Applications of Reinforcement Learning to Routing and Virtualization in Computer Networks

by

Soroush Haeri

B. Eng., Multimedia University, Malaysia, 2010

Dissertation Submitted in Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy

in the
School of Engineering Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Spring 2016

Approval

Name:	Soroush Haeri
Degree:	Doctor of Philosophy (Engineering Science)
Title:	*Applications of Reinforcement Learning to Routing and Virtualization in Computer Networks*
Examining Committee:	**Dr. Ivan V. Bajić** (chair)

Dr. Ljiljana Trajković
Senior Supervisor
Professor

Dr. R. H. Stephen Hardy
Supervisor
Professor Emeritus

Dr. Qianping Gu
Supervisor
Professor

Dr. Joseph G. Peters
Examiner
Professor

Dr. Azzedine Boukerche
External Examiner
Professor
School of Information
Technology and Engineering
University of Ottawa

Date Defended: March 17, 2016

Abstract

Computer networks and reinforcement learning algorithms have substantially advanced over the past decade. The Internet is a complex collection of inter-connected networks with a numerous of inter-operable technologies and protocols. Current trend to decouple the network intelligence from the network devices enabled by Software-Defined Networking (SDN) provides a centralized implementation of network intelligence. This offers great computational power and memory to network logic processing units where the network intelligence is implemented. Hence, reinforcement learning algorithms viable options for addressing a variety of computer networking challenges. In this dissertation, we propose two applications of reinforcement learning algorithms in computer networks.

We first investigate the applications of reinforcement learning for deflection routing in buffer-less networks. Deflection routing is employed to ameliorate packet loss caused by contention in buffer-less architectures such as optical burst-switched (OBS) networks. We present a framework that introduces intelligence to deflection routing (iDef). The iDef framework decouples design of the signaling infrastructure from the underlying learning algorithm. It is implemented in the ns-3 network simulator and is made publicly available. We propose the predictive Q-learning deflection routing (PQDR) algorithm that enables path recovery and reselection, which improves the decision making ability of the node in high load conditions. We also introduce the Node Degree Dependent (NDD) signaling algorithm. The complexity of the algorithm only depends on the degree of the node that is NDD compliant while the complexity of the currently available reinforcement learning-based deflection routing algorithms depends on the size of the network. Therefore, NDD is better suited for larger networks. Simulation results show that NDD-based deflection routing algorithms scale well with the size of the network and outperform the existing algorithms. We also propose a feed-forward neural network (NN) and a feed-forward neural network with episodic updates (ENN). They employ a single hidden layer and update their weights using an associative learning algorithm. Current reinforcement learning-based deflection routing algorithms employ Q-learning, which does not efficiently utilize the received feedback signals. We introduce the NN and ENN decision-making algorithms to address the deficiency of Q-learning. The NN-based deflection routing algorithms achieve better results than Q-learning-based algorithms in networks with low to moderate loads.

The second application of reinforcement learning that we consider in this dissertation is for modeling the Virtual Network Embedding (VNE) problem. We develop a VNE simulator (VNE-Sim) that is also made publicly available. We define a novel VNE objective function and prove its upper bound. We then formulate the VNE as a reinforcement learning problem using the Markov Decision Process (MDP) framework and then propose two algorithms (MaVEn-M and MaVEn-S) that employ Monte Carlo Tree Search (MCTS) for solving the VNE problem. In order to further improve the performance, we parallelize the algorithms by employing MCTS root parallelization. The advantage of the proposed algorithms is that, time permitting, they search for more profitable embeddings compared to the available algorithms that find only a single embedding solution. The simulation results show that proposed algorithms achieve superior performance.

Keywords: Computer networks; machine learning; reinforcement learning; deflection routing; virtual network embedding

For Inga, Baba, Maman, and Sina

Acknowledgements

Writing this dissertation would not have been possible without the intellectual and emotional contributions of the generous individuals I met throughout this journey.

I would like to thank my Senior Supervisor Dr. Ljiljana Trajković who dedicated countless hours of hard work for reviewing my works and guiding my research. She was very generous and encouraging to allow me to explore my ideas and I am truly grateful for the confidence she invested in me and my research. I would also like to thank my committee members Dr. Hardy, Dr. Gu, Dr. Peters, and Dr. Boukerche for reviewing my dissertation and providing constructive suggestions and comments.

I would like to thank Dr. Wilson Wang-Kit Thong. He piqued my interest in deflection routing and introduced me to the ns-3 tool. The ideas in the first portion of this dissertation were conceived during the time he was a visiting Ph.D. student at the Communication Networks Laboratory at Simon Fraser University.

I would like to express my gratitude to Marilyn Hay and Toby Wong of BCNET as well as Nabil Seddigh and Dr. Biswajit Nandy of Solana Networks, with whom I collaborated on industry-based projects. Although these projects are not part of this dissertation, many subjects I learned during our collaborations helped me develop my research skill.

I would like to thank my brother Dr. Sina Haeri who has always been a source of support and encouragement. His excellence in parallel programming helped me develop the parallel Monte-Carlo Tree Search that is used in this dissertation.

I would like to thank my Mom and Dad for their unconditional love and support throughout my life. Their support enabled me to study and pursue my educational goals.

I would also like to thank the scuba divers of Vancouver's Tec Club, especially John Nunes, Roger Sonnberger, and Eli Wolpin. Furthermore, I would like to extend my gratitude to Royse Jackson and Alan Johnson of the International Diving Center as well as Paul Quiggle of Vancouver's Diving Locker. Many ideas presented in this dissertation were conceived exploring the majestic waters of Vancouver.

I would like to extend my gratitude to my friends Majid Arianezhad, Alireza Jafarpour, Shima Nourbakhsh, Kenneth Fenech, and Luis Domingo Suarez Canales who have helped me and provided support in countless ways.

Last but not least, I would like to thank my lovely wife Inga. You supported me through the toughest times of my life. Without you, I would not have made it this far.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

BFS	Breadth First Search
BHP	Burst Header Packet
DEVS	Discrete Event System Specification
DfT	Deflection Time
DHC	Deflection Hop Counter
DN	Drop Notification
DrT	Drop Time
ENN-NDD	The Episodic Neural Network-based Node Degree Dependent Algorithm
FIFO	First In First Out
GRC	The Global Resource Capacity Algorithm
HMM	Hidden Markov Model
iDef	Intelligent Deflection Framework
InP	Infrastructure Provider
ISP	Internet Service Provider
MaVEn-M	Monte Carlo Virtual Network Embedding with Multimedia Flow
MaVEn-S	Monte Carlo Virtual Network Embedding with Shortest Path
MCF	Multicommodity Flow
MCTS	Monte Carlo Tree Search
MDP	Markov Decision Process
MIP	Mixed Integer Program

NDD	The Node Degree Dependent Algorithm
NN-NDD	The Neural Network-based Node Degree Dependent Algorithm
NSF	National Science Foundation
OBS	Optical Burst-Switched/Optical Burst Switching
OSPF	Open Shortest Path First
OXC	Optical Cross-Connect
PQDR	The Predictive Q-learning-based Deflection Routing Algorithm
PQR	Predictive Q-Routing
Q-NDD	The Q-learning-based Node Degree Dependent Algorithm
RLDRS	Reinforcement Learning-Based Deflection Routing Scheme
SDH	Synchronous Digital Hierarchy
SDN	Software Defined Networking
SONET	Synchronous Optical Network
SP	Service Provider
SVM	Support Vector Machine
TCP	Transport Control Protocol
TTT	Total Travel Time
UCT	Upper Confidence Bounds for Trees
VLiM	Virtual Link Mapping
VN	Virtual Network
VNE	Virtual Network Embedding
VNoM	Virtual Node Mapping
VNR	Virtual Network Request

Chapter 1

Introduction

Machine learning is a branch of artificial intelligence. Its focus is to improve machine behavior by analyzing examples of correct behavior rather than directly programming the machine. One application of machine learning algorithms is in situations when a precise specification of the desired behavior is unknown while examples of the desired behavior are available. Another application of machine learning algorithms is when the required behavior may vary over time or between users and, therefore, the system behavior may not be anticipated [31].

Supervised, unsupervised, and reinforcement learning are three major subcategories of machine learning algorithms. The goal of supervised learning algorithms is to learn a desired input-to-output mapping when samples of such mappings are available. Regression and various classification algorithms are examples of supervised learning algorithms. Unsupervised learning algorithms, in contrast, attempt to find regularities or certain patterns in a given input set. Clustering algorithms are examples of unsupervised learning algorithms [23]. Reinforcement learning algorithm attempt to learn a situation-to-action mapping with the main objective of maximizing a numerical reward. The learning agent is required to discover rewarding actions in various situations by executing trials [129].

Machine learning algorithms have not been used widely in computer networks. For example, various machine learning-based approaches proposed for routing in computer networks during the 1990's and early 2000's [41], [50], [109], [115] remained isolated and failed to receive much attention from the mainstream computer communication community. Machine learning algorithms, as a branch of machine intelligence algorithms, were mostly designed to solve problems with a high degree of complexity where the notion of probability and uncertainty plays an important role [119]. Therefore, such algorithms often required high computational power and/or large memory space. Early computer networks, in contrast, were mostly operated in controlled environments with a low degree of uncertainty involved [61]. As a result, the high level of complexity that machine learning and machine intelligence algorithms were designed to facilitate was not required in computer networks.

Traditionally, network intelligence was residing in the network nodes that were usually devices with rather limited computational power and memory. This made the network devices incapable of processing machine learning algorithms.

Computer networks have evolved over the years. The Internet, as we know it today, is a complex collection of inter-connected networks where a countless number of inter-operable technologies and protocols operate. The advent of Software-Defined Networking (SDN) and the current trend to decouple the network intelligence from the network devices enables a centralized implementation of the network intelligence unit. This centralized implementation may provide computational power and memory to the network logic processing units. Therefore, the increasing complexity of the computer networks and network processing entities, which have access to additional computational power and memory, make machine learning algorithms a viable option for solving computer networking problems.

Reinforcement learning algorithms have been proposed for deflection routing in optical burst-switching networks [33], [88]. Support Vector Machine (SVM), Hidden Markov Models (HMM), and Bayesian models have been employed to detect the Internet routing anomalies [21], [22]. HMM and Expectation Maximization have been used to identify the losses due to congestion and contention in optical burst-switching networks [85]. This identification may help improve the performance of Transport Control Protocol (TCP) in optical burst-switching networks. Support Vector Regression is used to predict TCP throughput [106].

Reinforcement learning algorithms enable development of self-improving agents that learn through interactions with their environment. This interaction-based learning capabilities is sought-after in computer networking because the issues that often arise in this area are required to be solved in real time based on interactions with the network.

In this Dissertation, we propose two applications of reinforcement learning algorithms in computer networks. We first investigate the applications of reinforcement learning for deflection routing in buffer-less networks. Deflection routing is employed to ameliorate packet loss caused by contention in buffer-less architectures such as optical burst-switched (OBS) networks. The main goal of deflection routing is to successfully deflect a packet based only on a limited knowledge that network nodes possess about their environment. We propose the *iDef* framework that introduces intelligence to deflection routing. The term "intelligence" refers to the class of machine intelligence algorithms. *iDef* decouples the design of the signaling infrastructure from the underlying learning algorithm. It consists of a signaling and a decision-making module. Signaling module implements a feedback management protocol while the decision-making module implements a reinforcement learning algorithm. *iDef* is compliant with the current Software-Defined Networking architecture that decouples the network intelligence from network devices. We also propose several learning-based deflection routing protocols, implement them in *iDef* using the ns-3 network simulator [17], and compare their performance.

We also consider applications of reinforcement learning for Virtual Network Embedding (VNE). Network virtualization helps overcome shortcomings of the current Internet architecture. The virtualized network architecture enables coexistence of multiple virtual networks on a physical infrastructure. The VNE problem, which deals with the embedding of virtual network components onto a physical network, is known to be NP-hard [151]. We formulate the VNE as a reinforcement learning problem and then propose two VNE algorithms: MaVEn-M and MaVEn-S. These algorithms formalize the Virtual Node Mapping (VNoM) problem by using the Markov Decision Process (MDP) framework and devise action policies (node mappings) for the proposed MDP using the Monte Carlo Tree Search (MCTS) algorithm. We develop a discrete event VNE simulator named *VNE-Sim* to implement and evaluate performance of MaVEn-M, MaVEn-S, and several recently proposed VNE algorithms.

1.1 Optical Burst-Switching and Deflection Routing

The Internet is an example of a complex network that has been extensively studied [99], [108], [127], [133]. Optical networks are part of the Internet infrastructure intended to carry high-bandwidth backbone traffic. Optical networks carry the majority of TCP/IP traffic in the Internet. Optical burst switching [117] combines the optical circuit switching and the optical packet switching paradigms. In optical burst-switched (OBS) networks, data are optically switched. Optical burst switching offers the reliability of the circuit switching technology and the statistical multiplexing provided by packet switching networks. Statistical multiplexing of bursty traffic enhances the network utilization. Various signaling protocols that have been proposed enable statistical resource sharing of a light-path among multiple traffic flows [27], [47].

Deflection routing is a viable contention resolution scheme that may be employed to ameliorate packet loss caused by contention in buffer-less architectures such as networks on chips or OBS networks. It was first introduced as "hot-potato" routing [30] because packets arriving at a node should be immediately forwarded [18], [104]. Contention occurs when according to a routing table, multiple arriving traffic flows at a node need to be routed through a single outgoing link. In this case, only one flow is routed through the optimal link defined by the routing table. In the absence of a contention resolution scheme, the remaining flows are discarded because the node possesses no buffers. Instead of buffering or discarding packets, deflection routing helps to temporarily deflect them away from the path that is prescribed by the routing table.

Deflection routing may benefit from the random nature of reinforcement learning algorithms. A deflection routing algorithm coexists in the network along with an underlying routing protocol that usually generates a significant number of control signals. Therefore, it is desired that deflection routing protocols generate few control signals. Reinforcement

learning algorithms enable a deflection routing protocol to generate viable deflection decisions by adding a degree of randomness to the decision-making process.

1.2 Virtual Network Embedding

The best-effort service, supported by the current Internet architecture, is not well-suited for all applications. A significant barrier to innovation has been imposed by the inability of the current Internet architecture to support a diverse array of applications [134]. The great success of the Internet has increased its ubiquity that, consequently, has lead to various challenges that the current Internet architecture is unable to address [25]. Network virtualization overcomes these shortcomings [62], [134]. The virtualized network model divides the role of Internet Service Providers (ISPs) into two independent entities: Infrastructure Providers (InPs) and Service Providers (SPs). The InPs manage the physical infrastructure while the SPs aggregate resources from multiple InPs into multiple Virtual Networks (VNs) to provide end-to-end services [52], [62].

In the virtualized network architecture, an InP owns and operates a *substrate network* composed of physical nodes and links that are interconnected in an arbitrary topology. Combinations of the substrate network nodes and links may be used to embed various virtualized networks. Virtual networks that are embedded in a substrate network are isolated thus enabling end-to-end service provisioning without requiring unified protocols, applications, or control and management planes [35].

An InP's revenue depends on the resource utilization within the substrate network that, in turn, depends on the efficiency of the algorithm that allocates the substrate network resources to virtual networks [52]. This resource allocation is known as the virtual network embedding (VNE) [159], which may be formulated as a mixed-integer program (MIP) [51] or may be reduced to the multiway separator problem [24], [151]. Both problems are $\mathcal{NP}$-hard making the VNE problem is also $\mathcal{NP}$-hard. This is one of the main challenges in network virtualization.

MIPs have been employed to solve the VNE problem [40], [51], [79]. The R-Vine and D-Vine algorithms use a rounding-based approach to attain a linear programming relaxation of the MIP that corresponds to the VNE problem [51]. Their objective is to minimize the cost of accommodating the Virtual Network Requests (VNRs). Node-ranking-based algorithms are among the most recent approaches to solve the VNE [48], [69], [157]. This family of algorithms computes a score/rank for substrate and virtual nodes based on various heuristics. Then, using the computed rank, a *large-to-large and small-to-small* [48] mapping scheme is employed to map the virtual nodes to substrate nodes. The Global Resource Capacity (GRC) [69] is among the most recent node-ranking-based algorithms that outperforms the earlier similar algorithms. Subgraph isomorphism detection [96], particle swarm optimization [158], and ant colony optimization [59] are among other employed methods.

