7-27-2022 11:30 AM

Developing Intelligent Routing Algorithm over SDN: Reusable Reinforcement Learning Approach

Wumian Wang, *The University of Western Ontario*

Supervisor: Anwar Haque, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science

Abstract

Traffic routing is vital for the proper functioning of the Internet. As users and network traffic increase, researchers try to develop adaptive and intelligent routing algorithms that can fulfill various QoS requirements. Reinforcement Learning (RL) based routing algorithms have shown better performance than traditional approaches. We developed a QoS-aware, reusable RL routing algorithm, RLSR-Routing over SDN. During the learning process, our algorithm ensures loop-free path exploration. While finding the path for one traffic demand (a source destination pair with certain amount of traffic), RLSR-Routing learns the overall network QoS status, which can be used to speed up algorithm convergence when finding the path for other traffic demands. By adapting Segment Routing, our algorithm can achieve flow-based, source packet routing, and reduce communications required between SDN controller and network plane. Our algorithm shows better performance in terms of load balancing than the traditional approaches. It also has faster convergence than the non-reusable RL approach when finding paths for multiple traffic demands.

Keywords

Reinforcement Learning (RL), Traffic Routing, Software Defined Networking (SDN), Segment Routing (SR), Quality of Service (QoS).

Summary for Lay Audience

In the past decades, the number of the Internet users and the type of services that rely on the Internet have increased dramatically. Traffic routing, the process of sending data from the source to its specified destination, is vital for the proper function of the Internet. By combing novel network architecture and techniques, researchers hope to develop routing algorithms that provide better performance than traditional approaches.

Based on Reinforcement Learning (RL) and Segment Routing (SR), we developed a routing protocol, RLSR-Routing, over Software Defined Networking (SDN). RLSR-Routing can self-explore the network and find a path for a given traffic demand based on user-defined optimality. Compared with previous work, our approach adopted some modifications to RL algorithm, such that the cost of finding the path is minimized. In addition, our approach can reuse previously learned knowledge about network status, when it is working on new traffic demands. In experiment settings, RLSR-Routing outperforms non-RL based routing algorithm in terms of load-balancing among network links. Compared with the RL approach that does not reuse previous learning results, RLSR-Routing shows faster convergence speed.

Acknowledgments

I would like to express my sincere gratitude to my supervisor Dr. Anwar Haque. I really appreciate the opportunity you offered to me. Your continuous support, encouragement, and guidance during my study allow me to face any challenges during the research. It is my great honor to be your student and a member of the WING lab team.

I would also like to thank Dr. Greg Sidebottom of Juniper Networks, who served as the industry advisor/collaborator for this research project. His valuable feedback, and assistance in accessing the essential resources such as industry tools and data have been a great help to the success of this research work.

I appreciate guidance from Dr. Ed Knorr, Dr. Alan Wagner, your recommendations encouraged me, a former student major in Microbiology and Immunology, to step forward in Computer Science.

I also want to say thank you to Bingyao Wang. You are the one who led me to the path of Computer Science, and I will never forget the time we spent at the UBC ICCS building, room X350. It is my fortune to be one of your friends.

At last, but not least, I sincerely thank my parents for their love and support. Without love, writing a thesis during a pandemic will be much harder.

Table of Contents

List of Tables

List of Figures

Chapter 1

1 Introduction

Over the past decade, the number of people who have access to the Internet has been steadily increasing. CISCO's report predicts that by 2023, there will be 5.3 billion Internet users, which is about two third of the World's population [1]. Routing, which is the process of determining and forwarding packets from their source to destination, is vital for the transmission of data and information between network of networks, a.k.a., the Internet [2]. Traditionally, traffic routing relies on protocols like OSPF, which aims to find the shortest path between source and destination [3], or Routing Information Protocol (RIP), which is a distance-vector based algorithm that mainly considers hop count [4]. OSPF is one of the most widely used Interior Gateway Protocol, and the shortest paths between source and destination pairs are calculated based on assigned weights to every link of a network [5]. To obtain good network performance instead of only considering number of hops along a routing path, the links weights should be optimized [5]. However, a previous study has proved that optimizing OSPF weights is a NP-complete problem [6].

To provide better network performance for increasing demands over a variety of applications, new network architectures and traffic routing methods have been proposed. Software Defined Networking (SDN) is an architecture that decouples the control plane and data plane of a router [7, 8, 9]. Traditional network routers have their own control and data plane [8], and in a protocol like OSPF, each router maintains its own forwarding table [3]. For SDN, the control plane is in a logically central controller which has a global view of the network; and the SDN controller enables better interaction with applications, Quality of Service (QoS) provisioning, network monitoring and so on [7, 8]. SDN can also be applied to the wireless network in addition to traditional Wide Area Network [9]. Segment Routing (SR) [10] is a source packet routing architecture in that each packet encodes a list of Segments in its header to guide the packet travel through the specified path [10, 11]. As stated in [12], SR's ability to control packet forwarding can provide better network programmability, fast reroute, load balancing etc. In addition to the new routing

architecture, researchers also try to apply Reinforcement Learning (RL) in traffic routing [13]. One recent implementation of RL in SDN can also be found in [14].

1.1 Motivation and Objectives

Reinforcement Learning has been studied to develop adaptive, intelligent routing algorithms over the past three decades [13]. It has shown extraordinary flexibility to combine with other techniques for traffic routing over a variety of network types, with different QoS requirements [13, 15]. Compared with traditional routing methods which are based on assumptions about network conditions, applying RL allows routing algorithms to automatically learn the dynamic of the network, such as traffic flows, link quality and so on [15]. Therefore, based on RL's self-learning ability, some of the previous research's objectives are to improve service quality to end users, while optimizing network resources [15]. However, many previously proposed algorithms do not fully exploit the global view of SDN controller, or the ability to control the whole forwarding path (i.e., flow-based routing) by combining with technique like SR [15]. In addition, many previous works only focus on one traffic demand at a time, and some require pre-defined algorithms (like OSPF) or initial paths as input [15].

Our objective is to develop a RL-based routing algorithm for SDN, such that it can address previous work's limitations. Our algorithm should be light-weighted, efficient, and allow users to customize their QoS requirements for definition of users' preferred routing paths. While the algorithm aiming to minimize costs during the path finding process, the final routing path from the algorithm should be as good as possible in terms of satisfying user defined QoS requirements.

1.2 Contribution

We developed a RL-based, SR-based, QoS-aware routing algorithm for SDN, RLSR-Routing. The algorithm considers various QoS factors, which can be customized by users, to find a user preferred path for traffic demands. The main contributions of this thesis are listed below:

- We modified State-Action-Reward-State-Action (SARSA), an on-policy RL method [13], so that our routing algorithm aggregates action selections first. Such modification can further reduce the number of message exchanges (e.g., sending packets, receiving QoS info) required between SDN controller and network switches.

- In addition, during the learning process, our algorithm ensures that no packet will be stuck in a loop, which means that packets either reach the destination or stop being forwarded if all next-hop nodes have been visited by themselves. In other words, a packet will be sent to any node in the network at most once.

- We divided an action's reward into local and global rewards. Local rewards are used for finding a path for one traffic demand, whereas global rewards are purely based on network status like link utilizations. Global rewards can be used to initialize local Q-table to speed up algorithm convergence.

- RLSR-Routing does not require prior knowledge of the network. In addition, it does not require pre-defined paths or initial link weights as a starting point. The algorithm can self-learn network status and assign a list of traffic demands over the network in sequence.

We compared our RLSR-Routing with a routing algorithm currently used by one of the major telecom solutions providers. A mesh topology network is setup with randomly generated traffic to be placed over the network. We compared the quality of selected paths for generated traffic, in terms of maximum link utilization. Our experiment result shows that the proposed RLSR-Routing approach contributes to minimizing link utilization in the network. In addition, when tested on a sequence of traffic demands, the proposed RLSR-Routing approach speeds up the algorithm convergence by applying knowledge of the network from its previous learning processes.

1.3 Thesis Structure

We divide our thesis into the following chapters. Chapter 2 presents a review of related concepts, such as RL and DRL. In chapter 3, we define our problem, network environment, and introduce previous researchers that use RL approach to solve traffic routing problems. Chapter 4 presents our methodology, including overall architecture, pseudocode of RLSR-Routing, and rationale behind our algorithm designs. In chapter 5, we summarize our experiment results, including a comparative study with a non-RL routing algorithm. In chapter 6, we conclude our study and propose potential updates and objectives for future work.

Chapter 2

2 Review of Related Concepts

An intelligent, adaptive routing algorithm can bring better performance in terms of QoS. It can also be designed to work under specific network conditions or exploit certain network architecture characteristics. In this chapter, we present a brief review about legacy routing schemes and the Software Defined Networking (SDN) architecture and Segment Routing (SR). We also discuss Reinforcement Learning (RL) vs Deep Reinforcement Learning (DRL), and some early work of RL-based routing.

2.1 Legacy Routing Algorithms

In general, a network router that resides on the network layer consists of a control plane and data plane: the data plane stores a forwarding table and uses it to decide next-hop router of a packet; the control plane executes routing algorithms to update forwarding table [2]. In this section, we mainly focus on two Interior Gateway Protocol (IGP): OSPF and RIP.

2.1.1 Routing Information Protocol (RIP)

RIP is specified in [4], it is a distance-vector (DV) algorithm which also referred as the Bellman-Ford algorithm. RIP is a distributed algorithm that each router runs in an asynchronous manner: routers exchange routing information, and an optimal path is calculated based on the cost for sending the packet to the next-hop neighbor and distance towards the destination [2]. In terms of RIP implementation, the distance between routers can be expressed as a number of hops, such that an optimal path is the one with the lowest hop-count from source to destination [4].

2.1.2 Open Shortest Path First (OSPF)

The second version of OSPF is defined in [3], similar to RIP, each router inside a network runs OSPF in a distributed manner. Each router maintains a database that provides a complete view of the network topology [3]. OSPF uses the Dijkstra algorithm to construct a shortest-path tree, with the router running the algorithm as root and other network routers as child nodes. Unlike RIP that is based on distance or hop-count, OSPF’s shortest path is

based on link weights [3] which can be manually assigned by a network administer [2]. When multiple paths have same lowest weights for one source-destination pair, traffic will be equally split among these paths [3]. In practice, setting different link weights will alter routing paths and thus affect network performance [5]. However, optimization of OSPF weights is proven to be NP-complete [6].

2.2 Software Defined Networking (SDN)

For traditional network architecture, both the control and data planes are deployed within every network device. SDN architecture decouples the control plane and data plane, by placing the control plane on a logically centralized SDN controller [5, 7, 8]. The logically centralized controller has a global view of the network, and it may be viewed externally as a single controller but consists of one or more physical devices [7]. Communications between network devices and controllers are through the controller-data plane interface, such as OpenFlow protocol [7], so that controllers can access link-state information [2, 7]. In addition, the control plane provides an interface (such as REST API) for interacting with the application plane, which consists of multiple network applications [7, 8]. Based on interaction with the control plane, network applications provide a variety of support, such as routing, access control, and load balance, to utilize network performance [2, 7]. Compared with traditional architecture, SDN provides better programmability and simplified network management via a logically centralized control plane [7, 8]. As a result, SDN supports flow-based routing with the potential of QoS provisioning and network monitoring [7]. SDN architecture can be combined with OSPF, and the previous study has shown better performance for SDN/OSPF hybrid networks [5]. In addition, SDN architecture can not only be deployed to traditional wired network, but also other network types such as wireless network [9].

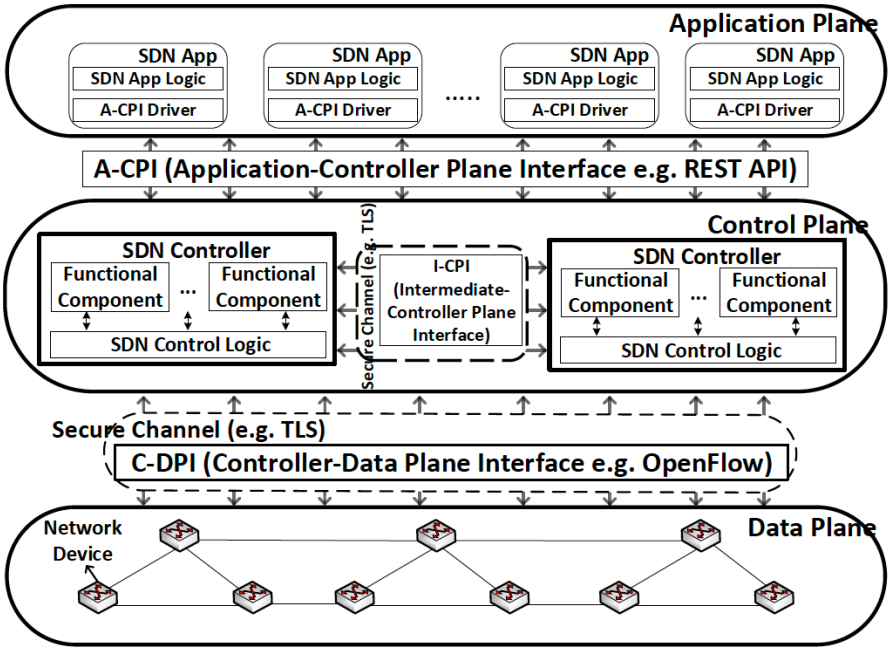


Figure 2.1: SDN architecture. Cited from [7]

2.3 Segment Routing (SR)

As standardized in RFC 8402, Segment Routing (SR) is a routing technique that leverages the source routing paradigm [10]. In this section, we first review the specifications of source packet routing, then we briefly discuss SR architecture.

2.3.1 Source Packet Routing

Traditionally, packet routing is done through a series of IP lookup through a bunch of routers from the source to the destination of a packet: when each router receives the packet, it examines the packet's destination IP address, and forwards the packet to the next-hop routers based on its forwarding table. Even with SDN architecture, routing is still determined by local forwarding tables among network devices. In contrast, the source packet routing technique allows the implementation of Traffic Engineering, which enables control of specific traffic flow within the network. For example, the source node of a traffic flow can alter paths for some packets instead of their shortest paths [12].

2.3.2 Segment Routing Architecture

In a network that deploys SR, the set of nodes that are participated in source-based routing forms an SR domain [10]. By definition, a segment can be viewed as an instruction for a node to execute on the incoming packet, such as steering the packet through the given path, or delivering the packet to a specified service [11, 10]. Each segment has its identifier as Segment ID (SID), and based on participation over different SR domains, a node can have one global segment and multiple local segments [10]. When combined with IGP, a segment can represent an IGP prefix as IGP-prefix segment, or an anycast group as IGP-anycast segment that is advertised by a set of nodes. A segment can also be assigned to a specific router as a node segment. When executing a node segment, it usually means forwarding the packet to the node identified by the node SID using the shortest path [11].

For packets that apply SR, a list of segments is encoded in their packets head. In general, SR can adopt MPLS which use label stack to encode segments. SR can also apply IPv6, such that each segment is expressed as an IPv6 address [11]. By modifying a number of segments placed in a packet's header, SR can achieve strict traffic engineering that each hop along the path is specified; or a loose option that only a few hops are specified, as described in source packet routing [12].

2.4 Reinforcement Learning (RL) & Deep Reinforcement Learning (DRL)

Reinforcement Learning is a type of machine learning that generally does not require using labeled or unlabeled datasets to create models that make prediction or classification [13]. The concept of reinforcement is adapted from study animal behaviors in which they respond to stimuli from surroundings [16]. In general, a RL algorithm consists of the following main components: an agent, which executes a RL algorithm, explores the environment in a trial-and-error manner; an environment, consists of a set of states; the agent can take an action at each step, which leads the agent to a new state; after taking an action, the reward is sent back from the environment to the agent, to evaluate the quality of action selection policy [15, 16, 17]. Since 1990s, dozens of published papers have

applied RL in traffic routing. They address different QoS considerations over a variety of network topology, and RL can be directly or indirectly used in the path finding process [13, 15]. Based on survey paper [15], the two most used RL algorithms are Q-learning and State-Action-Reward-State-Action (SARSA). Both algorithms have similar workflow: an agent selects an action from Q-table, based on the current state and defined action selection policy (e.g., greedy), performs an action, observes reward, and updates Q-table for previously involved state-action pair [13, 15]. In Q-learning, updating an action's Q-value depends on the maximum reward (Q-value) the agent can get in the next state; whereas in SARSA, such update depends on the action performed in the next state [13].

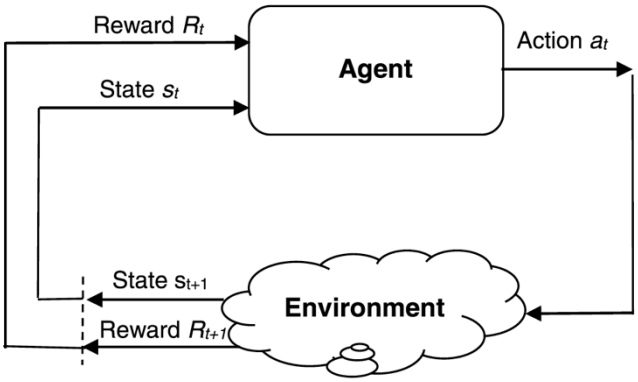


Figure 2.2: RL workflow, from [15]

2.4.1 Deep Reinforcement Learning (DRL)

Deep Reinforcement Learning (DRL) is the combination of Neural Network and RL to improve the scalability of RL algorithms to solve complex problems [16, 15]. For RL algorithms like Q-learning and SARSA, action selection usually involves looking up an action in a table, that stores every possible combination of state-action pairs' estimated cumulative reward [13]. In policy-based DRL, a Neural Network replaces Q-table: when the agent inputs the current state, the network outputs an action [16]. In value-based DRL, both state and action are sent into the network, which outputs an approximated Q-value of the state-action pair [16].

2.4.2 RL Based Routing: Early Work

One of the earliest works was Q-routing proposed by Boyan and Littman in 1993 [18], which is a direct application of Q-learning. In Q-routing, every network node is a learning agent and state x is a packet currently held by node x. An action A(d, y) is sending a packet to the next-hop node y to reach the packet's destination d. Action's reward is measured by the estimated latency for a packet to reach the destination through node y, which is sent back to the previous packet-holding node x from y [18]. The authors compared Q-routing with the shortest paths-based algorithm, and the result showed that on a 6*6 grid topology with high traffic load, Q-routing achieved a lower average delivery time due to its ability to detour some traffic to longer paths to mitigate congestions [18].

