Received September 3, 2020, accepted September 17, 2020, date of publication September 21, 2020,
date of current version October 5, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3025432

QR-SDN: Towards Reinforcement Learning States, Actions, and Rewards for Direct Flow Routing in Software-Defined Networks

JUSTUS RISCHKE[1], PETER SOSSALLA[1], HANI SALAH[1], FRANK H. P. FITZEK[1,3], (Senior Member, IEEE), AND MARTIN REISSLEIN[2,3], (Fellow, IEEE)
[1]Deutsche Telekom Chair, 5G Lab Germany, Technische Universität Dresden, 01062 Dresden, Germany
[2]School of Electrical, Computer, and Energy Engineering, Arizona State University, Tempe, AZ 85287, USA
[3]Centre for Tactile Internet with Human-in-the-Loop (CeTI), Technische Universität Dresden, 01062 Dresden, Germany

Corresponding author: Martin Reisslein (reisslein@asu.edu)

This work was Funded by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as part of Germany's Excellence Strategy–EXC 2050/1–Project ID 390696704–Cluster of Excellence ''Centre for Tactile Internet with Human-in-the-Loop'' (CeTI) of Technische Universität Dresden.

ABSTRACT Flow routing can achieve fine-grained network performance optimizations by routing distinct packet traffic flows over different network paths. While the centralized control of Software-Defined Networking (SDN) provides a control framework for implementing centralized network optimizations, e.g., optimized flow routing, the implementation of flow routing that is adaptive to varying traffic loads requires complex models. The goal of this study is to pursue a model-free approach that is based on reinforcement learning. We design and evaluate QR-SDN, a classical tabular reinforcement learning approach that directly represents the routing paths of individual flows in its state-action space. Due to the direct representation of flow routes in the QR-SDN state-action space, QR-SDN is the first reinforcement learning SDN routing approach to enable multiple routing paths between a given source (ingress) switch–destination (egress) switch pair while preserving the flow integrity. That is, in QR-SDN, packets of a given flow take the same routing path, while different flows with the same source-destination switch pair may take different routes (in contrast, the recent DRL-TE approach splits a given flow on a per-packet basis incurring high complexity and out-of-order packets). We implemented QR-SDN in a Software-Defined Network (SDN) emulation testbed. Our evaluations demonstrate that the flow-preserving multi-path routing of QR-SDN achieves substantially lower flow latencies than prior routing approaches that determine only a single source-destination route. A limitation of QR-SDN is that the state-action space grows exponentially with the number of network nodes. Addressing the scalability of direct flow routing, e.g., through routing only high-rate flows, is an important direction for future research. The QR-SDN code is made publicly available to support this future research.

INDEX TERMS Flow routing, Q-table, software-defined networking (SDN), state-space design.

I. INTRODUCTION

A. MOTIVATION: HOW TO OPTIMIZE FLOW ROUTING IN A SOFTWARE DEFINED NETWORK?

Flow routing is a fundamental problem in packet-switched communication networks. In flow routing, the packets of a given flow, e.g., a flow from a given source host to a given destination host or a Transmission Control Protocol (TCP) flow, should follow the same routing path through the network of packet-switching nodes (switches). However, distinct flows, even between the same source-destination switch pair (whereby the source switch may be the ingress switch of a routing domain and the destination switch may be the egress switch of the routing domain for the considered flows), may follow different routing paths so as to optimize the network performance [1]. Flow routing is thus fundamentally different from the classical path routing based only on the destination host (as considered in the Internet Protocol (IP)) or on the source-destination switch pair. Flow routing enables a wide range of adaptations to optimize the loads on network links,

The associate editor coordinating the review of this manuscript and approving it for publication was Peng-Yong Kong.

i.e., to conduct traffic engineering, so as to optimize various network performance metrics [1].

Software Defined Networking (SDN) enables centralized decision making by a controller that can adaptively configure the packet-switching nodes. The centralized controller has global knowledge of the network status, i.e., all the switching nodes and their interconnecting links, as well as network monitoring metrics. SDN appears therefore very well suited to enable the optimization of flow routing [2]–[4].

However, flow routing algorithms are relatively complex [1], and while significant progress has been made in recent decades, flow routing remains a complicated routing approach that requires detailed models of the communication network and traffic [5], [6]. In recent years, model-free artificial intelligence techniques based on reinforcement learning have been successfully applied to a wide range of complex control and optimization problems. In particular, classical tabular reinforcement learning, also referred to as tabular Q-Learning, stores the state-action representations in a table; however, this table can become very large for large state-action spaces [7]. Deep reinforcement learning [8]–[10] approximates the state-action table with a deep neural network to overcome the scalability problem of large tables. A plethora of recent studies has applied the reinforcement and deep reinforcement learning approach to complex problems in the area of communication networks [11]–[15].

A critical aspect of applying reinforcement learning to flow routing is to represent the specific features and characteristics of the flow routing problem in communication networks in the states, actions, and rewards for reinforcement learning. As further elaborated in Section II, this representation of flow routing has *not* been thoroughly researched to date. The existing studies have typically represented the communication network routing problem in a simplistic or indirect manner. For instance, routing studies have considered link weights, which are an indirect way to influence routing [16]; however, the existing studies have not directly (explicitly) specified routing paths as outcomes of the reinforcement learning. Thus, there is a pronounced lack of fundamental understanding of how to best represent the intricacies of flow routing for the application of reinforcement learning strategies. For instance, how should states, actions, and rewards be designed for communication networking flow routing problems? What exploration strategies for potential solutions to flow routing problems should be employed within the state-action space? How should state-action space changes, e.g., due to communication load or flow changes, be handled?

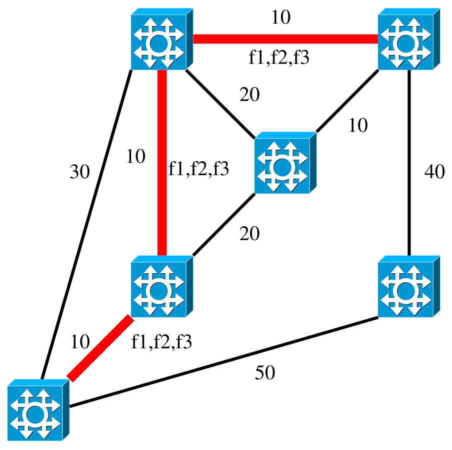
B. CONTRIBUTION: DIRECT FLOW ROUTING REPRESENTATION FOR FLOW-PRESERVING MULTI-PATH ROUTING

In this article, we address the representation of the intricacies of unicast flow routing in a Software-Defined Network (SDN). With the direct flow routing representation, the reinforcement learning produces actual routing paths. After a brief review of the background on SDNs and reinforcement learning as well as the related work on applying machine learning to networking problems, including routing, in Section II, we examine the representation of the SDN flow routing problem for reinforcement learning in Section III. More specifically, Section III examines the design of the state-action space to directly represent the SDN flow routing problem. Section III also presents the reward design for the SDN flow routing problem. Motivated by ultra low latency communication and control applications as well as the tactile internet [17], [18], we focus on flows that require short latencies and consider the flow latency as the reward.

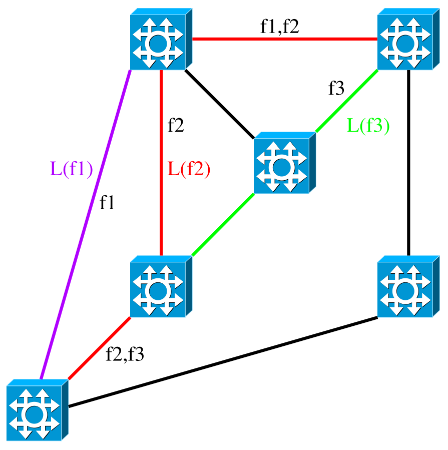
Our direct flow routing representation in the QR-SDN state-action space enables flow-preserving multi-path routing from a given source (ingress) switch to a given destination (egress) switch, as illustrated in Fig. 1b. Prior studies with indirect routing representations, e.g., through link weights, have typically only considered single-path routing [19], [20], see Fig. 1a. However, single routing paths tend to become congested for skewed traffic matrices, e.g., when a few source-destination flows dominate the current network traffic. Multi-path routing can effectively mitigate congestion and thus reduce latencies by utilizing several routing paths between a given source-destination pair. However, the indirect routing representation, e.g., through link weights, implies random flow splitting, typically on a per-packet basis, for utilizing multiple routing paths [21]. In contrast, through the direct flow routing representation, QR-SDN preserves flows by explicitly routing specific individual flows over specific individual routes.

A limitation of the direct flow routing approach is that the state-action space scales in the number of ongoing flows, creating a potential scalability problem. We note that the scalability problem arises primarily due to the storage space required for the Q-table and due to the learning time till convergence that is required to explore this large state-action space. On the other hand, the computational effort is low (only requiring an update of Q-values in every step, e.g., once every few seconds). We believe that this scalability problem can be addressed to some degree through employing multiple SDN controllers so that each controller has fewer nodes and flows to manage and accordingly a smaller Q-table to manage and a smaller state-action space to explore. Each SDN controller would optimize the flow routing within a reasonably small network domain. The SDN controllers would then exchange routes, similar to the current approach in Internet routing, where each autonomous system (AS) optimizes the routing inside its AS domain, and the ASs exchange routes via the Border Gateway Protocol (BGP).

Also, judicious path exploration strategies may help to address the scalability problem. For instance one potential path exploration strategy could limit explorations to highly utilized links (which are more likely to become congested). Also, the path exploration could be limited to high-rate flows; the vast majority of low-rate flows could be routed with conventional shortest-path routing (or some other routing approach that helps to relieve the highly utilized links, e.g.,



(a) Traditional link weight based single-path routing.



(b) Proposed approach: QR-SDN: Flow-preserving multi-path routing.

FIGURE 1. Illustration of the proposed QR-SDN routing in contrast to traditional link weight based routing. With traditional link weight based routing, all three flows $f1, f2, f3$ from the bottom left source switch to the top right destination switch follow the same routing path. (To avoid clutter, we omitted the source and destination hosts, which are directly attached to the bottom left and top right switches, respectively.) QR-SDN either uses only one communication path, such as the shortest-path, under low load or distributes the flows to achieve a lower average latency (average of latencies $L(f1), L(f2), L(f3)$) during traffic peaks. The distribution of flows over the routing paths is determined by the reinforcement learning agent.

routing over the longest paths). Furthermore, in operational networks, not all flows may require short latencies, some flows may only require reliable communication while tolerating long delays. These delay-tolerant flows could be routed with conventional routing, thus reserving the path exploration strategy to flows requiring short latencies. Moreover, improved learning algorithms [22] may help address the scalability problem in future research. The present study does not examine such flow scalability techniques in detail; rather, this present study seeks to lay the groundwork for effective low-latency flow-preserving multi-path SDN routing through reinforcement learning.

Section III-E details our implementation of the SDN controller that carries out the flow routing. We make the source code of QR-SDN publicly available [23] so as to spur future research on further improving the representation of the communication network characteristics in reinforcement learning states, actions, and rewards as well as the examination of different learning modules, e.g., deep reinforcement learning.

Our performance results in Section IV comprehensively evaluate the direct flow routing approach in QR-SDN. We find that the flow-preserving multi-path QR-SDN routing achieves substantially shorter flow latencies than the single-path shortest path routing. Our evaluation results also elucidate tradeoffs in state-action space representations of the SDN routing problem. The results indicate, for instance, that an action design that directly re-routes the entire ensemble of ongoing flows has worse scalability, but achieves shorter average flow latency than an action design that re-routes one flow at a time. To the best of our knowledge, the QR-SDN evaluation is the first evaluation of a reinforcement learning flow routing approach for dynamic scenarios, such as load changes and flows joining or leaving the network. We examine the tradeoffs in flows joining and leaving, which change the flow routing based state space. We acknowledge that while the flow-preserving multi-path routing of QR-SDN achieves good performance in small networks, the state space scales as the number of network nodes to the power of the number of flows. The presented QR-SDN is therefore not directly applicable for large networks with many flows. However, we believe that QR-SDN can serve as an impetus for re-examining the representation of the flow routing problem for reinforcement learning and spur future research that addresses the open scalability problem.

We also acknowledge that this study focuses on reinforcement learning and does not consider deep reinforcement learning. This study approach of focusing first on the representation of the flow routing problem is based on a number of disadvantages of deep reinforcement learning based approaches that have not been sufficiently examined in the communication network routing context. The so-called "sample inefficiency" of deep reinforcement learning [9], [10] (in contrast to the classical tabular Q-Learning [7]) leads to very long training times for the deep neural network, resulting in potentially excessive solution computation times. For instance, the recent TIDE study [19] considered around one hundred training episodes, each with one thousand time steps, whereby each time step is on the order of one second, resulting in training times on the order of days. Novel computation strategies can reduce the time complexity with appropriate initialization [24], [25], however timely deep reinforcement learning continues to be a challenge. Also, the function approximation through the deep neural network in deep reinforcement learning may lower the achievable reward or return and lead to suboptimal configurations [26, Fig. 1 and Sec. 4.2]. Moreover, for direct representations of the flow routing problem, the deep learning network would become very large as it would have to accommodate very high numbers of possible outputs (routing paths), leading to new scalability problems. Therefore, we believe that the usage of deep reinforcement learning as the learning module in machine learning based control of communication network flow routing should be examined in future research that can build on the flow routing representation studied in this article. Various enhancements of deep reinforcement learning [27] and alternate approaches, e.g., episodic control [28], [29] should be considered in this future research.