1.3 Roadmap

The reminder of this Dissertation is organized as follows.

In Chapter 2, we present a brief survey of reinforcement learning algorithms. We also describe the Q-learning algorithm and applications of feed-forward neural networks for reinforcement learning. We then present MDP as a framework for modeling decision-making problems, present two exact algorithms for solving MDPs and their computational complexity, and finally describe the MCTS algorithm as a promising approach for finding *near-optimal* solutions for large MDPs. Most algorithms and approaches discussed in Chapter 2 are employed in this Dissertation.

In Chapter 3, we describe the OBS technology and introduce deflection routing as a viable scheme to reduce burst loss probability in OBS networks. We then introduce the *iDef* framework for implementing reinforcement learning-based deflection routing algorithms and propose the Predictive Q-learning-based deflection routing algorithm (PQDR). We then propose the Node Degree Dependent (NDD) signaling algorithm whose complexity depends on a node degree rather than the size of the network. We combine the NDD signalling algorithm with two feed-forward neural network-based reinforcement learning algorithms into two deflection routing algorithms: *NN-NDD* and *ENN-NDD*. The feed forward neural network of *NN-NDD* operates on single episodes while *ENN-NDD* operates based on episodic updates. In the remainder of Chapter 3, we compare the performance of the algorithms using the National Science Foundation (NSF) network topology [11], [105] as well as using larger synthetic network topologies that are generated by the Waxman [140] algorithm.

In Chapter 4, we first present the VNE problem and its objective function, establish its upper bound, and introduce profitability as a performance metric. Some of the known VNE algorithms are then presented in detail. We propose an MDP formulation of the VNoM problem and introduce two MaVEn algorithms that utilize MCTS for finding optimal action policies for the proposed MDP. In order to improve the performance of the MaVEn algorithms by employing MCTS root parallelization [44], [45]. We then evaluate the performance of the MaVEn algorithms and compare them with the existing VNE algorithms using a synthesized network topology as well as BCube [71] and Fat-Tree [20], [92] data center network topologies. We conclude the Chapter with discussions.

In Chapter 5, a brief user's manual of the virtual network embedding simulator *VNE-Sim* is presented. The overall structure of the simulator is first described including a list of external libraries that are required by *VNE-Sim*. Various packages that *VNE-Sim* comprises and their dependencies are then described. Finally, we describe in details the content of the simulator's *core* package that defines the basic operation and classes required for simulating a virtual network embedding algorithm.

Finally, we conclude this Dissertation with Chapter 6.

Chapter 2

Reinforcement Learning

Reinforcement learning algorithms perform situation-to-action mappings with the main objective to maximize numerical rewards. These algorithms may be employed by agents that learn to interact with a dynamic environment through trial-and-error [86]. Reinforcement learning encompasses three abstract events irrespective of the learning algorithm: 1) an agent observes the state of the environment and selects an appropriate action; 2) the environment generates a reinforcement signal and transmits it to the agent; 3) the agent employs the reinforcement signal to improve its subsequent decisions. Therefore, a reinforcement learning agent requires information about the state of the environment, reinforcement signals from the environment, and a learning algorithm.

Some of the reinforcement leaning algorithms presented in this Chapter are used in this Dissertation. Q-learning [138] introduced in Section 2.1 is a simple table-based reinforcement learning algorithm. It is used in Chapter 3 for designing reinforcement learning-based deflection routing algorithms. Feed-forward neural networks and the REINFORCE algorithm [141], which are presented in Section 2.2, are also used in Chapter 3 for designing the NN-NDD and ENN-NDD deflection routing algorithms. In Section 2.3, we first present Markov Decision Processes (MDPs) [116], [130] and exact algorithms for solving them. We then present the Monte Carlo Tree Search (MCTS) [53], [89] algorithm as an approach for solving large MDPs that are computationally challenging. We then present two MCTS parallelization techniques: root and leaf parallelizations. In Chapter 4, MDP is used to formulate the Virtual Network Embedding (VNE) problem as a reinforcement learning process. For realistic cases, the size of the MDP that corresponds to the VNE problem is prohibitively large. Hence, exact methods are not applicable for solving such MDPs. Therefore, we employ MCTS to solve the proposed MDP. Root parallelization is also employed in Chapter 2.3 to enhance the MCTS solution.

2.1 Q-Learning

Q-learning [138] is a simple reinforcement learning algorithm that has been employed for path selection in computer networks. The algorithm maintains a Q-value $Q(s, a)$ in a Q-table for every state-action pair. Let s_t and a_t denote the state and the action executed by an agent at a time instant t, respectively. Furthermore, let r_{t+1} denote the reinforcement signal that the environment has generated for performing action a_t in the state s_t. When the agent receives the reward r_{t+1}, it updates the Q-value that corresponds to the state s_t and action a_t as:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \times \Big[r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\Big], \tag{2.1}$$

where $0 < \alpha \leq 1$ is the learning rate and $0 \leq \gamma < 1$ is the discount factor.

Q-learning has been considered as an approach for generating routing policies. The Q-routing algorithm [41] requires that nodes make their routing decisions locally. Each node learns a local deterministic routing policy using the Q-learning algorithm. Generating the routing policies locally is computationally less intensive than finding a global routing solution. However, the Q-routing algorithm does not generate an optimal routing policy in networks with low traffic loads nor does it learn new optimal policies in cases when network load decreases. Predictive Q-routing [50] addresses these shortcomings by recording the best experiences learned, which may then be reused to predict traffic behavior.

2.2 Feed-Forward Neural Networks for Reinforcement Learning

In a neural network of learning agents with weighted interconnections, the following steps are executed for each decision made:

- neural network receives an input from its environment;
- the input propagates through the network and it is mapped to an output;
- the output is fed to the environment for evaluation;
- environment evaluates the output signal and generates a reinforcement signal r;
- based on a learning algorithm, each neural network unit modifies its weights using the received reinforcement signal.

Let w_{ij} denote the weight of the connection between agents i and j. The behavior of the agent i is defined by a vector $\mathbf{w}_i$ that contains the weights of all connections to the ith agent. The agent i receives an input vector $\mathbf{x}_i$ from the environment and/or other agents

and then maps the input to an output y_i according to a probability distribution function $g_i(\zeta, \mathbf{w}_i, \mathbf{x}_i) = \Pr\{y_i = \zeta \mid \mathbf{w}_i, \mathbf{x}_i\}$ [141]. In case of binary agents where the input is mapped to either 0 or 1 with probabilities p_i and $1 - p_i$, respectively, g_i is defined as a Bernoulli semilinear unit:

$$g_i(\zeta, \mathbf{w}_i, \mathbf{x}_i) = \begin{cases} 1 - p_i & if \quad \zeta = 0 \\ p_i & if \quad \zeta = 1 \end{cases}, \tag{2.2}$$

where

$$\begin{aligned} p_i &= f_i(\mathbf{w}_i^T \mathbf{x}_i) \\ &= f_i\Big(\sum_j w_{ij} x_j\Big). \end{aligned} \tag{2.3}$$

Function f_i is a differentiable squashing function. A commonly used function is the logistic map:

$$f_i(\sum_j w_{ij} x_j) = \frac{1}{1 + e^{-\sum_j w_{ij} x_{ij}}}, \tag{2.4}$$

where x_{ij} is the jth element of the input vector $\mathbf{x}_i$. Agents that use Bernoulli semilinear unit along with the logistic function are called *Bernoulli-logistic* units.

Upon receiving a feedback signal r, the neural network updates its weights. The update rule for Bernoulli-logistic units suggested by the REINFORCE algorithm [141] updates the weights w_{ij} as:

$$\Delta w_{ij} = \alpha r (y_i - p_i) x_{ij}, \tag{2.5}$$

where α is a non-negative rate factor, y_i is the output of the ith agent, p_i is the probability of $y_i = 1$ given the input vector $\mathbf{x}_i$ and weight vector $\mathbf{w}_i$, and x_{ij} is the jth element of the input vector $\mathbf{x}_i$. Let matrix $\mathbf{W}$ denote the collection of weights that determines the behavior of an arbitrary feed-forward network. Under the assumption that the environment's inputs to the network and the received reinforcement signals for any input/output pair are determined by stationary distributions, (2.5) maximizes the expected value $\mathcal{E}$ of the reinforcement signal r, given the weight matrix $\mathbf{W}$:

$$\mathcal{E}\{r \mid \mathbf{W}\}. \tag{2.6}$$

The REINFORCE algorithm utilizes an episodic update rule in order to operate in environments where delivery of the reinforcement signals has unknown delays. A k-episode is defined when a network selects k actions between two consecutive reinforcement receptions. The k-episode is terminated upon receiving a reinforcement signal. The REINFORCE algorithm suggests that a network may update its weighs at the end of the k-episode as:

$$\Delta w_{ij} = \alpha r \sum_{l=1}^{k} [(y_i(l) - p_i(l)) x_j(l-1)]. \tag{2.7}$$

2.3 Markov Decision Process

MDP may be used to model a sequential decision-making problem. A discrete time MDP $\mathcal{M}$ is a quintuple $(\mathcal{T}, \Phi, \mathcal{A}, R, P)$, where $\mathcal{T}$ is the set of decision-making instances, Φ is the state space, $\mathcal{A}$ is the action space, $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function that assigns real-valued rewards to state-action pairs, and $P : \Phi \times \mathcal{A} \times \Phi \to [0, 1]$ is a transition probability distribution. Therefore, $R(\phi_t = \phi, a_t = a)$ is the reward for performing action a in state ϕ and

$$P(\phi', a, \phi) = \Pr(\phi_{t+1} = \phi' | a_t = a, \phi_t = \phi) \sim P \tag{2.8}$$

is the probability of a transition to state ϕ' when selecting action a in state ϕ. A state ρ is called a *terminal* state if $P(\rho, a, \rho) = 1$. We denote the reward of entering a terminal state by R_ρ. An MDP $\mathcal{M}$ is called *episodic* if it possesses a terminal state [130].

The behavior of a decision-making agent may be defined by its policy π for selecting actions. The overall return of a given policy π may be calculated as the terminal reward R_ρ plus the discounted sum of all step rewards:

$$\mathcal{R}^\pi = R^\pi_\rho + \sum_{t=0}^{T} \gamma^t R^\pi_{\phi_t}, \tag{2.9}$$

where T defines the decision-making *horizon* and $0 \leq \gamma \leq 1$ is a discount factor. If $T < \infty$, the MDP is called *finite-horizon* while $T \to \infty$ defines an *infinite-horizon* MDP. The goal of the decision-making agent is to find policy π^* that maximizes the expected cumulative rewards given an initial state ϕ_1 calculated as:

$$\mathcal{E}(\pi|\phi) = R^\pi_\phi + \gamma \sum_{\phi' \in \Phi} P(\phi', \pi(\phi), \phi) \mathcal{E}(\pi|\phi') \qquad \forall \phi \in \Phi, \tag{2.10}$$

where $\pi(\phi)$ is the preferred action in state ϕ according to the policy π [116], [130].

2.3.1 Solution of Markov Decision Processes: The Exact Algorithms

Linear programming and iterative methods are employed to find solutions of MDPs. The two major iterative algorithms that are widely used are the policy and value iteration algorithms.

Linear Programming

Finding an optimal reward function may be formulated as a linear program [56]. Linear programming is a general technique and, therefore, it does not take advantage of the special structure of MDPs. The *primal* linear program may be defined as:

Objective:

$$\text{maximize} \sum_{\phi \in \Phi} v_\phi \tag{2.11}$$

Constraints:

$$v_\phi \leq \mathcal{R}^a_\phi + \gamma \sum_{\phi' \in \Phi} P(\phi', a, \phi) v_{\phi'} \qquad \forall \phi \in \Phi,\ \forall a \in \mathcal{A}, \tag{2.12}$$

where $v_\phi,\ \forall \phi \in \Phi$ are the linear program variables that determine the optimal reward function of the original MDP.

The Policy Iteration Algorithm

This iterative algorithm is executed in two-stages: value determination and policy improvement [80]. In the value determination phase, current policy is evaluated. Then in the policy improvement phase, an attempt is made to improve the current policy. The computational complexity of the value determination phase is $\mathcal{O}(|\Phi|^3)$ while the complexity of the policy improvement phase is $\mathcal{O}(|\mathcal{A}||\Phi|^2)$, where $|\Phi|$ denotes the size of state space and $|\mathcal{A}|$ denotes the size of the action space [97].

The pseudocode of the policy iteration method is shown in Algorithm 1. The value of every improved policy is strictly larger than the previous one. Furthermore, there are $|\mathcal{A}|^{|\Phi|}$ distinct policies for an MDP with $|\Phi|$ states and $|\mathcal{A}|$ actions. Hence, termination of the algorithm is guaranteed in at most an exponential number of steps [97], [116].

Algorithm 1 Pseudocode of the policy iteration algorithm

procedure PolicyIteration
  Select a random policy π'
  $\pi \leftarrow \emptyset$
  while $\pi \neq \pi'$ **do**
    $\pi \leftarrow \pi'$
    $\forall \phi \in \Phi$ Calculate $E(\pi|\phi)$ by solving $|\Phi| \times |\Phi|$ system of linear equations (2.10)
    for all $\phi \in \Phi$ **do**
      if $\exists a \in \mathcal{A}$ such that $\left[R^a_\phi + \sum_{\phi' \in \Phi} P(\phi', a, \phi) E(\pi|\phi')\right] < E(\pi|\phi)$ **then**
        $\pi'(\phi) = a$
      else
        $\pi'(\phi) = \pi(\phi)$
      end if
    end for
  end while
  return π
end procedure

The Value Iteration Algorithm

This is a successive approximation algorithm [36] . The optimal value function is computed by successively expanding the horizon. This calculation method is guaranteed to converge to the optimal value function as the horizon is expanded. The policy that is associated with

the value function will also converge to the optimal policy in finite number of iterations [37], [97].

The value iteration algorithm is shown in Algorithm 2. The maximum number of iterations *max_itr* may be set in advance or an appropriate stopping condition may be employed. A stopping condition that guarantees an ϵ-optimal policy may be achieved [142] by terminating the main loop when:

$$\max_{\phi \in \Phi} \left| E^n(\phi) - E^{n-1}(\phi) \right| < \frac{\epsilon(1-\gamma)}{2\gamma}. \tag{2.13}$$

The computational complexity of each iteration of the value iteration algorithm is $\mathcal{O}(|\mathcal{A}||\Phi|^2)$. Hence, the total computational time is polynomial if total number of required iterations is polynomial [97].

Algorithm 2 Pseudocode of the value iteration algorithm. The maximum number of iterations *max_itr* may either be set in advance or an appropriate stopping rule may be employed.

procedure VALUEITERATION
  Initialize $E^0(\phi) \ \ \forall \phi \in \Phi$
  $n \leftarrow 1$
  while $n < max\_itr$ **do**
    for all $\phi \in \Phi$ **do**
      for all $a \in \mathcal{A}$ **do**
        $E^n(\phi, a) \Big[R^a_\phi + \sum_{\phi' \in \Phi} P(\phi', a, \phi) E^{n-1}(\phi') \Big]$
      end for
      $E^n(\phi) = \max_{a \in \mathcal{A}} E^n(\phi, a)$
    end for
    $n \leftarrow n + 1$
  end while
  for all do$\phi in \Phi$
    $\pi(\phi) = \operatorname{argmax}_{a \in \mathcal{A}} E^n(\phi, a)$
  end for
  return π
end procedure

2.3.2 Solution of Large Markov Decision Processes and Monte Carlo Tree Search

The size of the MDP state space grows exponentially with the number of state variables. Complexity of the exact algorithms for solving MDP is polynomial in the size of the state space [97]. Therefore, finding exact solutions for MDPs with large number of state variables is intractable. Solving such MDPs often involves finding a *near-optimal* solution.

Various approaches have been proposed for finding near-optimal solutions of large MDPs [26], [87], [126], [129]. Recent approaches [26], [126] are based on the Monte Carlo Tree Search (MCTS) [53], [89]. They assume that the decision-making agent has access to a generative model $\mathcal{G}$ of the MDP. The model is capable of generating samples of successor

states and rewards given a state and an action [126]. The agent uses the model $\mathcal{G}$ to perform a sampling-based look-ahead search [87]. MCTS builds a sparse search tree and selects actions using Monte Carlo samplings. These actions are used to deepen the tree in the most promising direction [26].