Since the publication of Q-routing, several research efforts were aimed to improve the performance of Q-learning based routing. In 1996, Samuel et al. proposed a memory-based RL routing algorithm, predictive Q-routing (PQ-routing), as an extension of original Q-routing [19]. PQ-routing also uses latency as the main QoS parameter to calculate actions' rewards. Under low traffic load, PQ-routing performs like the shortest paths algorithm. Under high traffic load, PQ-routing's learning agent periodically explores previously congested paths [19]. Shailesh and Risto developed a Dual Reinforcement Q-routing algorithm in 1997 with both forward and backward exploration [20]. In addition to the packet sender, the receiver of a packet also updates its Q-table based on feedback from the source [20]. Both [19, 20] papers' algorithms showed better performance than Q-learning under experiment conditions. A Modified Q-routing algorithm introduced in [21] also has distributed learning agents, and each agent exchanges an immediate reward after a packet is sent through a link. The notable feature of this algorithm also includes using link cost-based reward, and a forgetting factor that gradually devalues Q-table to improve performance [21].

In addition to the traditional wired network, some research focused on RL-based routing in other network types. Both [22, 23] applied RL approach on mobile Ad-hoc networks (MANETs). Both papers address the dynamic feature of MANETs and describe how RL-based algorithm handles node joining or leaving the network [22, 23]. For example, [23]'s adaptation of Q-routing encourages a node to send packets to its neighbours that are newly

discovered [23]. Ping and Ting presented an adaptive routing algorithm, AdaR, over a Wireless Sensor Network (WSN) [24]. Three notable features of AdaR are: 1) It is one of the earliest works that consider multiple QoS factors, such as node's residual energy and link reliability for an action's reward [15, 24]. 2) It uses LSPI (Least Squares Policy Iteration), which has a faster convergence speed and does not require tuning of initial parameters such as learning rate [24]. 3) The base station, instead of every node of the network, is the learning agent.

2.5 Quality of Service (QoS)

As mentioned in [25], Quality of Service (QoS) can be specific to applications or users; it can also refer to the overall, end-to-end QoS of the network. Some QoS parameters that are commonly used are delay, packet loss, jitter, throughput, and available bandwidth (link utilization) [25, 26, 27]. These parameters are often correlated with the others. For example, high jitter causes delay variance; such variation would affect queuing delay as routers may receive higher traffic in a certain time period and decrease overall throughput [26]. Higher link utilization means more available bandwidth is consumed by traffic flow; thus, it is an important performance measurement in flow-based network [27].

In addition, some networks may have special QoS requirements due to the physical deployment of the network or overall network architecture. For example, in Wireless Sensor Network, each sensor node may be powered by a battery; therefore, energy conservation is important to enabling proper function of WSN under resource-constraint conditions [25].

Chapter 3

3 Problem Statement and Related Work

In this chapter, we formally define our research problem and present a literature review of recent work related to our problem and existing solutions. This chapter also highlights current research gap and sets transitions to introducing our proposed methodology.

3.1 Problem Statement

We focused on traffic routing in SDN; therefore, we define our network as a set of SDN switches with a SDN controller that has a global view of the network, such that a controller knows the topology of the network, including every node and link. The controller has dedicated links to communicate with network nodes, but neither the controller nor these links are used for routing packets between the network's end users. Our routing algorithm interacts with the SDN controller to perform path calculation and installs defined routing paths into the network. From the routing algorithm's point of view, a network can be abstracted as a directed graph, *G (V, E)*. *V* is the set of nodes that each corresponds to a physical device (SDN switch), and *E* is the set of links that connects nodes. $l_{i,j}$ denotes the unidirectional link from node *i* to *j* where $i,j \in V$. We define a traffic demand as a flow of data transmitted from a source node to a destination node. One source and destination pair can have multiple traffic demands, each with a specified amount of data as demand traffic (expressed in bits/second). However, each traffic demand can only be assigned one routing path, which is defined as a list of nodes that connects from source to destination.

Each link and node of a network has maximum capacity. For a link, we use the term "maximum bandwidth" (in bits/second) to represent maximum traffic that can travel through the link, and "used bandwidth" (in bits/second) to represent the current amount of data on the link. A node's processing rate (in bits/second) is the maximum amount of incoming data it can handle. In addition, all links have a field called "link reliability", which varies from 0% to 100%. The higher the link reliability, the lower chances of link failure will occur.

Based on the above settings, our research objective is to give a network with SDN architecture with all nodes supporting SR and develop a RL-based routing algorithm that can find user preferred paths for traffic demands. Our definition of ‘user preferred’ paths are those paths found by our routing algorithm based on weights of one or more QoS parameters, which are adjusted by our algorithm’s users.

The following assumptions are made:

- Communications between SDN controllers and network switches are through dedicated links, in a reliable transfer manner.
- SDN controllers are not involved in routing packets that originated from the network’s end users. For better visibility, sometimes we do not draw the controllers and those dedicated links on network topology diagrams in this thesis.
- All SDN switches of the networks considered in this thesis support SR. i.e., they can handle packet’s SR header and execute segments properly.
- All nodes are reliable, and if total incoming traffic does not exceed a node’s processing rate; no packets will be dropped by nodes.
- For each pair of adjacent nodes *i-j* that connects with each other, at most two links exist: link *i->j* and link *j->i*, that directly connect *i* and *j*.
- If a link’s used bandwidth does not exceed its maximum bandwidth, and link failure does not happen, no packet will be lost when traveling on that link.

3.2 Related Work

3.2.1 RL on Non-SDN Types of Networks

Similar to [24], several recent research focused on RL approach for routing in WSNs [28, 29, 30]. One character of WSNs is network devices are often powered by battery, therefore, energy consumption or energy conservation is an important factor to be optimized when designing RL-based routing algorithm [28, 29, 30]. Both [28, 29]’s work involves exchanging information related to energy conservation, which is accomplished by adding extra fields in data packets. In [28], the sender node’s current Q-value is sent to a packet’s next hop node, along with information about previous nodes energy level. In [30], the

authors combine Q-learning and transmission gradient for optimizing energy consumption over the WSNs. For transmission gradient, each node maintains an estimated number of transmissions for sending a packet from the node to the base station [30]. Estimating the number of transmissions and energy level are used for calculating rewards; and nodes use status packets to send their Q-value and estimated number of transmissions to other nodes [30].

Another type of network which many studies focused on is Ad-hoc Networks (ANETs). In general, ANETs do not have a centralized controller, nodes are wirelessly connected, and network topology is dynamic [15]. Mobile Ad-hoc Networks (ANETs) is a subtype of ANETs that nodes have mobility; common types of MANETs include VANETs for vehicles moving in certain area like a city [15]. In this review, all 6 recent papers which focus on ANETs used packet loss/retransmission rate/delivery ratio as one of metric for performance evaluation [31, 32, 33, 34, 35, 36]. Unlike early work that is only based on RL, many of these ANETs related papers combine RL with other techniques or have additional steps in addition to RL algorithm. Authors in [31] proposed QLMAODV, a combination of Q-learning and AODV protocol. AODV is defined in RFC3561, it is a routing protocol for MANETs, and one notable feature is that AODV ensures loop free routing [37]. QLMAODV considers stability when Q-learning evaluates a path to achieve a stable route [31]. In [32], the authors proposed PFQ-AODV, an extension of their previous work in [38]. PFQ-AODV also combines AODV with Q-learning. In addition, it considers bandwidth, node's movement, and link quality in route selection by fuzzy logic [32]. Compared with their previous work, PFQ-AODV can use neighbor information, in addition to position information, to calculate vehicle movements [32]. QGeo proposed in [33] also considers nodes mobility: each node exchanges their location information periodically through HEELO packets [33].

Authors in [34] proposed a routing algorithm based on fuzzy logic, game theory and RL. Fuzzy logic groups vehicles (as nodes) into clusters and selects cluster heads, and game theory coordinates commutations between nodes and their cluster heads [34]. Nodes are enforced to use cluster heads for multi-hop transmission and RL is used for path evaluation [34]. Similarly, Q2-R proposed in [35] has additional bootstrapping steps before RL

routing. Q-learning starts with initial paths that discovered from bootstrapping steps, and agent gradually improves paths' Q-value during learning process [35]. A hierarchical routing scheme is proposed in [36], which first divides geological area into grids without the selection of cluster heads. Q-learning is used for selecting next grid and can be used to select the next hop (vehicle) inside the optimal grid [36].

[39, 40] focused on Wireless Mesh Networks (WMNs), in which nodes with multiple interfaces serve as a gateway that connects other nodes to the Internet [15, 40]. In [39]'s framework, each node predefined a set of routing algorithms, and Q-learning is used for adaptively selecting routing algorithms to execute. In [40], gateway nodes periodically broadcast their traffic load, each node first selects the gateway router, then uses RL to select next-hop node to send packets to the selected gateway [40]. Authors in [41] focused on multi-hop wireless networks, particularly for video streaming. The proposed RLOR algorithm combines RL and Opportunistic Routing, which exploit the broadcast nature of the wireless network and determines packets' next-hop on the fly [41].

3.2.2 RL on SDN

As illustrated in [14], when acting as an RL agent, SDN controller(s) can find optimal path from source to destination, in addition to next-hop node. To address the scalability issue of a single controller, the authors of [42] proposed QAR, a QoS-aware routing algorithm on hierarchical SDN. Nodes are divided into subnets, and each subnet has a domain controller, assisted by one or more "slave" controllers [42]. When a routing request's source and destination are at different subnets, the main controller with a global view calculates a subnet path that connects the source and destination's subnets [42]. Then each subnet along the path runs RL on their domain controller to travel packets within subnets, and such path finding can be done in parallel [42]. Another notable feature of QAR is that it uses SARSA, an on-policy RL algorithm that compared with Q-learning, updates of a state-action pair's Q-value depends on action performed on the next state [42]. [43, 44] is also based on SARSA, but on a non-hierarchical SDN. VS-routing in [43] applies a variable ε-greedy for action selection, such that the probability of randomly selecting an action instead of a greedy approach varies with hop count and a dynamic factor [43]. In addition, VS-routing has no immediate rewards, Q-value updates are based on the next state's associated Q-

values and ε [43]. A multiple controllers SDN is considered in [44], and its objective is load-balancing for communication between nodes and controllers. [44] also improved ε-greedy action selection, by using the Bayesian method: during exploration, agent tends to select frequently selected actions [44].

For Q-learning based routing algorithms in SDN, we also observed diversity in terms of network architecture and algorithms' workflows. Authors of [45] focused on congestion control in SDN. Learning is not based on selected actions and immediate rewards but update all edges reward in each episode of the routing algorithm [45]. Congestion identification is based on the current bandwidth of potential bandwidth, same for updating each edge's corresponding reward [45]. In [46], Q-learning based routing is used for load-balancing in Wireless SDN. In general, a Wireless SDN still has a central controller and set of SDN switches, in addition, SDN base stations are used to connect with SDN mobile devices [46]. States are assigned to users and depend on whether users' demands are satisfied, actions are connecting every user's traffic flow to a base station, and rewards are calculated based on available resource and a fairness function [46]. QR-SDN in [47] also adapts flow-based routing. A flow is defined as data transmission between a given source and destination pair, and each flow is assigned one path for routing traffic [47]. QR-SDN will not compute the path for a given flow, instead, it selects a path from a set of pre-calculated possible paths. During the learning process, QR-SDN can take a list of flows, change path assignment for one or more flows in each episode, and gradually learns optimal path assignments [47].

[48, 49] use Q-learning, but on a "distributed" SDN. In [48], the authors implemented a "distributed" SDN over several computers (PC), with each PC containing a part of SDN network with one controller. Links are initially assigned weights, and the objective of Q-learning is to optimize link weights assignments [48]. SDCIV proposed in [49] aims to achieve adaptive routing in SDN based Internet of Vehicles (IoV). The logical central controller is made up with a SDN controller cluster, and thus we classify it as a "distributed" operation mode [49]. SDCIV defines states as average vehicle speeds and densities, and the learning agent gradually learns the best routing protocol to execute for

different states [49]. Similar to [39], RL is not used for routing or path finding directly, but to find an appropriate protocol to execute from pre-defined routing algorithms [49].

3.2.3 DRL based Routing

Using RL in traffic routing started in 1990s, in contrast, DRL based approach is a relatively emerging field. Figure 3 shows a number of published papers related to DRL and routing between 2017 and 2021, from Web of Science database. Since this literature review focuses on traffic routing, we do not cover how neural network is trained in different DRL based routing algorithms.

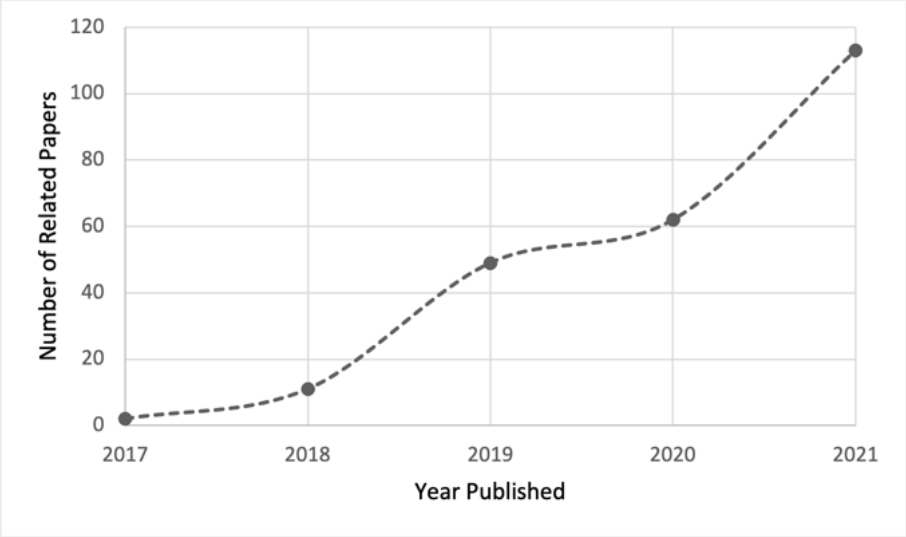


Figure 3.1: Papers related to DRL and Routing between 2017-2021

In [50], the authors proposed DROM, a routing optimization algorithm for SDN. DROM takes a traffic matrix of current network load as input, during the learning process, the DRL agent changes some links' weight so that some traffics alters their routing path [50]. WA-SRTE in [51] works on a partially deployed Segment Routing IPv6 network, and it has two phases. In offline network design phases, WA-SRTE uses DRL to optimize OSPF link weights and SR-enabling nodes' deployment based on historical traffic data. Once the offline phase is complete, link weights and SR nodes deployment is fixed [51]. In the online routing optimization phase, WA-SRTE uses other techniques like linear programming to optimize routing paths [51]. ENERO proposed in [52] also combines Segment Routing and

its objective is also to minimize maximum link utilization. Taking initial OSPF weights and a set of traffic demands as input, ENERO uses DRL to assign each traffic demand to an intermediate SR node to optimize links utilizations [52]. To further improve the result, local search (LS) is applied to DRL's output in ENERO [52].

In [53], the authors applied Deep Q-learning for routing in data center SDN. The authors classified different traffic flows into two types: the mice-flows and the elephant-flows based on the amount of data and duration of traffic. [53] thus builds two Deep Q-networks to assign paths for two traffic flows, respectively, and each flow type has a different QoS objective. [54] proposed a hierarchical deep double Q-routing algorithm, which groups nodes into different clusters at different hierarchal levels, each cluster with a group leader. During the recursive route-finding process, the source's routing request is sent to the highest level's cluster leader, such that the source and destination nodes are at two different sub-clusters [54]. The cluster leader uses DRL to select one link that connects two sub-clusters, and the process is repeated until a whole path is built [54]. The double Q-learning technique is adapted from [55]; when combined with deep learning, two Deep Q-networks are interchangeably used for action selection and evolution [54].

3.2.4 Analysis and Current Research Gap

Most of our reviewed papers proposed an RL-based routing algorithm (21 of 26), although the DRL approach has attracted researchers' interest dramatically in the past 5 years. Among studies that focus on non-SDN networks, 85.7% (12 out of 14) are on kind of wireless network and/or Ad-hoc network. In addition to latency-related factors (like delay or hop-count), RL-based approaches can consider a variety of factors that affect network performance. However, when working on a non-SDN network like WSNs or ANETs, applying RL-based routing may generate additional commutations overhead. For example, [28-30] routing algorithms require exchanging energy consumption or device residual energy information among network nodes. With SDN controllers' global view of their networks, or a hierarchical network with some nodes being cluster heads (like [34, 54]) could reduce commutations overhead placed by RL-based routing over a network.

61.5% (16 out of 26) of reviewed papers proposed a composite routing protocol, where RL or DRL is part of the whole routing protocol. Common reasons of combining RL with other techniques are: 1. To speed up the algorithm's convergence (like QAR in [42]); 2. To provide additional control, such as loop free routing [31, 32] or source packet routing [52, 51]. It should be noticed that for some studies on SDN, like QAR, their routing algorithm does not guarantee loop-free routing during the learning phase, even though SDN controller(s) are RL agents [42]. The usage of RL is not limited to finding the best next-hop node to forward a packet. RL agents can learn the whole path from a source to a destination or assign a path to traffic flows from a set of possible paths. In addition, several studies focus on OSPF optimization by using RL approach, which we define as the indirect usage of RL in traffic routing. Most papers' framework only focuses on forwarding one-packet, or one source-destination pair's traffic. None of our reviewed literature, which is based on RL over SDN, use RL to directly find the path for a set of traffic flows in parallel.

In summary, RL has been studied to develop adaptive, intelligent routing algorithms over the past three decades. It has shown extradentary flexibility to combine with other techniques for traffic routing over a variety of network types, with different QoS requirements. However, currently proposed algorithms do not fully exploit the global view of the SDN controller, or the ability to control the whole forwarding path (i.e., flow-based routing) by combining with a technique like SR. Additional work needs to be done in order to develop a RL-based, QoS aware flow-based routing over SDN, such that RL agent can find path for multiple traffic demands in parallel. Alternatively, while RL agent still focuses on one traffic demand at a time, the agent can reuse the network status knowledge it has learned previously to speed up algorithm convergence.

Chapter 4

4 Methodology

As the number of Internet users and types of services that relies on networking increases, traditional distance vector or shortest path based algorithms may not be able to provide optimal network performance. Previous studies demonstrate the potential of using RL or DRL to develop an intelligent routing algorithm that is adaptive to a dynamic network topology and fulfills a variety of QoS requirements. To the best of our knowledge, existing approaches do not fully exploit SDN architecture combined with flow-based source packet routing paradigm. In this study, we propose RLSR-Routing, a RL-based routing algorithm over SR enabled SDN architecture to address the current research gap. RLSR-Routing reduces network operation costs during path finding process and can have faster convergence than the previous RL-based approach. This chapter described the RLSR-Routing framework and the rationale behind RLSR-Routing's design.