II. BACKGROUND AND RELATED WORK

A. SOFTWARE-DEFINED NETWORKING (SDN)

SDN refers to the ability of software applications to dynamically program individual network devices, and thus control the behavior of the network as a whole. The goal of SDN is to program the network via open interfaces to dynamically initialize, control, change, and manage the network behavior. In SDN, software plays a central role in the operation of networks by introducing an abstraction for the data plane and by separating the data plane from the control plane. To achieve such a separation, three types of components are needed: (*i*) a (logically) centralized SDN controller, (*ii*) SDN-capable switches, and (*iii*) a management protocol, such as OpenFlow [30]. An advantage of SDN is the capability of centrally updating routes of existing traffic on demand by the controller [31]–[38] in a consistent manner [16].

B. REINFORCEMENT LEARNING

In reinforcement learning, a decision-making *agent* observes an *environment* and decides according to its *state* which *action* has to be performed. A classical approach to solve decision-making problems is to represent problems as a Markov process. A Markov decision process (MDP) consists of (*i*) a set of states $\mathcal{S}$, (*ii*) a set of actions $\mathcal{A}$, which can be state dependent, and (*iii*) transition probabilities $p(s', r|s, a) \doteq \Pr(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a)$, which define the probability to reach a succeeding state $S_{t+1} = s'$ from a previous state $S_t = s$ by performing action $A_t = a$, and receiving a reward $R_{t+1} = r$, as elaborated in Section III.

When detailed models of the underlying system are lacking or intractable, model-free learning approaches [39] can be employed. Knowledge about unknown probability functions $p(s', r|s, a)$ can be acquired by sampling over the environment and by using experiences to learn optimal action-value functions $q(s, a)$. This $q(s, a)$ function can then be used to determine the best action for a particular state s. The approximation of such an action-value function $q(s, a)$ is achieved by storing the *Q-values* $Q(S_t, A_t)$ of state-action pairs in arrays or tables and is therefore also referred to as *tabular* learning.

The actual learning of optimal policies or value-functions from raw experiences is also called the *Monte Carlo Method*. The disadvantage of the *Monte Carlo Method* is that always entire episodes of learning need to be finished, which is unsuitable for routing since we want to adapt flow routes at runtime. *Bootstrapping* can update the estimates without waiting for the final outcome of an episode. This bootstrapping is applied by *Temporal-Difference Learning*, whereby *Q-Learning* [7] is one of the most common temporal-difference learning algorithms.

C. REVIEW OF RELATED WORK

The emergence of tabular reinforcement learning [7], [39], also referred to as tabular Q-Learning, in the early 1990s spurred several studies that attempted to exploit Q-Learning for distributed routing in classical Internet Protocol (IP) networks, see e.g., [40]–[44], as well as for optical burst routing [45]. Tabular Q-Learning has recently also been employed for optimizing the video quality in multimedia networking [46], mobile ad hoc network routing [47], and for load scheduling in the energy Internet [48]. While Q-Learning was generally found to achieve good routing performance, the distributed nature of classical IP network operation makes implementation challenging. In contrast, SDN provides a centralized controller for tracking the network status, running the learning agent, and instructing the switches to implement the routes. We believe that it is therefore important to re-examine Q-Learning in the context of centralized SDN operation and control.

With the emergence of deep reinforcement learning [9], [10], several recent articles have suggested to apply deep reinforcement learning to the centralized SDN routing problem [49]–[55]. Generally, existing routing approaches have not examined the representation of the SDN routing problem for deep reinforcement learning in detail. Typically, the neural networks in the existing approaches produce floating point numbers as outputs, instead of flow or path routes. These floating point numbers are interpreted as a characteristic for a link, e.g., as link weights [19]–[21], [56], [57] or a communication session, e.g., as probabilistic split ratios for communication flows [21]. Thus, the existing deep reinforcement learning based routing mechanisms control the routing *indirectly* through controlling link or session characteristics. Then, the link and session characteristics are fed into a standard routing algorithm, e.g., a variation of a shortest-path routing algorithm, to find a single routing path. In contrast, our proposed QR-SDN approach represents the actual flow routes for the Q-Learning reasoning. Thus, our QR-SDN *directly* controls the routing, i.e., the reinforcement learning produces a specific routing path to be deployed for a given flow. This direct flow routing avoids the splitting of any individual source-destination host flow and can exploit multiple routing paths for flows sharing the same source-destination switch pair. To the best of our knowledge, our study is the first to directly represent flow routing in the state-action space of reinforcement learning.

For completeness we note that deep reinforcement learning has been applied to a wide variety of other communication network problems, including distributed routing [58], [59], congestion control [60], data center networks [61], wireless network routing [62]–[71], vehicular ad hoc network routing [72], [73], optical networking [74]–[76], caching [77], and mobile edge computing [78], [79]. We also note that a preprocessing approach for efficiently representing virtual network embeddings for subsequent algorithm processing has been examined in [80].

III. QR-SDN: REPRESENTATION DESIGN OF SDN ROUTING

The representation of the SDN flow routing problem for efficient decision making by a reinforcement learning agent has not been examined in detail in the existing literature. Especially the design of the states and actions so as to effectively represent the flow routing problem for processing by a reinforcement learning agent has to be investigated.

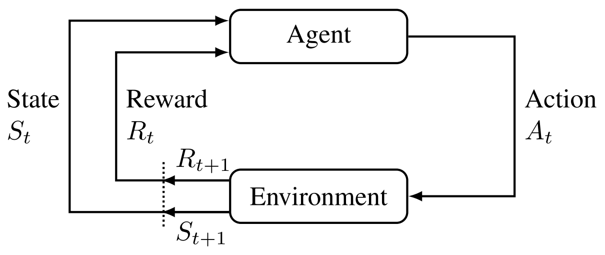


FIGURE 2. Interaction between the agent and environment after Sutton [39].

A. STATE-SPACE DESIGN

Consider the network $G(\mathcal{V}, \mathcal{E})$ with $\mathcal{E}$ as a set of edges connecting the set of vertices $\mathcal{V}$. We focus on unicast communication flows, i.e., flows that transmit data from a given sender to a single receiver. The data transmission for a given application or transport layer context, e.g., a given Transport Control Protocol (TCP) flow, from a given sender (source host) s_f to a given receiver (destination host) d_f is referred to as flow f. We denote $\mathcal{F}$ for the set of all flows. We assume that a flow f transmits a prescribed traffic rate R^f out of the source host s_f into the network. A path P_{s_f,d_f} is a sequence of vertices $P = (v_1, \ldots, v_n)$ from the set of all possible paths $P \in \mathcal{P}_{s_f,d_f} = \{P_{s_f,d_f,1}, P_{s_f,d_f,2}, \ldots\}$ connecting source host s_f to destination host d_f, whereby the set $\mathcal{P}_{s_f,d_f}$ may be determined by a graph search algorithm, such as Depth-First Search (DFS) [34], [81].

The reinforcement learning agent, which may operate at the SDN controller, observes the environment, (i.e., the network) by measuring the desired key performance indicators, such as latency or bandwidth, at discrete time steps $t = 0, 1, 2, \ldots$. The observation consists of the environment's *state* S_t from the set of states $\mathcal{S} = \{S^1, S^2, \ldots\}$ and a *reward* $R_t \in \mathcal{R} \subset \mathbb{R}$. We define the state S_t to consist of a table, which contains the currently selected path P for each flow f:

$$S_t = \begin{cases} f_{s_1,d_1} : & P_{s_1,d_1,t} \\ \vdots & \\ f_{s_i,d_i} : & P_{s_i,d_i,t}. \end{cases} \tag{1}$$

The state space is essentially a key-value dictionary with the flows as keys and the current path as value for each key. We note that this dictionary is just one possible implementation of the state space. The states could also be represented as a list, which could be directly mapped to the inputs of a neural network used for deep Q-Learning. We preferred the dictionary with the flows as keys and the paths as values, as this is a straightforward representation of the state space from a programming implementation point of view.

B. ACTION-SPACE DESIGN

Depending on the state S_t and its corresponding reward R_t, an *action* $A_t \in \mathcal{A}$ is selected (whereby the set of possible actions $\mathcal{A}$ may generally depend on the state S_t). The set of actions $\mathcal{A} = \{A_{t,1}, A_{t,2}, \ldots\}$ is determined by the set of possible paths, including the current path, i.e., $\mathcal{A} = \mathcal{P}_{s_f,d_f}$ for flow f. One of these possible paths is then selected to either replace or keep the current path. Thus, an action essentially changes the value, i.e., the current path, of a key, i.e., a flow, in the key-value dictionary that represents the state space. The procedure for re-routing is described in Section III-E1.

An action A_t applied to a single flow can be described by

$$A_t = \{f_{s_1,d_1} : P_{s_1,d_1,t} \Rightarrow P_{s_1,d_1,t+1}\}. \tag{2}$$

Now the question remains whether only one flow or several flows should be changed with one action. We can change one flow at a time step, i.e., conduct a *OneFlow Change* as specified in Eq. (2). Alternatively, we can change all flows at a time step, i.e., take the action

$$A_t = \begin{cases} \{f_{s_1,d_1} : P_{s_1,d_1,t} \Rightarrow P_{s_1,d_1,t+1}\}, \\ \vdots \\ \{f_{s_i,d_i} : P_{s_i,d_i,t} \Rightarrow P_{s_i,d_i,t+1}\}, \end{cases} \tag{3}$$

which we refer to as *Direct Change*. Clearly, as Eqs. (2) and (3) indicate, the design of action A_t has an impact on the size of the action space $|\mathcal{A}|$. A disadvantage of the *Direct Change* approach is that the action space scales with the product of the numbers of possible paths for the flows; in contrast, the action space of *OneFlow Change* scales with the sum of the numbers of possible paths for the flows, as analyzed in more detail in Section IV-C4. On the other hand, *Direct Change* allows direct switching in a single time step to a desired state (routing configuration) in order to achieve a higher flow routing performance. We will quantitatively evaluate this tradeoff in Section IV with measurements in an emulated SDN.

In principle, MDP state changes are non-deterministic. However, our environment is an SDN, i.e., we know how the routing paths will change after we select new routing paths by performing an action A_t. Thus, our states transition deterministically (and the new state S_{t+1} does not need to be observed). Therefore, only the reward R_{t+1} achieved with the action A_t needs to be observed.

We briefly contrast our state-action space design, which directly represents the flow routes from the state-action space designs in related studies on SDN routing. The deep reinforcement learning for traffic engineering (DRL-TE) study [21] adopts a vector of the throughput-latency tuples of all ongoing flows as state space, while the DROM study [20] considers the flow source-destination traffic matrix as state space, and the TIDE study [19] considers a time series of network status matrices, which contain the link utilization levels. The prior studies use indirect specifications of the routing actions. Specifically, the actions-spaces in the DROM and TIDE studies are link weight settings, while the action space in the DRL-TE study is a set of probabilistic split ratios for each ongoing flow. The probabilistic split ratio specifies the probability with which a given packet should be sent on a particular path. We note that while the probabilistic split action in the DRL-TE study [21] contributed an important initial understanding of SDN routing with deep reinforcement learning, the probabilistic flow splitting approach is not practical for high-speed networks. In particular, the probabilistic flow splitting incurs prohibitive computational effort for generating the random numbers for splitting the flow as each packet requires an independently generated random number. Moreover, the flow splitting generates an excessive amount of out-of-order packets, which the receiver has to buffer and process. In order to avoid these complications and to design a practical state-action space for SDN routing, we directly and consistently consider flow routes in both our state space and our action space. Thus, with our state-action space design, the routing actions relate directly to the state and can be directly implemented in an SDN.

C. REWARD DESIGN

We use the reward R_{t+1} to measure how well an action A_t solved the flow routing problem. Motivated by the recent interest in low-latency networking [17], [18], our evaluations in this study consider the latency for the reward. Also, congestion generally increases the latency, but not the consumed transmission bitrate. A consideration of the throughput for the reward would require knowledge of the required transmission bitrate per application. Without knowledge of the requirements of the sending hosts, it would be unclear whether a throughput change was due to a bad routing decision or just because the sending host had lowered the transmission bitrate. If multiple performance metrics should be considered, then a weighted formula as proposed in [50] can be used.

Our proposed reward R_t consists of the sum of latencies L_f along the current paths P_{s_f,d_f} of the flows $f \in \mathcal{F}$. In order to weigh outliers relatively heavily we employ the root mean square

$$R_t = -\sqrt{\frac{\sum_{\forall f \in \mathcal{F}} L_f^2}{|\mathcal{F}|}}. \tag{4}$$

Note that the minus sign is required since the reinforcement learning agent strives to maximize its reward; however, a higher latency is less desirable.