The MCTS algorithm is shown in Algorithm 3. It begins with a tree that only consists of the root node. It then executes four phases until a predefined computational budget β is exhausted:

Algorithm 3 Pseudocode of the MCTS algorithm

```
1: procedure MCTS(root)
2:     while β > 0 do
3:         current_node ← root
4:         while current_node ∈ search_tree do                ▷ Tree Traversal Phase
5:             last_node ← current_node
6:             current_node ← SELECT(current_node)
7:         end while
8:         last_node ← EXPANDlast_node                        ▷ A node is added to the search tree
9:         Reward ← SIMULATE(last_node)                       ▷ A simulated game is played
10:        current_node ← last_node                           ▷ The Reward is backpropagated
11:        while current_node ∈ search_tree do
12:            BACKPROPAGATIO(current_node, Reward)
13:            current_node ← current_node.parent
14:        end while
15:        β ← β − 1
16:    end while
17:    return The action that leads to the root's child with highest Reward
18: end procedure
```

1. *Selection*: The tree is traversed from the root until a non-terminal leaf node is reached. At each level of the tree, a child node is selected based on a *selection strategy*. A selection strategy may be exploratory or exploitative. An exploratory strategy probes the undiscovered sections of the search tree to find better actions while an exploitative strategy focuses on the promising subtrees that have already been discovered. The *exploration vs. exploitation* trade-off [101] must be considered when employing a selection strategy [86].

   The Upper Confidence Bounds for Trees (UCT) [89] is one of the most commonly used selection strategies. Let u denote the current node of the search tree and $\mathcal{I}$ the set of all its children. Furthermore, let v_i and σ_i denote the value and visit count of a node i. UCT selects a child κ from:

$$\kappa \in \arg\max_{i \in \mathcal{I}} \left(\frac{v_i}{\sigma_i} + D\sqrt{\frac{\ln \sigma_u}{\sigma_i}}\right), \tag{2.14}$$

   where D is an exploration constant that determines the balance between exploration and exploitation. If $D = 0$, the selection strategy is strictly exploitative.

   The MCTS algorithm was originally introduced to design intelligent computer-based Go players [53]. Go is a two-player board game with possible outcomes: win, draw, or

loss that may be encoded as 1, 0, or -1, respectively. It is considered as one of the most challenging games to be played by computer agents. It has more than 10^{170} states and up to 361 legal moves [68]. Value of nodes in a search tree of a two-player game belong to the interval $[-1, 1]$. In contrast, in single-player puzzle games, the objective of the player is not to win the game but rather to achieve a high score. Therefore, the values of search tree nodes in such games belong to the interval $[0, Max\_Score]$, where Max_Score is the maximum possible score in the game. For example, the maximum score in the single-player puzzle *SameGame* may be as high as $6,000$ [122].

The UCT selection strategy (2.14) does not consider possible deviations in the values of children nodes. In a two-player game, this deviation does not play an important role because the node values are in $[-1, 1]$ and the deviation from the mean is small. The deviation becomes more important in single-player games. Single-Player MCTS (SP-MCTS) [122] is a variant of MCTS that has been proposed for solving single-player puzzles. It introduces a *deviation* term to UCT (2.14). Hence:

$$\kappa \in \arg\max_{i \in \mathcal{I}} \left(\frac{v_i}{\sigma_i} + D\sqrt{\frac{\ln \sigma_u}{\sigma_i}} + \sqrt{\frac{\sum r_i^2 - \sigma_i v_i + C}{\sigma_i}}\right) \tag{2.15}$$

is used as the selection strategy, where r_i^2 is the sum of the squared rewards that the i^{th} child node has received so far and C is a large positive constant.

2. *Expansion*: After a non-terminal leaf node is selected, one or more of its successors are added to the tree. The most common *expansion strategy* is to add one node for every execution of the four MCTS phases. The new node corresponds to the next state [53].

3. *Simulation*: From the given state of the non-terminal node that has been selected, a sequence of actions is performed until a terminal state is reached. Even though MCTS converges with randomly selected actions [89], utilizing domain knowledge may improve the convergence time [26].

4. *Backpropagation*: The reward is calculated after a terminal state is reached in the *Simulation* phase. This reward is then propagated from the terminal node to the root. Every tree node in the current trajectory is updated by adding the reward to its current value v and incrementing its count σ.

The computational budget β may be defined as the number of evaluated action samples per selection cycle. After repeating the four phases of MCTS β times, the child of the root with the highest average value is selected as the optimal action. The MDP then enters its next state and the selected child is chosen to be the new root of the search tree.

2.3.3 Parallel Monte Carlo Tree Search

There are various available techniques for parallelizing the MCTS algorithm. We consider a symmetric multiprocessor (SMP) system as the platform for parallelization. Since the memory is shared in such platforms, mutual exclusions (mutex) should be employed in order to avoid corruption of the search tree when multiple threads attempt to access and modify the search tree during the 1st, 2nd, and 4th phases of the MCTS algorithm. The simulation phase (3rd phase) of the MCTS algorithm does not require any information from the search tree. Hence, the simulations may be executed independently without any mutex requirements [45]. Root and leaf parallelizations [44] are the common techniques that do not require any mutex. Hence, they are simpler to implement and may be implemented on distributed memory architectures such as clusters. Root and leaf parallelization techniques are shown in Fig. 2.1 left and right, respectively.

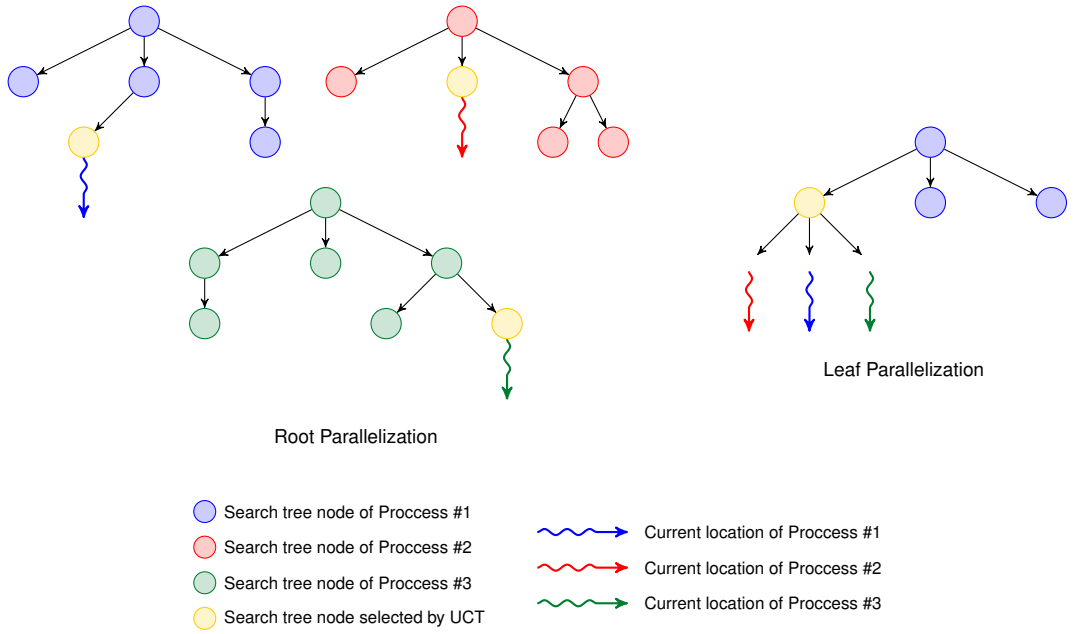

Figure 2.1: Root and leaf Monte Carlo Tree Search parallelizations.

In *root parallelization*, each process creates its own search tree and the processes do not share information. Each process should be initialized with a unique random number generator seed to ensure that the constructed search trees are not identical. When the pre-allocated time for simulations (β) is over. Each process communicates to the master process the value v and count σ of the children of its root. The master process then calculates the best action based on the information it receives from other processes [44].

In *leaf parallelization*, only one process traverses and adds nodes to the tree (1st and 2nd MCTS phases). When the master process reaches a leaf node, it distributes the simulation tasks among the available processes. Each process executes the 3rd MCTS phase in isolation.

When the simulations are completed, the master process collects the outcomes from other processes and propagates the results up the tree to the root (4th MCTS phase) [44].

Root parallelization requires less coordination and communication between the processes compared to the leaf parallelization. Furthermore, performance evaluations have shown that root parallelization produces superior results [44], [45].

Chapter 3

Reinforcement Learning-Based Deflection Routing in Buffer-Less Networks

Reinforcement learning-based algorithms were proposed in the early days of the Internet to generate routing policies [41], [50], [115]. Q-learning [138] is a reinforcement learning algorithm that has been employed for generating routing policies. The Q-routing algorithm [41] requires that nodes locally make their routing decisions. Each node learns a local deterministic routing policy using the Q-learning algorithm. Generating the routing policies locally is computationally less intensive. However, the Q-routing algorithm does not generate an optimal routing policy in networks with low loads nor does it learn new optimal policies in cases when network load decreases. The Predictive Q-routing algorithm [50] addresses these shortcomings by recording the best experiences learned that may then be reused to predict traffic behavior. Packet routing algorithms in large networks such as the Internet have to consider the business relationships between Internet Service Providers. Therefore, in such environment, randomness is not a desired property of a routing algorithm. Consequently, Internet routing algorithms have not employed reinforcement learning because of its inherent randomness.

In this Chapter, we present related applications of reinforcement learning algorithms for deflection routing in OBS networks [74]–[78]. The contributions are:

1. We develop a framework named iDef that simplifies implementation and testing of deflection routing protocols by employing a modular architecture. The ns-3 [17] implementation of iDef is made publicly available [16].

2. The predictive Q-learning deflection routing (PQDR) [74] algorithm that combines the learning algorithm of the predictive Q-routing [50] and the signaling algorithm of the Reinforcement Learning-Based Deflection Routing Scheme (RLDRS) [33] is

proposed. PQDR enables a node to recover and reselect paths that have small Q-values as a results of a link failure or congestion. This improves the decision making ability of the node in high load conditions. When deflecting a traffic flow, the PQDR algorithm stores in a Q-table the accumulated reward for each deflection decision. It also recovers and reselects decisions that are not well rewarded and have not been used over a period of time.

3. We introduce the novel Node Degree Dependent (NDD) signaling algorithm [75]. The complexity of the algorithm only depends on the degree of the node that is NDD compliant while the complexity of the other currently available reinforcement learning-based deflection routing algorithms depends on the size of the network. Therefore, NDD is better suited for larger networks. Simulation results show that NDD-based deflection routing algorithms scale well with the size of the network and perform better than the existing algorithms. Furthermore, the NDD signaling algorithm generates fewer control signals compared to the existing algorithms.

4. For the decision-making, we propose a feed-forward neural network (NN) and a feed-forward neural network with episodic updates (ENN) [78]. They employ a single hidden layer neural network that updates its weights using an associative learning algorithm. Currently available reinforcement learning-based deflection routing algorithms employ Q-learning, which does not utilize efficiently the gathered feedback signals. NN and ENN decision-making algorithms address the deficiency of Q-learning by introducing a single hidden layer NN to generate deflection decisions. The NN-based deflection routing algorithms achieve better results than Q-learning-based algorithms in networks with low to moderate loads. Efficiently utilizing control signals in such cases is important because the number of deflections is small and a deflection routing algorithm receives fewer feedback signals.

The algorithms presented in this Chapter are summarized in Table 3.1.

Table 3.1: Summary of the presented algorithms.

Deflection routing algorithm	Signaling algorithm	Signaling algorithm complexity	Learning algorithm
NN-NDD [78]	NDD	Degree of a node	Neural network-based using REINFORCE algorithm [141]
ENN-NDD [78]	NDD	Degree of a node	Neural network-based with episodic updates using REINFORCE algorithm [141]
Q-NDD [75]	NDD	Degree of a node	Q-Learning
PQDR [74]	RLDRS	Size of the network	Predictive Q-Learning
RLDRS [33]	RLDRS	Size of the network	Q-Learning

The remainder of this Chapter is organized as follows. In Section 3.1, we describe bufferless network architectures and contention in such networks. In Section 3.2, we introduce

deflection routing as a contention resolution scheme and provide a brief survey of work related to applications of reinforcement learning in deflection routing. We present the iDef framework in Section 3.3. The PQDR algorithm is then introduced in Section 3.4. We present the NDD signaling algorithm in Section 3.5. Designs of the NN and ENN decision-making modules are presented in Section 3.6. The performance of the algorithms is evaluated in Section 3.8. We conclude this Chapter with discussions in Section 3.9.

3.1 Buffer-Less Architecture, Optical Burst Switching, and Contention

Nodes in buffer-less networks do not posses memory (buffer) to store packets. Buffers are usually implemented as first-in-first-out (FIFO) queues that are used to store packets contending to be forwarded to the same outgoing link. Examples of buffer-less architectures are OBS networks and network-on-chips. OBS is a technology designed to share optical fiber resources across data networks [117]. Other optical switching technologies for data communication such as Synchronous Optical Network (SONET) and Synchronous Digital Hierarchy (SDH) [114] reserve the entire light-path from a source to a destination. Even though a light-path is not fully utilized, it may not be shared unless its reservation is explicitly released. The OBS technology overcomes these limitations. Switching in OBS networks is performed optically, allowing the optical/electrical/optical conversions to be eliminated in the data plane. This permits high capacity switching with simpler switching architecture and lower power consumption [146].

3.1.1 Optical Burst Switching and Burst Traffic

In OBS networks, data are optically switched. At an ingress node of such a network, multiple packets are aggregated into one optical burst. Before transmitting the burst, a burst header packet (BHP) is created and sent ahead of the burst with an offset time t_{offset}. The BHP contains information needed to perform OBS switching and IP routing, such as the burst length t_{length}, offset time t_{offset}, and the destination IP address. When an OBS node receives a BHP, it has t_{offset} time to locate the outgoing link l_o in the routing table, reserve the link for the burst to pass through, and reconfigure the optical cross-connect (OXC) module that connects the incoming and the outgoing links. The duration of t_{offset} should be selected based on the processing time of the routing and switching algorithms. In simulations scenarios presented in this dissertation, we do not consider the effect of t_{offset} on the performance of the algorithms.

Simulation of computer networks requires adequate models of network topologies as well as traffic patterns. Traffic measurements help characterize network traffic and are basis for developing traffic models. They are also used to evaluate performance of network protocols

and applications. Traffic analysis provides information about the network usage and helps understand the behavior of network users. Furthermore, traffic prediction is important to assess future network capacity requirements used to plan future network developments.

It has been widely accepted that Poisson traffic model that was historically used to model traffic in telephone networks is inadequate to capture qualitative properties of modern packet networks that carry voice, data, image, and video applications [112]. Statistical processes emanating from traffic data collected from various applications indicate that traffic carried by the Internet is self-similar in nature [93]. Self-similarity implies a "fractal-like" behavior and that data on various time scales have similar patterns. Implications of such behavior are: no natural length of bursts, bursts exist across many time scales, traffic does not become "smoother" when aggregated, and traffic becomes more bursty and more self-similar as the traffic volume increases. This behavior is unlike Poisson traffic models where aggregating many traffic flows leads to a white noise effect.

A traffic burst consists of a number of aggregated packets addressed to the same destination. Assembling multiple packets into bursts may result in different statistical characteristics compared to the input packet traffic. Short-range burst traffic characteristics include distribution of burst size and burst inter-arrival time. Two types of burst assembly algorithms may be deployed in OBS networks: time-based and burst length-based. In time-based algorithms, burst inter-arrival times are constant and predefined. In this case, it has been observed that the distribution of burst lengths approaches a Gamma distribution that reaches a Gaussian distribution when the number of packets in a burst is large. With a burst length-based assembly algorithms, the packet size and burst length are predetermined and the burst inter-arrival time is Gaussian distributed. Long-range traffic characteristics deal with correlation structures of traffic over large time scales. It has been reported that long-range dependency of incoming traffic will not change after packets are assembled into bursts, irrespective of the traffic load [152].