4.1 RLSR-Routing Architecture

Figure 4.1 presents an abstract view of our approach's framework. RL routing algorithm is one component of the SDN controller, and it relies on the SDN control plane to provide network topology and link state information. Although different literature may categorize routing as an application plane or control plane's function [2, 7], such difference does not affect RLSR-Routing's framework. Because in either case RLSR-Routing does not directly interact with network devices. Figure 4.2 describes RLSR-Routing's components in greater detail. These components can be classified into three categories. 1) Storage of information: such as Global Q-table which represents previously learned network status, Default parameters and Network topology. 2) RL algorithm: this is the central part of RLSR-Routing, it finds paths for specified traffic demands, as well as learns network status during execution. 3) Communication: these components interact with SDN controller to help other components send instructions to SDN controller, and to collect network information like QoS data, and topology information from SDN controller.

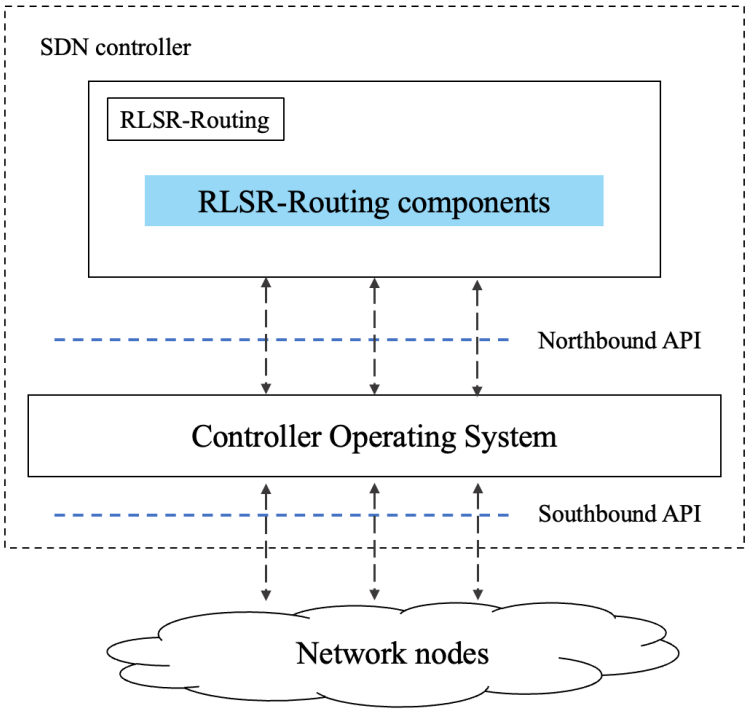


Figure 4.1: RLSR-Routing Architecture

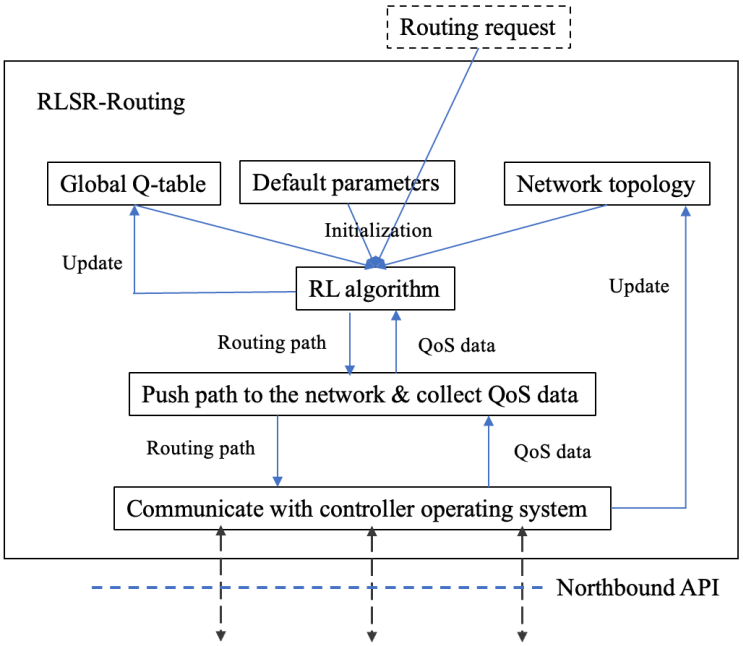


Figure 4.2: RLSR-Routing Components

4.2 RL: State, Action, Reward, and Modifications

The RL algorithm used in RLSR-Routing can be viewed as an extension to previous work, particularly QAR from [42]. RLSR-Routing applies modified SARSA, an on-policy RL algorithm and a complete Markov Decision Process is included in RLSR-Routing. In this chapter, we define the state space, the action space, and rewards' QoS considerations for RL algorithm used in RLSR-Routing. We also explain the rationale for two major modifications we made in our RL algorithms.

4.2.1 State and Action Space

RLSR-Routing focuses on routing for one traffic demand at a time: during each episode of algorithm execution, RLSR-Routing instructs the SDN controller to send one packet from the traffic demand's source to its destination. Based on the above settings, our definition of state space and action space is as follows.

- State space: let S denotes the set of all possible states of a network. $|S| = |V|$ where V is set of nodes of the network. A state $s \in S$ means node s is holding the packet. Notations s_t and s_{t+1} are also used, which represent the state at time t and time $t+1$, respectively.
- Action space: let A denotes the set of all possible actions that can be performed over a network. $|A| = |E|$ where E is set of edges of the network. An action $a_{i,j} \in A$ represents sending the packet along link $a_{i,j}$, from its source node i to its destination node j. Notations a_t and a_{t+1} are also used, which represent selected action at time t and time $t+1$, respectively.

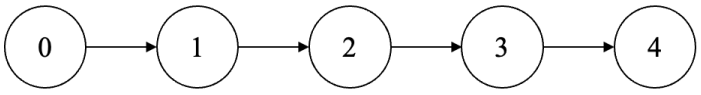


Figure 4.3: Trivial Topology 1 (T1)

Figure 4.3 shows a trivial topology T1 with 5 nodes, and 5 unidirectional links (followed the arrows). SDN controller and its connections with nodes are not shown in the figure. If a packet is sent from node *0* to node *5*, at time t_0, node *0* holds packet, the chosen action is

$l_{0,1}$. At time t_1, the packet reaches node *1*, and therefore state at t_1 is node 1 holding the packet.

4.2.2 QoS Considerations for Actions' Rewards

Previous studies have demonstrated the flexibility of RL-based routing that multiple QoS metrics can be used to calculate an action's reward. In RLSR-Routing, we considered five QoS parameters which can be categorized into three groups: latency; reliability; and load balancing. Assuming the performed action is sending one packet from node *i* to node *j* through link *l*, calculation of each QoS parameter's reward is presented below.

For latency related QoS metrics, we considered 1) number of hops, 2) transmission rate.

Number of hops related reward (R_{hop}) is calculated as follow:

$$R_{hop} = {}^{1}/_{number\ of\ hops} \tag{4.1}$$

Transmission rate related reward ($R_{transmission}$) reflects transmission delay of a node, and it is calculated as follows. The higher a node's processing rate, the less time it requires to transmit all bits of a packet to the link. Therefore, the related action will receive higher reward.

$$R_{transmission} = \frac{2}{\pi} \times \arctan(node\ i\ processing\ rate) \tag{4.2}$$

For reliability related QoS metric, we considered 1) link reliability. Since, in our assumption, all network switches and SDN controllers are 100% reliable, only the possibility of link failure is considered. Calculation of link reliability related reward ($R_{reliability}$) is:

$$R_{reliability} = link\ l's\ reliability\ value \tag{4.3}$$

For load balance related QoS metrics, we considered 1) traffic intensity, 2) link utilization. For these two metrics, both current and estimated (i.e., estimated traffic intensity and link utilization after placing the traffic demand's flow on the link) values are considered. The

calculation of current traffic intensity (R_{inten}) and estimated traffic intensity ($R_{inten\text{-}est}$) related reward are shown blow:

$$R_{inten} = 1 - \frac{Node\ j\ current\ total\ incoming\ traffic}{Node\ j\ processing\ rate} \quad (4.4)$$

$$R_{inten\text{-}est} = 1 - \frac{Node\ j\ estimated\ total\ incoming\ traffic}{Node\ j\ processing\ rate} \quad (4.5)$$

Calculation of current link utilization (R_{util}) and estimated link utilization ($R_{util\text{-}est}$) related reward are shown below:

$$R_{util} = 1 - \frac{Link\ l\ current\ used\ bandwidth}{Link\ l\ maximum\ bandwidth} \quad (4.6)$$

$$R_{util\text{-}est} = 1 - \frac{Link\ l\ estimated\ used\ bandwidth}{Link\ l\ maximum\ bandwidth} \quad (4.7)$$

Equations (4.1) – (4.7) ensure that all QoS related rewards values are no greater than 1. The closer a reward's value is towards 1, the better quality of the performed action is in terms of the reward. Take T1 as an example; suppose a packet is sent from node *0* to *4* follow the path *0->1->2->3->4*. All links have 10Mb/s maximum bandwidth, 95% of time are working; all nodes have 50Mb/s processing rate. Currently, there is 5Mb/s traffic sending from node *3* to node *4*, and the traffic demand from node *0* to *4* has 0.5Mb/s estimated traffic. If we evaluate the action of sending packet from node *3* to *4*, the corresponding rewards are:

$R_{hop} = {}^{1}/_{4} = 0.25$ (Since link $l_{3,4}$ is the fourth hop the packet has travelled).

$R_{transmission} = \frac{2}{\pi} \times \arctan(50Mb/s) = 0.9873$ (Round to 4 decimals).

$R_{reliability} = 95\% = 0.95$.

$R_{inten} = 1 - \frac{5\ Mb/s}{50\ Mb/s} = 0.9$ (Currently there is 5Mb/s traffic towards node *4*).

$R_{inten\text{-}est} = 1 - \frac{5+0.5\ Mb/s}{50\ Mb/s} = 0.89$ (After placed demand's traffic, node *4* are expected to have 5.5 Mb/s total incoming traffic).

$R_{util} = 1 - \frac{5\ Mb/s}{10\ Mb/s} = 0.5.$

$R_{util\text{-}est} = 1 - \frac{5+0.5\ Mb/s}{10\ Mb/s} = 0.45.$

4.2.3 RL Modification: Aggregate Action Selection

This section describes and explains one of the major modifications we made for SARSA used in RLSR-Routing – aggregate action selection of one episode. Based on our literature review, the two most commonly used RL algorithms in traffic routing are Q-learning and SARSA. Both Q-learning and SARSA follow a similar workflow during each episode of algorithm execution.

1. At time *t*, from current state s_t, select an action a_t based on action selection policy.
2. Perform action a_t.
3. Observe and/or calculate action's reward, and new state s_{t+1} at time $t+1$.
4. Update Q(s_t, a_t)
5. Time t <– t + 1.

   Current state s_t <– s_{t+1}.

The general workflow described above for Q-learning is similar to a "stop and wait" format: Until the reward is observed, and the state action pair's Q-value has been updated, RL learning agent cannot choose another action to perform. In our modified RL with aggregation of action selection, the workflow in each episode is as follows:

1. Let current state be s_0, select an action $a_{0,1}$ which leads to a never reached state s_1, if the action is successfully performed.
2. Add $a_{0,1}$ to a list $\{a_{0,1}, \ldots\}$.
3. Update the current state to be s_1 and repeat step 1 and 2. Stop repeating when the selected action leads to traffic demand's destination, or to avoid stuck in a loop.
4. Perform the actions $\{a_{0,1}, a_{1,2}, a_{2,3}\ldots\}$ in order.
5. Observe and calculate rewards and store them in an order list.
6. Updated corresponding state-action pairs, follow the order of performed actions.

Compared with unmodified workflow, in RLSR-Routing's approach, the RL agent first selects all actions that will be performed during one episode. Then the agent instructs the network to perform all the actions in order, and passively waits for returning QoS data which are used to calculate each action's reward. The reason why we can aggregate action selection before update Q-values are based on the following observations.

Observation I: update $Q(s_t, a_t)$ does not affect action selection at state s_{t+1}, if selected action a_{t+1} leads to a never-reached state during this episode.

Observation II: update $Q(s_{t+1}, a_{t+1})$ does not affect updating $Q(s_t, a_t)$, if Q-values are updated in order of performed actions, and path is non-cyclic.

When RL agent tries to explore the environment, it usually randomly selects an action without considering Q-value of the state-action pair. When RL agent exploits knowledge which it learned previously about the environment, it usually applies a greedy selection policy to find the action with the maximum Q-value of the state-action pair. In either case, action selection at state s_{t+1} does not consider $Q(s_t, a)$. On the other hand, action selection does not alter any Q-value on the Q-table. Therefore, we may say that update $Q(s_t, a_t)$ and action selection at state s_{t+1} are two independent steps.

When an SARSA's learning agent updates a state-action pair's Q-value, it uses the following equation.

$$Q_{t+1}(s_t, a_t) = (1 - \alpha) \times Q_t(s_t, a_t) + \alpha \times (R + \gamma \times Q_t(s_{t+1}, a_{t+1})) \qquad (4.8)$$

R stands for rewards, the Greek letter α represents the learning rate, and γ indicates the importance of long-term rewards. Update of (s_t, a_t)'s Q-value at time $t+1$ depends on its old Q-value at time t (i.e., $Q_t(s_t, a_t)$) and old Q-value for state-action pair at the next state (i.e., $Q_t(s_{t+1}, a_{t+1})$). When the path a.k.a. the list of actions, is non-cyclic, each state will only be included at most once in the path. If the update of Q-values follows the same order as actions performed, previously updated state-action pairs' Q-values will not be affected by updating the subsequent state-action pairs' Q-values.

Take topology T1 as an example. Suppose now the aggregated selected actions are {$a_{0,1}$, $a_{1,2}$, $a_{2,3}$, $a_{3,4}$}, which is a non-cyclic path from node *0* to node *4*. Starting from time *t*, update of Q-values are as follows:

1. $Q_{t+1}(s_0, a_{0,1}) = (1 - \alpha) \times Q_t(s_0, a_{0,1}) + \alpha \times (R_{0,1} + \gamma \times Q_t(s_1, a_{1,2}))$
2. $Q_{t+1}(s_1, a_{1,2}) = (1 - \alpha) \times Q_t(s_1, a_{1,2}) + \alpha \times (R_{1,2} + \gamma \times Q_t(s_2, a_{2,3}))$
3. $Q_{t+1}(s_2, a_{2,3}) = (1 - \alpha) \times Q_t(s_2, a_{2,3}) + \alpha \times (R_{2,3} + \gamma \times Q_t(s_3, a_{3,4}))$
4. $Q_{t+1}(s_3, a_{3,4}) = (1 - \alpha) \times Q_t(s_3, a_{3,4}) + \alpha \times (R_{2,3} + \gamma \times 1)$

Since action $a_{3,4}$ is the last one performed, here we just use 1 to represent its next state-action pair's Q-value. As explained early, an update of any state-action pair's Q-value does not affect previously updated state-action pairs. Based on the above observations, we think that our modification will not affect the accuracy of Q-values updates. On the other hand, aggregate action selection can reduce network costs during the learning process, particularly communication costs between network nodes and the SDN controller which is explained below.

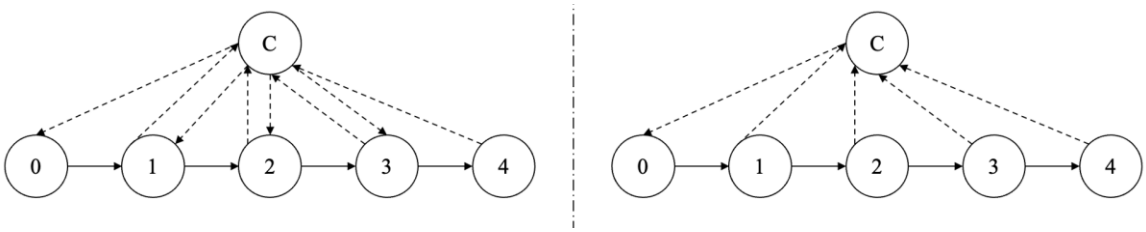


Figure 4.4: Comparing Number of Communications

Using the same example, the left of Figure 4.4 shows communications between the controller (labelled as C) and nodes without action selection aggregation, whereas the right is with action selection aggregation. Without aggregation, the controller needs to send a new packet to the network for every new selected action. In previous RL-based routing like QAR in [42], RL agent only selects and performs one action during one episode. Even if the path between node *0* and *4* is trivial, when the current state is "node *0* holds the packet" for example, RL agent can only instruct node *0* sends the packet to node *1*. After the action is performed and the packet reaches node 1 (current state becomes "node *1* holds the

packet"), the RL agent has to select and perform a new action, i.e., sends an instruction to node *1* about the next hop for the packet.

With aggregation and SR technique, the controller can encode all selected actions into a packet's header, send the packet to start routing, and passively waits for QoS data returned from the network when packet loss has not happened. In general, if *n* actions will be performed during one RL learning episode, without action selection aggregation, there will be *2n* number of communications between controller and nodes. With action selection aggregation, there will be *n+1* number of communications. In addition, if the RL approach does not combine any technique to prevent infinite loop formation, the number *n* will have a greater possibility of being equal to the maximum time-to-live (TTL) a packet allowed inside the network. Whereas our modification prevents an infinite loop since RL agent knows what states have been included in a path during aggregated action selection process. The above summarization does not consider packet loss or being dropped, which will be covered in section 4.3; it does not consider details of how QoS monitoring is implemented either, which is beyond the research interest of this thesis.

4.2.4 RL Modification: Local and Global Reward

In addition to the aggregate action selection process, we also aimed to use what RL agent has learned during each episode more efficiently to speed up algorithm convergence. Take QAR proposed in [42] as an example, its RL algorithm workflow can be summarized as follows:

1. Take source and destination node from a traffic demand.
2. Initialized Q-table with all entries set to 0.
3. In each episode, repeat Markov Decision Process.
4. After RL converges, output routing path from updated Q-table.

The Q-table is updated after the algorithm above converges, including but not limited to entries that together form a path with maximal rewards from source to destination. Since Q-values represent estimated cumulative rewards, we can view the updated Q-table containing knowledge about some of the network QoS status, if the reward function is

based on those QoS parameters. As the RL agent handles a new traffic demand and initializes Q-table's entries to 0 again, the previously learned knowledge about network QoS status is lost and the RL agent needs to explore the network from scratch again.