D. REINFORCEMENT LEARNING AGENT

1) Q-LEARNING

This section briefly reviews the principles of *Q-Learning* [7] and describes our Q-table structure for QR-SDN. The Q-value $Q(S_t, A_t) \in (-\infty, 0)$ is in principle an expected quality measure of an action A_t that was taken in state S_t at time t. Q-Learning is based on an iterative update rule:

$$\underbrace{Q(S_t, A_t)}_{\text{new Q-value}} \leftarrow (1-\alpha)\underbrace{Q(S_t, A_t)}_{\text{old Q-value}} + \underbrace{\alpha}_{\text{learning rate}} \Big(\underbrace{R_{t+1}}_{\text{observed reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_{a \in \mathcal{A}} Q(S_{t+1}, a)}_{\text{expected value of future state}} \Big), \tag{5}$$

whereby the learning rate α determines how quickly the newly learned Q-values are adopted. The discount rate γ expresses how future expected rewards will be considered. The future expected rewards are represented by $\max_{a \in \mathcal{A}} Q(S_{t+1}, a)$, which basically expresses how much reward we could get if the highest valued action a is taken.

Each state-action pair and its corresponding Q-value need to be saved in a tabular data structure. For the SDN flow routing problem, we implemented the Q-table using nested dictionaries, whereby the different states S are the keys. The values, in turn, are dictionaries with an action, see Eqs. (2) and (3), as key and the actual Q-value as value, i.e.,

$$\text{Q-Table} = \begin{cases} S^1 : \begin{cases} A^1 : Q(S^1, A^1) \\ \vdots \\ A^j : Q(S^1, A^j) \end{cases} \\ \vdots \\ S^i : \begin{cases} A^1 : Q(S^i, A^1) \\ \vdots \\ A^j : Q(S^i, A^j). \end{cases} \end{cases} \tag{6}$$

This Q-table information of the flow routing will be the basis for the exploration strategy in Section III-D2.

An important aspect for practical operation is the initialization of the Q-table. In our current implementation we initialize the table with a Q-value of $-\infty$ and use initially a random routing action. This initial Q-value of $-\infty$ is set because we generally select the action according to the highest Q-value. Moreover, we treat a Q-value of $-\infty$ as 0 in the initial iteration of Eq. (5), i.e., we initially set the old Q-value to zero in Eq. (5) to allow for the newly learned values to take over. An improved initialization could be based on shortest-path routing, which may converge faster to an optimal routing configuration.

2) EXPLORATION STRATEGIES

Unsupervised learning requires an exploration strategy to try new action-state combinations, which have not been experienced before. On the other hand, once knowledge has been gained, the knowledge should be exploited to achieve the maximum reward. We implemented and evaluated three common exploration strategies, namely ϵ-*greedy*, *Softmax*, and *Upper Confidence Bound (UCB)* for the SDN flow routing problem.

In ϵ-*greedy*, the action with the highest Q-value is chosen, i.e.,

$$A_t \doteq \operatorname*{argmax}_{a \in \mathcal{A}} Q_t(s, a), \tag{7}$$

whereby there is a probability of $\epsilon \in [0, 1]$ that a random action is chosen. This helps to explore new states, but also lowers the exploited reward, even after a long learning time.

The *Softmax* strategy converts the Q-values into selection probabilities

$$Pr\{A_t = a\} = \frac{\exp Q(s, a)/\tau}{\sum_{b \in \mathcal{A}} \exp Q(s, b)/\tau} \tag{8}$$

for each action of the states and samples over the results. The *Softmax* function is controlled by the *temperature* τ, whereby a low τ favors exploitation (i.e., the action with the highest Q-value is chosen more often), and a high τ leads to an explorative character to acquire new knowledge.

Since our Q-values are initialized with $-\infty$, we need to modify Eq. (8) to

$$Pr\{A_t = a\} = \frac{\exp\left[-1/(Q(s, a) \cdot \tau)\right]}{\sum_{b \in \mathcal{A}} \exp\left[-1/(Q(s, b) \cdot \tau)\right]} \tag{9}$$

in order to mimic the behavior of *Softmax* for negative value ranges. The Q-values never reach zero, therefore it is safe to place $(Q(s, a) \cdot \tau)$ in the denominator. In Eq. (8), all $\exp Q(s, a)/\tau$ would be zero for $Q(s, a) = -\infty$. Thus, the initial randomly selected state-action combination would always be preferred compared to all others, since their probability is zero. In contrast, in Eq. (9), the probabilities are non-zero for the initialization $Q(s, a) = -\infty$. Another possibility would be to initialize the Q-values with zero. In Eq. (9), this would cause all action-state combinations to be tried, since the probabilities for $Q(s, a) = 0$ are the highest. This forced exploration of all action-state combinations, however, would contradict the idea that exploration is controlled by the temperature τ.

Irrespective of whether Eq. (8) with $Q(s, a) = 0$ or Eq. (9) with $Q(s, a) = -\infty$ is used, the initialization has a strong influence on the convergence of the reinforcement learning agent. The DRL-TE approach [21] uses an initialization based on a traffic engineering (TE) solution for its exploration. As suggested in [21], an initial solution could be shortest-path as baseline. For an optimized initialization, we adapt the TE approach by using the worst possible path in terms of cost as a starting point. We then use the latency of this worst possibly path to calculate a hypothetical Q-value, assuming that the reinforcement learning agent would converge to this worst possible path, as detailed in Algorithm 1.

Algorithm 1 Optimistic Q-Value Initialization Calculation

```
1:  procedure OPTIMISTICQVALUEINITIALIZATION
2:      $\alpha \leftarrow 0.8$
3:      $\gamma \leftarrow 0.8$
4:      $Reward \leftarrow$ LATENCYWORSTPOSSIBLEPATH
5:      ▷ For consistency for all exploration strategies
6:      ▷ we initialize with $-\infty$
7:      $q \leftarrow -\infty$
8:      for iter in RANGE50 do
9:          if q == $-\infty$ then
10:             $q \leftarrow 0$
11:         end if
12:         $q \leftarrow (1 - \alpha) \cdot q + \alpha \cdot (Reward + \gamma \cdot q)$
13:     end for
14:     return $q$
15: end procedure
```

Algorithm 1 illustrates our optimistic Q-value initialization calculation. We set the reward to the latency of the worst path and set the initial Q-value to $-\infty$ (Lines 4–7) Subsequently, we let the Q-value converge by iterating over Eq. (5) (Lines 8–13). We treat the initial old Q-value as 0 (Lines 9–10). We compare the *Softmax* exploration strategy with and without the optimistic Q-value initialization in Section IV-C3.

Generally, it is often desired to decrease exploration over time, which can be achieved by continuously reducing the values of ϵ and τ. This process is called *annealing* and allows to move smoothly from exploration to exploitation. The *Upper Confidence Bound (UCB)* automatically conducts *annealing*, i.e., controls the exploration, by counting the times an action has been chosen. Thereby, actions are chosen based on their Q-value and their potential. More specifically, a bonus b^+ is introduced:

$$A_t \doteq \operatorname*{argmax}_{a \in \mathcal{A}} \Big(Q_t(s, a) + cb^+\Big). \tag{10}$$

The bonus b^+ is decreased over time by counting how often a state has been visited $N(s)$ and the number of times a corresponding action-state combination has been selected $N(s, a)$; in particular, $b^+ = \sqrt{\ln N(s)/N(s, a)}$. The degree of exploration can be adjusted with the parameter $c > 0$, whereby a higher c gives the bonus b^+ more leverage, resulting in a more explorative character.

The respective employed exploration strategy selects the action A_t [based on Eqn. (7), (9), or (10)] from the respective pre-configured OneFlow Change or Direct Change action space. All these exploration strategies have strengths and weaknesses, as measured by the time to a converged state and average latency achieved after converging. We will quantitatively examine these tradeoffs in Section IV-C3.

E. SDN CONTROLLER IMPLEMENTATION

We implemented a centralized SDN controller using the *Ryu* framework [82]. The SDN controller performs three tasks simultaneously: (*i*) latency monitoring, (*ii*) reinforcement learning as described in Section III-D, and (*iii*) the actual path deployment with the *OpenFlow* protocol. Each task is implemented as a single thread to facilitate the parallelization of the processing since the latency monitoring is time critical. The SDN controller sends probing packets, as described in [83], to monitor the experienced flow latencies L_f.

1) REROUTING

A distinct advantage of SDN is the capability to modify the routing paths of existing traffic flows at runtime without incurring any losses or down time. Since we use this flexibility to update our flow routes, we need an algorithm to find the required modifications for existing routes to change to new routes. To the best of our knowledge, there is no existing algorithm for finding these modifications, and therefore we designed our own algorithm (Algorithm 2).

The challenge is to modify the forwarding rules in such a way that existing traffic flows are not interrupted. Our approach is to iterate over a new path, which is a list of switches proposed by the reinforcement learning agent, and compare the new path with the old path (Lines 6–7). If the old and new paths share the same switches in the correct sequence, then nothing is changed (Lines 9–11). On the other hand, if there is a new switch, it will be added to the *flowAddList* and its predecessor switch will be added to the *flowModList* (Lines 18–21). Moreover, if old and new switches do not share the same predecessor switch, then the forwarding rule needs to be modified, and therefore the predecessor switch is added to the *flowModList* (Lines 13–15). To ensure a seamless connection, the forwarding rules are first added to the new switches (Lines 24–28). The forwarding rules of the existing switches are then changed, but starting from the destination to the source switch to avoid disruptions or gaps in the routing path (Lines 29–33).

The modification of forwarding rules relies on the knowledge of the topology and the relations between the ports and the neighboring switches. This knowledge is gathered by the monitoring module and used to determine the outgoing port to reach the succeeding switch on the new path. The manipulation of the actual forwarding rule is implemented with OpenFlow's `FlowMod` command.

Finally, the currently unused switches on the old path need to be removed. We compare the old path with the new path to build the relative complement of the old path and store it in the *flowDelList* (Line 34). Subsequently, each forwarding rule per switch in the *flowDelList* is deleted for this specific flow (Lines 35-38).

Generally, we note that for the case of a leaving flow we need some timeout to remove entries from the Q-table and switches, since flows do not give notice when leaving the network.

Algorithm 2 (Re)routing Algorithm

```
 1: procedure REROUTING(OLDPATH, NEWPATH, FLOWID)
 2:     ▷ New switches on the path
 3:     flowAddList ← [ ]
 4:     ▷ Modify routes of switches on the path
 5:     flowModList ← [ ]
 6:     for index, switch in ENUMERATEnewPath do
 7:         if switch in oldPath then
 8:             oldIndex ← GETINDEXoldPath, switch
 9:             ▷ Same previous switch?
10:             if oldPath[oldIndex – 1] == newPath[index – 1] then
11:                 continue
12:             else
13:                 if newPath[index – 1] not in flowAddList then
14:                     flowModList ← flowModList + newPath[index − 1]
15:                 end if
16:             end if
17:         else
18:             flowAddList ← flowAddList + switch
19:             if newPath[index – 1] not in flowAddList then
20:                 flowModList ← flowModList + newPath[index − 1]
21:             end if
22:         end if
23:     end for
24:     ▷ Add forwarding rule for flow to new switches
25:     for switch in flowAddList do
26:         nextSwitch ← newPath[GETINDEXnewPath, switch + 1]
27:         ADDFLOWSWITCHswitch, flowID, nextSwitch
28:     end for
29:     ▷ Modify forwarding rule for flow of persistent switches
30:     for switch in REVERSEDflowModList do
31:         nextSwitch ← newPath[GETINDEXnewPath, switch + 1]
32:         MODFLOWSWITCHswitch, flowID, nextSwitch
33:     end for
34:     flowDelList ← SETDIFFoldPath, newPath
35:     ▷ Delete forwarding rule for flow of old switches
36:     for switch in flowDelList do
37:         DELFLOWSWITCHswitch, flowID
38:     end for
39: end procedure
```

2) LATENCY MONITORING

An important part of our implementation is the gathering of link latency statistics. We used the approach of Phemius *et al.* [83] that creates artificial Ethernet (probing)

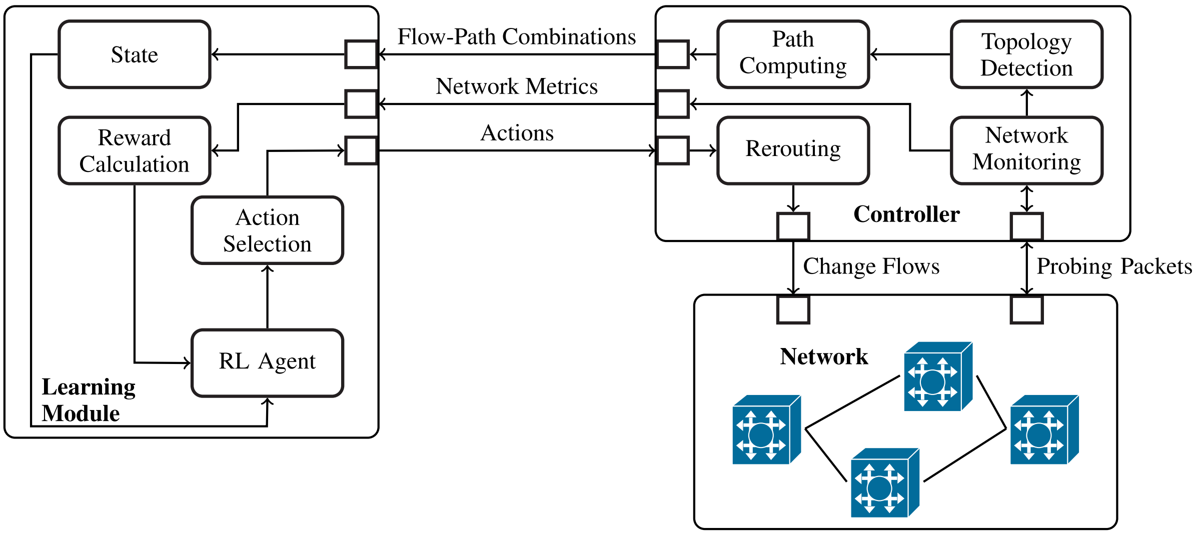


FIGURE 3. QR-SDN Controller Architecture: The controller feeds the flow-path combinations and the network metrics to the learning module. The learning module, which is co-located with the SDN controller and stores the Q-table, then conducts the reinforcement learning and selects the routing actions. The routing actions are implemented by the controller in the data plane.

packets (type $0 \times 07\mathrm{c}3$). The payload contains a timestamp and the datapath ID from the sending switch. The packets are padded with zeros to reach the minimum Ethernet packet size of 64 bytes. The packets are broadcasted every second to neighboring switches (creating a probing packet bitrate of 64 bytes/s on each link). The benefit of this approach is that the measured latency is the same as the latency that is perceived by the flows. The packet probes experience the packet processing, queuing, transmission, and propagation delays on the one-way path from the source host to the destination host. Only packet probes that arrive at the destination are considered in the latency measurement, dropped probe packets (due to full switch buffers) are ignored. We measure the latencies every second; however, we perform one learning iteration (step) every two seconds. This permits the latency monitoring and the learning process to operate asynchronously. By measuring the latency twice as frequently as the learning iterates, we ensure that the asynchronously measured values are reasonably up to date. We define the 2-second duration of one learning iteration (step) as the *step time* T_{step}.