A Poisson-Pareto burst process has been proposed [19], [160] to model the Internet traffic in optical networks. It may be used to predict performance of optical networks [75]. The inter-arrival times between adjacent bursts are exponentially distributed (Poisson) while the burst durations are assumed to be independent and identically distributed Pareto random variables. Pareto distributed burst durations capture the long-range dependent traffic characteristics. Poisson-Pareto burst process has been used to fit the mean, variance, and the Hurst parameter of measured traffic data and thus match the first order and second order statistics. The Hurst parameter H expresses the speed of decay of the autocorrelation function of a time series. Self-Similar series with long-range dependencies have Hurst parameters in the range $0.5 < H < 1$. The degree of self-similarity and long-range dependency increases as H increases [55], [93].

Traffic modeling affects evaluation of OBS network performance. The effect of the arrival traffic statistics depends on the time scale. Short time scales greatly influence the

behavior of buffer-less high-speed networks. Self-similarity does not affect blocking probability even if the offered traffic is long-range dependent over large time scales. Poisson approximation of the burst arrivals provides an upper bound for blocking probability [84]. Furthermore, assuming Poisson arrival processes introduces errors that are shown to be within acceptable limits [154]. Hence, we adopt this approach commonly accepted in the literature and consider Poisson inter-arrival times. The impact of traffic self-similarity on the performance of the algorithms presented in this dissertation remains an open research topic.

3.1.2 Contention in Optical Burst-Switched Networks

Consider an arbitrary OBS network shown in Fig. 3.1, where $\mathcal{N} = \{x_1, x_2, \ldots, x_n\}$ denotes the set of all network nodes. Node x_i routes the incoming traffic flows f_1 and f_2 to the destination nodes x_{d_1} and x_{d_2}, respectively. According to the shortest path routing table stored in x_i, both flows f_1 and f_2 should be forwarded to node x_j via the outgoing link l_0. In this case, the flows f_1 and f_2 are contending for the outgoing link l_0. The node x_i forwards flow f_1 through l_0 to the destination x_{d1}. However, in the absence of a contention resolution scheme, the flow f_2 is discarded because node x_i is unable to buffer it. Deflection routing [154] is a contention resolution scheme that may be employed to reduce packet loss. Contention between the flows f_1 and f_2 is resolved by routing f_1 through the preferred outgoing link l_0 while routing f_2 through alternate outgoing link $l \in \mathcal{L}\backslash\{l_0\}$, where the set $\mathcal{L}$ denotes the set of all outgoing links connected to x_1. Various other methods have been proposed to resolve contention. Wavelength conversion [98], fiber delay lines [136], and control packet buffering [57] are among the contention resolution schemes that are applicable to optical networks.

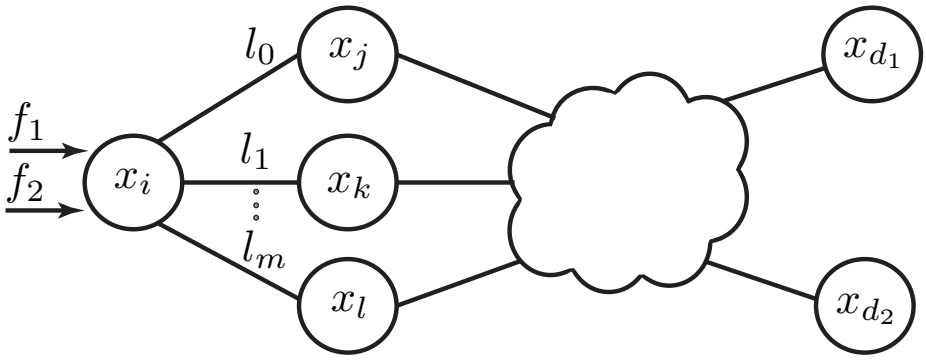

Figure 3.1: A network with buffer-less nodes.

When employing deflection routing, two routing protocols should operate simultaneously: an underlying routing protocol such as the Open Shortest Path First (OSPF) that primarily routes packets and a deflection routing algorithm that only deflects packets in case of a contention.

Slotted and unslotted deflection schemes were compared [39], [49] and performance of a simple random deflection algorithm and loss rates of deflected data were analyzed [38], [70], [144], [155]. Integrations of deflection routing with wavelength conversion and fiber delay lines were also proposed [149], [154]. Deflection routing algorithms generate deflection decisions based on a deflection set, which includes all alternate links available for deflection. Several algorithms have been proposed to populate large deflection sets while ensuring no routing loops [83], [147].

3.2 Deflection Routing by Reinforcement Learning

Deflection routing has attracted significant attention as a viable method to resolve contention in buffer-less networks [42], [94], [95]. Studies show that multiple deflections may significantly improve utilization in networks with multiple alternate routes such as fully meshed networks [144]. Performance evaluation of various deflection routing algorithms using scale-free Barabási-Albert [29] topologies, which resemble the autonomous system-level view of the Internet, shows that deflection routing effectively reduces the burst loss probability in optical networks [76].

Deflection routing algorithms generate deflection decisions based on a deflection set $\mathcal{D}$, which includes all viable alternate links available for deflection. The size of the set $\mathcal{D}$ determines the flexibility of deflection module in resolving contention. One approach includes all available links in the set $\mathcal{D}$, which would maximize the flexibility while possibly introducing routing loops. Another approach includes only outgoing links that lie on the shortest path, which avoids routing loops while reducing the flexibility in decision-making [38]. Several algorithms have been proposed to populate large deflection sets $\mathcal{D}$ while ensuring no routing loops [83], [147]. The proposed NDD signaling algorithm begins by including all outgoing links in the deflection set $\mathcal{D}$. As time progresses and deflection module receives feedback signals, the probability of selecting alternate links that may result in routing loops decreases.

Performance analysis of deflection routing based on random decisions shows that random deflection may effectively reduce blocking probability and jitter in networks with light traffic loads [131]. Advanced deflection routing algorithms improve the quality of decision-making by enabling neighboring nodes to exchange traffic information [33], [66], [88], [113], [132]. This information provides to the deflection module an integral neighborhood view for decision-making. Heuristic methods may then be employed to make deflection decisions based on the collected traffic information. Reinforcement learning provides a systematic framework for processing the gathered information. Various deflection routing protocols based on reinforcement learning [33], [88] employ the Q-learning algorithm or its variants.

Enhancing a node in buffer-less networks with a reinforcement learning agent that generates deflection decisions requires three components: function that maps a collection of

the environment variables to an integer (state); decision-making algorithm that selects an action based on the state; and signaling mechanism for sending, receiving, and interpreting the feedback signals. The decision-making instances in deflection routing are intermittent. Hence, a learning algorithm need not consider the effect of all possible future system trajectories for making its decisions. Agents that do not consider future system trajectories for decision-making are known as immediate reward or associative learning agents.

The Q-learning Path Selection algorithm [88] calculates a priori a set of candidate paths $P = \{p_1, \ldots, p_m\}$ for tuples (s_i, s_j), where $s_i, s_j \in S$ and $S = \{s_1, \ldots, s_n\}$ denotes the set of all edge nodes in the network. The Q-table stored in the i^{th} edge node maintains a Q-value for every tuple (s_j, p_k), where $s_j \in S \backslash \{s_i\}$ and $p_k \in P$. The sets S and P are states and actions, respectively. The Q-value is updated after each decision is made and the score of the path is reduced or increased depending on the received rewards. The algorithm does not specify a signaling method or a procedure for handling the feedback signals. RLDRS [33] also employs the Q-learning algorithm for deflection routing. The advantages of RLDRS are its precise signaling and rewarding procedures.

The Q-learning Path Selection algorithm and RLDRS have two drawbacks: they are not scalable and their underlying learning algorithms are inefficient.

Scalability

Q-learning Path Selection algorithm and RLDRS are not scalable because their space complexity depends on the network size. For example, the size of the Q-table generated by the Q-learning Path Selection algorithm [88] depends on the size of the network and the set of candidate paths. Therefore, it may be infeasible to store a large Q-table emanating from a large network.

Learning Deficiency

Q-learning is the only learning algorithm that has been employed for the deflection routing [33], [88]. Q-learning has two deficiencies:

1. Decisions that repeatedly receive poor rewards have low Q-values. Certain poor rewards might be due to a transient link failure or congestion and, hence, recovery and reselection of such decisions may improve the decision-making. Q-learning-based deflection routing algorithms do not provide a procedure to reselect the paths that have low Q-values as a consequence of transient network conditions [50];

2. Even though Q-learning guarantees eventual convergence to an optimal policy when finding the best action set, it does not efficiently use the gathered data. Therefore, it requires gaining additional experience to achieve good results [86].

The proposed NDD signaling algorithm addresses the scalability issue of the current reinforcement learning-based deflection routing algorithms. Its complexity depends only on a node degree. The NN and ENN decision-making modules are introduced to address the learning deficiency of Q-learning. They learn based on the REINFORCE associative learning algorithm [141], [143] that utilizes the gained experience more efficiently than Q-learning.

3.3 The iDef Framework

The proposed iDef framework is designed to facilitate development of reinforcement learning-based deflection routing protocols. We implemented the iDef framework in the ns-3 network simulator. In iDef, a reinforcement learning-based deflection routing algorithm is abstracted using mapping, decision-making, and signaling modules. iDef is designed to minimize the dependency among its modules. This minimal dependency enables implementation of portable deflection routing protocols within the iDef framework, which enables modules to be replaced without changing the entire design. For example, replacing the decision-making module requires no changes in the implemented signaling module. iDef components are shown in Fig. 3.2.

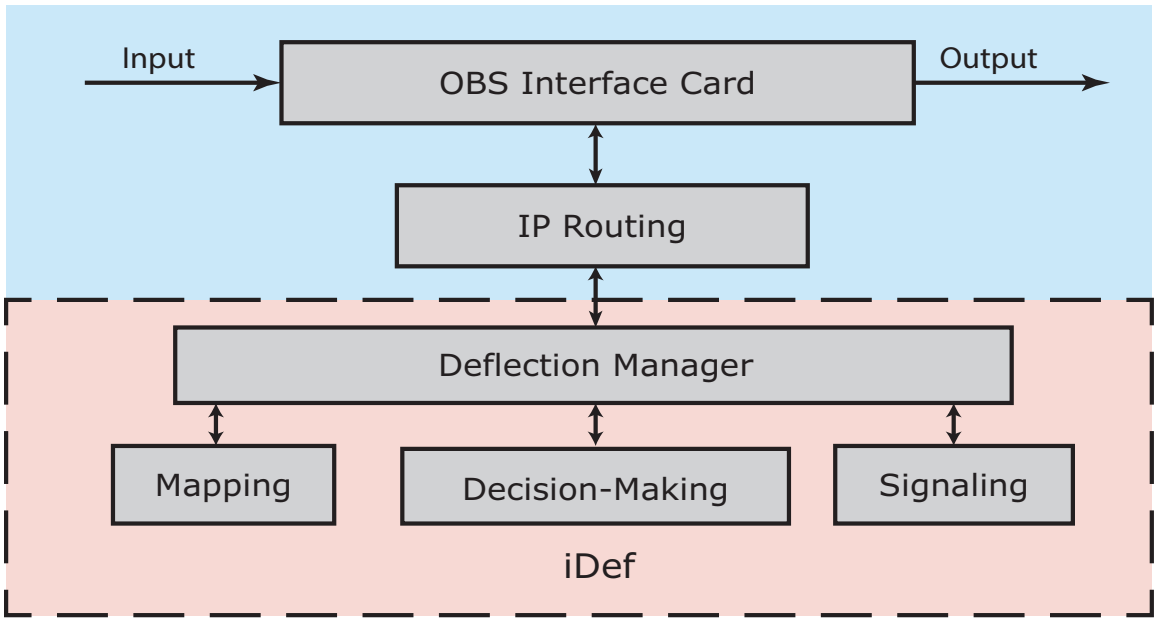

Figure 3.2: iDef building blocks: The iDef is composed of deflection manager, mapping, signaling, and decision-making modules. The deflection manager module coordinates the communication between modules. Its purpose is to remove dependencies among modules.

A deflection manager glues together the iDef modules. It has access to the OBS network interface cards and the IP routing table. The deflection manager makes iDef portable

by eliminating communication among mapping, decision-making, and signaling modules. The burst header messages received by a node are passed to the deflection manager. The deflection manager inspects the IP routing table for the next hop and then checks the status of the optical interfaces. If the desired optical interface is available, the optical cross-connects are configured according to the path defined by the IP routing table. If the interface is busy, the deflection manager passes the environment variables such as destination of the burst, output links blocking state, and the next hop on the shortest path, to the mapping module.

The *mapping module* maps all or a subset of the received variables to an integer called the *state*. For example, in the proposed NDD signaling algorithm, one possibility of such mapping is to append the binary ID of the port number obtained from the routing table to a string of 0s and 1s that represents the outgoing links status. This binary string may then be converted to an integer.

The *decision-making module* implements the learning algorithm. Therefore, the statistics, the history, and other required information for decision-making are stored in this module. It implements two main functions that are used by the deflection manager: a function that generates actions given a state and a function that updates the statistics when a reinforcement signal is received. The mapped state is passed to the decision-making module where an alternate output link (action) is selected for the burst. The generated decisions are then passed to the deflection manager. The signaling module passes the received reinforcement signals to the decision-making module where they are used for statistic updates.

The *signaling module* adds header fields to the deflected bursts. It also inspects the burst headers received by the deflection manager for special deflection header fields and tags. It assembles and sends feedback messages when required. Upon receiving a feedback message, the signaling module interprets the reinforcement signal and translates it to a scalar reward. This reward is then used by the deflection manager to enhance the decision-making module.

3.4 Predictive Q-Learning-Based Deflection Routing Algorithm

In this Section, we present details of the predictive Q-learning deflection routing algorithm (PQDR) [74]. PQDR determines an optimal output link to deflect traffic flows when contention occurs. The algorithm combines the predictive Q-routing (PQR) algorithm [50] and RLDRS [33] to optimally deflect contending flows. When deflecting a traffic flow, the PQDR algorithm stores in a Q-table the accumulated reward for every deflection decision. It also recovers and reselects decisions that are not well rewarded and have not been used over a period of time.

An arbitrary buffer-less network is shown in Fig. 3.1. $\mathcal{N} = \{x_1, x_2, \ldots, x_n\}$ denotes the set of all network nodes. Assume that each node possesses a shortest path routing table and a module that implements the PQDR algorithm to generate deflection decisions. Consider an arbitrary node x_i that is connected to its m neighbors through a set of outgoing links $\mathcal{L} = \{l_0, l_1, \ldots, l_m\}$. Node x_i routes the incoming traffic flows f_1 and f_2 to the destination nodes x_{d_1} and x_{d_2}, respectively. According to the shortest path routing table stored in x_i, both flows f_1 and f_2 should be forwarded to node x_j via the outgoing link l_0. In this case, node x_i forwards flow f_1 through l_0 to the destination x_{d1}. However, flow f_2 is deflected because node x_i is unable to buffer it. Hence, node x_i employs the PQDR algorithm to select an alternate outgoing link from the set $\mathcal{L}\backslash\{l_0\}$ to deflect flow f_2. It maintains five tables that are used by PQDR to generate deflection decisions. Four of these tables store statistics for every destination $x \in \mathcal{N}\backslash\{x_i\}$ and outgoing link $l \in \mathcal{L}$:

1. $Q_{x_i}(x, l)$ stores the accumulated rewards that x_i receives for deflecting packets to destinations x via outgoing links l.

2. $B_{x_i}(x, l)$ stores the minimum Q-values that x_i has calculated for deflecting packets to destinations x via outgoing links l.

3. $R_{x_i}(x, l)$ stores recovery rates for decisions to deflect packets to destinations x via outgoing links l.

4. $U_{x_i}(x, l)$ stores the time instant when x_i last updated the (x, l) entry of its Q-table after receiving a reward.