Inspired by previous work, which provides some starting point during RL agent's initialization [35], we proposed a "dual rewards scheme" for our modified RL algorithm to improve the efficacy of using RL results. Instead of one reward for one action, our algorithm calculates two rewards for each action: local and global rewards. The concept of *local reward* is similar to action's reward in traditional Q-learning or SARSA: RL agent uses *local rewards* to find actions that lead to maximum expected rewards (a.k.a. towards a preferred path defined by the user's customized QoS requirements). On the other hand, *global rewards* preserve the knowledge RL agent learned about the network, which can be used in future learning processes.

As a result of using two kinds of reward, RLSR-Routing needs to construct two Q-tables: local Q-table and global Q-table. At the initialization phase, users of RLSR-Routing decide whether to use a global Q-table to initialize local Q-table, instead of setting every entry to a random value like 0. During the learning process, the agent only uses a local Q-table to select actions, since the main objective is to find path for given traffic demands rather than learning overall network status. By adjusting weights of different QoS parameters, users of RLSR-Routing can customize the calculation of local rewards, and thus customize evaluation of a path's quality for different traffic demands. Meanwhile, calculation of global rewards and updating the global Q-table are hidden from incoming traffic demands. To support potential future upgrades of RLSR-Routing that enables multi-threading, i.e., running multiple RL algorithm components to find a path for multiple traffic demands concurrently, we separated the global Q-table from RL algorithm and made it another component of RLSR-Routing, as illustrated in figure 4.2.

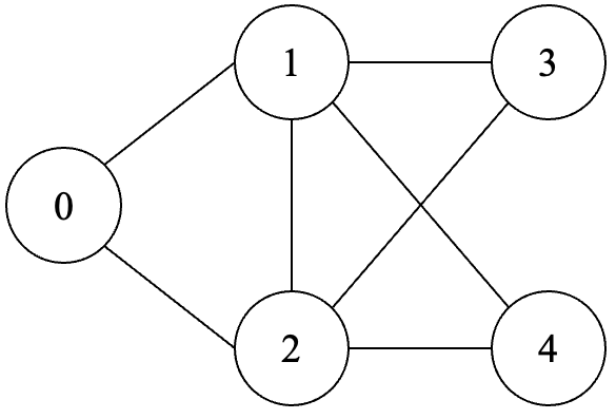


Figure 4.5: A Simple Topology (T2)

Topology T2 in figure 4.5 is sufficient to demonstrate the advantage of using dual rewards scheme. All links represent the two unidirectional links between two end nodes. RL algorithm finds the best path between node *0* and *3* based on the number of hops and link reliability, and it has reached convergence. Assuming the final local and global Q-tables are as follows:

Local table	Node *0*	Node *1*	Node *2*	Node *3*	Node *4*
Node *0*	X	-1.5	-1.8	X	X
Node *1*	0	X	-1.1	-1.5	-1.2
Node *2*	0	-1.0	X	-0.8	-1.3
Node *3*	X	0	0	X	X
Node *4*	X	-1.2	-1.1	X	X

Table 4.1: Local Q-table after Convergence (Hypothesized)

Global table	Node *0*	Node *1*	Node *2*	Node *3*	Node *4*
Node *0*	X	-1.9	-1.7	X	X
Node *1*	0	X	-1.5	-1.4	-1.1
Node *2*	0	-0.2	X	-0.9	-0.9
Node *3*	X	0	0	X	X
Node *4*	X	-0.4	-0.7	X	X

Table 4.2: Global Q-table after Convergence (Hypothesized)

Each row represents a state for both tables, and each column represents an action. Letter "X" means the state-action pair does not exist, i.e., there is no direct link from the source to the destination of the node pair. Value 0 indicates the agent did not explore the state-action pair during the learning process. Based on local Q-table, a non-cyclic path with maximal reward from node *0* to *3* will be: *0 -> 1 -> 2 -> 3*. While RL agent was exploring the best path of the traffic demand, it also updated the global Q-table based on link utilization, traffic intensity, and link reliability. Suppose now RL agent needs to find a path for a traffic demand from node *0* to node *4*, focusing on link utilization and traffic intensity instead of the number of hops. If the agent initializes all entries of the local Q-table to 0, it has zero knowledge about network load-balancing status before the learning process begins. In contrast, if it uses global Q-table to initialize local Q-table, the agent will have some prior knowledge about the network, e.g., the first action should better choose to send packet from node *0* to *2*. We believe that in some cases, a priori knowledge of network QoS status will speed up algorithm convergence.

Discussions in sections 4.2.3 and 4.2.4 are mainly under ideal cases where packet loss does not happen. Packet loss indicates the action is not successfully performed, which results in an early termination of the performing actions process. In practice, RLSR-Routing needs to handle action failure. In addition, under the dual reward scheme, definition of "action failure" might be different, depending on whether RL agent is calculating local or global

reward. We will discuss how RLSR-Routing's workflow in greater detail, including the calculation of local and global rewards, and handling action failure in section 4.3.

4.3 RLSR-Routing Workflow

In section 4.1 and 4.2 we demonstrated RLSR-Routing's architecture and explained the rationale behind some of our modifications to RL algorithm. In this section, we describe RLSR-Routing's workflow in pseudo-code, especially on how our RL routing takes traffic demand from input and produces a user preferred path for a given demand.

4.3.1 RL Algorithm: Overall Workflow

```
function RL_findRoute(TrafficDemand Demand, Boolean B, QoSWeights W,
|                     Hyperparameters H)
|       // * * * * * * * * Initialization Phase * * * * * * * *
|       get Graph G from RLSR-Routing; //network topology
|       get QTable GT from RLSR-Routing; //global Q-table
|       get QoSWeights Wd from RLSR-Routing; //default QoS weights
|       get Hyperparameters Hd from RLSR-Routing; //default hyperparameters
|       QTable localT = InitLTable(G, GT, B); //local Q-table
|       QoSWeights wl = (W == NULL) ? Wd : W; //QoS weights for local rewards
|       Hyperparameters hl = (H == NULL) ? Hd : H; //hyperparameters
|       // * * * * * * * * Learning Phase * * * * * * * *
|       int episodes = 0;
|       while (episodes < hl.E)
|       |       // -------- each learning episode --------
|       |       // * * * * Actions Selection * * * *
|       |       //selects a temporary path for traffic demand to explore network
|       |       Path tempPath = FindTempPath(Demand, localT, hl);
|       |       // * * * * Perform Actions, Observe QoS data * * * *
|       |       Push_Path_for_Routing(tempPath);
|       |       // count is number of actions performed
|       |       QoSData[] qData, int count = Collect_QoS_Data(tempPath);
|       |       // * * * * Rewards Calculation * * * *
|       |       //local rewards for performed actions
|       |       Reward[] lRewards = CalculateLRewards(qData, wl, count, Demand);
|       |       //global rewards for performed actions
|       |       Reward[] gRewards = CalculateGRewards(qData, Wd, count);
|       |       // * * * * Tables Update * * * *
|       |       UpdateTable(localT, lRewards, hl, count);
|       |       UpdateTable(GT, gRewards, Hd, count);
|       |       // end of current episode
|       |       episodes += 1;
|       end
|       //* * * * * * * * Return final result * * * * * * * *
|       return FindFinalPath(Demand, localT, hl);
end
```

RL_findRoute() describes the overall workflow of RL algorithm component: user of RLSR-Routing initiates a routing request by providing traffic demand and optional customized factors. RL algorithm explores the network and gradually learns the user preferred path during the learning phase. Once the algorithm converges or finishes the learning process, the final path is retrieved from the local Q-table. In practice, RLSR-Routing will be implemented and deployed on the SDN controller. By using the controller's API, RLSR-Routing receives requests of finding path for given traffic demands. Similarly, RLSR-Routing explores the network by using the controller's API to send packets to SR-enabled network nodes. As network nodes route the packets based on their header defined segments, RLSR-Routing waits for link-state information from network nodes. The final routing path for a traffic demand will be sent to the source node in the network, in order to achieve flow-based routing.

4.3.2 RL Algorithm: Initialization

Before RL agent starts learning process, it needs to extract information from user input as well as other RLSR-Routing components. This information is necessary to initialize local Q-table, QoS weights, hyperparameters and most importantly, the traffic demand to handle.

function InitLTable (Graph G, QTable GT, Boolean B)

```
|   //check user's provided option
|   if B is true
|   |   //user wants to use global Q-table to initialize local Q-table
|   |   QTable localT = DeepCopy(GT);
|   |   return localT;
|   end
|   //user does not want to use global Q-table for initialization
|   QTable localT = new QTable(G.numberOfNodes)
|   Link[][] links = G.getLinkMatrix()
|   for i=0; i<links.length; i++:
|   |   for j=0; j<links[0].length; j++:
|   |   |   if link[i][j] == NULL
|   |   |   |   localT[i][j] = -0x80000000;
|   |   |   end
|   |   |   else localT[i][j] = 0;
|   |   end
|   end
|   return localT;
end
```

InitLTable() is a helper function for initializing local Q-table. Both global and local Q-table are implemented as a 2-dimensional array of doubles, which array entry [i][j] represents the Q-value for state-action pair "at node *i*, sends packet to node *j*". If the user wants to apply global Q-table, the helper function creates a deep copy of global Q-table as starting point of local Q-table. We appreciate that in some cases, users of RLSR-Routing want to apply global Q-table, but not copy every entry from global to local Q-table exactly. We

may add more options for applying global Q-table for initialization in the future. On the other hand, users can initialize local Q-table from scratch: for every existed link, initializes the corresponding value to 0. For links that do not exist, initialize the corresponding state-action pair's value to a minimal integer value.

During initialization steps, RL agent also determines whether to use default QoS weights and/or hyperparameters, based on user input. The structure of QoS weights and hyperparameters are presented as follows.

Structure QoSWeights

| double lConstant; *//constant used when calculating global reward*
| double gConstant; *//constant used when calculating local reward*
| double Wc; *//hop-count reward's weight*
| double Wt; *//transmission rate reward's weight*
| double Wr; *//link reliability reward's weight*
| double Wi; *//traffic intensity reward's weight*
| double Wu; *//link utilization reward's weight*
end

Structure Hyperparameters

| double ε; *//for ε-greedy action selection*
| double α; *//learning rate*
| double γ; *//importance of long-term rewards*
| int TTL; *//maximum number of hops a packet is allowed in the network*
| int E; *//number of training episodes RL agent should perform*
end

Users can provide customized weights to emphasize the importance of different factors, and variables gConstant and lConstant depends on these weights, as illustrated below. Hyperparameters contain fields for RL's action selection and Q-value updating (ε, α, γ) and for adjusting training duration (TTL, E). Details about calculating global and local rewards will be illustrated in section 4.3.5.

$$lConstant = Wc + Wt + Wr + Wi + Wu + 0.1 \quad (4.9)$$

$$gConstant = Wr + Wi + Wu \quad (4.10)$$

4.3.3 RL Algorithm: Actions Selection

```
function FindTempPath(TrafficDemand D, QTable L, Hyperparameters H)
|   Link[][] links = get link matrix from Graph G, from RLSR-Routing;
|   //an array to record all nodes have been added to a path.
|   int[] visitedNodes = new int[G.numberOfNodes];
|   initialize all visitedNodes entries to 0;
|   //current node's id, start with traffic demand's source node
|   int currentId = D.srcId;
|   //next hop node's id
|   int nextId = -1;
|   Path tempPath;
|   //how many actions can still be selected
|   int remainTTL = H.TTL;
|   while (remainTTL > 0 && nextId != D.dstId)
|   |   Node[] neighbors = get current node's neighbors from Graph;
|   |   //get neighbor nodes that have not been included in tempPath
|   |   Node[] unvisited = neighbors ∩ (visitedNodes with entry = 0);
|   |   if unvisited is empty:
|   |   | //all current node's neighbors have been included in the tempPath
|   |   | break;
|   |   end
|   |   if Probability P < H. ε:
|   |   | //randomly selects an action (exploration)
|   |   | nextId = randomly selects one unvisited neighbor;
|   |   end
|   |   else:
|   |   | //greedily selects an action (exploitation)
|   |   | nextId = one with highest Q-value from L;
|   |   end
|   |   //add selected action(next hop) to tempPath, and updates related fields
|   |   tempPath.add(links[currentId][nextId]);
|   |   unvisited[nextId] = 1; //selected node now been included in tempPath
|   |   currentId = nextId; //next time, find next hop for node with nextId
|   |   remainTTL = remainTTL – 1;
|   end
|   return tempPath;
end
```

FindTempPath() describes how RL algorithm uses aggregate action selection to produce a non-cyclic path. The function keeps track of every node that has been added to the

temporary path, and because RLSR-Routing assigns each network node an integer ID that starts from 0, visitedNodes[i] is enough to express whether node *i* has been included. Starting from the traffic demand's source node, *FindTempPath()* builds a consecutive path by selecting from the current node's unvisited neighbors. The path building process ends due to one of the following 1) a dead-end, i.e., all current node's neighbors have been included in tempPath. 2) the path length reaches time-to-live a packet allows to travel in the network. 3) the path has reached traffic demand's destination node.

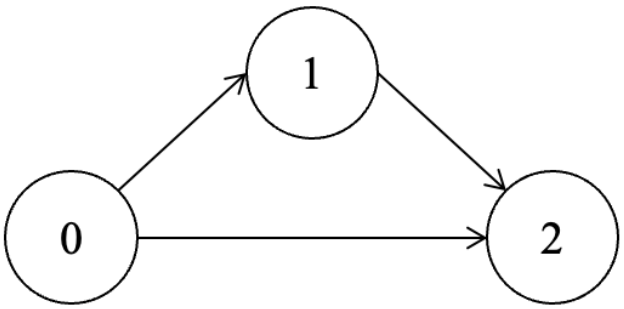


Figure 4.6: Edge Case Topology (T3)

It should be noted that *FindTempPath()* follows an action selection policy throughout its running process, even if traffic demand's destination node is one of the current node's neighbors. Considering topology T3 in figure 4.6, all links are unidirectional and now there is a traffic demand from node *0* to node *2*, with QoS consideration on link utilization. Suppose link $l_{0,2}$ currently has 99% utilization rate, whereas link $l_{0,1}$ and $l_{1,2}$ has 0% utilization. Although node *2* is adjacent to node *0*, the preferred path is *0->1->2* instead of *0->2*.

4.3.4 RL Algorithm: Perform Actions & Observe QoS Data

Push_Path_for_Routing() and *Collect_QoS_Data()* represent the process of generating a packet, encodes tempPath to packet's header, sends packet to network via SDN controller for routing, and waiting for related link-state QoS data collected by SDN controller. In this study, we only focused on simulating the overall workflow. However, during actual deployment on SDN architecture, these functions will be implemented by other components of RLSR-Routing. We may also need to modify SR protocol, so that packets

generated by RLSR-Routing in the learning process are distinguished from other packets in the network. Therefore, network devices only send related QoS data to SDN controller when they process packets created by RLSR-Routing's RL algorithm. Another way of modifying protocol can be adding a field called "SEND QoS DATA" on all packet's header. Network switches only send QoS data to the controller for certain packets (such as those created by the RL algorithm during the learning process) that set "SEND QoS DATA" on.

Although we assumed that communications between network devices and the controller are reliable, the returned QoS data may be out-of-order due to factors like the physical distance between a node and the controller. *Collect_QoS_Data()* will sort incoming data in order of tempPath's added links. In addition, *Collect_QoS_Data()* should be able to detect packet loss by a mechanism like a timer. When packet loss happens, it means one performed action failed to complete (i.e., transform current state to next state), and the function will receive no QoS data of the failed action. The perform actions process will terminate and this failed action now becomes the last action performed. After that, *Collect_QoS_Data()* should actively send a request to the controller to explicitly require related link-state information. To distinguish failed action from the action without packet loss, we added a filed in *QoSData* structure, *hasLost*, to indicate whether packet loss happened when the action was performed.

4.3.5 RL Algorithm: Rewards Calculation

$$R_{local} = Wc \times R_{hop} + Wt \times R_{transmission} + Wr \times R_{reliability} + Wi \times R_{inten\text{-}est} + Wu \times R_{util\text{-}est} - lConstant \quad (4.11)$$

$$R_{global} = Wr \times R_{reliability} + Wi \times R_{inten} + Wu \times R_{util} - gConstant \quad (4.12)$$

Equation (4.11) and (4.12) are calculation of local reward and global reward, respectively. Each individual QoS factor's reward is calculated based on equation (4.1) – (4.7), and two constant values calculations (lConstant and gConstant) are based on equations (4.9) and (4.10). In our traffic routing problem, local rewards can be viewed as a reflection of path quality for given traffic demand. Therefore, we used the estimated link destination node's

traffic intensity ($R_{inten\text{-}est}$) and estimated link utilization if we place the demand's traffic on that link, when considering local rewards. On the other hand, global rewards are used to learn current network status. Thus, the current traffic intensity (Rinten) and current link utilization (Rutil) are used to calculate global rewards. Equation (4.11) ensures all local rewards are no greater than -0.1, and (4.12) ensures all global rewards are no greater than 0.

```
function CalculateLRewards(QoSData D, QoSWeights W, int Count,
                           TtrafficDemand Demand)
|       Reward[] lRewards; //array to store local rewards
|       for int i = 0; i < Count–1 ; i++:
|       |       lRewards[i].srcId = D.srcId;
|       |       lRewards[i].dstId = D.dstId;
|       |       lRewards[i].actionSuccess = True;
|       |       lRewards[i].value = use (4.11), D, W to calculate;
|       end
|       //only the last performed action may has packet loss, handle separately
|       lRewards[Count-1].srcId = D.srcId;
|       lRewards[Count-1].dstId = D.dstId;
|       if D.hasLost OR D.dstId != Demand.dstId:
|       |       // the last performed action either has packet loss, or final node is not
|       |       //       traffic demand's destination node
|       |       lRewards[Count-1].actionSuccess = False;
|       |       lRewards[Count-1].value = -W.lConstant;
|       end
|       else:
|       |       lRewards[Count-1].actionSuccess = True;
|       |       lRewards[i].value = use Eq. (4.11), D, W to calculate;
|       end
|       return lRewards;
End
```

function CalculateGRewards(QoSData D, QoSWeights W, int Count)

```
|       Reward[] gRewards; //array to store global rewards
|       for int i = 0; i < Count–1 ; i++:
|       |       gRewards[i].srcId = D.srcId;
|       |       gRewards[i].dstId = D.dstId;
|       |       gRewards[i].actionSuccess = True;
|       |       gRewards[i].value = use Eq. (4.12), D, W to calculate;
|       end
|       //only the last performed action may has packet loss, handle separately
|       lRewards[Count-1].srcId = D.srcId;
|       lRewards[Count-1].dstId = D.dstId;
|       if D.hasLost:
|       |       // the last performed action has packet loss
|       |       lRewards[Count-1].actionSuccess = False;
|       |       lRewards[Count-1].value = -W.gConstant;
|       end
|       else:
|       |       gRewards[Count-1].actionSuccess = True;
|       |       gRewards[i].value = use Eq. (4.12), D, W to calculate;
|       end
|       return gRewards;
end
```

CalculateLRewards() and *CalculateGRewards()* show the pseudo code for calculating local and global rewards. Both functions have a similar workflow. For the first performed action till the second last, these actions are guaranteed not experienced packet loss, since RL algorithm passively received related QoS data from the controller. Therefore, two functions apply Eq. (4.11) and (4.12), respectively to calculate local/global rewards. For the last performed action, however, packet loss may happen as explained in section 4.3.4. We want to set punishment for unsuccessfully performed actions, so that RL agent learns to avoid select those actions in the future.

