pp. 330–335.

[39] Z. Shu, J. Wan, J. Lin, S. Wang, D. Li, S. Rho, and C. Yang, "Traffic engineering in software-defined networking: Measurement and management," *IEEE access*, vol. 4, pp. 3246–3256, 2016.

[40] Ryu Project Team, "Ryu application API," [Accessed: March 16, 2021]. [Online]. Available: https://ryu.readthedocs.io/en/latest/ryu_app_api.html

[41] L. Al Shalabi and Z. Shaaban, "Normalization as a preprocessing engine for data mining and the approach of preference matrix," in *IEEE DepCoS*, 2006, pp. 207–214.

[42] Z. Yang, Y. Xie, and Z. Wang, "A theoretical analysis of deep q-learning," in *PMLR Learning for Dynamics and Control*, 2020, pp. 486–489.

[43] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart, "An architectural approach to autonomic computing," in *IEEE ICAC*, 2004, pp. 2–9.

[44] P. T. Kirstein, "European international academic networking: A 20 year perspective." in *TERENA Networking Conference*, 2004.

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

[46] R. L. S. De Oliveira, C. M. Schweitzer, A. A. Shinoda, and L. R. Prete, "Using mininet for emulation and prototyping software-defined networks," in *IEEE COLCOM*, 2014, pp. 1–6.

[47] A. L. Stancu, S. Halunga, A. Vulpe, G. Suciu, O. Fratu, and E. C. Popovici, "A comparison between several software defined networking controllers," in *IEEE TELSIKS*, 2015, pp. 223–226.

[48] ONF, "Openflow switch specification v1.5.0," Open Network Foundation, Technical Specification TS-020, December 2014. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf

[49] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *USENIX OSDI*, 2016, pp. 265–283.

[50] T. E. Oliphant, *A guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[51] W. McKinney, "Data structures for statistical computing in python," in *SciPy*, S. van der Walt and J. Millman, Eds., 2010, pp. 51–56.

[52] D. M. Casas-Velasco, O. M. Caicedo, and N. L. S. Da Fonseca, "Routing based on (deep) reinforcement learning for software-defined networking," in *https://github.com/danielaCasasv/DRSIR_DRL_routing_approach_for_SDN*, 2021.

[53] C. Werle, S. Mies, and M. Zitterbart, "On benchmarking routing protocols," in *IEEE ICON*, 2011, pp. 305–310.

[54] S. Uhlig, B. Quoitin, J. Lepropre, and S. Balon, "Providing public intradomain traffic matrices to the research community," *SIGCOMM Computer Communication Review*, vol. 36, no. 1, pp. 83–86, 2006.

[55] V. Heorhiadi, "Tmgen traffic matrix generator," https://github.com/progwriter/TMgen, 2018.

[56] M. Roughan, "Simplifying the synthesis of internet traffic matrices," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 5, p. 93–96, 2005.

[57] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR*. Ithaca, 2015.

[58] E. Okewu, P. Adewole, and O. Sennaike, "Experimental comparison of stochastic optimizers in deep learning," in *Springer ICCSA*, 2019, pp. 704–715.

[59] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *PMLR AISTATS*, vol. 9, 2010, pp. 249–256.

[60] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1, no. 2.

[61] K. Hornik, M. Stinchcombe, H. White *et al.*, "Multilayer feedforward networks are universal approximators." *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[62] J. Rischke, P. Sossalla, H. Salah, F. H. Fitzek, and M. Reisslein, "Qr-sdn: towards reinforcement learning states, actions, and rewards for direct flow routing in software-defined networks," *IEEE Access*, vol. 8, pp. 174 773–174 791, 2020.

[63] Z. Wang, Z. Lu, and C. Li, "Research on deep reinforcement learning multi-path routing planning in SDN," *Journal of Physics: Conference Series*, vol. 1617, p. 012043, 2020.

[64] A. Kanervisto, C. Scheller, and V. Hautamäki, "Action space shaping in deep reinforcement learning," in *IEEE CoG*, 2020, pp. 479–486.

[65] D. Bienstock, G. Muñoz, and S. Pokutta, "Principled deep neural network training through linear programming," in *ICLR*, 2019.

Daniela M. Casas-Velasco (GS'13) is pursuing her Ph.D degree in Computer Science at the State University of Campinas, Campinas, Brazil.

Oscar M. Caicedo (GS'11–M'15–SM'20) is a Full Professor at the Departamento de Telematica, Universidad del Cauca.

Nelson L. S. da Fonseca (M'88–SM'01) is a Full Professor with the Institute of Computing, State University of Campinas, Brazil.

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Universidade Estadual de Campinas. Downloaded on March 16,2022 at 22:17:31 UTC from IEEE Xplore. Restrictions apply.