Each node stores a record for each of the $n-1$ other nodes in the network. Hence, the size of each table is $m \times (n-1)$, where m and n are the number of elements in the sets $\mathcal{L}$ and $\mathcal{N}$, respectively. The fifth table $P_{x_i}(l)$ records the blocking probabilities of the outgoing links connected to the node x_i. A time window τ is defined for each node. Within each window, the node counts the successfully transmitted packets λ_{l_i} and the discarded packets ω_{l_i} on every outgoing link $l_i \in \mathcal{L}$. When a window expires, node x_i updates entries in its P_{x_i} table as:

$$P_{x_i}(l_i) = \begin{cases} \dfrac{\omega_{l_i}}{\lambda_{l_i} + \omega_{l_i}} & \lambda_{l_i} + \omega_{l_i} > 0 \\ 0 & \text{otherwise} \end{cases} . \tag{3.1}$$

The PQDR algorithm needs to know the destination node x_{d_2} of the flow f_2 in order to generate a deflection decision. For every outgoing link $l_i \in \mathcal{L}$, the algorithm first calculates a Δt value as:

$$\Delta t = t_c - U_{x_i}(x_{d_2}, l_i), \tag{3.2}$$

where t_c represents the current time and $U_{x_i}(x_{d_2}, l_i)$ is the last time instant when x_i had received a feedback signal as a result of selecting the outgoing link l_i for deflecting a traffic

flow that is destined for node x_{d_2}. The algorithm then calculates $Q'_{x_i}(x_{d_2}, l_i)$ as:

$$Q'_{x_i}(x_{d_2}, l_i) = \max\Big(Q_{x_i}(x_{d_2}, l_i) + \Delta t \times R_{x_i}(x_{d_2}, l_i), B_{x_i}(x_{d_2}, l_i)\Big). \tag{3.3}$$

$Qx_i(x_{d_2}, l_i)$ is then used to generate the deflection decision (action) ζ:

$$\zeta \leftarrow \arg\min_{l_i \in \mathcal{L}}\{Q'_{x_i}(x_{d_2}, l_i)\}. \tag{3.4}$$

The deflection decision ζ is the index of the outgoing link of node x_i that may be used to deflect the flow f_2. Let us assume that $\zeta = l_1$ and, therefore, node x_i deflects the traffic flow f_2 via l_1 to its neighbor x_k. When the neighboring node x_k receives the deflected flow f_2, it either uses its routing table or the PQDR algorithm to forward the flow to its destination through one of its neighbors (x_l). Node x_k then calculates a feedback value ν and sends it back to node x_i that had initiated the deflection:

$$\nu = Q_{x_k}(x_{d_2}, l_{kl}) \times D(x_k, x_l, x_{d_2}), \tag{3.5}$$

where l_{kl} is the link that connects x_k and x_l, $Q_{x_k}(x_{d_2}, l_{kl})$ is the (x_{d_2}, l_{kl}) entry in x_k's Q-table, and $D(x_k, x_l, x_{d_2})$ is the number of hops from x_k to the destination x_{d_2} through the node x_l. Node x_i receives the feedback ν for its action ζ from its neighbor x_k and then calculates the reward r:

$$r = \frac{\nu \times (1 - P_{x_i}(\zeta))}{D(x_i, x_k, x_{d_2})}, \tag{3.6}$$

where $D(x_i, x_k, x_{d_2})$ is the number of hops from x_i to the destination x_{d_2} through x_k while $P_{x_i}(\zeta)$ is the entry in the x_i's link blocking probability table P_{x_i} that corresponds to the outgoing link ζ (l_1). The reward r is then used by the x_i's PQDR module to update the (x_{d_2}, ζ) entries in the Q_{x_i}, B_{x_i}, and R_{x_i} tables. The PQDR algorithm first calculates the difference ϕ between the reward r and $Q_{x_i}(x_{d_2}, \zeta)$:

$$\phi = r - Q_{x_i}(x_{d_2}, \zeta). \tag{3.7}$$

The Q-table is then updated using ϕ as:

$$Q_{x_i}(x_{d_2}, \zeta) = Q_{x_i}(x_{d_2}, \zeta) + \alpha \times \phi, \tag{3.8}$$

where $0 < \alpha \leq 1$ is the learning rate. Table B_{x_i} keeps the minimum Q-values and, hence, its (x_{d_2}, ζ) entry is updated as:

$$B_{x_i}(x_{d_2}, \zeta) = \min(B_{x_i}(x_{d_2}, \zeta), Q_{x_i}(x_{d_2}, \zeta)). \tag{3.9}$$

Table R_{x_i} is updated as:

$$R_{x_i}(x_{d_2},\zeta) = \begin{cases} R_{x_i}(x_{d_2},\zeta) + \beta \dfrac{\phi}{t_c - U_{x_i}(x_{d_2},\zeta)} & \phi < 0 \\ \gamma R_{x_i}(x_{d_2},\zeta) & \text{otherwise} \end{cases}, \tag{3.10}$$

where t_c denotes the current time and $0 < \beta \leq 1$ and $0 < \gamma \leq 1$ are recovery learning and decay rates, respectively. Finally, the PQDR algorithm updates table U_{x_i} with current time t_c as:

$$U_{x_i}(x_{d_2},\zeta) = t_c. \tag{3.11}$$

Signaling algorithms implemented in RLDRS and PQDR are similar. Their main difference is in the learning algorithm. RLDRS uses the Q-learning algorithm and, therefore, it only stores a Q-table $Q_{x_i}(x,l)$ that records the accumulated rewards that the node x_i receives for deflecting packets to destinations x via outgoing links l. As a result, a deflection decision ζ is generated using only the Q-table. Hence, instead of (3.3) and (3.4), RLDRS generates a deflection decision using:

$$\zeta \leftarrow \arg\max_{l_i \in \mathcal{L}} \{Q_{x_i}(x_{d_2}, l_i)\}. \tag{3.12}$$

3.5 The Node Degree Dependent Signaling Algorithm

We describe here the proposed NDD [75] signaling algorithm and the messages that need to be exchanged in order to enhance a buffer-less node with a decision-making ability. The NDD algorithm provides a signaling infrastructure that nodes require in order to learn and then optimally deflect packets in a buffer-less network. It is a distributed algorithm where nodes make deflection decisions locally.

The flowchart of the signaling algorithm is shown in Fig. 3.3. We consider a buffer-less network with $|\mathcal{N}|$ nodes. All nodes are iDef compatible and, hence, possess mapping, decision-making, and signaling modules. The headers of the packets received by a node are passed to the signaling module. The module inspects the routing table for the next hop and then checks the status of the network interfaces. If the desired interface is available, the packet is routed according to the path defined by the routing table. If the interface is busy and the packet has not been deflected earlier by any other node, the current states of the network interfaces and the output port obtained from the routing table are passed to the mapping module. The mapping module maps these inputs to a unique representation known as the system *state*. It then passes this information (*state*) to the decision-making module. The *state* representation depends on the underlying learning algorithm. Therefore, it may change based on the design of the decision-making module. Various decision-making modules require specifically designed compatible mapping modules. For example, Q-learning-based decision-making module requires a mapping module that transforms the

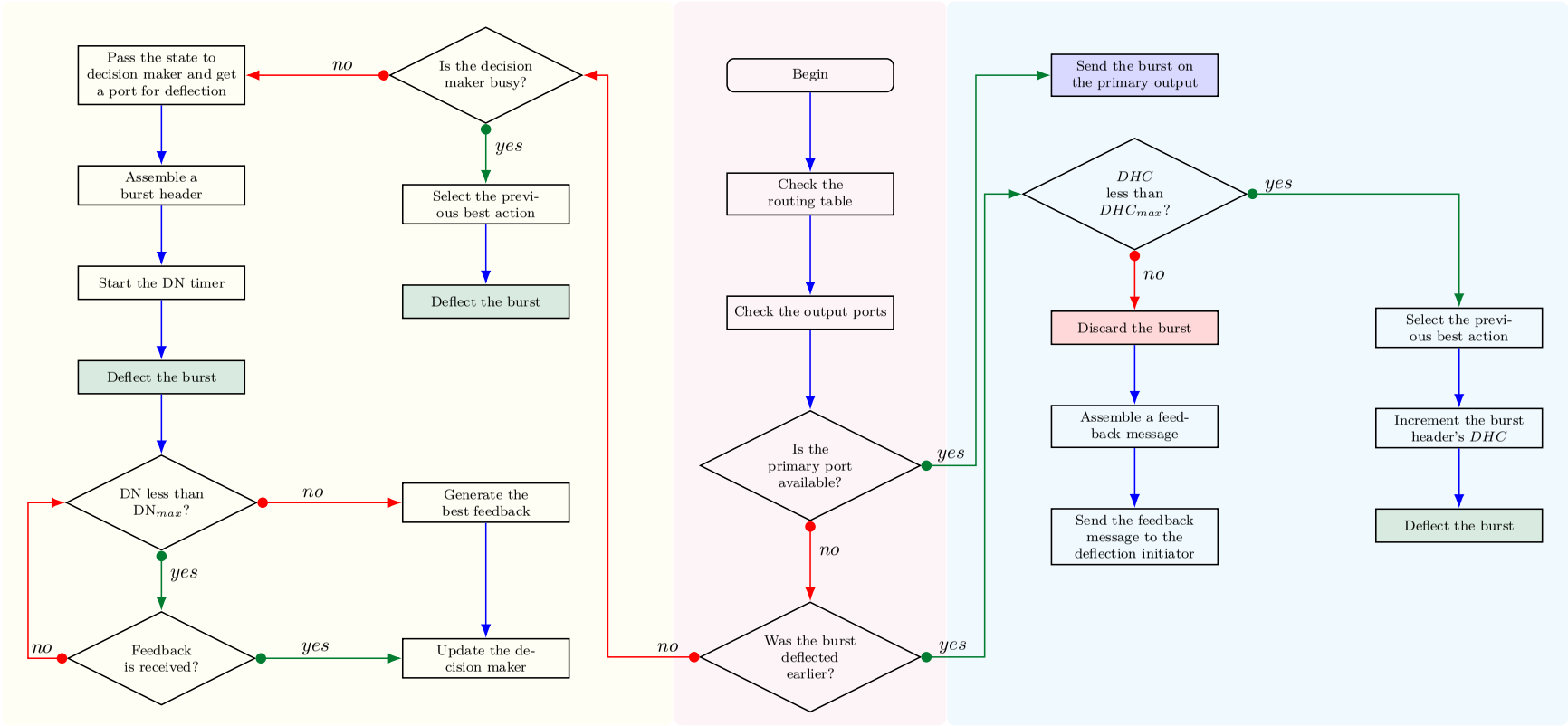

Figure 3.3: The flowchart of the proposed signaling algorithm. The DN timer denotes the drop notification timer. Nodes wait for feedback signals until this timer reaches DHC_{max}. DHC denotes the deflection hop counter. This counter is a field in the burst header that is incremented by one each time the burst is deflected. DHC_{max} is set in order to control the volume of deflected traffic. A burst is discarded when its DHC value reaches the maximum.

current states of the network interfaces and the output port suggested by the routing table to a real number while NN and ENN-based decision-making modules require binary vectors. The mapping module maps the states of the network interfaces to an ordered string of 0s and 1s, where idle and busy interfaces are denoted by 0 and 1, respectively.

The decision-making module returns the index of the best network interface for deflecting the packet based on the current *state*. The signaling module adds the following information to the packet header: a unique *ID* number used to identify the feedback message that pertains to a deflection; the address of the node that initiated the deflection, to be used by other nodes as the destination for the feedback messages; a deflection hop counter *DHC*, which is incremented each time other nodes deflect the packet.

When a packet is to be deflected at a node for the first time, the node records the current time as the deflection time *DfT* along with the *ID* assigned to the packet. The Drop Notification *DN* timer is initiated and the packet is deflected to the network interface that is selected by the decision-making module. The maximum value of the DN timer is set to DN_{max}, which indicates expiration of the timer. The purpose of the timer is to reduce the number of feedback signals.

After a deflection decision is made, the decision-making module waits for the feedback. It makes no new decisions during this idle interval. The deflected packet is discarded when either it reaches a fully congested node or its *DHC* reaches the maximum permissible number of deflections DHC_{max}.

The node that discards the deflected packet assembles a feedback message consisting of the packet *ID*, *DHC*, and the time instant *DrT* when the packet was discarded. The feedback message is then sent to the node that initiated the deflection.

When the node that initiated the deflection receives the feedback message, it calculates the total travel time TTT that the packet has spent in the network after the first deflection:

$$TTT = DrT - DfT. \tag{3.13}$$

The TTT and *DHC* values are used by the decision-making module to update its statistics. If no feedback message is received until the *DN* timer expires, the node assumes that the packet has arrived successfully to its destination. The node may then update its decision-making module with the reinforcement signal having $TTT = 0$ and $DHC = 0$. A decreasing function with the global maximum at $(0, 0)$ is used as a reward function to map TTT and *DHC* to a real value r.

A node records the best action selected by the decision-making module. These records are used if a node needs to deflect a packet that has been deflected earlier or during an idle interval.

In order to reduce the excess traffic generated by the number of feedback messages, a node receives feedback messages only when it deflects packets that have not been deflected earlier. Hence, only deflecting such packets enhances the node's decision-making ability.

3.6 Neural Networks for Deflection Routing

In this Section, we propose the feed-forward neural networks that is used for generating deflection decisions. The neural network decision-making module implements a network of interconnected learning agents and uses the REINFORCE algorithm [141] described in Section 2.2.

3.6.1 Feed-Forward Neural Networks for Deflection Routing with Single-Episode Updates

We propose a single-hidden-layer feed-forward neural network for deflection routing because fewer number of algebraic operations are required when the neural network selects actions and updates its weights. Furthermore, such neural networks have reasonable learning capabilities [81], [82]. The proposed single hidden-layer neural network achieves acceptable performance and, hence, we did not consider larger neural networks. The feed-forward neural network for generating deflection decisions is composed of the input, middle, and output layers, as shown in Fig. 3.4.

Consider a buffer-less node with n outgoing links. The input layer of such a node consists of two partitions denoted by binary vectors $\mathbf{I}^l = [i_1^l \ \ldots \ i_n^l]$ and $\mathbf{I}^d = [i_1^d \ \ldots \ i_n^d]$. If the k^{th} outgoing link of the node is blocked, i_k^l is set to 1. It is 0 otherwise. If the burst that is to be deflected contends for the j^{th} outgoing link of the node, the j^{th} entry of the input vector $\mathbf{I}^d$ is set to 1 while the remaining elements are set to 0. The $\mathbf{I}^l$ partition of the input has weighted connections to the output layer. The $n \times n$ matrix of wights $\mathbf{W}^{lo}$ is defined as:

$$\mathbf{W}^{lo} = \begin{bmatrix} w_{11}^{lo} & 0 & \cdots & 0 \\ 0 & w_{22}^{lo} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & w_{nn}^{lo} \end{bmatrix}. \tag{3.14}$$

Selecting a link that is blocked for deflection should be avoided because it results in an immediate packet loss. Let us assume that the ith outgoing link of the buffer-less node is selected as the preferred link for deflection when the output y_i of the feed-forward neural network is 1. If Bernoulli-logistic agents are employed, selecting a blocked link for deflection may be prohibited by setting the elements w_{kk}^{lo}, $k = 1, \ldots, n$, of the weight matrix $\mathbf{W}^{lo}$ to a large negative value. Regardless of the state of the buffer-less network and its behavior, selecting a blocked link will always result in immediate packet drop. This behavior does

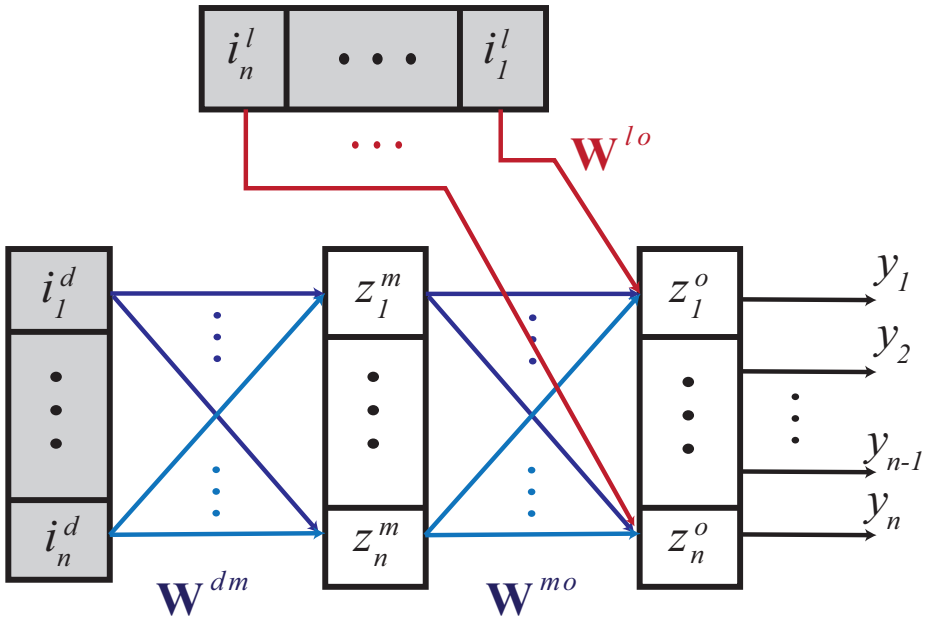

Figure 3.4: The proposed design of the feed-forward neural network for deflection routing. The input layer consists of two partitions denoted by binary vectors $\mathbf{I}^l = [i_1^l \ \ldots \ i_n^l]$ and $\mathbf{I}^d = [i_d^l \ \ldots \ i_n^d]$. The $\mathbf{I}^l$ partition of the input has weighted connections to the output layer. The binary vector $\mathbf{Z}^m = [z_1^m \ \ldots \ z_n^m]$ denotes the mid-layer of the proposed feed-forward neural network while $\mathbf{Z}^o$ denotes the output layer.

not change with time and does not depend on the reinforcement signals received from the environment. Hence, $\mathbf{W}^{lo}$ is a deterministic weight matrix that is not updated when reinforcement signals are received.