However, definition of “unsuccessfully performed last action” is different from the local or global reward’s perspective. For local rewards calculation, the last action is considered unsuccessful if it failed to deliver the packet to its traffic demand’s destination, even if no packet loss happened. For global rewards calculation, the last action is considered unsuccessful only when packet loss occurred. In addition to assigning punishment values for unsuccessful actions, RL algorithm adopted in RLSR-Routing has different ways to update related Q-values, which is illustrated in section 4.3.6.

4.3.6 RL Algorithm: Tables Update

function UpdateTable(QTable T, Rewards R, Hyperparameters H, int Count)

```
|   //update Q-values for the first till the second last performed actions
|   for int i=0; i<Count-1; i++:
|   |   int state = R[i].srcId;
|   |   int action = R[i].dstId;
|   |   T[state][action] = use Eq. (4.8) to update Q-value;
|   end
|   //handle potential unsuccessfully performed last action
|   int state = R[Count-1].srcId;
|   int action = R[Count-1].dstId;
|   //next state Q-value for last state-action pair is set to 0
|   double nextQVal = 0;
|   if R[count-1].actionSuccess == False:
|   |   T[state][action] += R[Count-1].value;
|   end
|   else:
|   |   Table[state][action] = use Eq. (4.8) and nextQVal to update;
|   end
end
```

RL algorithm uses *UpdateTable()* for both local and global Q-table update. For performed actions that are guaranteed to be successful (first action till the second last), the function just apply SARSA’s typical function (Eq. (4.8)) to update corresponding Q-values. For the

last action that is classified as unsuccessfully performed, the associated Q-value accumulates the penalty by adding the reward's value. The reason why UpdateTable() does not use Eq. (4.8) for unsuccessful actions' Q-values update is as follows.

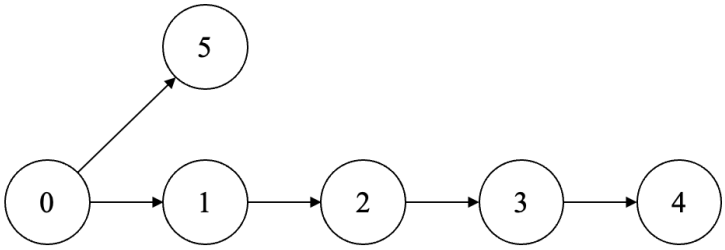


Figure 4.7: Edge Case Topology (T4)

Topology T4 illustrates an edge case which using Eq. (4.8) to update unsuccessful action's Q-value may cause problem in RLSR-Routing. Suppose a traffic demand is to send packets from node *0* to node *4*. Every successfully performed action has local reward value -2; whereas unsuccessfully performed ones have local reward value -3. RL agent uses $\alpha = \gamma = 1$, $\varepsilon = 0.5$, TTL=16 for hyperparameters. No packet loss happened during the learning process. After running enough episodes that the local Q-table converges, the final local Q-table should look like this:

Local Table	Node *0*	Node *1*	Node *2*	Node *3*	Node *4*	Node *5*
Node *0*	X	-8	X	X	X	-3
Node *1*	X	X	-6	X	X	X
Node *2*	X	X	X	-4	X	X
Node *3*	X	X	X	X	-2	X
Node *4*	X	X	X	X	X	X
Node *5*	X	X	X	X	X	X

Table 4.3: Local Table after convergence in T4 (Hypothesized)

Based on setting of RL algorithm, when node *0* chooses to send the packet to node *5*, such action is regarded as unsuccessful by local rewards calculation. Because from node *5*, RL agent cannot find a never visited next-hop node to deliver the packet to the traffic demand's

destination. If we use Eq. (4.8) to update $Q(s_0, a_{0,5})$, the value will be converged at -3 based on our settings above. As a result, at node *0*, choosing action $a_{0,5}$ has a higher cumulative reward than choosing action $a_{0,1}$ (-3 vs -8), even if sending a packet to node *1* is the only way to reach the final destination node *4*. In contrast, the accumulative penalty mechanism in *UpdateTable()* ensures that the more times an action is performed unsuccessfully, the lower Q-value it will receive. For example, the first time agent chooses $a_{0,5}$, $Q(s_0, a_{0,5})$ will become -3; the second time $Q(s_0, a_{0,5})$ will become -6; the third time $Q(s_0, a_{0,5})$ will become -9, and so on.

4.3.7 RL Algorithm: Return final result

After all learning episodes are completed, the RL algorithm uses *FindFinalPath()* to retrieve the final routing path from the source node to the destination node of the input traffic demand. *FindFinalPath()* executes *FindTempPath()*, but with Hyperparameters' ε value set to 0 – i.e., using greedy selection policy. With an adequate number of learning episodes, the final path should be non-cyclic and successfully reaches the destination node, with the highest estimated cumulative reward from local Q-table. However, the final path may not be the best solution based on the user's QoS requirements, and one possible reason could be the action selection policy used during the learning process.

4.4 Action Selection Policy and Final Path's Quality

In section 4.3 we introduced the overall workflow of RLSR-Routing's RL algorithm component. Within the RL algorithm, each sub steps (e.g., action selections, rewards calculation) can also be designed and implemented independently, giving additional flexibility to fulfill various objectives. For actions selection, we designed it to apply the ε-greedy policy which aims to balance exploration of the network and exploitation of agent's learned knowledge. However, during the implementation of RLSR-Routing, we noticed the inconsistency of the final routing path caused by ε-greedy during RL agent's learning process.

4.4.1 Drawback of ε-greedy for SARSA Based Routing

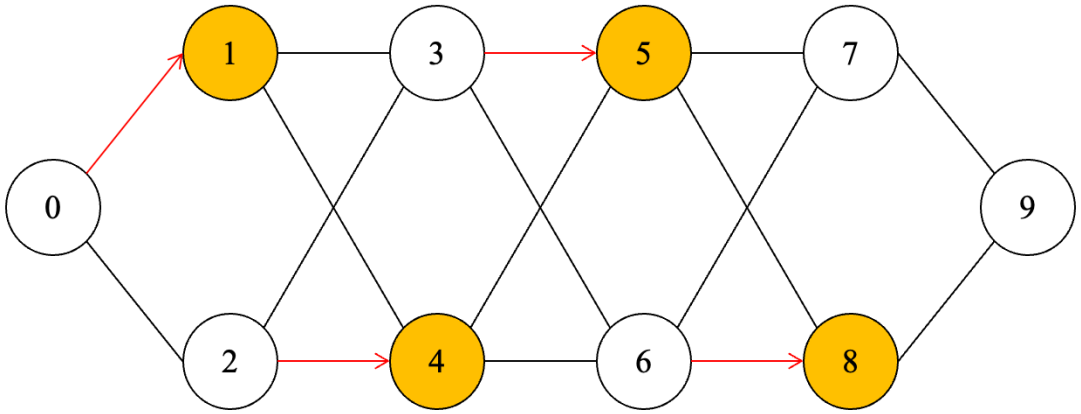


Figure 4.8: 10 Nodes Topology (T5)

Figure. 4.8 presents a 10 nodes network with mesh topology (T5). Each node has 10Mb/s processing rate, and each link has 10Mb/s maximum bandwidth. Each link on T5 represents two unidirectional links that connect two adjacent nodes, with link $l_{0,1}$, $l_{2,4}$, $l_{3,5}$ and $l_{6,8}$ (marked as red, single arrow) currently having 9.9Mb/s traffic. Suppose we want to find a path to send packets from node *0* to node *9*, with link utilization and link destination node's traffic intensity as QoS considerations. Based on settings above, the user preferred path from *0* to *9* should be *0 -> 2 -> 3 -> 6 -> 7 -> 9*. When RL agent uses $\alpha = \gamma = 0.9$ and a non-zero ε value (e.g., 0.3), the final path produced by RL algorithm is inconsistent, for example:

i. *0 -> 1 -> 4 -> 6 -> 7 -> 9*; (A less preferred path)
ii. *0 -> 2 -> 3 -> 6 -> 7 -> 9*; (Same as the user preferred path)
iii. *0 -> 2 -> 4 -> 5 -> 8 -> 9*. (Another less preferred path)

For unmodified ε-greedy action selection, the probability of randomly selecting an action is consistent throughout the learning process. For example, the agent in the above settings still has 0.3 probability of randomly selecting an action, even for the last few episodes of the learning process. Recall that for SARSA, update of the current state-action pair's Q-value depends on the action performed at the next state (Eq. (4.8)). Suppose at the beginning of episode 95 (out of 100), $Q(s_0, a_{0,2})$ is higher than $Q(s_0, a_{0,1})$, a greedy selection

will add $a_{0,2}$ as the first action. At node *2*, if RL agent randomly selects $a_{2,4}$ as the next action, the update of $Q(s_0, a_{0,2})$ will be:

$$Q_{t=96}(s_0, a_{0,2}) = (1 - \alpha) \times Q_{t=95}(s_0, a_{0,2}) + \alpha \times (R + \gamma \times Q_{t=95}(s_2, a_{2,4})).$$

Since link $l_{2,4}$ currently has high traffic load, $Q_{t=95}(s_2, a_{2,4})$ is probably going to make $Q_{t=96}(s_0, a_{0,2})$ lower than $Q_{t=96}(s_0, a_{0,1})$. At episode 96, RL agent may not have enough trials to learn that at node *0*, choosing link $l_{0,2}$ is better than link $l_{0,1}$. As a result, we saw a less preferred path produced in the trail (i) has *0 -> 1* as the first link on the path.

To sum up, the drawback of ε-greedy is that the probability of randomly selecting an action is constant throughout the learning process. And as noted in [42], all available actions have an equal chance to be chosen when using random selection. A better solution should be a kind of variable ε-greedy policy, such that at the beginning, the agent has a high probability of exploring the environment; whereas at the end the agent has a high probability or even completely using greedy policy to exploit the knowledge it learned.

4.4.2 Greedy Approach: Temporary Solution to Local Optimality

To our surprise, after we set ε value to 0 (i.e., all action selection are using greedy approach, so that the algorithm always selects the action which has the maximum Q-value among the current state's available actions), RL algorithm can find the user preferred path for the experiment on topology T5, and the results are consistent through different trials. The rational greedy approach still able to find the preferred path is probably due to the value-based nature of RL. In the beginning, we initialized all local Q-table's entry to 0, and no action's reward can be greater then -0.1. As a result, the greedy approach still ensures the exploration of network during an early stage of the learning process. Moreover, after Q-table's values converged, the tempPath selected in each episode will be consistent, thus, randomly chosen bad action will not occur at the end of the learning process when applying the greedy approach.

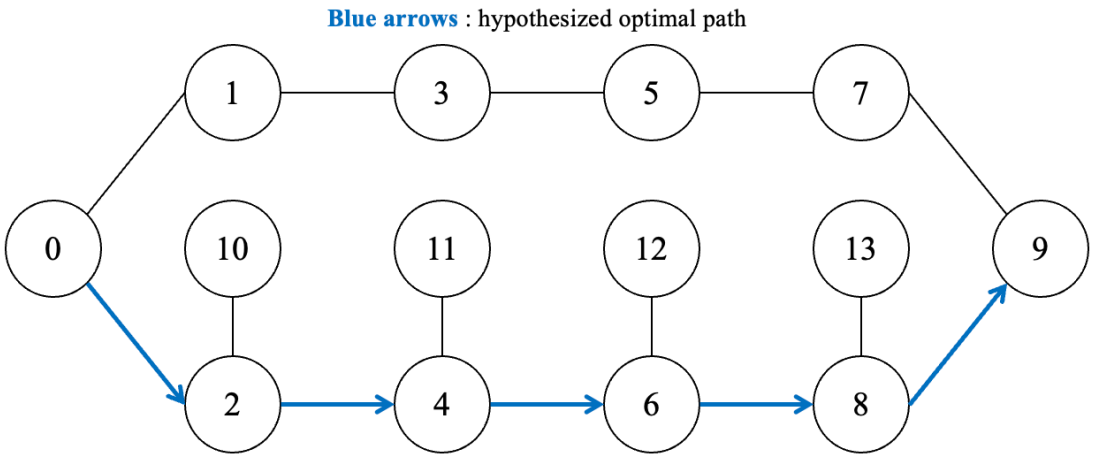


Figure 4.9: Edge Case Topology (T6)

However, the pure greedy approach may not guarantee that RL algorithm always produces the best solution. T6 illustrates a ring-like topology with two non-cyclic paths from node *0* to node *9* (one starts with link $l_{0,1}$ and the other starts with $l_{0,2}$). We assumed that if RL algorithm sends packets through both paths with enough episodes, at node *0*, choosing link $l_{0,2}$ will have higher reward (e.g., $Q(s_0, a_{0,1}) = -11$, $Q(s_0, a_{0,2}) = -10$). During the initial episodes, RL agent has not fully explored the network. The selected temporary path may be *0->2->10*, *0->2->4->11*, *0->2->4->6->12*… etc., since initially all entries have same Q-value. As a result, Q-value for state-action pair s_0, $a_{0,2}$ may be lower than its converged value during the early training process, e.g., $Q(s_0, a_{0,2}) = -12$. Meanwhile, if the path starts with $l_{0,1}$ converges, we now face the situation that $Q(s_0, a_{0,1}) = -11$ which is greater than $Q(s_0, a_{0,2}) = -12$. The final produced path will be *0->1->3->5->7->9* instead of *0->2->4->6->8->9*.

Throughout the rest of our implementation and experiment setup, we applied a greedy action selection policy since it is easy to achieve while still producing users preferred results. Since our main focus is to prove the ability of RLSR-Routing to satisfy various QoS demands and to validate two of our major modifications (described in section 4.2.3 and 4.2.4) on SARSA. We will leave an upgrade of a better action selection policy as future work.

4.5 Network Simulation

We provided two ways to create a network topology for performing experiments. First is to use our implemented methods to write code, which step by step creates a graph representation of a network, adds nodes and links, and possibly assigns the initial used bandwidth of certain links. Another way is to write a JSON file with a specified format, which can be parsed using an open-source JSON parser.

4.6 General Settings for Hyperparameters

Authors who proposed QAR in [42] conducted experiments to study the effects of learning rate and the importance of long-term rewards. We followed their guidelines and unless specified explicitly, the hyperparameters used in Chapter 5's experiments are as follows:

- $\varepsilon = 0$;
- $\alpha = 0.9$;
- $\gamma = 0.9$;
- TTL = 32;
- E = 75.

Chapter 5

5 Implementation and Results

RLSR-Routing is a framework that directly addresses path finding issues in SDN, without requiring prior knowledge of the network (e.g., link weights) or external libraries to train a neural network. Deployment of RLSR-Routing should be flexible as long as it can interact with the SDN controller and parse JSON file. This study mainly focused on the RL algorithm component's implementation, with additional code to support network simulation. All code is written in JAVA but transfer to other programming languages should not be a complicated task. This chapter presents our comparative study and validation experiments' results with discussion.

5.1 Comparative Study with Non-RL routing

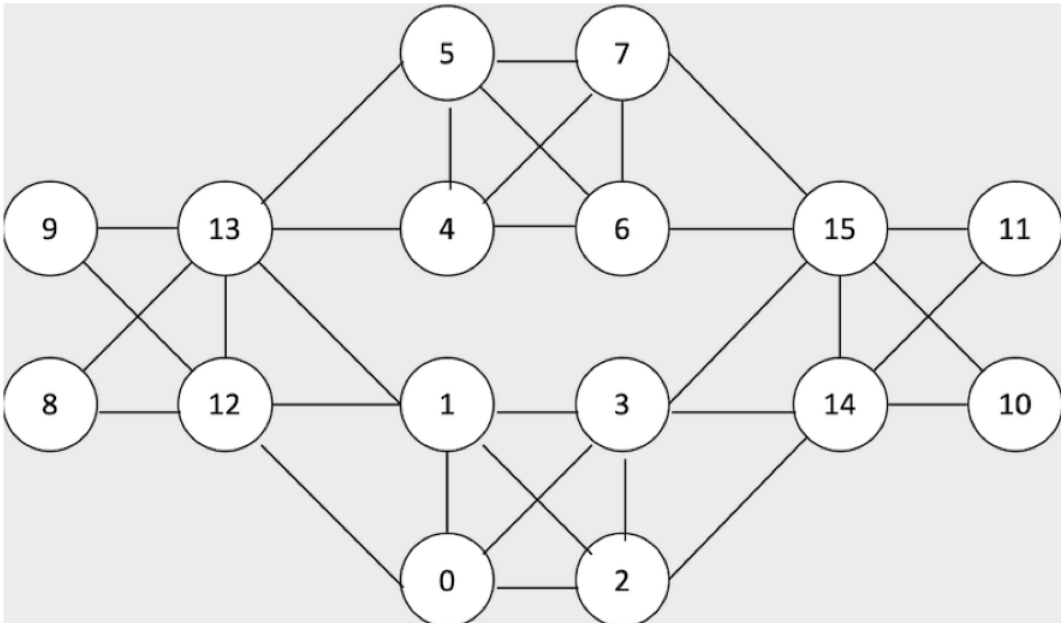


Figure 5.1: 16 Nodes Topology (T7)

We performed a comparative study between RLSR-Routing and a non RL-based routing algorithm, NR-Routing, which is used by a telecom solutions provider. Similar to our proposed RLSR-Routing, NR-Routing is able to assign paths for given traffic flows between source and destination pairs; but it uses a non-RL based agent to explore the traffic

engineering space by placing traffic demands greedily (from largest to smallest) and/or randomly on the network capacity using multi-path segment routing based traffic engineering. Due to the confidentiality issue, we cannot provide additional information about NR-Routing's implementation as well as other details. The overall objective is to compare two algorithms' load balancing ability in terms of minimizing maximum link utilization over given network. The testing network topology T7 is shown in Fig. 5.1, with 16 nodes and each pair of adjacent nodes has two unidirectional links connecting with each other. All links have 2.0Mb/s maximum bandwidth, 100% reliability rate, and all nodes have 20Mb/s processing rate. NR-Routing is an executable program embodying two main functions:

- Generation of traffic engineering problems consisting network topologies and sets of traffic demands to place the network capacity
- Optimization of traffic demand engineering by placing demands on paths in the network with segment routing based traffic steering techniques.