We acknowledge that packet reordering may occur when transitioning from an old routing configuration to a new routing configuration. Therefore, routing configuration changes should be minimized so as to minimize packet reordering. Generally, the time till convergence reflects the routing configuration changes required by a learning approach and thus, the time till convergence should be minimized. After convergence, the traffic is rarely re-routed (depending on the exploration strategy). The step time T_{step}, which is equivalent to the time between routing changes, can be chosen dependent on the duration of short-lived TCP flows so as to avoid degrading their performance, while still achieving re-routing benefits for long-lived flows. In summary, there is a trade-off between either experiencing persistent high levels of congestion and long latencies with long step times, or congestion mitigation through reasonably frequent routing configuration changes according to a short step time at the expense of some out-of-order packets. The detailed investigation of this trade-off is an interesting direction for future research. We chose the step time to be quite short, i.e., only 2 seconds, in order to complete the emulation based evaluation within a reasonable time in a statistical reliable manner with multiple independent measurement replications. In practice, the step time should be set depending on how frequently or how fast the network load changes, and therefore how fast the routing should react.

We also note that a frequent execution of Algorithm 2 with a short step time can be readily accommodated by contemporary SDN controllers which can handle multiple thousands of requests per second [84].

IV. PERFORMANCE EVALUATION

A. PERFORMANCE METRICS

We use the following two metrics in our evaluation, namely time till convergence and average flow latency.

1) TIME TILL CONVERGENCE

Time till convergence describes how quickly the reinforcement learning agent finds a low latency state. This search for a low latency state should not take too long, because in reality and in our emulations, an acceleration (e.g., by faster computing hardware) is not possible; rather, the search duration is inherently coupled to the design of the search algorithm, i.e., the reinforcement learning algorithm and the duration of a time step (i.e., the time interval for one learning iteration). Thus, the reinforcement learning agent should learn as fast and efficiently as possible. To the best of our knowledge, there is no common definition for convergence in reinforcement learning. In order to quantitatively evaluate the convergence times, we first smoothed the measured average latencies of the flows with a moving average of $N = 40$ consecutive steps. Subsequently, we calculated the finite differences of the smoothed average latencies. If the variations, i.e., the finite differences of the smoothed average latencies, are smaller than a prescribed threshold value of 0.4 ms/step, we consider the system as converged. We arrived at the 0.4 ms/step threshold value through experimenting with different threshold values and observing the transient behaviors of the latencies, similar to the transient behaviors examined in Fig. 5a.

2) AVERAGE FLOW LATENCY

Latency is a useful indicator of the performance of a network as it detects congestion, which not only increases latency, but also reduces throughput. Since in a real network we would not know the latencies experienced by the actual traffic, we measure the flow latencies via probing packets, as explained in Section III-E2. We define the average latency as the unweighted average flow latency across all flows along their respective source-to-destination paths. We note that the latency variance across all the flows would not be a meaningful performance metric as the flows have vastly different source-destination switch pairs. However, we do evaluate the 5% and 95% percentiles of the average latency across multiple independent measurement replications to evaluate the variance of the evaluated QR-SDN routing approach.

B. EVALUATION SETUP

1) EVALUATION APPROACH AND HARDWARE

We evaluated our proposed approach in the network emulator *Mininet* [85], which closely resembles the operation on a network with distributed hardware. In the emulation, the execution takes place in real time and incurs the actual processing delays in real hardware. We parallelized the emulation by spawning multiple virtual machines that executed the experiments. We used a commercial workstation with an *Intel Xeon W-2155* CPU and 128 GB DDR4 memory. As hypervisor we used KVM with the *Ubuntu 18.04* host operating system and the *Debian 10* operating system for the guest machines. We used *Mininet*'s standard *OpenFlow* software switch *OpenvSwitch*.

2) LINK AND TRAFFIC SETTINGS

We decreased the standard queue capacities for *Mininet* interfaces from 1000 packets per queue to 30 packets per queue, which is consistent with recent recommendations for small router buffers [86]–[88]. The reason is that congestion should result in degradations within one step time T_{step} or between two actions, respectively. Otherwise, a change, e.g., to a redirected flow, would not be reflected in the reward. In order to have a change reflected we need:

$$\frac{\text{MTU [bytes/packet]} \cdot \text{Queue size [packets]}}{\text{Link capacity [bytes/s]}} \overset{!}{\leq} T_{step}. \quad (11)$$

Service providers can adapt the step time according to the Maximum Transfer Unit (MTU) and link capacities. Since the MTU and link capacities are fixed, only the queue sizes or step times can be adapted. For our setup, we chose a lower queue capacity of 30 packets in accordance with our considered link transmission bitrates of up to 10 Mbit/s and packet size of up to 1500 bytes, since we then have a margin of about $1500\,\text{bytes} \cdot 8 \cdot 30\,\text{packets}/250\,\text{kbit/s} = 1.44$ seconds to degrade the congestion within the step time of our implementation of 2 seconds, as mentioned in Section III-E2. For larger buffer sizes, a longer step time would be needed which would reduce the responsiveness of the routing control. The margin of about 250 kbit/s will be subtracted from the nominal flows bitrates to avoid congestion, because congestion will occur even if the nominal flow transmission bitrate and link capacity are equal. Table 1 lists the actual flow transmission bitrates for different load levels calculated as (Nominal flow bitrate – Margin) · Load level. This flow bitrate adjustment is necessary because the Max-flow min-cut specifies an achievable theoretical upper limit; however, practically congestion occurs through queuing at 100% utilization. To avoid this discrepancy, we subtract the margin from the nominal flow bitrate. The actual traffic was generated with *Iperf*.

We compare our QR-SDN approach to the classic Shortest-Path First (SPF) routing protocol, whereby we set the latency as cost. We initialize SPF in an uncongested network which only considers the pre-set default link latencies and do not adapt the SPF routes at runtime. Adapting SPF routing at runtime without a specific route stabilization mechanism leads to route oscillations [89]–[91]. For a fair evaluation, we do not consider any specific routing stabilization mechanisms, neither for the SPF benchmark nor for the QR-SDN approach. We briefly note for completeness that multi-path routing could also be considered as a benchmark; however existing multi-path routing approaches are typically limited to equal-cost links [92], [93], or require path weights [94]. Also, as noted in Section II-C, the existing reinforcement learning based routing schemes are not suitable for exploiting multiple routing paths for flows sharing the same source-destination switch pair while preserving the integrity of source-destination host pair flows (i.e., avoiding flow splitting).

The link latencies are created with *NetEm* [95], a *Mininet*'s standard tool for network emulation. Due to the limited link transmission bitrates, each instance of exceeding of the link capacities leads to more packets in the queues and thus to longer queuing delays. In particular, *NetEm* evaluates the link latency as the maximum of the pre-set default link latency and the queuing delay. Thus, for low link load, the link latency is equal to the pre-set default link latency, while for high link load, the link latency is equal to the queuing delay.

C. RESULT OF BASIC EVALUATION

We first evaluate how QR-SDN performs in comparison to SPF, and how fast QR-SDN converges. We consider the ele-

TABLE 1. Transmission bitrates (Mbits/s) of flows with a margin of 250 kbit/s for different load levels in basic evaluation topology in Fig. 4.

Flows: Nominal bitrate	Load level in %									
	30	40	50	60	70	80	90	100	110	120
f_1 : 3 Mbits/s	0.825	1.1	1.375	1.65	1.925	2.2	2.475	2.75	3.025	3.3
f_2, f_3 : 2 Mbits/s	0.525	0.7	0.875	1.05	1.225	1.4	1.575	1.75	1.925	2.1

mentary example topology in Fig. 4. The distribution of flows that achieves the lowest average flow latency can be readily identified for this topology. Also, the QR-SDN approach can be readily emulated with Mininet in this elementary example topology. Thus, the main open question is how quickly can QR-SDN find this minimum average flow latency configuration and how often will QR-SDN try different configurations due to explorations. We load the network with the flows f_1 : 3 Mbits/s and f_2, f_3 : 2 Mbits/s. As described in Section IV-B2, we leave a margin of 250 kbit/s to the nominal flow bitrate.

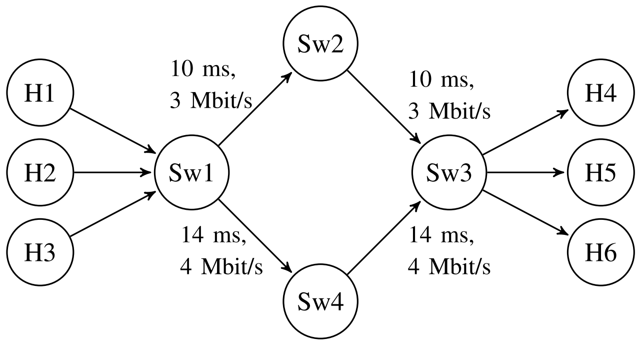


FIGURE 4. Basic evaluation topology: Three source-destination host pairs (H1-to-H4, H2-to-H5, and H3-to-H6) send flows over a network consisting of four switches with prescribed default link latencies and link bitrates, e.g., the Sw1 → Sw2 link has a default latency of 10 ms and a bitrate of 3 Mbit/s.

1) TRANSIENT BEHAVIOR UNDER HIGH LOAD

First we evaluate the transient behavior of the system under high load (100% load level of the three flows f_1, f_2, f_3). SPF chooses the lowest-latency path Sw1 → Sw2 → Sw3 as default route. However, the best flow distribution, in terms of the minimum average flow latency, would be f_1 via route Sw1 → Sw2 → Sw3, and f_2, f_3 via route Sw1 → Sw4 → Sw3. Any other flow distribution would lead to congestion. Fig. 5(a) shows the sample average of 20 independent measurement replications as thick lines and the corresponding 5% and 95% percentiles as shaded areas. Each step on the x-axis represents one action-measurement cycle, as depicted in Fig. 2.

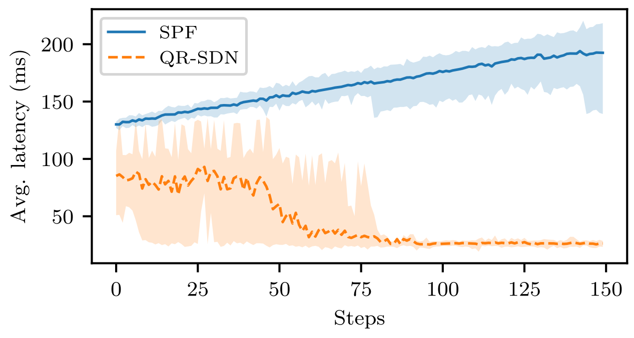
With SPF, the Sw1 → Sw2 link is congested right away, since the queue of 30 packets is filled quickly by the total flow rate of 7 Mbit/s arriving to Sw1 (SPF achieves a throughput of only 3 Mbit/s, the other 4 Mbit/s are lost). The latency rises therefore essentially immediately to about 130 ms, which is the sum of the queuing delay of the congested Sw1 → Sw2 link (MTU $\cdot$ 8 $\cdot$ Queue size/Link capacity $\approx$ 120 ms) and the 10 ms default latency of the initially uncongested Sw2 → Sw3 link. The Sw2 → Sw3 link is also gradually getting congested as it is 100% loaded and some random variations of the processing delays occur due to the emulation hardware.