Let the binary vector $\mathbf{Z}^m = [z_1^m \ \ldots \ z_n^m]$ denote the mid-layer of the proposed feed-forward neural network shown in Fig. 3.4. This layer is connected to the input and the output layers using the weight matrices:

$$\mathbf{W}^{dm} = \begin{bmatrix} w_{11}^{dm} & w_{12}^{dm} & \cdots & w_{1n}^{dm} \\ w_{21}^{dm} & w_{22}^{dm} & \cdots & w_{2n}^{dm} \\ \vdots & \vdots & & \vdots \\ w_{n1}^{dm} & w_{n2}^{dm} & \cdots & w_{nn}^{dm} \end{bmatrix} \tag{3.15}$$

and

$$\mathbf{W}^{mo} = \begin{bmatrix} w_{11}^{mo} & w_{12}^{mo} & \cdots & w_{1n}^{mo} \\ w_{21}^{mo} & w_{22}^{mo} & \cdots & w_{2n}^{mo} \\ \vdots & \vdots & & \vdots \\ w_{n1}^{mo} & w_{n2}^{mo} & \cdots & w_{nn}^{mo} \end{bmatrix}. \tag{3.16}$$

These matrices reflect the preferences for deflection decisions. If there is no initial preference for the output links, a uniform distribution of the output links is desirable. This

is achieved by setting $\mathbf{W}^{dm} = \mathbf{W}^{mo} = 0$. However, these weight matrices get updated and new probability distributions are shaped as reinforcement signals are received from the environment. For an arbitrary $h \times k$ matrix $\mathbf{Q}$, we define the Bernoulli semilinear operator $\mathcal{F}$ as:

$$\mathcal{F}(\mathbf{Q}) = \begin{bmatrix} \frac{1}{1+e^{q_{11}}} & \cdots & \frac{1}{1+e^{q_{1k}}} \\ \frac{1}{1+e^{q_{21}}} & \cdots & \frac{1}{1+e^{q_{2k}}} \\ \vdots & & \vdots \\ \frac{1}{1+e^{q_{h1}}} & \cdots & \frac{1}{1+e^{q_{hk}}} \end{bmatrix}. \tag{3.17}$$

In each decision-making epoch, a probability vector $\mathbf{P}^m = [p_1^m \ \ldots \ p_n^m]$ is calculated using the Bernoulli semilinear operator as:

$$\mathbf{P}^m = \mathcal{F}(\mathbf{I}^d \times \mathbf{W}^{mo}). \tag{3.18}$$

Each $z_k^m \in \mathbf{Z}^m$ is then calculated as:

$$z_k^m = \begin{cases} 0 & \text{if } p_k^m < 0.5 \\ \mathcal{U}(0,1) & \text{if } p_k^m = 0.5 \\ 1 & \text{if } p_k^m > 0.5 \end{cases}, \tag{3.19}$$

where $\mathcal{U}(0,1)$ denotes 0 or 1 sampled from a uniform distribution.

Let the $1 \times 2n$ row vector $\mathbf{I}^o = [\mathbf{I}^l \, \mathbf{Z}^m]$ denote the input to the output layer $\mathbf{Z}^o$. The $n \times 2n$ matrix $\mathbf{W}^o = [\mathbf{W}^{lo} \, \mathbf{W}^{mo}]$ denotes the weight matrix of the output layer $\mathbf{Z}^o$. The probability vector $\mathbf{P}^o = [p_1^o \ \ldots \ p_n^o]$ is first calculated as:

$$\mathbf{P}^o = \mathcal{F}\Big(\mathbf{I}^o \times \big(\mathbf{W}^o\big)^T\Big). \tag{3.20}$$

We then calculate the binary output vector $\mathbf{Z}^o = [z_1^o \ \ldots \ z_n^o]$ (3.19). Indices of 1s that appear in the output vector $\mathbf{Z}^o$ may be used to identify an outgoing link for the burst to be deflected. If $\mathbf{Z}^o$ contains multiple 1s, then the tie-break procedure described in Algorithm 4 is used for selecting the outgoing link. The output vector $\mathbf{Z}^o$ and associated probability vector $\mathbf{P}^o$ are the inputs to the algorithm. Multiple indices of 1s in $\mathbf{Z}^o$ may have different values in $\mathbf{P}^o$ because (3.19) maps to 1 all $\mathbf{P}^o$ elements greater than 0.5 when updating $\mathbf{Z}^o$. Therefore, in order to break a tie, Algorithm 4 considers all indices of 1s in $\mathbf{Z}^o$ and selects the index that has the maximum value in $\mathbf{P}^o$. If there are multiple indices with the same maximum value, Algorithm 4 randomly selects one of these indices according to a uniform distribution. It then sets to 0 the value of the elements of $\mathbf{Z}^o$ that were not selected. The algorithm needs to inspect the values of $\mathbf{Z}^o$ and $\mathbf{P}^o$ at most once. Furthermore, since vectors $\mathbf{Z}^o$ and $\mathbf{P}^o$ of a buffer-less node with n outgoing links have n elements, the time complexity of Algorithm 4 is $\mathcal{O}(n)$.

Algorithm 4 The tie-break procedure used to select an action based on the output vector $\mathbf{Z}^o$ of the feed-forward neural network

Input: Output vector: $\mathbf{Z}^o = [z_1^o \ldots z_n^o]$
Probability vector: $\mathbf{P}^o = [p_1^o \ldots p_n^o]$
Output: Index of the outgoing link for the burst to be deflected: a

$p_{max} \leftarrow 0$
$S \leftarrow \varnothing$
for $k \in \{1, 2, \ldots, n\}$ **do**
  if $z_k^o = 1$ && $p_k^o = p_{max}$ **then**
    $S \leftarrow k$
  else if $z_k^o = 1$ && $p_k^o > p_{max}$ **then**
    $S \leftarrow \varnothing$
    $S \leftarrow k$
    $p_{max} \leftarrow p_k^o$
  end if
end for
$a \leftarrow$ sample uniformly from S
for $k \in \{1, 2, \ldots, n\} \backslash \{a\}$ **do**
  $z_k^o \leftarrow 0$
end for
return a

After the reinforcement signal is received from the environment, the neural network updates its weight matrices $\mathbf{W}^{dm}$ and $\mathbf{W}^{mo}$ according to (2.5):

$$\Delta w_{ij} = \alpha r (y_i - p_i) x_{ij},$$

where α is a non-negative rate factor, y_i is the output of the ith agent, p_i is the probability of $y_i = 1$ given the input vector $\mathbf{x}_i$ and weight vector $\mathbf{w}_i$, and x_{ij} is the jth element of the input vector $\mathbf{x}_i$.

3.6.2 Feed-Forward Neural Networks for Deflection Routing with k-Episode Updates

The difference between an episodic feed-forward neural network decision-making module and a single-episode module is that the episodic neural network may generate deflection decisions while it waits for reward signals that belong to its previous decisions. An episodic neural network generates deflection decisions similar to the feed-forward neural network with a single-episode. It keeps an episode counter $C_{episode}$ that is incremented when the network generates a deflection decision and a reward counter C_r that is incremented when the neural network receives a reward signal from the environment.

An episode starts when $C_{episode} = 0$ and terminates when the neural network receives a reward signal for all generated deflection decisions ($C_{episode} = C_r$). During an episode, the decision-making module maintains the input vector $\mathbf{I}^d$, weight matrices $\mathbf{Z}^m$ and $\mathbf{Z}^o$, and probability vectors $\mathbf{P}^m$ and $\mathbf{P}^o$.

Let us assume that an episode terminates after k decisions. Let $\mathcal{R}_k$ denote the sum of all k reward signals that have been received from the environment. When the k-episode terminates, the feed-forward neural network resets the counters $C_{episode}$ and C_r to zero.

Furthermore, it updates its weights as:

$$w_{uv}^{dm} \leftarrow w_{uv}^{dm} + \left(\alpha \mathcal{R}_k \sum_{q=2}^{k} \left[((z_u^m)_q - (p_u^m)_q)(i_v^d)_{q-1} \right] \right) \tag{3.21}$$

$$w_{uv}^{mo} \leftarrow w_{uv}^{mo} + \left(\alpha \mathcal{R}_k \sum_{q=2}^{k} \left[((z_u^o)_q - (p_u^o)_q)(z_v^m)_{q-1} \right] \right), \tag{3.22}$$

where w_{uv}^{dm} and w_{uv}^{mo} are elements in the u^{th} row and v^{th} column of the weight matrices $\mathbf{W}^{dm}$ and $\mathbf{W}^{mo}$, respectively, and $(z_u^m)_q$ is the u^{th} element of the vector $\mathbf{Z}^m$ that was calculated during the q^{th} decision of the current episode.

3.6.3 Time Complexity Analysis

Q-NDD, PQDR, and RLDRS employ the table-based Q-learning algorithm. Hence, selecting an action depends on the table implementation. If the Q-table is implemented as a hash table, then an actions may be generated in constant time $\mathcal{O}(1)$. The update procedure executed after receiving a reward signal may also be completed in constant time. The NN-NDD and ENN-NDD algorithms employ the neural network shown in Fig. 3.4. In the case of NN-NDD, time complexity of selecting an action and updating the neural network weights after receiving a reward signal is no longer constant. When selecting an action, the neural network needs to calculate vectors $\mathbf{P}^m$ and $\mathbf{P}^o$ using (3.18) and (3.20), respectively. This results in $\mathcal{O}(n^2)$ complexity, where n is the number of a buffer-less node neighbors. The reward procedure requires inspection of elements of the weight matrices $\mathbf{W}^{dm}$ and $\mathbf{W}^{mo}$. Each of these inspections is quadratic in n, yielding a complexity of $\mathcal{O}(n^2)$ for the update procedure. In the case of ENN-NDD, the complexity of selecting an action is also $\mathcal{O}(n^2)$ while the complexity of the update procedure is $\mathcal{O}(kn^2)$, where k is the length of an episode.

The polynomial time complexity of the NN-NDD and ENN-NDD algorithms may affect their real-time decision-making performance. However, this complexity only depends on the degree of a buffer-less node. For example, the neural network of a node with the degree equal to 1,000 performs 10^6 operations for an action selection or for an update. An average general-purpose 2 GHz CPU is capable of processing 2×10^9 operations per second and, therefore, it is capable of processing the neural network in less than 0.5 ms.

3.7 Network Topologies: A Brief Overview

In this Section, we present a historical overview of the Internet topology research as a motivation and background for the topologies used in Section 3.8.

Many natural and engineering systems have been modeled by random graphs where nodes and edges are generated by random processes. They are referred to as Erdös and Rényi models [58]. Waxman [140] algorithm is commonly used to synthetically generate

such random network topologies. In a Waxman graph, an edge that connects nodes u and v exists with a probability:

$$\Pr\left(\{u,v\}\right) = \eta \exp\Big(\frac{-d(u,v)}{L\delta}\Big), \tag{3.23}$$

where $d(u,v)$ is the distance between nodes u and v, L is the maximum distance between the two nodes, and η and δ are parameters in the range $(0,1]$. Graphs generated with larger η and smaller δ values contain larger number of short edges. These graphs have longer hop diameter, shorter length diameter, and larger number of bicomponents [156]. Graphs generated using Waxman algorithm do not resemble the backbone and hierarchal structure of the current Internet. Furthermore, the algorithm does not guarantee a connected network [43].

Small-world graphs where nodes and edges are generated so that most of the nodes are connected by a small number of nodes in between were introduced rather recently to model social interactions [139]. A small-world graph may be created from a connected graph that has a high diameter by randomly adding a small number of edges. (The graph diameter is the largest number of vertices that should be traversed in order to travel from one vertex to another.) This construction drastically decreases the graph diameter. Generated networks are also known to have "six degrees of separation." It has been observed in social network that any two persons are linked by approximately six connections.

Most computer networks may be modeled by scale-free graphs where node degree distribution follows power-laws. Nodes are ranked in descending order based on their degrees. Relationships between node degree and node rank that follow various power-laws have been associated with various network properties. Eigenvalues vs. the order index as well as number of nodes within a number of hops vs. number of hops also follow various power-laws that have been associated with Internet graph properties [46], [60], [108], [125], [127]. The power-law exponents are calculated from the linear regression lines $10^{(a)}x^{(b)}$, with segment a and slope b when plotted on a log-log scale. The model implies that well-connected network nodes will get even more connected as Internet evolves. This is commonly referred as the "rich get richer" model [29]. Analysis of complex networks also involves discovery of spectral properties of graphs by constructing matrices describing the network connectivity.

Barabási-Albert [29] algorithm generates scale-free graphs that possess power-law distribution of node degrees. It suggests that incremental growth and preferential connectivity are possible causes for the power-law distribution. The algorithm begins with a connected network of n nodes. A new node i that is added to the network connects to an existing node j with probability:

$$\Pr(i,j) = \frac{d_j}{\sum_{k\in N} d_k}, \tag{3.24}$$

where d_j denotes the degree of the node j, N is the set of all nodes in the network, and $\sum_{k\in N} d_k$ is the sum of all node degrees.

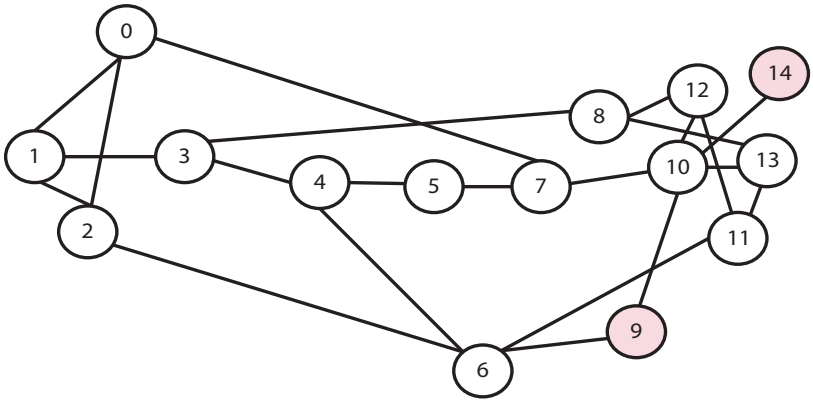

Figure 3.5: Topology of the NSF network after the 1989 transition. Node 9 and node 14 were added in 1990.

The Internet is often viewed as a network of Autonomous Systems. Groups of networks sharing the same routing policy are identified by Autonomous System Numbers [12]. The Internet topology on Autonomous System-level is the arrangement of autonomous systems and their interconnections. Analyzing the Internet topology and finding properties of associated graphs rely on mining data and capturing information about the Autonomous Systems. It has been established that Internet graphs on the Autonomous System-level exhibit the power-law distribution properties of scale-free graphs [60], [108], [127]. Barabási-Albert algorithm has been used to generate viable Internet-like graphs.