We first randomly generate networks and demands and let NR-Routing assign one or more paths for each demand. These generated paths were saved as JSON file. We then removed unwanted demands, adjusted the amount of data for the remaining, and then used them as NR-Routing's input for load balancing. This time the final output is saved at another JSON file and to be used for comparison. The traffic and paths assignments for selected tunnels that we used in NR-Routing program's load balancing test are as follows:

- From node *4* to *14*: 400kb/s total traffic, 2 paths.
- From node *1* to *5*: 400kb/s total traffic, 2 paths.
- From node *0* to *6*: 1.0Mb/s total traffic, 5 paths.
- From node *13* to *11*: 600kb/s total traffic, 4 paths.
- From node *3* to *13*: 600kb/s total traffic, 3 paths.
- From node *3* to *4*: 1.2Mb/s total traffic, 6 paths.

For RLSR-Routing, it interoperated each tunnel as *n* traffic demands, where *n* equals to number of paths been assigned to the tunnel by NR-Routing. Every traffic demand from

the same tunnel will split the tunnel's total traffic. For example, for tunnels from node 3 to 4, each of the six traffic demands will have 200kb/s traffic. RL algorithm only considered link utilization when calculating local rewards during the learning process. After the RL algorithm found a path for one traffic demand, the demand's traffic was placed over the network along the path. The above steps were repeated until RL algorithm found path for all traffic demands and placed all demand traffic over the network.

5.1.1 Experiment Result: Routing Paths by RLSR-Routing

The paths generated for each tunnel (source and destination pair) by RLSR-Routing are listed below.

Paths for two traffic demands from node *4* to *14*:

i. *4 -> 7 -> 15-> 14;*
ii. *4 -> 6 -> 15 -> 14.*

Paths for two traffic demands from node *1* to *5*:

i. *1 -> 13 -> 5;*
ii. *1 -> 13 -> 5.*

Paths for five traffic demands from node *0* to *6*:

i. *0 -> 3 -> 15 -> 6;*
ii. *0 -> 2 -> 14 -> 15 -> 6;*
iii. *0 ->12 -> 13 -> 4 -> 6;*
iv. *0 -> 3-> 15 -> 7 -> 6;*
v. *0 -> 1 -> 3 -> 15 -> 6.*

Paths for four traffic demands from node *13* to *11*:

i. *13 -> 1 -> 2 -> 14 -> 11;*
ii. *13 -> 1 -> 3 -> 14 -> 11;*
iii. *13 -> 4 -> 7 -> 15 -> 11;*
iv. *13 -> 5 -> 6 -> 15 -> 11.*

Paths from node *3* to *13*:

i. *3 -> 1 -> 13;*

ii. *3 -> 0 -> 12 -> 13;*

iii. *3 -> 1 -> 13.*

Paths from node *3* to *4*:

i. *3 ->15 -> 7 -> 4;*

ii. *3 -> 2 -> 1 -> 12 -> 8 -> 13 -> 4;*

iii. *3 ->14 -> 15 -> 7 -> 4;*

iv. *3 -> 15 -> 6 -> 4;*

v. *3 ->1 -> 13 -> 4;*

vi. *3 -> 0 -> 12 -> 13 -> 4.*

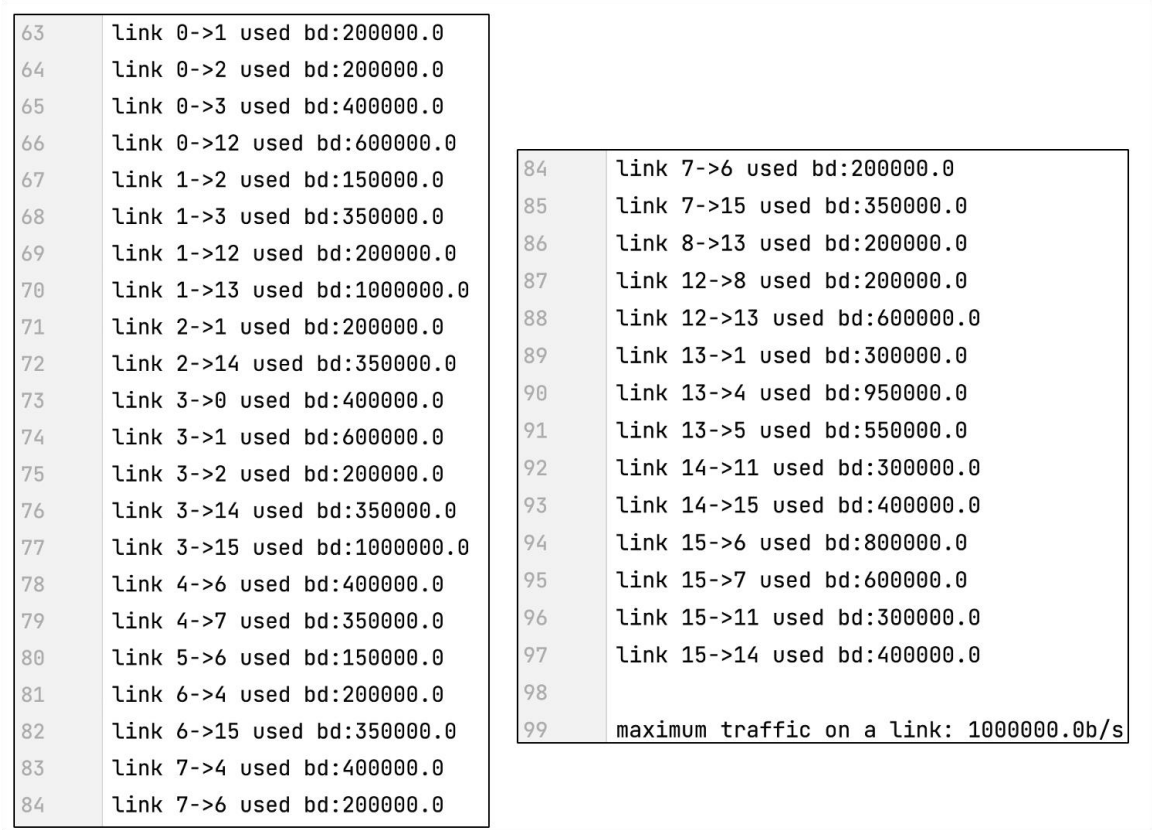
All the paths produced by RLSR-Routing are valid routing paths: they are loop-free and consist of a consecutive set of nodes that connect the source and destination nodes of specified traffic demands.

5.1.2 Experiment Result: Links Utilizations

```
15  R3(14) -> R4(15)used BD: 800000.0
16  R4(15) -> R3(14)used BD: 400000.0
17  B2(5) -> B3(6)used BD: 316666.0
18  R1(12) -> R2(13)used BD: 1433333.0
19  B3(6) -> R4(15)used BD: 500000.0
20  R4(15) -> B3(6)used BD: 966666.0
21  A2(1) -> R2(13)used BD: 500000.0
22  A1(0) -> R1(12)used BD: 933333.0
23  A4(3) -> R4(15)used BD: 466666.0
24  R2(13) -> B2(5)used BD: 866666.0
25  B1(4) -> B3(6)used BD: 516666.0
26  B3(6) -> B1(4)used BD: 300000.0
27  A4(3) -> R3(14)used BD: 466666.0
28  R2(13) -> B1(4)used BD: 1066666.0
29  A2(1) -> R1(12)used BD: 500000.0
```

```
30  A3(2) -> R3(14)used BD: 333333.0
31  B1(4) -> B4(7)used BD: 350000.0
32  B4(7) -> B1(4)used BD: 300000.0
33  A1(0) -> A4(3)used BD: 333333.0
34  A4(3) -> A1(0)used BD: 600000.0
35  A1(0) -> A3(2)used BD: 333333.0
36  A4(3) -> A2(1)used BD: 600000.0
37  B4(7) -> R4(15)used BD: 500000.0
38  R4(15) -> B4(7)used BD: 300000.0
39  B2(5) -> B4(7)used BD: 150000.0
40  R4(15) -> PE4(11)used BD: 600000.0
41
42  maximum traffic on a link: 1433333.0b/s
```

Figure 5.2: Screenshot for NR-Routing's Links with Traffic



```
63  link 0->1 used bd:200000.0
64  link 0->2 used bd:200000.0
65  link 0->3 used bd:400000.0
66  link 0->12 used bd:600000.0
67  link 1->2 used bd:150000.0
68  link 1->3 used bd:350000.0
69  link 1->12 used bd:200000.0
70  link 1->13 used bd:1000000.0
71  link 2->1 used bd:200000.0
72  link 2->14 used bd:350000.0
73  link 3->0 used bd:400000.0
74  link 3->1 used bd:600000.0
75  link 3->2 used bd:200000.0
76  link 3->14 used bd:350000.0
77  link 3->15 used bd:1000000.0
78  link 4->6 used bd:400000.0
79  link 4->7 used bd:350000.0
80  link 5->6 used bd:150000.0
81  link 6->4 used bd:200000.0
82  link 6->15 used bd:350000.0
83  link 7->4 used bd:400000.0
84  link 7->6 used bd:200000.0
```

```
84  link 7->6 used bd:200000.0
85  link 7->15 used bd:350000.0
86  link 8->13 used bd:200000.0
87  link 12->8 used bd:200000.0
88  link 12->13 used bd:600000.0
89  link 13->1 used bd:300000.0
90  link 13->4 used bd:950000.0
91  link 13->5 used bd:550000.0
92  link 14->11 used bd:300000.0
93  link 14->15 used bd:400000.0
94  link 15->6 used bd:800000.0
95  link 15->7 used bd:600000.0
96  link 15->11 used bd:300000.0
97  link 15->14 used bd:400000.0
98
99  maximum traffic on a link: 1000000.0b/s
```

Figure 5.3: Screenshot for RLSR-Routing's Links with Traffic

Figures 5.2 and 5.3 show the links that placed traffic on it by NR-Routing and RLSR-Routing, respectively. For NR-Routing, traffic assigned on different links of the network was saved in the JSON output file. We extracted the information and saved in another text file. Although the same network topology was used in both NR-Routing and RLSR-Routing, the labels for each node used in NR-Routing are different. Therefore, we included the translated node's ID in brackets, for example, node with ID R3 in NR-Routing is node with ID (14) in RLSR-Routing. For NR-Routing, the maximum traffic assigned to one link is 1.433333Mb/s, thus the maximum link utilization in NR-Routing's output is 1.433333 / 2.0 = 71.67%. For RLSR-Routing, the maximum traffic assigned to one link is 1.0Mb/s, thus the maximum link utilization in RLSR-Routing's links assignment is 1.0 / 2.0 = 50.0%.

5.2 Cost of Exploration During Learning Process

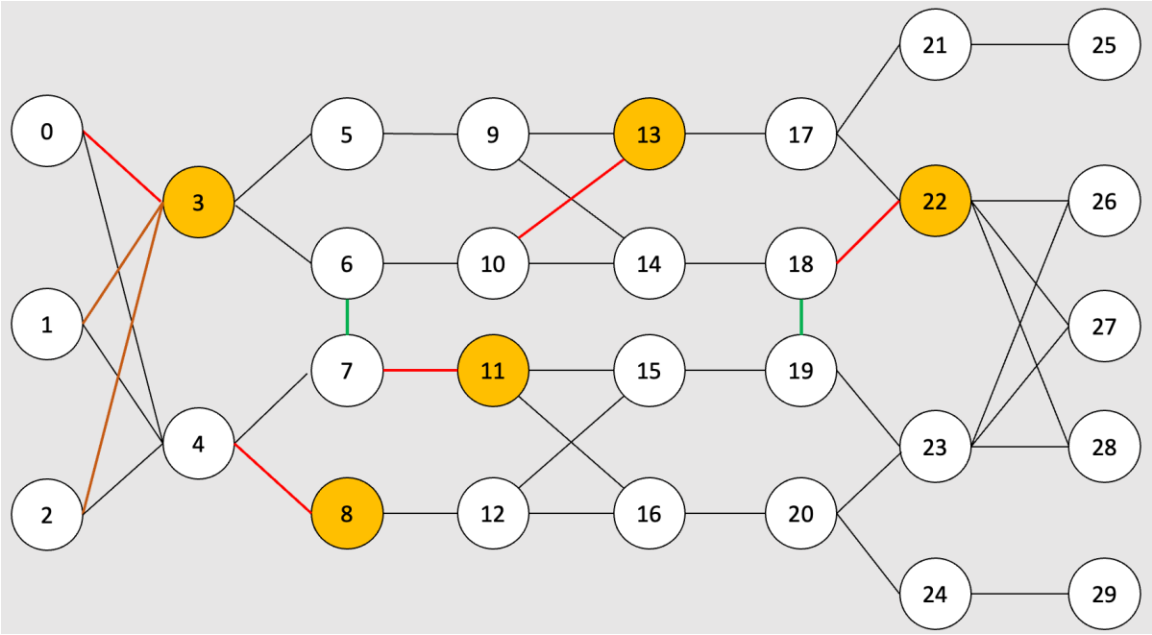


Figure 5.4: 30 Nodes Topology (T8)

We used a 30 nodes network as shown in figure 5.4 to test the efficacy of RLSR-Routing in learning process. In topology T8, each pair of adjacent nodes are connected by two unidirectional links to support bi-directional communications. All nodes have 100Mb/s processing rate, and all links have 10Mb/s maximum bandwidth. Before any traffic demand's data been placed over the network, links used bandwidth are as follows:

- Link $l_{6,7}$, $l_{7,6}$, $l_{18,19}$, $l_{19,18}$: 0Mb/s used bandwidth;
- Link $l_{1,3}$, $l_{2,3}$: 5Mb/s used bandwidth;
- Link $l_{0,3}$, $l_{4,8}$, $l_{7,11}$, $l_{10,13}$, $l_{18,22}$: 9Mb/s used bandwidth;
- The rest of links: 1Mb/s used bandwidth.

In this experiment, we gave RLSR-Routing nine traffic demands, each with 0.1Mb/s estimated traffic. RL algorithm found one path for every given traffic demand, with traffic intensity and link utilization as considered QoS parameters (Wi = Wu = 1). Global Q-table was not used during this experiment since the experiment was not focused on convergence

speed. After one path was returned from RL algorithm, corresponding traffic demand data was placed over the network (i.e., all links on the path added 0.1Mb/s to their used bandwidth).

5.2.1 Experiment Result: Final Paths

Traffic Demand	Final Path
0 -> 26	*0 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 26*
1 -> 26	*1 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 26*
2 -> 26	*2 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 26*
0 -> 27	*0 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 27*
1 -> 27	*1 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 27*
2 -> 27	*2 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 27*
0 -> 28	*0 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 28*
1 -> 28	*1 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 28*
2 -> 28	*2 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 28*

Table 5.1: Final Path for Traffic Demands

Table 5.1 summarizes the final routing path returned by RLSR-Routing for every traffic demand. All routing paths are valid: every node in a path is only included once, and every path reaches the specified destination. During the learning process, RLSR-Routing gradually explored the network, so that highly utilized links (like link $l_{0,3}$, $l_{4,8}$, $l_{7,11}$, $l_{10,13}$, $l_{18,22}$) and nodes with relatively high traffic intensity (link node *3*, *8*, *11*, *13*, *22*) are excluded in final paths. As a result, all the paths include the same set of nodes, node *4*, *7*, *6*, *10*, *14*, *18*, *19*, *23* as intermediate nodes between source and destination. Although these are not the shortest paths in terms of hop-count, they are the user preferred paths in terms of link utilization and traffic intensity.