Due to the random initialization of QR-SDN, the latency is already lower at the beginning since the probability of routing all three flows over one of the two paths is only $\frac{1}{2^3} \cdot 2$, i.e., 25%. Up to step 60, QR-SDN tries different combinations, as indicated by the latency peaks. Around step 90, the exploration decreases and QR-SDN converges to a steady state. QR-SDN nearly reaches the minimum average flow latency of $(2 \cdot 28 + 20)/3$ ms $\approx$ 25.33 ms, which is the ideal routing configuration with f_1 over Sw1 → Sw2 → Sw3 and flows f_2, f_3 over Sw1 → Sw4 → Sw3. However, due to some ongoing explorations there will always be some latency overhead compared to the minimum latency. This tradeoff between exploration and exploitation to achieve low latency will be evaluated in Section IV-C3.

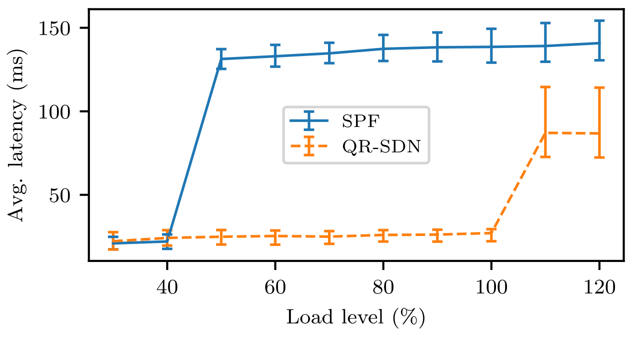
2) AVERAGE LATENCY VS. LOAD LEVEL

Next, we examine the average latency after convergence as a function of the load level. Fig. 5(b) shows the average flow latencies and the corresponding 5% and 95% percentiles as whiskers obtained from 20 independent measurement replications. For very low load levels up to 40%, SPF achieves very slightly shorter latencies than QR-SDN. This is because SPF routes only over the lowest latency path. In contrast, QR-SDN continuously explores other flow-path combinations (even after having converged), which creates a minuscule latency overhead.

We observe from Fig. 5(b) that for a load level of 50%, which exceeds the capacity of the Sw1 → Sw2 → Sw3 path (which provides 3 Mbps out of the total 3 + 4 Mbit/s network capacity), the SPF delays increase to nearly 150 ms, In contrast, QR-SDN maintains low average flow latencies up to a load of 100% and then gives increased average delays around 100 ms while SPF gives average delays around 150 ms. With SPF, all flows traverse the congested Sw1 → Sw2 → Sw3 path. On the other hand, with QR-SDN, the flows are distributed over both paths, so that the flows experience varying levels of congestion, as indicated by the relatively wide span of the 5% to 95% percentiles, while the average latency remains below the SPF latency. Overall, we conclude from the average latency results in Fig. 5(b) that for low loads, SPF achieves very slightly shorter latencies than QR-SDN, while for moderate to high loads, QR-SDN achieves substantial latency reductions over SPF. Future research could develop a load-adaptive routing strategy that employs SPF at low loads and switches to QR-SDN for moderate to high loads.



(a) Transient behavior, high (100%) load.



(b) Average latency as a function of load.

FIGURE 5. Basic flow latency evaluation of QR-SDN: The flow-preserving multi-path QR-SDN with direct flow route representation achieves substantially shorter average flow latencies than the single-path shortest-path routing (SPF) for moderate to high loads; prior indirect routing representation studies with a single path, e.g., [19], [20], correspond roughly to SPF, the prior multi-path flow-splitting approach [21] splits flows on a per-packet basis (incurring complexity and out-of-order packets). Fixed parameters: *Softmax* with $\tau = 0.00005$ and *OneFlow* change.

Importantly, all routing approaches, including the recent deep reinforcement learning approaches, e.g., TIDE [19] and DROM [20], which determine a single routing path for a given source-destination switch pair, will give latencies similar to SPF. Only the random per-packet traffic splitting approach of DRL-TE [21] can achieve the same low latencies as QR-SDN. However, DRL-TE does not preserve the integrity of flows, as outlined in Section III-A. Therefore, to the best of our knowledge, QR-SDN is presently the only flow-preserving reinforcement learning based SDN multi-path routing approach for achieving low latencies for high loads that exceed the capacities of a single source-destination route.

3) EXPLORATION VS. EXPLOITATION

This section compares the different exploration strategies outlined in Section III-D2, namely *ϵ-greedy*, *Softmax*, and *UCB*. Generally, the exploration determines how fast the reinforcement learning agent can find a state with low latency. However, too much exploration limits the exploitation of a beneficial low-latency state that the learning had converged to. In our experiments, the *ϵ-greedy* approach did not converge within a reasonable time (of tens of hours); following the standard *ϵ-greedy* algorithm, we did not decrease the ϵ value over time (we tested the fixed values $\epsilon = 0.01, 0.05, 0.1, 0.15, 0.2, 0.25,$ and 0.3). Thus, we focus on *UCB* and *Softmax*. We compare the time until the learning converges to a steady state and the average flow latency (after convergence) for *UCB* and *Softmax* in Figs. 6 and 7.

Initially, we evaluate in Fig. 6 the optimistic Q-value initialization that we introduced in Section III-D2 for *Softmax* (see in particular Algorithm 1) compared to the initialization $Q(s, a) = -\infty$. We observe from Fig. 6 that the optimistic Q-value initialization achieves substantially shorter convergence times for the low temperature $\tau = 0.0001$ than the $Q(s, a) = -\infty$ initialization; whereas for the high temperature $\tau = 0.0005$, the optimistic Q-value initialization does not shorten the convergence time compared to the $Q(s, a) = -\infty$ initialization. These results are mainly attributed to the high temperature $\tau = 0.0005$ leading to an excessive amount of exploration, which cannot be effectively curtailed by the optimistic Q-value initialization.

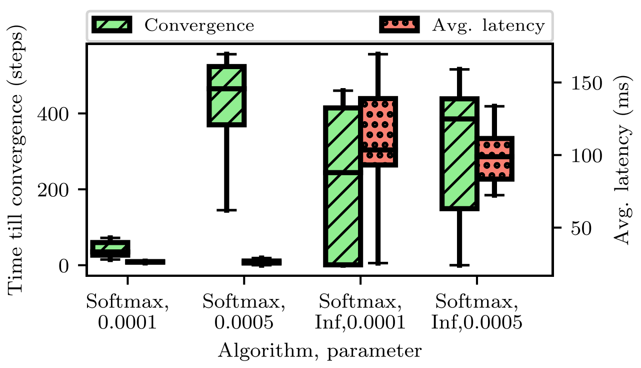


FIGURE 6. Comparison of initialization values for *Softmax*: Convergence time and average flow latency (after convergence) for *optimistic Q-value initialization* and the default initial value of $-\infty$ (denoted as `Inf`), see Section III-D2, for different τ values. Fixed parameters: Load level 100% and *OneFlow* change.

We also observe from Fig. 6 that the optimistic Q-value initialization achieves dramatically shorter flow latencies than the $Q(s, a) = -\infty$ initialization, irrespective of the temperature τ. This is because although the learning has converged, the learning may not necessarily have converged to a low latency state. This is also reflected in the wide range of measurement results for *Softmax* with the $Q(s, a) = -\infty$ initialization. In particular, *Softmax-Inf.* with $\tau = 0.0001$ had some iterations which converged as fast as *Softmax* with optimistic initialization and achieved a similar average latency (cf. whiskers). That is, the initialization appears to improve the probability of converging to a more beneficial state.

We observe from Fig. 7 that *UCB* with $c = 30$ achieves the lowest average flow latencies, while the average flow latencies for *Softmax* are slightly longer (albeit still quite short for *Softmax* with low temperature τ). The slightly lower latency with *UCB* is due to the *annealing* in *UCB*, which

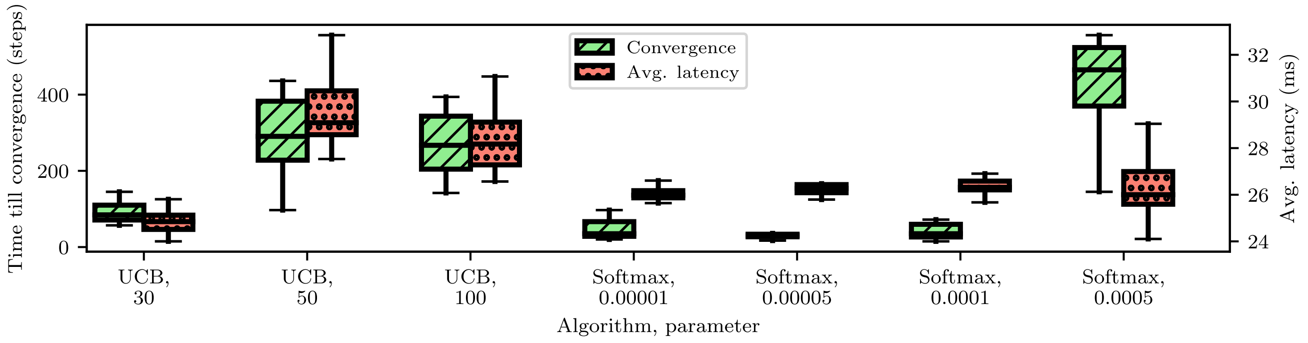
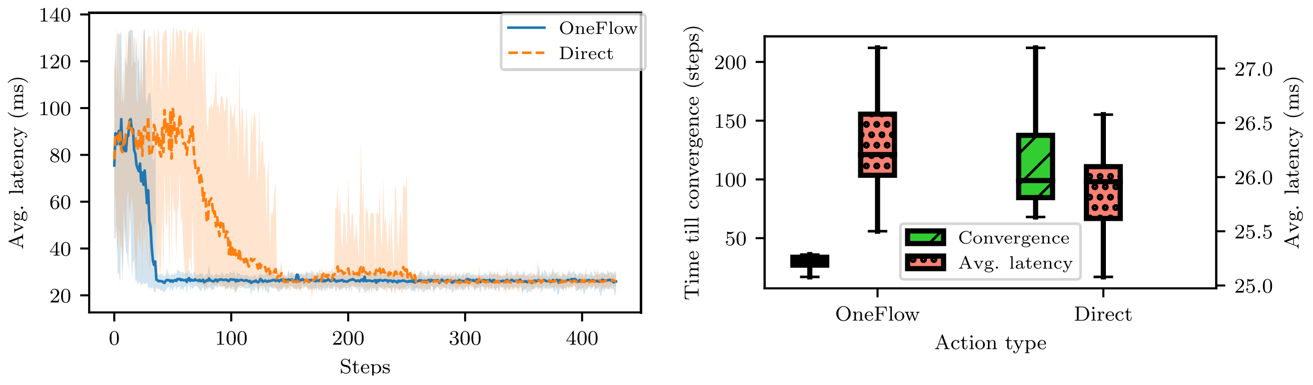


FIGURE 7. Comparison of exploration strategies: Convergence time and average flow latency (after convergence) for *UCB* with different *c* parameter values and *Softmax* for different temperature τ values with optimistic initialization (Section III-D2). Fixed parameters: Load level 100% and *OneFlow* change.



(a) Transient behavior of average flow latency.

(b) Convergence time.

FIGURE 8. Comparison of action design types: *OneFlow Change* and *Direct Change*. Fixed parameters: Load level 100%, *Softmax* with $\tau = 0.00005$.

decreases the exploration over time, and rather exploits the discovered optimal routing configuration.

We also observe from Fig. 7 that *Softmax* achieves the shortest convergence times for low temperature τ values (0.00001 – 0.0001), followed by *UCB* for the small $c = 30$. In contrast, *Softmax* for $\tau = 0.0005$ and *UCB* with the high $c = 50$ and 100 give long convergence times. Since both values, τ and c, control exploration, too high values cause too much exploration, so that a more advantageous state is only exploited later. On the other hand, too little exploration can lead to local minima, as shown in Section IV-D2.

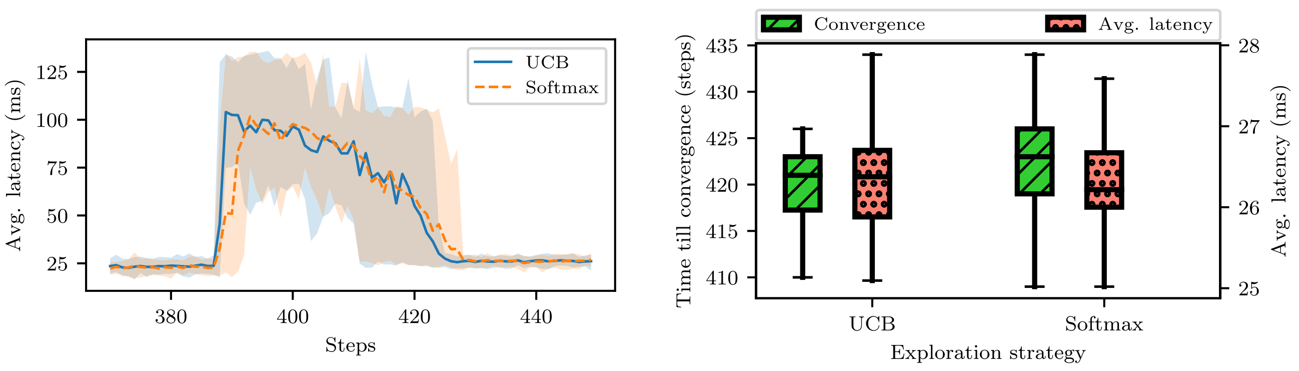
4) INFLUENCE OF ACTION DESIGN

Next, we evaluate the actions design tradeoff by comparing *OneFlow Change* and *Direct Change* in Fig 8. We repeated the experiment from before and used *Softmax* with $\tau = 0.00005$ as exploration strategy. Fig. 8 indicates that the *Direct Change* approach converges slower than the *OneFlow Change* approach. This is mainly because the *Direct Change* approach has a larger action space than the *OneFlow Change* approach. Thus, with *Direct Change*, the reinforcement learning agent has to evaluate more possible state-action pairs before it can converge.