In 1985, NSF envisioned creating a research network across the United States to connect the recently established supercomputer centers, major universities, and large research laboratories. The NSF network was established in 1986 and operated at 56 kbps. The connections were upgraded to 1.5 Mbps and 45 Mbps in 1988 and 1991, respectively [11]. In 1989, two Federal Internet Exchanges (FIXes) were connected to the NSF network: FIX West at NASA Ames Research Center in Mountain View, California and FIX East at the University of Maryland [105]. The topology of the NSF network after the 1989 transition is shown in Fig. 3.5.

3.8 Performance Evaluation

We evaluate performance of the proposed deflection routing protocols and RLDRS [33] by implementing them within the iDef framework. We compare the algorithms based on burst loss probability, number of deflections, average end-to-end delay, and average number of hops traveled by bursts. We first use the National Science Foundation (NSF) network topology shown in Fig. 3.5, which has been extensively used to evaluate performance of OBS networks [32]–[34], [88], [95], [144], [145]. We also use network topologies generated by the Waxman algorithm [140]. These networks consist of 10, 20, 50, 100, 200, 500, and

1,000 nodes. We compare the deflection routing algorithms in terms of memory and Central Processing Unit (CPU) time using the larger Waxman graphs. In all simulation scenarios, we allow up to two deflections per burst ($\mathrm{DHC}_{max} = 2$). The burst header processing time is set to 0.1 ms.

3.8.1 National Science Foundation Network Scenario

The topology of the NSF network is shown in Fig. 3.5. The nodes are connected using bidirectional 1 Gbps fiber links with 64 wavelengths. The learning rate is set to $\alpha = 0.1$ and the maximum drop notification timer to $\mathrm{DN}_{max} = 50$ ms.

Multiple Poisson traffic flows with a data rate of 0.5 Gbps are transmitted randomly across the network. Each Poisson flow is 50 bursts long with each burst containing 12.5 kB of payload. While the burst arrival process depends on the aggregation algorithm [107] deployed in a node, the Poisson process has been widely used for performance analysis of OBS networks because it is mathematically tractable [152], [153].

The simulation scenarios are repeated with five random assignments of nodes as sources and destinations. The simulation results are averaged over the five simulation runs. The burst loss probability and the average number of deflections as functions of the number of Poisson flows for 64 wavelengths scenarios are shown in Fig. 3.6 and Fig. 3.7, respectively.

Even though the space complexity of the NDD algorithm is reduced to the degree of a node, simulation results show that NDD-based protocols perform better than RLDRS and PQDR in the case of low to moderate traffic loads. However, NN-NDD and Q-NDD protocols initiate larger number of deflections compared to RLDRS and PQDR. In moderate to high loads, PQDR exhibits the best performance. It deflects fewer bursts and its burst-loss probability is lower compared to other algorithms. In the cases of higher traffic loads, ENN-NDD algorithm discards fewer bursts while deflecting more bursts compared to other NDD-based algorithms.

In scenarios with lower traffic loads, the bursts are deflected less frequently and, therefore, the decision-making modules (learning agents) learn based on a smaller number of trials and errors (experience). The learning deficiency of Q-learning based algorithms in these cases results in higher burst-loss probabilities. In the cases of low to moderate traffic loads, the NN-NDD algorithm has the lowest burst-loss probability. In scenarios with higher traffic loads, decision-making modules (learning agents) deflect bursts more frequently. This enables the learning agents to gain additional experience and make more informed decisions. In these cases, RLDRS and the PQDR algorithm make optimal decisions, which result in lower burst-loss probabilities because they collect and store additional information about the network dynamics.

The RLDRS and PQDR signaling algorithms take into account the number of hops to destination when they generates feedback signals. Therefore, RLDRS and PQDR have smaller average end-to-end delay and average number of hops traveled by bursts. The

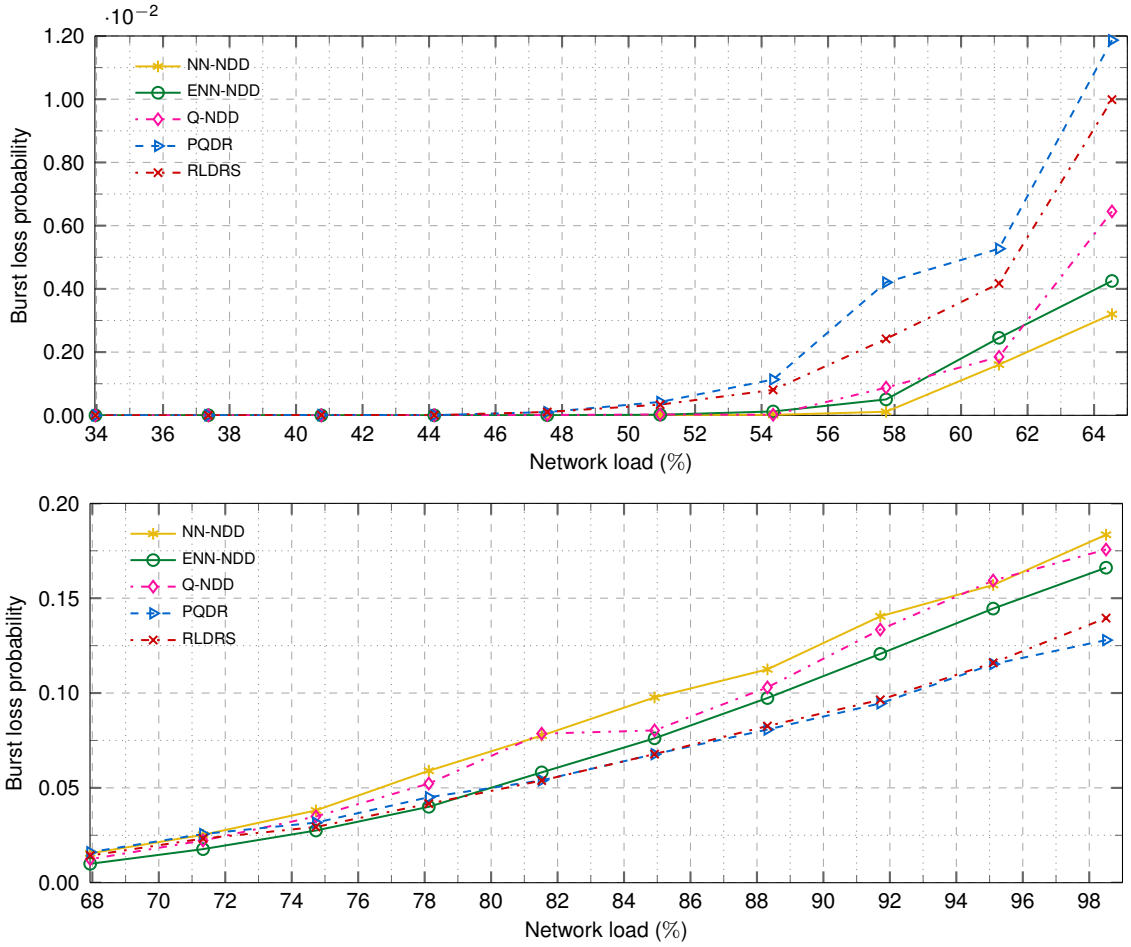

Figure 3.6: Burst loss probability as a function of the number of Poisson flows in the NSF network simulation scenario. For readability, two cases are plotted: 1,000 ($\approx$ 35% load) to 2,000 ($\approx$ 65% load) Poisson flows (top) and 2,000 ($\approx$ 65% load) to 3,000 ($\approx$ 100% load) Poisson flows (bottom). The NDD algorithms perform better than RLDRS and PQDR in case of low to moderate traffic loads. In the cases of higher traffic loads, ENN-NDD has smaller burst-loss compared to other NDD algorithms.

average end-to-end delay and average number of hops travelled by bursts as functions of traffic load are shown in Fig. 3.8 (top) and Fig. 3.8 (bottom), respectively.

Simulation results indicate that in the case of moderate loads (40%–65%), the NDD algorithms have much smaller burst-loss probability than RLDRS and PQDR, as shown in Fig. 3.6 (top). For example, at 65% load, the burst-loss probability of the NN-NDD algorithm is approximately 0.003, which is four times better than performance of PQDR ($\approx$ 0.012). The NDD algorithms maintain comparable performance in terms of end-to-end delay (within $\approx$ 0.04 ms), as shown in Fig. 3.8 (top). Similar trend is observed in the average number of hops travelled by bursts, as shown in Fig. 3.8 (bottom). In the case of high loads (80%–100%), the maximum difference in burst-loss probabilities is between

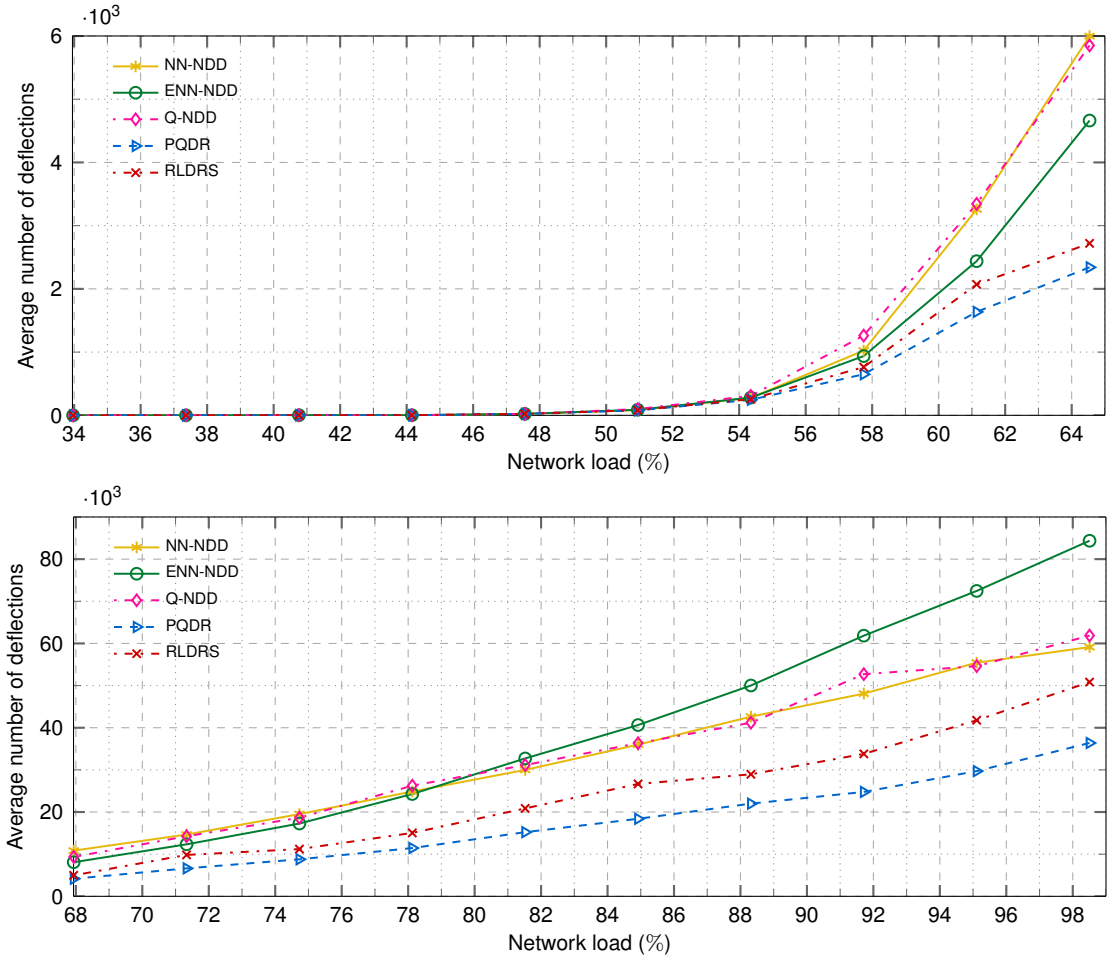

Figure 3.7: Average number of deflections as a function of the number of Poisson flows in the NSF network simulation scenario. For readability, two cases are plotted: 1,000 ($\approx$ 35% load) to 2,000 ($\approx$ 65% load) Poisson flows (top) and 2,000 ($\approx$ 65% load) to 3,000 ($\approx$ 100% load) Poisson flows (bottom). The PQDR and RLDRS algorithms perform better than the NDD algorithms in all cases. The PQDR algorithm has the smallest number of deflections.

NN-NDD ($\approx$ 0.18) and PQDR ($\approx$ 0.13) at 100% load, as shown in Fig. 3.6 (bottom). The maximum difference in end-to-end delays at 100% load is between Q-NDD ($\approx$ 0.5 ms) and RLDRS ($\approx$ 0.42 ms), as shown in Fig. 3.8 (top). Superior performance of NDD algorithms in case of moderate loads makes them a viable solution for deflection routing because the Internet backbone was engineered to keep link load levels below 50% [83]. Studies show that the overloads of over 50% occur less than 0.2% of a link life-time [64], [83].

3.8.2 Complex Network Topologies and Memory Usage

We use the Boston University Representative Internet Topology Generator (BRITE) [13] to generate random Waxman graphs [140] with 10, 20, 50, 100, 200, 500, and 1,000 nodes. In simulation scenarios, we use the Waxman parameters (3.23) $\eta = 0.2$ and $\delta = 0.15$ [156].

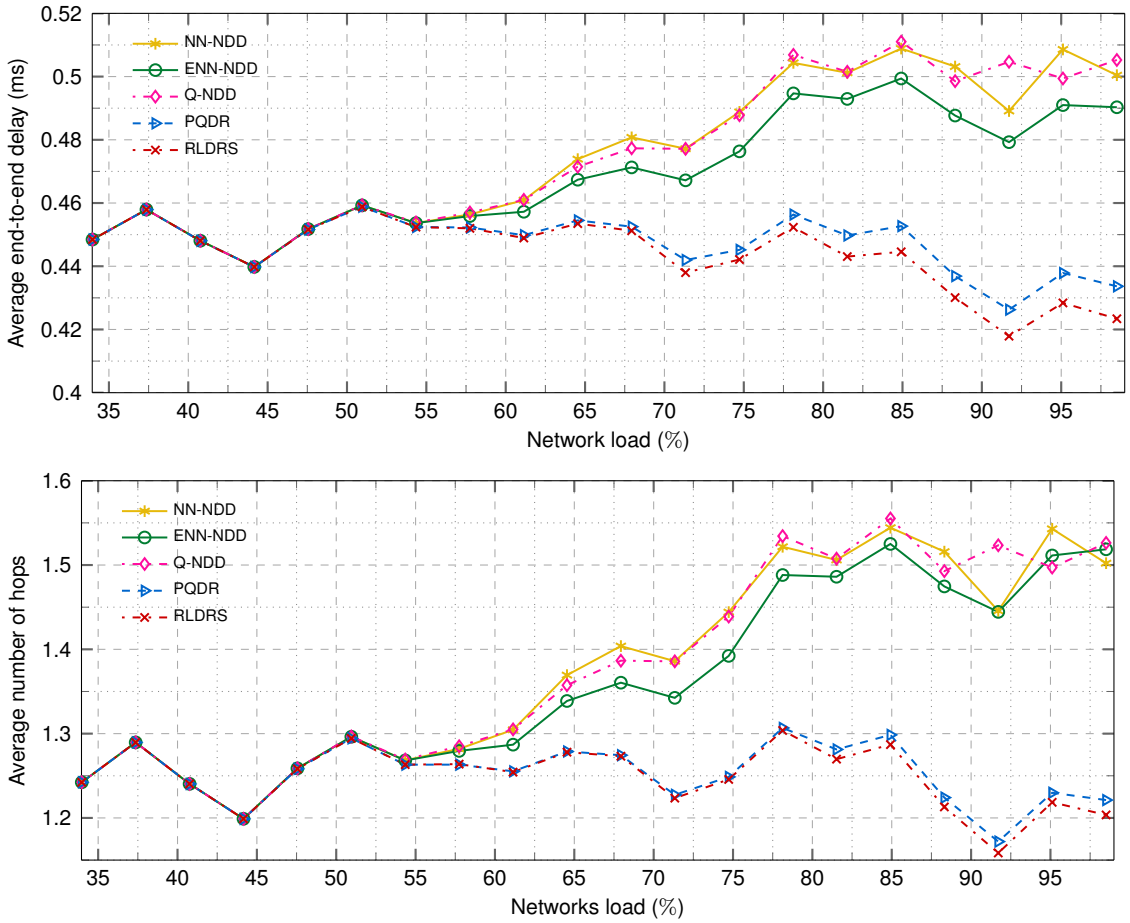

Figure 3.8: Average end-to-end delay (top) and average number of hops travelled by bursts (bottom) as functions of network traffic load in the NSF network scenario with 64 wavelengths. RLDRS and PQDR achieve better performance in both cases.