5.2.2 Experiment Result: Path Length vs Episode Number

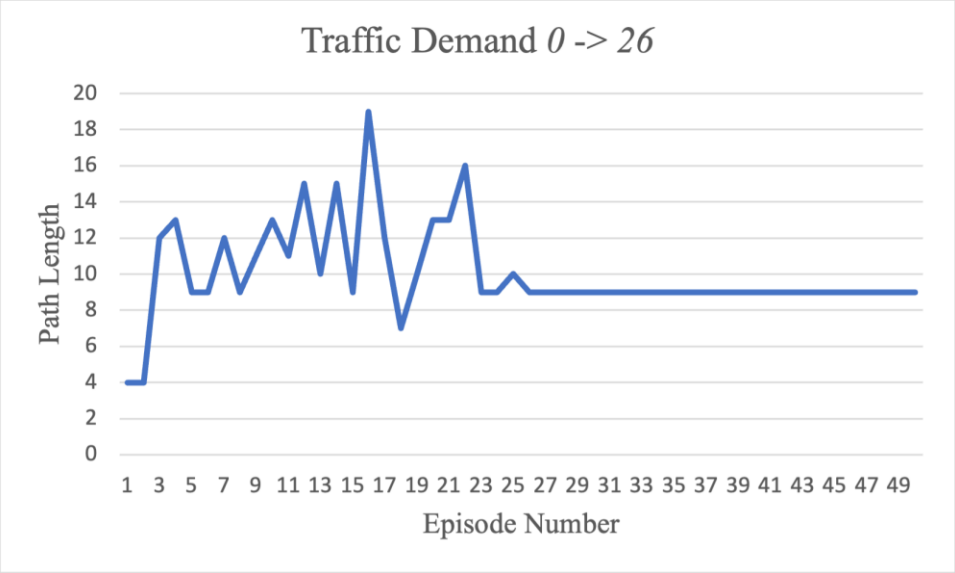


Figure 5.5: TempPath length for Demand *0* -> *26*

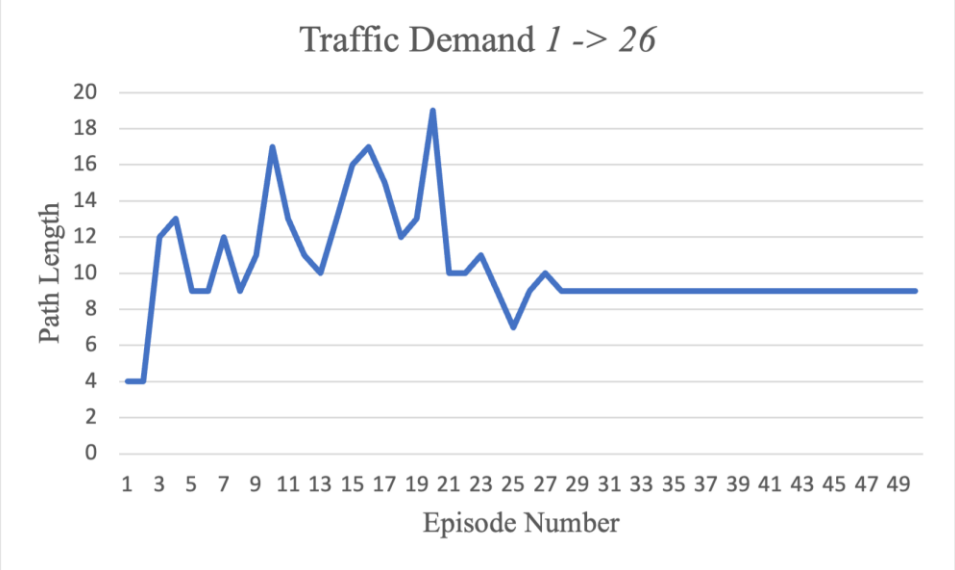


Figure 5.6: TempPath length for Demand *1* -> *26*

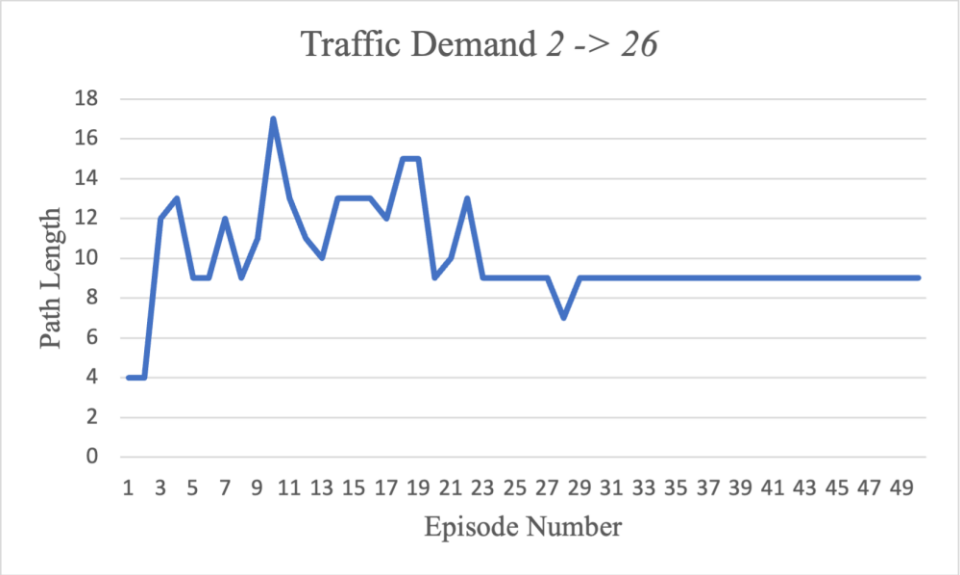


Figure 5.7: TempPath length for Demand *2 -> 26*

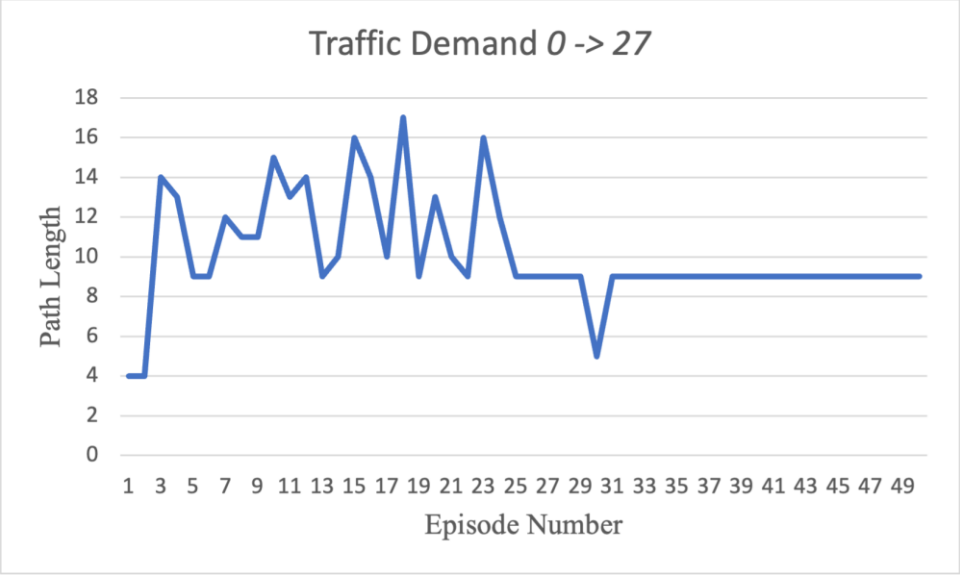


Figure 5.8: TempPath length for Demand *0 -> 27*

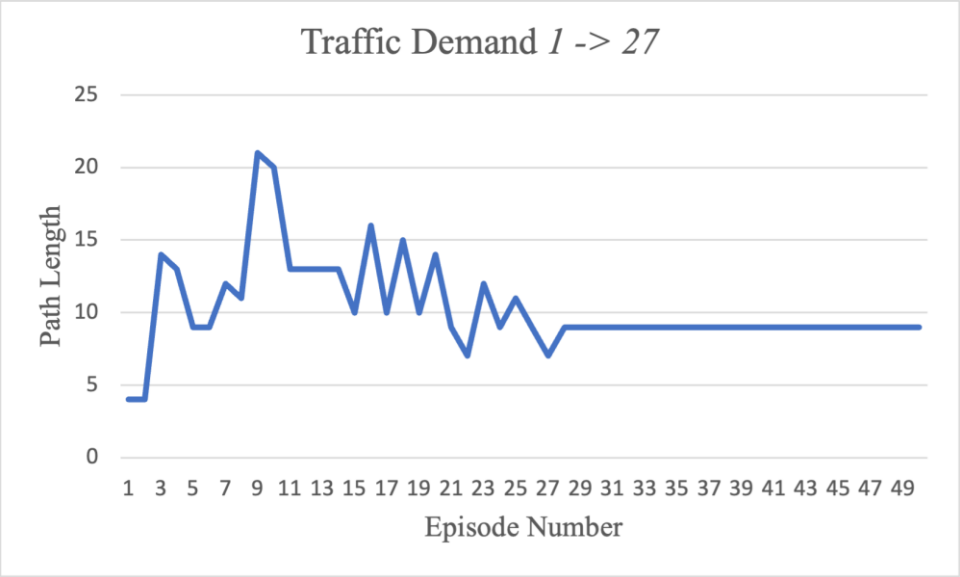


Figure 5.9: TempPath length for Demand *1* -> *27*

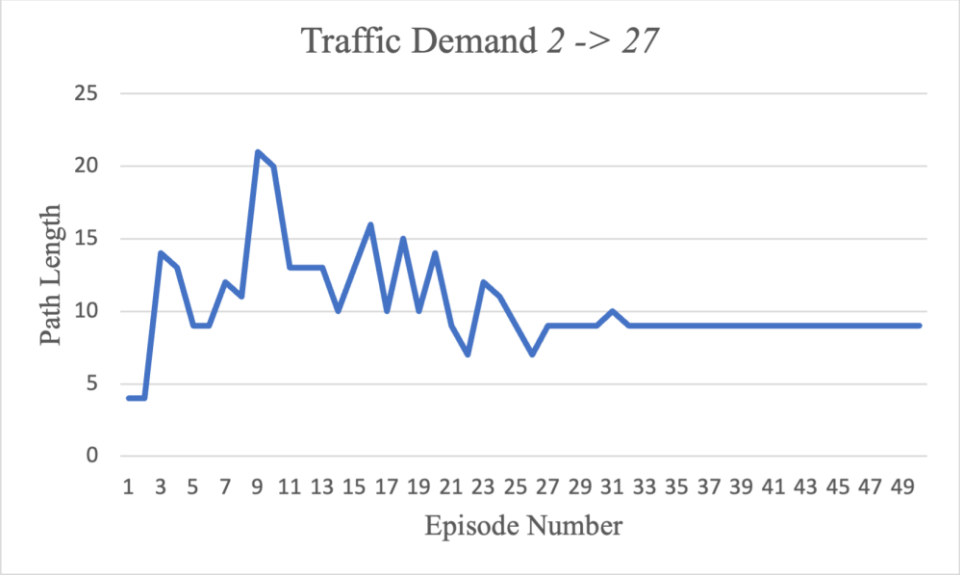


Figure 5.10: TempPath length for Demand *2* -> *27*

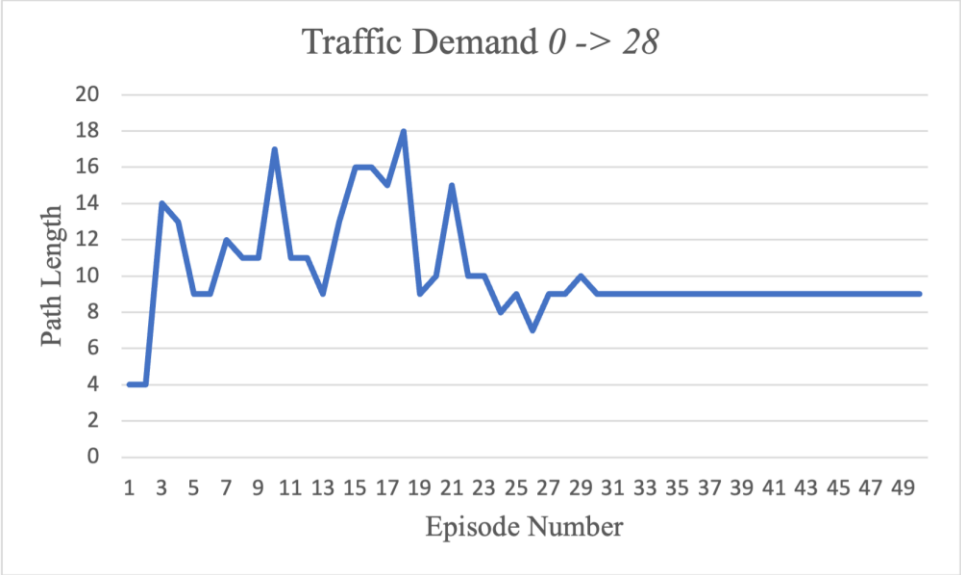


Figure 5.11: TempPath length for Demand *0 -> 28*

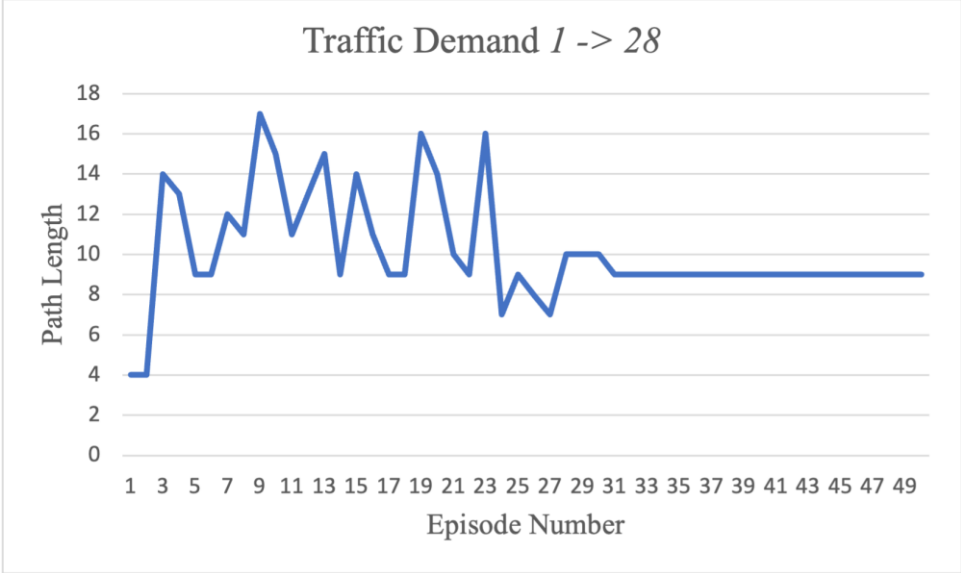


Figure 5.12: TempPath length for Demand *1 -> 28*

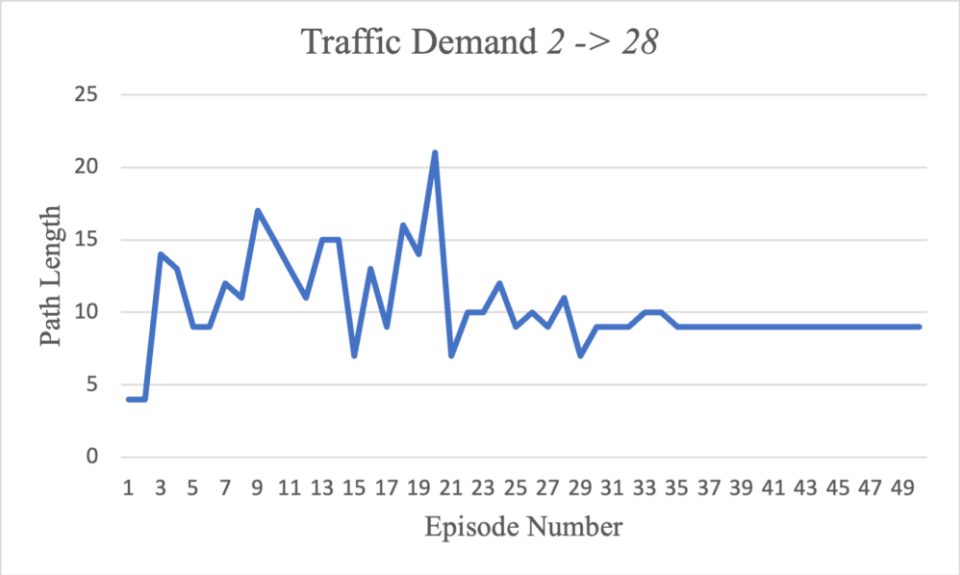


Figure 5.13: TempPath length for Demand *2 -> 28*

Figure 5.5 to 5.13 demonstrated when RLSR-Routing was finding the path for each traffic demand, the length (i.e., number of hops) of temporary paths (denote as TempPath) generated in the first 50 episodes. None of the temporary paths would result in a packet inside a loop, and most of the paths have less than 20 nodes included. Although at the beginning of the learning process, especially during the first 20 episodes, some temporary paths failed to reach the desired destination, the agent quickly learned to avoid choosing actions that lead to a dead end. For most of the testing traffic demands, RLSR-Routing was able to converge around 30 episodes; after that, the temporary paths produced in each episode are the same as the final paths.

5.3 Global Q-table and Effects of γ Value

In this section, we present our study about whether using global Q-table speeds up algorithm convergence. We applied the same network topology (T8) and the same initial links used bandwidth as in section 5.2. The same set of traffic demands were given to RLSR-Routing and the user preferred path was defined by the same QoS weights. The only difference is that RLSR-Routing used global Q-table to initialize local Q-table. To study the effect of γ (importance of long term reward) on applying global Q-table, we set up five experiment groups. The control group from section 5.2 did not use global Q-table; test

groups I to IV applied γ value 0.3, 0.5, 0.7 and 0.9 to update global Q-table's entries. The final paths for each traffic demand and the number of episodes RLSR-Routing used to converge were recorded for further analysis.

5.3.1 Experiment Result: Final Paths for Traffic Demands

Traffic Demand	Final Path
0 -> 26	*0 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 26*
1 -> 26	*1 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 26*
2 -> 26	*2 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 26*
0 -> 27	*0 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 27*
1 -> 27	*1 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 27*
2 -> 27	*2 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 27*
0 -> 28	*0 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 28*
1 -> 28	*1 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 28*
2 -> 28	*2 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 28*

Table 5.2: Final Paths by Test Group I (γ = 0.3)

Traffic Demand	Final Path
0 -> 26	*0 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 26*
1 -> 26	*1 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 26*
2 -> 26	*2 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 26*
0 -> 27	*0 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 27*
1 -> 27	*1 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 27*
2 -> 27	*2 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 27*
0 -> 28	*0 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 28*
1 -> 28	*1 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 28*
2 -> 28	*2 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 28*

Table 5.3: Final Paths by Test Group II ($\gamma = 0.5$)

Traffic Demand	Final Path
0 -> 26	*0 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 26*
1 -> 26	*1 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 26*
2 -> 26	*2 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 26*
0 -> 27	*0 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 27*
1 -> 27	*1 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 27*
2 -> 27	*2 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 27*
0 -> 28	*0 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 28*
1 -> 28	*1 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 28*
2 -> 28	*2 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 28*

Table 5.4: Final Paths by Test Group III ($\gamma = 0.7$)

Traffic Demand	Final Path
0 -> 26	*0 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 26*
1 -> 26	*1 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 26*
2 -> 26	*2 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 26*
0 -> 27	*0 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 27*
1 -> 27	*1 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 27*
2 -> 27	*2 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 27*
0 -> 28	*0 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 28*
1 -> 28	*1 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 28*
2 -> 28	*2 -> 4 -> 7 -> 6 -> 10 -> 14 -> 18 -> 19 -> 23 -> 28*

Table 5.5: Final Paths by Test Group IV (γ = 0.9)

Table 5.2 to 5.5 illustrate different test groups' final paths for given traffic demands. Sama as the control group in section 5.2, all test groups' final paths are the best in terms of traffic intensity and link utilization.