On the other hand, we observe from Fig. 8 that the *Direct Change* approach causes less latency overhead due to exploration, i.e., achieves a lower average latency after convergence, as clearly indicated by the red box plots in Fig. 8b as well as by the slightly lower *Direct* latency at the right side of Fig. 8a. The main reason for this behavior is that *Direct Change*, after having explored various state-action pairs, can directly switch between different beneficial (low-latency) states without having to traverse (in a step-by-step one-flow change fashion) disadvantageous (high-latency) states.

In summary, *Direct Change* is worse than *OneFlow Change* in terms of scalability and therefore convergence speed, but causes less latency overhead due to exploration. The size of the action space for *Direct Change* scales with the number of states $|\mathcal{S}|$; specifically, the size of the action space is equal to the size of the state space, excluding the current state S. We denote the size of the action space excluding the current state S by $|\mathcal{S}_{-S}|$. Noting that there is one additional "no transition" action gives the size of the *Direct Change* action space as

$$|\mathcal{A}_{\text{Direct}}| = |\mathcal{S}_{-S}| + 1 = \prod_{f \in \mathcal{F}} (|\mathcal{P}_{s_f,d_f}| - 1) + 1. \quad (12)$$



(a) Average latency of *UCB* and *Softmax* and their adaption to a changing load. (b) Steps till learning converges again and average latency afterwards.

FIGURE 9. Convergence time and average flow latency (after convergence) for load increase from 40% to 100%: *UCB*, $c = 30$ vs. *Softmax*, $\tau = 0.00005$ with *OneFlow change*.

In contrast, the number of actions for *OneFlow Change* grows with the total number of possible paths for all flows f, i.e.,

$$|\mathcal{A}_{\text{OneFlow}}| = \sum_{f \in \mathcal{F}} (|\mathcal{P}_{s_f,d_f}| - 1) + 1. \tag{13}$$

D. RESULTS OF ADVANCED EVALUATION

1) CHANGING ENVIRONMENT

So far, we have evaluated a static environment; in order to demonstrate that QR-SDN can handle dynamic conditions, we are now evaluating dynamic cases. Two scenarios in which the reinforcement learning approach is challenged are (*i*) changing the loads that require new optimal routing configurations, and (*ii*) when new flows join the network or ongoing flows leave the network. We note that prior reinforcement learning SDN routing studies, e.g., [19]–[21], have not covered such real-life changes of the network load or flows. To the best of our knowledge, we are the first to quantitatively examine how reinforcement based SDN routing performs for changing loads and flows.

We first compare how quickly *UCB* and *Softmax* adapt to load level changes within the context of the evaluation setup in Section IV-C. We observe from Fig. 9 that although *UCB* decreases its exploration over time, it adapts slightly faster than *Softmax* to the changing load level. The average flow latency after convergence is nearly the same for both exploration strategies; the difference is less than the accuracy of our latency measurements. We conclude that *UCB* has the advantage of annealing, i.e., *UCB* decreases the exploration over time. Nevertheless, *UCB* still manages to quickly find a new optimal routing configuration after a load level change.

In addition to changing loads, new flows can spontaneously join or leave the network. For our QR-SDN approach, this means that the state-action space changes. The question is whether we can reuse the previous experience to avoid having to completely restart the learning, which could result in a long adaption period. We propose two variants for the transfer of the already gained experience into a new Q-values table, namely *Merging* and *Reset*, and for the initial placement of the new flow, namely *Random* and *SPF*.

S_1	$A_{1,1} : Q(S_1, A_{1,1})$
	$A_{1,2} : Q(S_1, A_{1,2})$
S_2	$A_{2,1} : Q(S_2, A_{2,1})$
	$A_{2,2} : Q(S_2, A_{2,2})$

(a) Old Q-table for one flow with two possible paths (S_1 and S_2, which can be kept or changed during an action).

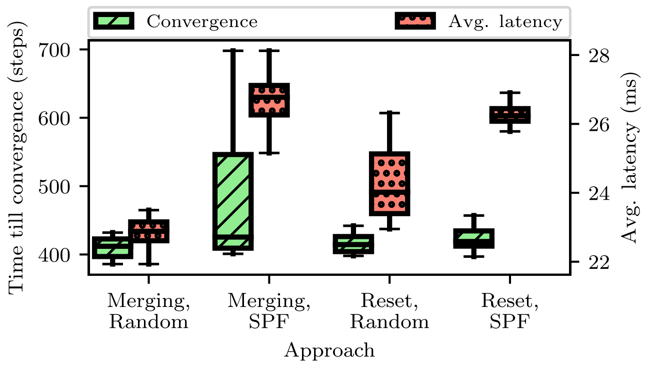
$\Rightarrow$

S'_1	$A'_{1,1} : Q'(S'_1, A'_{1,1})$
	$A'_{1,2} : Q'(S'_1, A'_{1,2})$
	$A'_{1,3} : Q'(S'_1, A'_{1,3})$
S'_2	$A'_{2,1} : Q'(S'_2, A'_{2,1})$
	$A'_{2,2} : Q'(S'_2, A'_{2,2})$
	$A'_{2,3} : Q'(S'_2, A'_{2,3})$
S'_3	$A'_{3,1} : Q'(S'_3, A'_{3,1})$
	$A'_{3,2} : Q'(S'_3, A'_{3,2})$
	$A'_{3,3} : Q'(S'_3, A'_{3,3})$
S'_4	$A'_{4,1} : Q'(S'_4, A'_{4,1})$
	$A'_{4,2} : Q'(S'_4, A'_{4,2})$
	$A'_{4,3} : Q'(S'_4, A'_{4,3})$

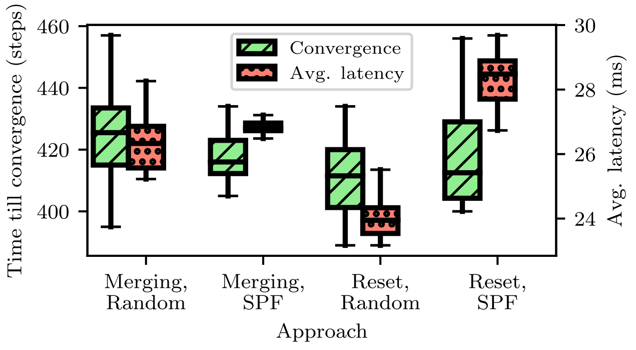
(b) New Q-table for old flow and new flow with four possible states.

FIGURE 10. Illustration of Q-table merging for OneFlow Change policy.

For transferring of the old Q-values into the new state-action space, we compare the old table $Q(s, a)$ and new table $Q'(s, a)$. For this comparison, we take the new states S' minus the entries consisting of the new flow f', i.e., $S' \setminus f'$. In the same manner, we inspect the corresponding new actions A'. If we find the entries $S' \setminus f'$ or $A' \setminus f'$ in the old table, we transfer their Q-values. All other values, which have nothing in common with the previous Q-value table, are initialized with the default Q-values. This merging approach is illustrated in Fig. 10: Consider initially one flow, which can be routed over path A (state S_1) or B (state S_2); thus, there are four actions namely stay in state S_1 or S_2, or transition to the other



(a) Measurements for topology Fig. 4.



(b) Measurements for modified topology with interchanged path latencies, i.e., with higher capacity for shorter latency path.

FIGURE 11. Comparison of approaches for joining flows. Fixed parameters: *Softmax* with $\tau = 0.00005$, 100% load level and *OneFlow Change*.

state (path), each of these four state-action pairs has a Q-value (top table). Now, consider that a new flow joins: The old flow is still routed over path A or B, i.e., the old state S_1 is now "included" in S'_1 or S'_2 and S_2 in S'_3 or $S4'$. Since the second flow can be routed over path A or B, there are a total of four combinations (states). For the first flow, there is an action to stay in the current state, or to change the path, whereby the merged approach transfers the previously learned Q-values to this new table (bottom). In addition, the new flow may stay on the current path or its path is changed; the Q-values for the new flow actions are unknown and therefore initialized with a default Q-value (gray). With the OneFlow Change policy, only one flow routing is changed at a time, i.e., the possible actions for the illustrative example in Fig. 10(b) are no routing path change, old flow changes path, or new flow changes path.

In the case of *Reset*, we simply reinitialize the Q-table with the default values, which deletes all learned Q-values. For the initial placement of a new flow, we either select randomly (*Random*) a possible path or use the current shortest path (*SPF*).

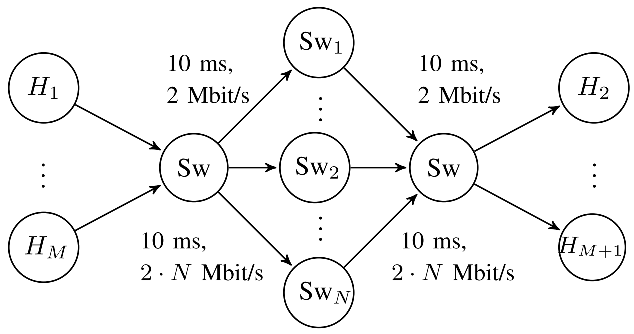


FIGURE 12. Scalability evaluation topology with *N* intermediate switches and *M* flows.

We consider the previously used evaluation topology and in addition (to demonstrate that *SPF* as flow initialization is highly dependent on the topology) we modified the topology depicted in Fig. 4 in such a way that the path latencies are

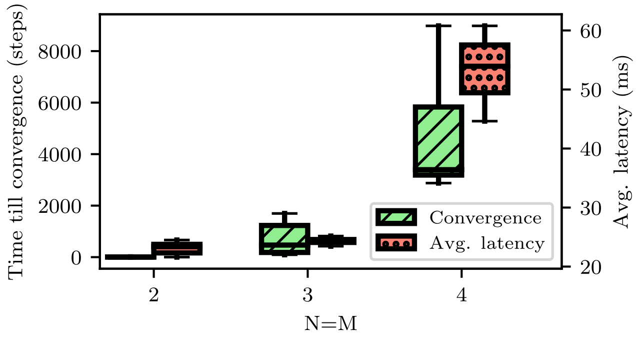


FIGURE 13. Convergence time and average flow latency for increasing number *N* of intermediate switches and number *M* of flows. Fixed parameters: *Softmax* with $\tau = 0.00005$ and *OneFlow Change*.

interchanged, i.e., Sw1 → Sw2 → Sw3: 28 ms, Sw1 → Sw4 → Sw3: 20 ms. The new flow is randomly selected from the three source-destination host pairs, see Fig. 4, and is started later, after the learning has settled for the first two source-destination host pairs. We observe from Fig. 11a that the initial placement of the new flow based on shortest-path converges slower and has the highest average flow latency after convergence. This is due to the shortest path, namely Sw1 → Sw2 → Sw3, which is getting congested if the new flow is routed over it. In contrast, with random placement, the new flow may also be routed via the other path, which avoids congestion. Even if—as in Fig. 11b—the shorter path has a higher capacity, the SPF based approach does not appear to offer an advantage. To conclude, random initial placement of new flows performs better than shortest path placement in high load scenarios, (however, for low load scenarios, SPF placement may be advantageous). Future research could examine placement decisions based on the remaining idle link bandwidth as higher utilization leads to a higher probability of the new flow congesting a path.

Regarding the transfer, we observe from Fig. 11a that *Merging* performs better than *Reset* for random placement, while both transfer approaches have similar mean convergence times and flow latencies for SPF placement, with *Merging* experiencing more long outlier convergence times and flow latencies. On the other hand, for the scenario in Fig. 11b, *Merging* gives shorter flow latencies than *Reset* for SPF placement, while *Merging* performs worse than *Reset* for random placement. Thus, *Merging* appears to have some performance advantages, although the transfer results are overall somewhat inconclusive. More evaluation scenarios need to be considered to conclusively evaluate the transfer approaches.

2) SCALABILITY

Scalability is a major challenge of reinforcement learning based routing. We examine scalability with the topology in Fig. 12 with N intermediate switches and M flows, whereby the links of the intermediate switch Swi support $i \cdot 2$ Mbits/s and each flow f_i from host H_i to H_{i+1} has a bandwidth demand of $i \cdot 2$ Mbits/s (minus 250 kbits/s margin). The optimal flow distribution can be easily deduced for this topology, namely f_1 over Sw1, f_i over Swi, and so on. For this topology, we can calculate the Q-table size for *OneFlow Change* as:

$$\underbrace{|Q(s,a)|}_{\text{Size of Q-table}} = \underbrace{N^M}_{\text{Number of states}} * \underbrace{\left[(N-1)\cdot M + 1\right]}_{\text{Number of actions}}. \quad (14)$$

Thus, the Q-value table size critically depends on the number of flows M. We evaluated the time till convergence and the average flow latency (after convergence) for the combinations of $N = M = 2, 3$, and 4 switches and flows and plot the results in Fig. 13. As expected, the time to convergence increases with increasing Q-table size. In addition, a new phenomenon occurs, namely the convergence to a local minimum. In the case of $N, M = 4$, QR-SDN converges to an average latency of about 55 ms; whereas, for $N, M = 2, 3$, QR-SDN converges nearly to the optimum of 20 ms. Thus, QR-SDN was unable to find the optimal distribution for $N, M = 4$, which could be due to a lack of exploration for the larger $N, M = 4$ topology. Future research needs to examine the adaptation of the QR-SDN exploration according to the size of the topology. Future research could also examine the reduction of the Q-table size by decreasing the number of possible paths as done by Xu *et al.* [21]. This, however, could eliminate possible states that could be more advantageous.