5.3.2 Experiment Result: RL Convergence Speed

Demand	Control	Test I γ = 0.3	Test II γ = 0.5	Test III γ = 0.7	Test IV γ = 0.9
0 -> 26	25	25	25	25	25
1 -> 26	27	26	25	21	18
2 -> 26	30	28	38	17	17
0 -> 27	30	21	24	23	20
1 -> 27	27	24	25	21	20
2 -> 27	31	25	26	24	25
0 -> 28	29	22	25	27	21
1 -> 28	30	26	28	29	22
2 -> 28	34	34	32	32	24

Table 5.6: Episode Number when RL Converged

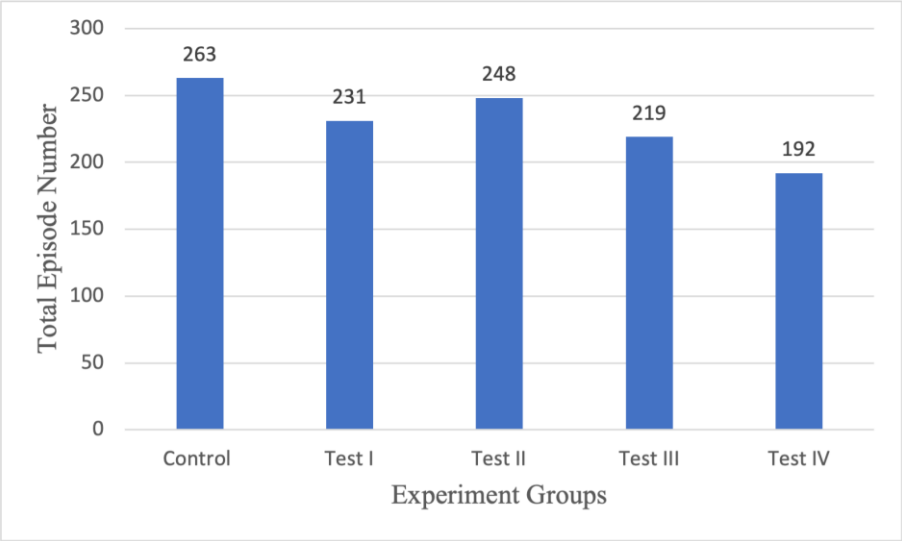


Figure 5.14: Total Episode Numbers when RL Converged for All Demands

Table 5.6 shows for each traffic demand, the number of episodes different experiment groups used to reach convergence. Compared with the control group, in most cases, test groups used fewer episodes to reach convergence for the same traffic demand. Figure 5.14 summarizes total episode numbers for each group to reach convergence on 9 traffic demands. All test groups used fewer total episode numbers to find paths for all traffic demands than the control group. Test group IV which used γ value 0.9 has the fastest convergence speed: it took 192 episodes to find paths for all given traffic demands.

5.4 Discussion

The overall objectives of the experiments are to test RLSR-Routing's ability to find user preferred paths, as well as its efficiency during path finding process. As suggested by the industry collaborator, we used the 16 nodes topology T7 and those traffic demands, which were both generated by our industry collaborator's provided NR-Routing program. Due to the limit of time, we only conducted section 5.1's experiment as the first stage of the comparative study. For the 30 nodes topology T8, we think the network is adequate to make the path finding less trivial and to provide good visualization, compared with 36 nodes topology in [18] and 25 nodes network used in [42]. We used path length, i.e., number of hops a packet travelled in the network during one training episode, as a measurement of training cost. The longer the path length, the longer time a packet will be

in the network. We used to number of episodes RLSR-Routing took to converge as a measurement of the algorithm's efficiency. The fewer the episodes RLSR-Routing took to reach convergence, the more efficient the algorithm is.

In the comparative study (Section 5.1), we observed that RLSR-Routing achieved lower maximum link utilization than NR-Routing after placing the same amount of traffic over the same network topology. When we compared the two algorithms' output, we noticed that RLSR-Routing used more links than NR-Routing. It seems that RLSR-Routing tried to balance network load by detouring some traffic through longer paths to their destination.

On the 30 nodes topology T8, RLSR-Routing still able to find user preferred paths for given traffic demands. Although constraints from the non-cyclic path policy resulted in some packets not being sent to the destination during exploration, RLSR-Routing converged within a reasonable number of episodes. Combined with the observation that most temporary paths consist of less than 20 nodes, we conclude that RLSR-Routing does not require much network resource during the learning process. The cost of path finding can be further reduced by applying previously learned network status, as we observed in section 5.3's experiment. The best test group uses 27% a smaller number of episodes to find paths than the control group (192 vs 263). Similar improvement can be found in all test groups that applied the global Q-table. We hypothesized that higher γ value used for updating global Q-table should have a better result, since RL agent will learn more about long term network QoS status. However, Test group II takes more episodes to reach convergence than test group I. This is perhaps due to a relatively small test size (9 traffic demands tested), or this reflects the sensitivity of RL on parameters, especially when using greedy action selection strategy.

Chapter 6

6 Conclusion

We developed a RL-based routing algorithm, RLSR-Routing, which works on SR-enabled SDN. We modified SARSA, an on-policy RL algorithm, by aggregating action selection and setting a dual rewards scheme. To handle potential packet loss or failure to reach the desired destination during the learning process, we added additional steps than the normal function to update Q-tables for failed actions. Experiment results indicate that RLSR-Routing can find user preferred paths for given traffic demand, based on customized QoS considerations. Our research contribution is listed as follows:

- RLSR-Routing directly applies RL in the path finding process, without a prior knowledge of the network. It does not rely on additional inputs such as pre-defined link weights; or a set of pre-calculated paths between a source and destination pair to find the user preferred path of a traffic demand.
- We further reduced the number of communications required between SDN controller and network data planes by exploiting Segment Routing, compared with previous RL-based routing in SDN.
- We ensured that no packet would be routed in a loop during the RLSR-Routing learning process.
- We gave users the ability to customize QoS weights to define what is their preferred path.
- Our proposed framework can reuse previously learned knowledge to speed up algorithm convergence.

6.1 Limitations

As we explained in section 4.4.2, action selection based on a greedy approach may not find the best path in some cases. Relative medium size networks and small number of traffic demands tested may be the reasons that we only observed user preferred final paths in experiment results. Another issue is that RLSR-Routing does not support finding paths for multiple paths in parallel. When working on a set of traffic demands, RLSR-Routing only

focuses on finding the best path for one traffic demand at a time, without considering the effect of placing traffic on the best path on other traffic demands that have not been assigned a path. To provide better demonstrations of paths assignments and links utilizations by RLSR-Routing and NR-Routing, we only used a subset of randomly generated demands during the comparative study. In practice, RLSR-Routing should assign paths for all traffic demands from the users of the network.

6.2 Future Work

As explained earlier, several upgrades can further improve RLSR-Routing's performance. For example, a better action selection policy, and more flexibility for users to decide how to apply a global Q-table to initiate a local Q-table. For the comparative study, we plan to test our RLSR-Routing in a more practical setting of traffic engineering problem, such as a 1000 nodes networks with thousands of traffic demands. In addition, future experiment should test RLSR-Routing's ability to find user preferred path in terms of other QoS parameters, such as link reliability, instead of just using link utilization and traffic intensity. Finally, we will modify the whole RL algorithm component so that it can compute the user preferred path for a traffic demand matrix in parallel.

References

[1] Cisco Annual Internet Report (2018-2023), CISCO Systems, 2020, [online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[2] J. F. Kurose and K. W. Ross, “Computer Networking: A Top-down approach, Seventh Edition.” Boston, MA, USA: Pearson. 2017.

[3] J. Moy, “RFC 2328: OSPF Version 2”, 1998.

[4] C. Hedrick, “RFC 1058: Routing Information Protocol”, 1988.

[5] Y. Guo, Z. Wang, X. Yin, X. Shi and J. Wu, "Traffic Engineering in SDN/OSPF Hybrid Network", 2014 IEEE 22nd International Conference on Network Protocols, pp. 563-568, 2014.

[6] Y. Wang, Z. Wang, and L. Zhang, “Internet Traffic Engineering without Full Mesh Overlaying”, Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society, pp. 565-571, vol. 1, 2001.

[7] M. Karakusa and A. Durresia, “Quality of service (QoS) in software defined networking (SDN): A survey”, Journal of Network and Computer Applications, pp. 200-218, 2017.

[8] R. Masoudi and A. Ghaffari, “Software defined networks: A survey”, Journal of Network and Computer Applications, 67, pp. 1-25, 2016.

[9] I. T. Haque and N. Abu-Ghazaleh, “Wireless Software Defined Networking: A Survey and Taxonomy”, IEEE Communications Surveys & Tutorials, vol. 18, no. 4, pp. 2713-2737, 2016.

[10] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, “RFC 8402: Segment Routing Architecture”, 2018.

[11] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardonay, and P. Francois, “The Segment Routing Architecture”, 2015 IEEE Global Communications Conference (GLOBECOM), pp. 106, 2015.

[12] S. Previdi, C. Filsfils, B. Decraene, S. Litkowski, M. Horneffer, R. Shakir, “RFC 7855: Source Packet Routing in Networking (SPRING) Problem Statement and Requirements”, 2016.

[13] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi et. al., “A comprehensive survey on machine learning for networking: evolution, applications and research opportunities”, Journal of Internet Services and Applications, 2018.

[14] D. M. C-Velasco, O. M. C. Rendon, and N. L. S. d. Fonseca, “Intelligent Routing Based on Reinforcement Learning for Software-Defined Networking”, IEEE Transactions on Network and Service Management, pp. 870-881, 2021.

[15] Z. Mammeri, “Reinforcement Learning Based Routing in Networks: Review and Classification of Approaches”, IEEE Access, vol. 7, pp. 55916-55950, 2019.

[16] F. Agostinelli, G. Hocquet, S. Singh, and P. Baldi, “From reinforcement learning to deep reinforcement learning: An overview”, Braverman readings in machine learning. key ideas from inception to current state, pp. 298-328, 2018.

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

[18] J. A. Boyan and M. L. Littman, “Packet Routing in Dynamically Changing Networks: A Reinforcement Learning Approach”, Advances in neural information processing systems, 1993.

[19] S. P.M. Choi and D.Y. Yeung, “Predictive Q-Routing: A Memory-based Reinforcement Learning Approach to Adaptive Traffic Control”, Advances in Neural Information Processing Systems 8, 1995.

[20] S. Kumar, and R. Miikkulainen, “Dual reinforcement Q-routing: An on-line adaptive routing algorithm”, Proceedings of the artificial neural networks in engineering Conference, pp. 231-238, 1997.

[21] R. Rudek, L. Koszalka and I. Pozniak-Koszalka, "Introduction to multi-agent modified Q-learning routing for computer networks," Advanced Industrial Conference on Telecommunications/Service Assurance with Partial and Intermittent Resources Conference/E-Learning on Telecommunications Workshop (AICT/SAPIR/ELETE'05), pp. 408-413, 2005.

[22] R. Sun, S. Tatsumi and G. Zhao, "Q-MAP: a novel multicast routing method in wireless ad hoc networks with multiagent reinforcement learning", 2002 IEEE Region 10 Conference on Computers, Communications, Control and Power Engineering. TENCOM '02. Proceedings., vol. 1, pp. 667-670, 2002.

[23] Yu-Han Chang, T. Ho and L. P. Kaelbling, "Mobilized ad-hoc networks: a reinforcement learning approach", Proceedings of the International Conference on Autonomic Computing (ICAC’04), pp. 240-247, 2004.

[24] P. Wang and T. Wang, "Adaptive Routing for Sensor Networks using Reinforcement Learning," The Sixth IEEE International Conference on Computer and Information Technology (CIT'06), pp. 219-219, 2006.

[25] D. Chen, and P. K. Varshney, “QoS Support in Wireless Sensor Networks: A Survey”, International conference on wireless networks, vol. 233, pp. 1-7, 2004.

[26] W. Sugeng, J.E. Istiyanto, K. Mustofa, and A. Ashari, “The Impact of QoS Changes towards Network Performance”, International Journal of Computer Networks and Communications Security, vol. 3, no. 2, pp. 48-53, 2015.

[27] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, et al., “Flowsense: Monitoring network utilization with zero measurement cost”, International Conference on Passive and Active Network Measurement, pp. 31-41, 2013.

[28] T. Hu and Y. Fei, "QELAR: A Machine-Learning-Based Adaptive Routing Protocol for Energy-Efficient and Lifetime-Extended Underwater Sensor Networks" IEEE Transactions on Mobile Computing, vol. 9, no. 6, pp. 796-809, 2010.

[29] S. Z. Jafarzadeh and M. H. Y. Moghaddam, "Design of Energy-aware QoS Routing Algorithm in Wireless Sensor Networks Using Reinforcement Learning", 2014 4th International Conference on Computer and Knowledge Engineering (ICCKE), pp. 722-727, 2014.

[30] B. Debowski, P. Spachos and S. Areibi, "Q-Learning Enhanced Gradient Based Routing for Balancing Energy Consumption in WSNs," IEEE 21st International Workshop on Computer Aided Modelling and Design of Communication Links and Networks (CAMAD), pp. 18-23, 2016.

[31] G. Santhi, A. Nachiappan, M. Z. Ibrahime, R. Raghunadhane and M. K. Favas, "Q-learning based adaptive QoS routing protocol for MANETs", International Conference on Recent Trends in Information Technology (ICRTIT), pp. 1233-1238, 2011.

[32] C. Wu, S. Ohzahata, and T. Kato, "Flexible, Portable, and Practicable Solution for Routing in VANETs: A Fuzzy Constraint Q-Learning Approach", IEEE Transactions on Vehicular Technology, vol. 62, no. 9, pp. 4251-4263, 2013.

[33] W. Jung, J. Yim, and Y. Ko, “QGeo: Q-Learning-Based Geographic Ad Hoc Routing Protocol for Unmanned Robotic Networks”, IEEE Communications Letters, vol. 21, no. 10, pp. 2258-2261, 2017.

[34] C. Wu, T. Yoshinaga, Y. Ji, and Y. Zhang, “Computational Intelligence Inspired Data Delivery for Vehicle-to-Roadside Communications”, IEEE Transactions on Vehicular Technology, vol. 67, no. 12, pp. 12038-12048, 2018.

[35] T. Hendriks, M. Camelo, and S. Latré, "Q2-Routing : A Qos-aware Q-Routing algorithm for Wireless Ad Hoc Networks", 14th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), pp. 108-115, 2018.

[36] F. Li, X. Song, H. Chen, X. Li and Y. Wang, "Hierarchical Routing for Vehicular Ad Hoc Networks via Reinforcement Learning", IEEE Transactions on Vehicular Technology, vol. 68, no. 2, pp. 1852-1865, 2019.

[37] C. Perkins, E. Belding-Royer, and S. Das, “RFC 3561: Ad hoc On-Demand Distance Vector (AODV) Routing”, 2003.

[38] C. Wu, S. Ohzahata and T. Kato, "Routing in VANETs: A fuzzy constraint Q-Learning approach", EEE Global Communications Conference (GLOBECOM), pp. 195-200, 2012.

[39] M. Nurchis, R. Bruno, M. Conti, and L. Lenzini, “A Self-Adaptive Routing Paradigm for Wireless Mesh Networks Based on Reinforcement Learning”, Proceedings of the 14th ACM international conference on Modeling, analysis and simulation of wireless and mobile systems (MSWiM '11), pp. 197-204, 2011.

[40] M. Boushaba, A. Hafid, A. Belbekkouche, and M. Gendreau, “Reinforcement learning based routing in wireless mesh networks”, Wireless Networks 19, pp. 2079-2091, 2013.

[41] K. Tang, C. Li, H. Xiong, J. Zou and P. Frossard, "Reinforcement Learning-Based Opportunistic Routing for Live Video Streaming over Multi-Hop Wireless Networks", IEEE 19th International Workshop on Multimedia Signal Processing (MMSP), pp. 1-6, 2017.

[42] 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", IEEE International Conference on Services Computing (SCC), pp. 25-33, 2016.

[43] Z. Yuan, P. Zhou, S. Wang, and X. Zhang, “Research on Routing Optimization of SDN Network Using Reinforcement Learning Method”, 2nd International Conference on Safety Produce Informatization (IICSPI), pp. 442-445, 2019.

[44] S. Yang, H. Shi, and H. Zhang, "Dynamic Load Balancing of Multiple Controller based on Intelligent Collaboration in SDN", International Conference on Computer Vision, Image and Deep Learning (CVIDL), pp. 354-359, 2020.

[45] S. Kim, J. Son, A. Talukder, and C. S. Hong, "Congestion prevention mechanism based on Q-leaning for efficient routing in SDN", International Conference on Information Networking (ICOIN), pp. 124-128, 2016.

[46] D. Tennakoon, S. Karunarathna, and B. Udugama, "Q-learning Approach for Load-balancing in Software Defined Networks", Moratuwa Engineering Research Conference (MERCon), pp. 1-6, 2018.

[47] J. Rischke, P. Sossalla, H. Salah, F. H. P. Fitzek, and M. Reisslein, "QR-SDN: Towards Reinforcement Learning States, Actions, and Rewards for Direct Flow Routing in Software-Defined Networks", IEEE Access, vol. 8, pp. 174773-174791, 2020.

[48] S. Sendra, A. Rego, J. Lloret, J. M. Jimenez, and O. Romero, "Including artificial intelligence in a routing protocol using Software Defined Networks", EEE International Conference on Communications Workshops (ICC Workshops), pp. 670-674, 2017.

[49] C. Wang, L. Zhang, Z. Li, and C. Jiang, "SDCoR: Software Defined Cognitive Routing for Internet of Vehicles", IEEE Internet of Things Journal, vol. 5, no. 5, pp. 3513-3520, 2018.

[50] 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.

[51] Y. Tian et al., "Traffic Engineering in Partially Deployed Segment Routing Over IPv6 Network With Deep Reinforcement Learning", IEEE/ACM Transactions on Networking, vol. 28, no. 4, pp. 1573-1586, 2020.

[52] P. Almasan, S. Xiao, X. Cheng, X. Shi, P. B. Ros and A. C. Aparicio, “ENERO: Efficient Real-Time WAN Routing Optimization with Deep Reinforcement Learning”, arXiv:2109.10883v2, 2021.

[53] Q. Fu, E. Sun, K. Meng, M. Li, and Y. Zhang, "Deep Q-Learning for Routing Schemes in SDN-Based Data Center Networks," in IEEE Access, vol. 8, pp. 103491-103499, 2020.

[54] R. E. Ali, B. Erman, E. Baştuğ and B. Cilli, "Hierarchical Deep Double Q-Routing", EEE International Conference on Communications (ICC), pp. 1-7, 2020.

[55] H. Hasselt, “Double Q-learning”, Advances in neural information processing systems (23), 2010.

Curriculum Vitae

Name:	Wumian Wang
Post-secondary Education and Degrees:	University of British Columbia Vancouver, British Columbia, Canada 2014-2019 B.Sc. (Microbiology & Immunology) The University of Western Ontario London, Ontario, Canada 2020-2022 M.Sc. (Computer Science)
Honours and Awards:	Western Graduate Research Scholarships 2021-2022
Related Work Experience	Teaching Assistant (CS-1033 Introduction to Multimedia and Communications) The University of Western Ontario 2022
Publications:	