V. CONCLUSION AND OUTLOOK

We have developed and evaluated a classical tabular reinforcement learning (Q-Learning) approach for flow routing in Software-Defined Networks (SDNs). Our approach, which we refer to as QR-SDN, directly represents the flow routes in the Q-Learning state and action spaces so as to enable flow-preserving multi-path routing. We have implemented QR-SDN in a network emulation testbed and make the full source code publicly available to facilitate future research [23].

We have extensively evaluated the direct representation of the SDN flow routing problem in QR-SDN in small networks. The evaluations have demonstrated that the flow-preserving multi-path routing of QR-SDN achieves substantially lower latencies than existing single-path routing approaches for moderate to high loads. We have compared action spaces that re-route one flow at a time versus action spaces that directly re-route the entire set of ongoing flows. We found that one-flow re-routing tends to converge faster, while direct re-routing of the complete set of flows tends to achieve lower average flow latencies. We have also found that QR-SDN effectively accommodates changes, such as load changes due to new flows or flows that terminate. However, the evaluations also revealed the scalability problems of the proposed direct flow routing state representation for reinforcement learning. Thus, this study presents overall mixed results: The direct flow routing representation is highly effective in achieving low latencies and preserving the flow integrity in small multi-path networks. However, in large networks with many flows, the direct flow routing representation leads to scalability problems in the form of slow convergence.

By formulating and evaluating direct flow representations for Q-Learning and providing the QR-SDN source code, the present study provides the groundwork for several important future research directions. One important future research direction is to address the scalability issues arising from the direct flow routing representation in the QR-SDN state-action space. Future research needs to examine how QR-SDN can operate in a scalable manner, e.g., by employing multiple SDN controllers, and through judicious scalability-oriented path exploration strategies, e.g., focusing on currently highly utilized links (which may become congested in the near future) and on high-rate flows that dominate congestion (while the vast majority of low-rate flows could be routed on the single shortest path). Also, improved learning algorithms, e.g., approaches that embed the discrete actions into a continuous space and apply efficient continuous policies [22], could aid to ensure scalability.

Moreover, we believe that a highly critical direction for future research is to develop and evaluate novel exploration strategies that reduce the convergence time of the reinforcement learning agent to an optimal routing configuration. A related interesting future research direction is to investigate novel exploration and reward strategies that are triggered by load levels on critical links. Another interesting direction is to explore approaches that proactively manage time-varying loads, e.g., by forecasting traffic peaks and proactively adjusting the flow routing. Also, a possible future research direction is to examine the use of a neural network as a function approximator to improve scalability, i.e., to examine deep reinforcement learning, for SDN routing.

REFERENCES

[1] R. K. Ahyja, J. B. Orlin, and T. L. Magnanti, *Network Flows: Theory, Algorithms, and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, 1993.

[2] M. Beshley, M. Seliuchenko, O. Panchenko, and A. Polishuk, "Adaptive flow routing model in SDN," in *Proc. IEEE Int. Conf. Exp. Designing Appl. CAD Syst. Microelectron. (CADSM)*, 2017, pp. 298–302.

[3] R. F. Diorio and V. S. Timóteo, "Per-flow routing with QoS support to enhance multimedia delivery in OpenFlow SDN," in *Proc. Brazilian Symp. Multimedia Web*, 2016, pp. 167–174.

[4] S. T. V. Pasca, S. S. P. Kodali, and K. Kataoka, "AMPS: Application aware multipath flow routing using machine learning in SDN," in *Proc. 23rd Nat. Conf. Commun. (NCC)*, Mar. 2017, pp. 1–6.

[5] W. Dai, K.-T. Foerster, D. Fuchssteiner, and S. Schmid. (2020). *Load-Optimization in Reconfigurable Networks: Algorithms and Complexity of Flow Routing*. [Online]. Available: https://www.univie.ac.at/ct/stefan/perf20loaddan.pdf

[6] X. Fei, F. Liu, H. Xu, and H. Jin, "Adaptive VNF scaling and flow routing with proactive demand prediction," in *Proc. IEEE INFOCOM*, Apr. 2018, pp. 486–494.

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

[8] V. Francois-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *Found. Trends Mach. Learn.*, vol. 11, nos. 3–4, pp. 219–354, 2018.

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2015, *arXiv:1509.02971*. [Online]. Available: http://arxiv.org/abs/1509.02971

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[11] 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 Commun. Surveys Tuts.*, vol. 21, no. 4, pp. 3133–3174, 4th Quart., 2019.

[12] D. Rafique and L. Velasco, "Machine learning for network automation: Overview, architecture, and applications [invited tutorial]," *IEEE/OSA J. Opt. Commun. Netw.*, vol. 10, no. 10, pp. D126–D143, Oct. 2018.

[13] J. Xie, F. R. Yu, T. Huang, R. Xie, J. Liu, C. Wang, and Y. Liu, "A survey of machine learning techniques applied to software defined networking (SDN): Research issues and challenges," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 1, pp. 393–430, 1st Quart., 2019.

[14] 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 Things J.*, vol. 5, no. 6, pp. 4319–4327, Dec. 2018.

[15] Y. Zhao, Y. Li, X. Zhang, G. Geng, W. Zhang, and Y. Sun, "A survey of networking applications applying the software defined networking concept based on machine learning," *IEEE Access*, vol. 7, pp. 95397–95417, 2019.

[16] K.-T. Foerster, S. Schmid, and S. Vissicchio, "Survey of consistent software-defined network updates," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1435–1461, 2nd Quart., 2019.

[17] J. Sachs, L. A. A. Andersson, J. Araújo, C. Curescu, J. Lundsjö, G. Rune, E. Steinbach, and G. Wikstrom, "Adaptive 5G low-latency communication for tactile Internet services," *Proc. IEEE*, vol. 107, no. 2, pp. 325–349, Feb. 2019.

[18] Z. Xiang, F. Gabriel, E. Urbano, G. T. Nguyen, M. Reisslein, and F. H. P. Fitzek, "Reducing latency in virtual machines: Enabling tactile Internet for human-machine co-working," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 5, pp. 1098–1116, May 2019.

[19] P. Sun, Y. Hu, J. Lan, L. Tian, and M. Chen, "TIDE: Time-relevant deep reinforcement learning for routing optimization," *Future Gener. Comput. Syst.*, vol. 99, pp. 401–409, Oct. 2019.

[20] 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. 64533–64539, 2018.

[21] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven networking: A deep reinforcement learning based approach," in *Proc. IEEE INFOCOM*, Apr. 2018, pp. 1871–1879.

[22] G. Dulac-Arnold, R. Evans, H. van Hasselt, P. Sunehag, T. Lillicrap, J. Hunt, T. Mann, T. Weber, T. Degris, and B. Coppin, "Deep reinforcement learning in large discrete action spaces," 2015, *arXiv:1512.07679*. [Online]. Available: http://arxiv.org/abs/1512.07679

[23] P. Sossalla and J. Rischke. *QR-SDN Code*. Accessed: Sep. 2, 2020. [Online]. Available: https://github.com/justus-comnets/qr-sdn

[24] A. Azzouni, R. Boutaba, and G. Pujolle, "NeuRoute: Predictive dynamic routing for software-defined networks," in *Proc. 13th Int. Conf. Netw. Service Manage. (CNSM)*, Nov. 2017, pp. 1–6.

[25] B. Mao, F. Tang, Z. M. Fadlullah, and N. Kato, "An intelligent route computation approach based on real-time deep learning strategy for software defined communication systems," *IEEE Trans. Emerg. Topics Comput.*, early access, Feb. 14, 2020, doi: 10.1109/TETC.2019.2899407.

[26] J. Fu, A. Kumar, M. Soh, and S. Levine, "Diagnosing bottlenecks in deep Q-learning algorithms," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 2021–2030.

[27] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Proc. AAAI Conf. Artif. Intell.*, 2018, pp. 3215–3222.

[28] C. Blundell, B. Uria, A. Pritzel, Y. Li, A. Ruderman, J. Z. Leibo, J. Rae, D. Wierstra, and D. Hassabis, "Model-free episodic control," 2016, *arXiv:1606.04460*. [Online]. Available: http://arxiv.org/abs/1606.04460

[29] S. Ritter, J. X. Wang, Z. Kurth-Nelson, and M. M. Botvinick, "Episodic control as meta-reinforcement learning," *bioRxiv*, to be published.

[30] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[31] E. Akin and T. Korkmaz, "Comparison of routing algorithms with static and dynamic link cost in SDN," in *Proc. 16th IEEE Annu. Consum. Commun. Netw. Conf. (CCNC)*, Jan. 2019, pp. 1–8.

[32] A. BinSahaq, T. Sheltami, and K. Salah, "A survey on autonomic provisioning and management of QoS in SDN networks," *IEEE Access*, vol. 7, pp. 73384–73435, 2019.

[33] G. Baggio, R. Bassoli, and F. Granelli, "Cognitive software-defined networking using fuzzy cognitive maps," *IEEE Trans. Cognit. Commun. Netw.*, vol. 5, no. 3, pp. 517–539, Sep. 2019.

[34] J. W. Guck, A. Van Bemten, M. Reisslein, and W. Kellerer, "Unicast QoS routing algorithms for SDN: A comprehensive survey and performance evaluation," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 1, pp. 388–415, 1st Quart., 2018.

[35] M. He, A. M. Alba, A. Basta, A. Blenk, and W. Kellerer, "Flexibility in softwarized networks: Classifications and research challenges," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 3, pp. 2600–2636, 3rd Quart., 2019.

[36] W. Kellerer, P. Kalmbach, A. Blenk, A. Basta, M. Reisslein, and S. Schmid, "Adaptable and data-driven softwarized networks: Review, opportunities, and challenges," *Proc. IEEE*, vol. 107, no. 4, pp. 711–731, Apr. 2019.

[37] B. Schlinker, H. Kim, T. Cui, E. Katz-Bassett, H. V. Madhyastha, I. Cunha, J. Quinn, S. Hasan, P. Lapukhov, and H. Zeng, "Engineering egress with edge fabric: Steering oceans of content to the world," in *Proc. ACM SIGCOMM*, 2017, pp. 418–431.

[38] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A survey on software-defined networking," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 1, pp. 27–51, 1st Quart., 2015.

[39] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA, USA: MIT Press, 2018.

[40] J. A. Boyan and M. L. Littman, "Packet routing in dynamically changing networks: A reinforcement learning approach," in *Proc. Int. Conf. Neural Inf. Syst. (NIPS)*. San Francisco, CA, USA: Morgan Kaufmann, 1993, pp. 671–678.

[41] S. Choi and D.-Y. Yeung, "Predictive Q-routing: A memory-based reinforcement learning approach to adaptive traffic control," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*. Cambridge, MA, USA: MIT Press, 1996, pp. 945–951.

[42] P. Marbach, O. Mihatsch, and J. N. Tsitsiklis, "Call admission control and routing in integrated services networks using neuro-dynamic programming," *IEEE J. Sel. Areas Commun.*, vol. 18, no. 2, pp. 197–208, Feb. 2000.

[43] L. Peshkin and V. Savova, "Reinforcement learning for adaptive routing," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, vol. 2, May 2002, pp. 1825–1830.

[44] Z. Hu and H. Chen, "Network load balancing strategy based on supervised reinforcement learning with shaping rewards," in *Proc. 4th Int. Conf. Intell. Control Inf. Process. (ICICIP)*, Jun. 2013, pp. 393–397.

[45] Y. Kiran, T. Venkatesh, and C. R. Murthy, "A reinforcement learning framework for path selection and wavelength selection in optical burst switched networks," *IEEE J. Sel. Areas Commun.*, vol. 25, no. 9, pp. 18–26, Dec. 2007.

[46] A. Al-Jawad, P. Shah, O. Gemikonakli, and R. Trestian, "LearnQoS: A learning approach for optimizing QoS over multimedia-based SDNs," in *Proc. IEEE Int. Symp. Broadband Multimedia Syst. Broadcast. (BMSB)*, Jun. 2018, pp. 1–6.

[47] V. Tilwari, K. Dimyati, M. Hindia, A. Fattouh, and I. S. Amiri, "Mobility, residual energy, and link quality aware multipath routing in MANETs with Q-learning algorithm," *Appl. Sci.*, vol. 9, no. 8, pp. 1582.1–1582.23, 2019.

[48] C. Qiu, S. Cui, H. Yao, F. Xu, F. R. Yu, and C. Zhao, "A novel QoS-enabled load scheduling algorithm based on reinforcement learning in software-defined energy Internet," *Future Gener. Comput. Syst.*, vol. 92, pp. 43–51, Mar. 2019.

[49] C. Fang, C. Cheng, Z. Tang, and C. Li, "Research on routing algorithm based on reinforcement learning in SDN," *J. Phys. Conf. Ser.*, vol. 1284, no. 1, pp. 012053-1–012053-10, 2019.

[50] 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 *Proc. IEEE Int. Conf. Services Comput. (SCC)*, Jun. 2016, pp. 25–33.

[51] T. Mahboob, Y. R. Jung, and M. Y. Chung, "Optimized routing in software defined networks—A reinforcement learning approach," in *Proc. Int. Conf. Ubiquitous Inf. Manage. Commun.* Cham, Switzerland: Springer, 2019, pp. 267–278.

[52] Z. Mammeri, "Reinforcement learning based routing in networks: Review and classification of approaches," *IEEE Access*, vol. 7, pp. 55916–55950, 2019.

[53] A. Outtagarts, "Deep reinforcement learning based QoS-aware routing in knowledge-defined networking," in *Proc. EAI Int. Conf. Quality, Rel., Secur. Robustness Heterogeneous Syst.*, vol. 272. Springer, Dec. 2018, pp. 14–26.

[54] P. Swain, U. Kamalia, R. Bhandarkar, and T. Modi, "CoDRL: Intelligent packet routing in SDN using convolutional deep reinforcement learning," in *Proc. IEEE Int. Conf. Adv. Netw. Telecommun. Syst. (ANTS)*, Dec. 2019, pp. 1–6.

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

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

[57] P. Sun, J. Li, Z. Guo, Y. Xu, J. Lan, and Y. Hu, "SINET: Enabling scalable network routing with deep reinforcement learning on partial nodes," in *Proc. ACM SIGCOMM Conf. Posters Demos*, 2019, pp. 88–89.

[58] X. You, X. Li, Y. Xu, H. Feng, J. Zhao, and H. Yan, "Toward packet routing with fully-distributed multi-agent deep reinforcement learning," 2019, *arXiv:1905.03494*. [Online]. Available: http://arxiv.org/abs/1905.03494

[59] Z. Zhang, L. Ma, K. Poularakis, K. K. Leung, and L. Wu, "DQ scheduler: Deep reinforcement learning based controller synchronization in distributed SDN," in *Proc. IEEE ICC*, May 2019, pp. 1–7.

[60] Z. Xu, J. Tang, C. Yin, Y. Wang, and G. Xue, "Experience-driven congestion control: When multi-path TCP meets deep reinforcement learning," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 6, pp. 1325–1336, Jun. 2019.

[61] Q. Fu, E. Sun, K. Meng, M. Li, and Y. Zhang, "Deep Q-learning for routing schemes in SDN-based data center networks," *IEEE Access*, vol. 8, pp. 103491–103499, 2020.

[62] 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. 37109–37120, 2019.

[63] Y. Du, Y. Xu, L. Xue, L. Wang, and F. Zhang, "An energy-efficient cross-layer routing protocol for cognitive radio networks using apprenticeship deep reinforcement learning," *Energies*, vol. 12, no. 14, pp. 2829-1–2829-21, 2019.

[64] T. Hendriks, M. Camelo, and S. Latré, "Q^2-routing: A QoS-aware Q-routing algorithm for wireless ad hoc networks," in *Proc. Int. Conf. Wireless Mobile Comput., Netw. Commun. (WiMob)*, 2018, pp. 108–115.

[65] R. Huang, L. Ma, G. Zhai, J. He, X. Chu, and H. Yan, "Resilient routing mechanism for wireless sensor networks with deep learning link reliability prediction," *IEEE Access*, vol. 8, pp. 64857–64872, 2020.

[66] Q. Medhat Salih, M. A. Rahman, F. Al-Turjman, and Z. R. M. Azmi, "Smart routing management framework exploiting dynamic data resources of cross-layer design and machine learning approaches for mobile cognitive radio networks: A survey," *IEEE Access*, vol. 8, pp. 67835–67867, 2020.

[67] A. Paul, A. Banerjee, and S. P. Maity, "Residual energy maximization in cognitive radio networks with Q-routing," *IEEE Syst. J.*, vol. 14, no. 3, pp. 3791–3800, Sep. 2020.

[68] S. Wang and Y. Shin, "Efficient routing protocol based on reinforcement learning for magnetic induction underwater sensor networks," *IEEE Access*, vol. 7, pp. 82027–82037, 2019.

[69] 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, Aug. 2019.

[70] W. Xia, C. Di, H. Guo, and S. Li, "Reinforcement learning based stochastic shortest path finding in wireless sensor networks," *IEEE Access*, vol. 7, pp. 157807–157817, 2019.

[71] C. Zhang, P. Patras, and H. Haddadi, "Deep learning in mobile and wireless networking: A survey," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 3, pp. 2224–2287, 3rd Quart., 2019.

[72] D. Wang, Q. Zhang, J. Liu, and D. Yao, "A novel QoS-awared grid routing protocol in the sensing layer of Internet of vehicles based on reinforcement learning," *IEEE Access*, vol. 7, pp. 185730–185739, 2019.

[73] J. Wu, M. Fang, H. Li, and X. Li, "RSU-assisted traffic-aware routing based on reinforcement learning for urban VANETs," *IEEE Access*, vol. 8, pp. 5733–5748, 2020.

[74] X. Chen, B. Li, R. Proietti, H. Lu, Z. Zhu, and S. J. B. Yoo, "DeepRMSA: A deep reinforcement learning framework for routing, modulation and spectrum assignment in elastic optical networks," 2019, *arXiv:1905.02248*. [Online]. Available: http://arxiv.org/abs/1905.02248

[75] M. R. Raza, C. Natalino, P. Öhlen, L. Wosinska, and P. Monti, "Reinforcement learning for slicing in a 5G flexible RAN," *IEEE/OSA J. Lightw. Technol.*, vol. 37, no. 20, pp. 5161–5169, Oct. 2019.

[76] S. Troia, D. E. Martinez, I. Martin, L. M. M. Zorello, G. Maier, J. A. Hernandez, O. G. de Dios, M. Garrich, J. L. Romero-Gazquez, F.-J. Moreno-Muro, P. P. Marino, and R. Casellas, "Machine learning-assisted planning and provisioning for SDN/NFV-enabled metropolitan networks," in *Proc. Eur. Conf. Netw. Commun. (EuCNC)*, Jun. 2019, pp. 438–442.

[77] A. Sadeghi, G. Wang, and G. B. Giannakis, "Deep reinforcement learning for adaptive caching in hierarchical content delivery networks," *IEEE Trans. Cognit. Commun. Netw.*, vol. 5, no. 4, pp. 1024–1033, Dec. 2019.

[78] Y. He, F. R. Yu, N. Zhao, V. C. M. Leung, and H. Yin, "Software-defined networks with mobile edge computing and caching for smart cities: A big data deep reinforcement learning approach," *IEEE Commun. Mag.*, vol. 55, no. 12, pp. 31–37, Dec. 2017.

[79] J. Wang, L. Zhao, J. Liu, and N. Kato, "Smart resource allocation for mobile edge computing: A deep reinforcement learning approach," *IEEE Trans. Emerg. Topics Comput.*, early access, Mar. 4, 2019, doi: 10.1109/TETC.2019.2902661.

[80] A. Blenk, P. Kalmbach, J. Zerwas, M. Jarschel, S. Schmid, and W. Kellerer, "NeuroViNE: A neural preprocessor for your virtual network embedding algorithm," in *Proc. IEEE INFOCOM*, Apr. 2018, pp. 405–413.

[81] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.

[82] Ryu SDN Framework Community. *Ryu*. Accessed: Sep. 18, 2020. [Online]. Available: https://osrg.github.io/ryu/

[83] K. Phemius and M. Bouet, "Monitoring latency with OpenFlow," in *Proc. 9th Int. Conf. Netw. Service Manage. (CNSM)*, Oct. 2013, pp. 122–125.

[84] Y. Zhao, L. Iannone, and M. Riguidel, "On the performance of SDN controllers: A reality check," in *Proc. IEEE Conf. Netw. Function Virtualization Softw. Defined Netw. (NFV-SDN)*, Nov. 2015, pp. 79–85.

[85] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proc. ACM SIGCOMM Workshop Hot Topics Netw.*, 2010, pp. 19.1–19.6.

[86] M. Enachescu, Y. Ganjali, A. Goel, N. McKeown, and T. Roughgarden, "Part III: Routers with very small buffers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 3, pp. 83–90, 2005.

[87] A. Lakshmikantha, C. Beck, and R. Srikant, "Impact of file arrivals and departures on buffer sizing in core routers," *IEEE/ACM Trans. Netw.*, vol. 19, no. 2, pp. 347–358, Apr. 2011.

[88] J. Ye and K.-C. Leung, "Adaptive and stable delay control for combating bufferbloat: Theory and algorithms," *IEEE Syst. J.*, vol. 14, no. 1, pp. 1285–1296, Mar. 2020.

[89] D. Bertsekas, "Dynamic behavior of shortest path routing algorithms for communication networks," *IEEE Trans. Autom. Control*, vol. AC-27, no. 1, pp. 60–74, Feb. 1982.

[90] P. Narvaez, K.-Y. Siu, and H.-Y. Tzeng, "New dynamic algorithms for shortest path tree computation," *IEEE/ACM Trans. Netw.*, vol. 8, no. 6, pp. 734–746, Dec. 2000.

[91] Z. Wang and J. Crowcroft, "Analysis of shortest-path routing algorithms in a dynamic network environment," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 22, no. 2, pp. 63–71, Apr. 1992.

[92] M. Chiesa, G. Kindler, and M. Schapira, "Traffic engineering with equal-cost-multipath: An algorithmic perspective," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 779–792, Apr. 2017.

[93] C. E. Hopps. *Analysis of An Equal-Cost Multi-Path Algorithm*, document RFC 2992, 2000, pp. 1–8.

[94] S. Dharmapurikar, M. A. Attar, N. Yadav, R. Vaidyanathan, and K. C. Chu, "Weighted equal cost multipath routing," U.S. Patent 9 502 111, Nov. 22, 2016.

[95] S. Hemminger, "Network emulation with NetEm," in *Proc. Australia's 6th Nat. Linux Conf. (LCA, Linux.Conf.Au)*, M. Pool, Ed. Sydney, NSW, Australia: Linux Australia, Apr. 2005. [Online]. Available: http://developer.osdl.org/shemminger/netem/LCA2005_paper.pdf

JUSTUS RISCHKE received the Dipl.-Ing. degree in electrical engineering from Technische Universität Dresden (TU Dresden), Dresden, Germany, in 2017. He is currently pursuing the Ph.D. degree with the Deutsche Telekom Chair of Communication Networks. His research interests include network coding and reinforcement learning in software-defined networks (SDN) for low-latency communications.

PETER SOSSALLA received the Dipl.-Ing. degree in electrical engineering, in 2019. He is currently pursuing the Ph.D. degree with the Deutsche Telekom Chair of Communication. His current research interests include robotics, software-defined networking (SDN), and time-sensitive networking (TSN).

HANI SALAH received the B.Sc. degree in computer systems engineering from Palestine Polytechnic University, the M.Sc. degree in internetworking from the Royal Institute of Technology (KTH), Sweden, and the Ph.D. degree in computer science from TU Darmstadt, in 2015, specialized in measurements, performance, and security of large-scale distributed systems, under the supervision of Prof. Dr. T. Strufe. He is currently a Senior Researcher with the Deutsche Telekom Chair of Communication Networks, TU Dresden, Dresden, Germany. His current research interests include distributed systems and modern networking technologies (SDN, NFV, ICN), with the emphasis on their performance, security, and monitoring.

FRANK H. P. FITZEK (Senior Member, IEEE) received the Dipl.-Ing. degree in electrical engineering from the University of Technology–Rheinisch-Westfälische Technische Hochschule (RWTH), Aachen, Germany, in 1997, and the Ph.D. (Dr.-Ing.) in electrical engineering from the Technical University Berlin, Germany, in 2002. He is currently a Professor and the Head of the Deutsche Telekom Chair of Communication Networks, Technical University Dresden, Germany, coordinating the 5G Lab Germany. He is the Spokesman of the DFG Cluster of Excellence CeTI. In 2003, he joined Aalborg University as an Associate Professor and later became a Professor. In 2015, he was awarded the honorary degree Doctor Honoris Causa from the Budapest University of Technology and Economy (BUTE). His current research interests include wireless and mobile 5G communication networks, mobile phone programming, network coding, cross layer, energy efficient protocol design, and cooperative networking.

MARTIN REISSLEIN (Fellow, IEEE) received the Ph.D. degree in systems engineering from the University of Pennsylvania, in 1998. He is currently a Professor with the School of Electrical, Computer, and Energy Engineering, Arizona State University (ASU), Tempe, and an External Associated Investigator with the Centre for Tactile Internet with Human-in-the-Loop (CeTI), Technische Universität Dresden, Germany. He is currently an Associate Editor-in-Chief of the IEEE Communications Surveys and Tutorials, a Co-Editor-in-Chief of *Optical Switching and Networking*, and chaired the steering committee of the IEEE Transactions on Multimedia, from 2017 to 2019.