Nodes are randomly placed and each node is connected to three other nodes using bidirectional single wavelength fiber links. Sources and destinations of traffic flows are randomly selected. For all scenarios, we keep the network load at 40%. Hence, scenarios with 10, 20, 50, 100, 200, 500, and 1,000 nodes have 24, 48, 120, 240, 480, 1,200, and 2,400 Poisson traffic flows, respectively. Simulations were performed on a Dell Optiplex-790 with 16 GB memory and the Intel Core i7 2600 processor.

Burst-Loss Probability

Performance of deflection routing algorithms in terms of burst-loss probability as a function of number of nodes is shown in Fig. 3.9. Note that the burst-loss probability has a logarithmic trend. The NN-NDD and Q-NDD algorithms scale better as the size of the network grows.

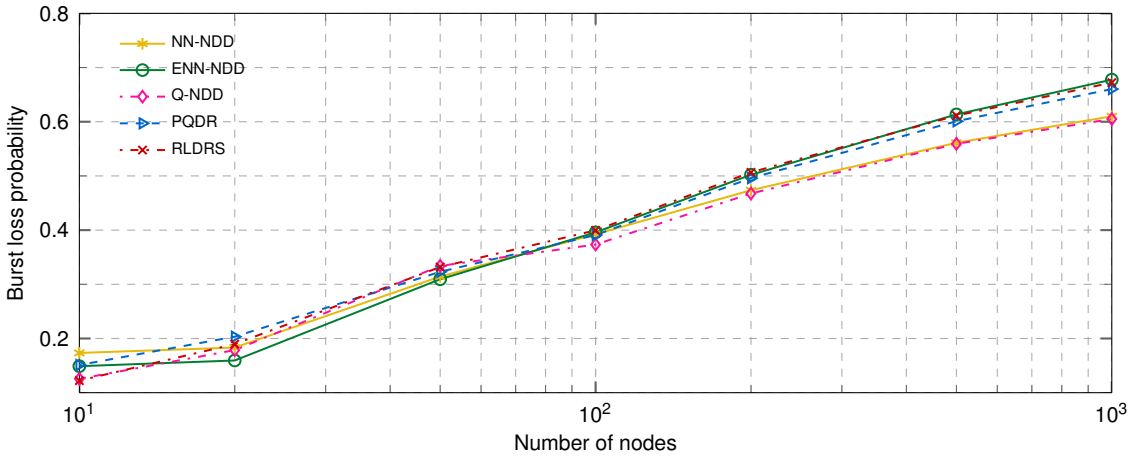

Figure 3.9: Burst loss probability as a function of the number of nodes in the Waxman graphs at 40% traffic load. These results are consistent with the results shown in Fig. 3.6, which were derived for the NSF network consisting of 14 nodes. Shown burst loss probabilities for networks of similar size (14 nodes) illustrate that NN-NDD, ENN-NDD, and Q-NDD algorithms have comparable performance to other algorithms.

The burst-loss probability of the NN-NDD and Q-NDD algorithms is smaller and bursts are deflected less frequently in larger networks. However, bursts travel through additional hops and thus experience longer end-to-end delays. Therefore, smaller burst-loss probability and smaller number of deflections come at the cost of selecting longer paths, which are less likely to be congested. RLDRS and the PQDR algorithm consider the number of hops to destination when deflecting bursts, which causes the bursts to travel through shorter paths. The probability of congestion along shorter paths is usually higher because the majority of the routing protocols tend to route data through such paths. As a result, burst-loss probability and probability of deflecting bursts is higher along the paths that PQDR and RLDRS select for deflection.

Number of Deflections

Although burst deflection reduces the burst-loss probability, it introduces excess traffic load in the network. This behavior is undesired from the traffic engineering point of view. Therefore, the volume of the deflected traffic should also be considered as a performance measure. Performance of the deflection routing algorithms in terms of number of deflections as a function of number of nodes is shown in Fig. 3.10. Simulation results show that the NN-NDD and Q-NDD algorithms deflect fewer bursts compared to RLDRS and the PQDR algorithm.

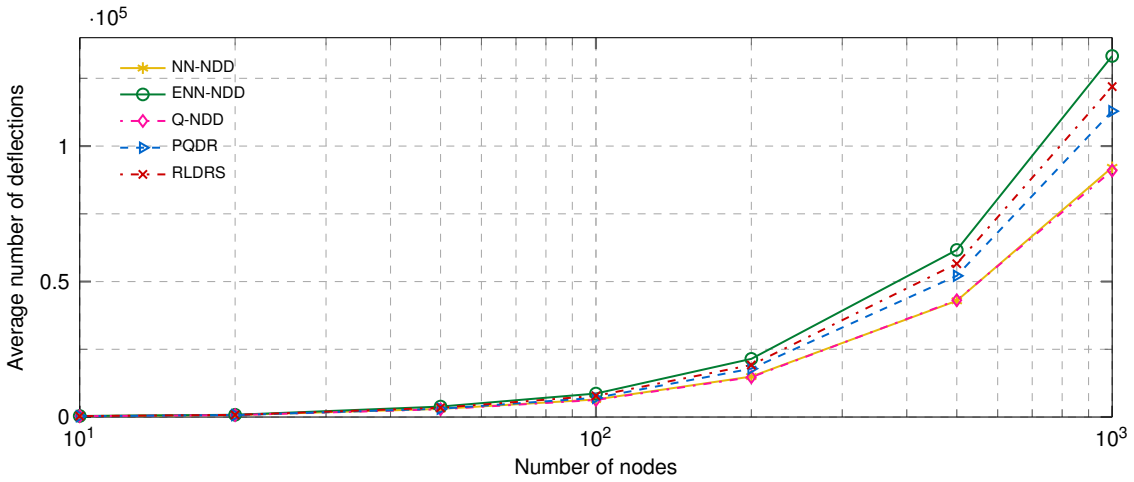

Figure 3.10: Number of deflections as a function of the number of nodes in the Waxman graphs at 40% traffic load.

End-to-End Delay and Average Number of Hops

Average end-to-end delay and average number of hops travelled by bursts as functions of the number of nodes are shown in Fig. 3.11 (top) and Fig. 3.11 (top), respectively. Simulation results indicate that in the case of NDD algorithms, bursts travel through additional hops compared to RLDRS and the PQDR algorithm. When deflecting a burst, RLDRS and the PQDR algorithm consider the number of hops to the destination. Furthermore, the underlying topology and the connectivity of nodes affect the number of hops traveled by bursts [76].

Memory and CPU Requirements

The memory and CPU time requirements of NN-NDD, ENN-NDD, Q-NDD, RLDRS, and PQDR are shown in Table 3.2. These values are based on the executed simulation scenarios.

All algorithms initially have comparable memory requirements. However, as the simulations proceed and the Q-tables are populated by new entries, the memory usage of RLDRS and the PQDR algorithm grows faster compared to NN-NDD and Q-NDD. The ENN-NDD algorithm memory usage grows faster than NN-NDD, Q-NDD, and RLDRS. This may be attributed to a larger number of bursts that ENN-NDD deflects, as shown in Fig. 3.10. The simulation results also show that NDD-based algorithms require less CPU time compared to RLDRS and PQDR. The memory usage of algorithms in the network with 1,000 nodes is shown in Fig. 3.12. The graphs were generated by using 100 equally spaced time instances over each simulation run.

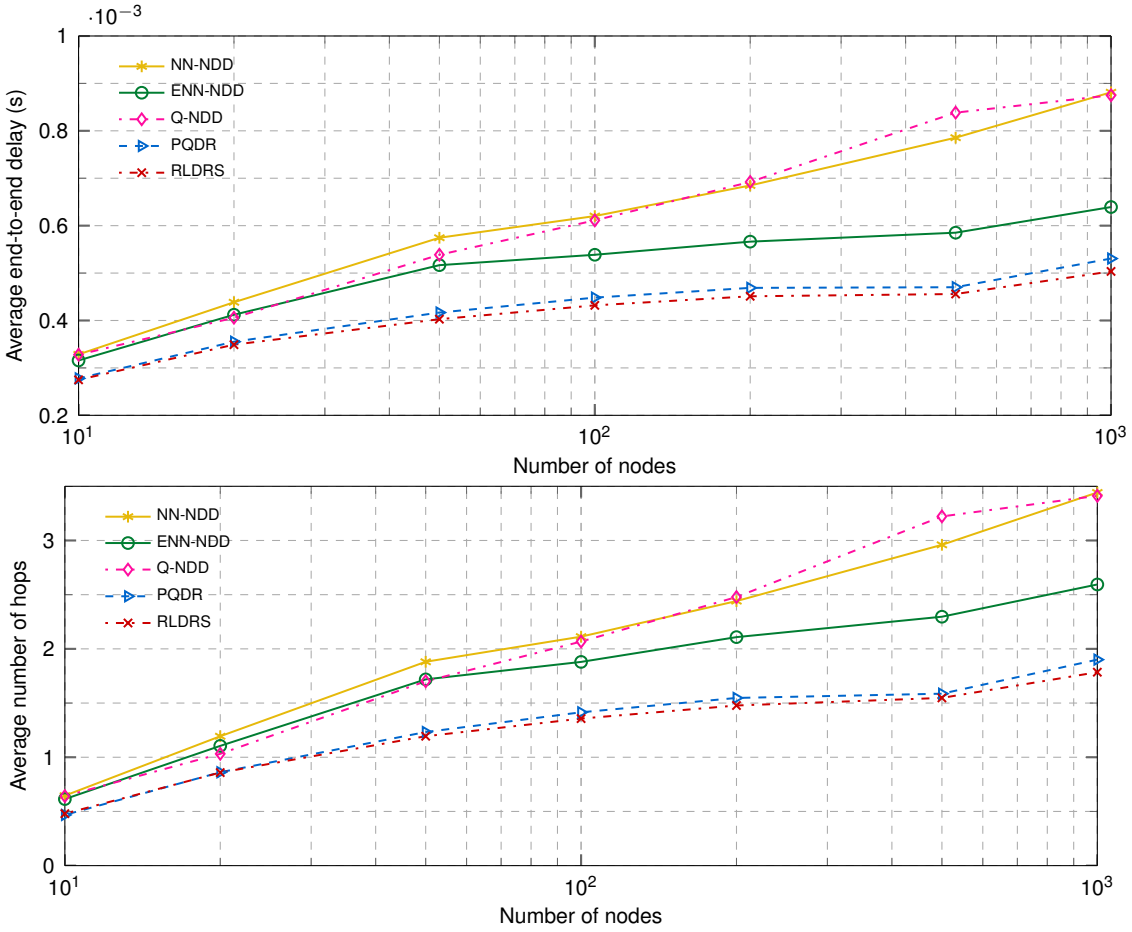

Figure 3.11: Average end-to-end delay (top) and average number of hops travelled by bursts (bottom) as functions of the number of nodes in the Waxman graphs at 40% traffic load.

3.9 Discussion

In this Chapter, we introduced the iDef framework that was implemented in the ns-3 network simulator. iDef may be employed for implementation and testing of various reinforcement learning-based deflection routing algorithms. Its independent modules enable users to integrate various learning and signaling algorithms when designing deflection routing protocols. We then introduced the PQDR algorithm that employs the predictive Q-routing to generate optimal deflection decisions. The proposed algorithm recovers and reselects paths to generate viable deflection decisions. We also introduced the NDD signaling algorithm for deflection routing. Its space complexity depends only on the node degree. Furthermore, we proposed a neural network-based associative learning algorithm and an NN-based associative learning algorithm with episodic updates. We combined the NDD signaling algorithm with neural network (NN-NDD), Episodic Neural Network (ENN-NDD), and a Q-learning-based (Q-NDD) decision-making modules for deflection routing.

Table 3.2: Comparison of Memory and CPU Usage of NN-NDD, ENN-NDD, Q-NDD, PQDR, and RLDRS

Algorithm	Number of nodes	Number of links	Number of flows	Minimum memory usage (MB)	Maximum memory usage (MB)	Average CPU usage (%)	Total CPU time (mm:ss)
NN-NDD	500	1,500	1,200	264	271	8.1	8:28.23
	1,000	3,000	2,400	983	1,001	14.03	32:47.91
ENN-NDD	500	1,500	1,200	264	294	7.74	12:03.48
	1,000	3,000	2,400	989	1,050	12.69	42:40.16
Q-NDD	500	1,500	1,200	264	271	8.22	8:25.29
	1,000	3,000	2,400	987	1,000	13.56	32:35.44
PQDR	500	1,500	1,200	264	290	9.83	13:28.11
	1,000	3,000	2,400	987	1,048	15.93	52:58.90
RLDRS	500	1,500	1,200	264	276	9.53	14:27.04
	1,000	3,000	2,400	987	1,013	15.41	56:13.34

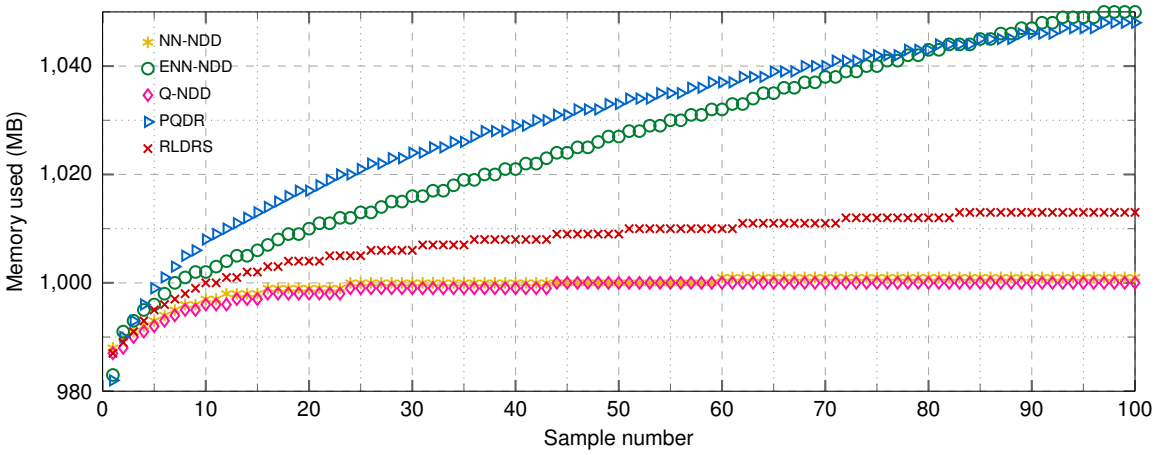

Figure 3.12: Memory used in the network with 1,000 nodes. The graphs were generated by using 100 equally spaced time instances over each simulation run.

Performance of the NN-NDD, ENN-NDD, and Q-NDD and PQDR algorithms was compared with the existing RLDRS algorithm. We implemented these algorithms within the iDef framework. For simulations, we employed the National Science Foundation (NSF) network topology and random graphs that consisted of 10, 20, 50, 100, 200, 500, and 1,000 nodes. These graphs were generated using the Waxman algorithm.

In simulations using the NSF network topology, NN-NDD achieves the lowest burst loss probability in the cases of low to moderate loads. Simulations with Waxman topologies indicate that NN-NDD and Q-NDD achieve smaller burst-loss probabilities while they deflect bursts less frequently. However, bursts travel through additional hops and thus experience longer end-to-end delays. Therefore, smaller burst-loss probability and smaller number of deflections come at the cost of selecting longer paths, which are less likely to be congested. RLDRS and the PQDR algorithm consider the number of hops to destination when deflect-

ing bursts. This, in turn, causes the bursts to travel through shorter paths. However, the probability of congestion along shorter paths is usually higher because the majority of the routing protocols tend to route data through such paths. Consequently, burst-loss probability and probability of deflecting bursts are higher along the paths that RLDRS and the PQDR algorithm select for deflection. The proposed NDD signaling algorithm also requires less memory and CPU resources, which are more significant as the size of the network grows.

Several OBS testbed implementations that have been reported [91], [128] are rather small and use field-programmable gate arrays (FPGA) for optical cross-connect implementation. Furthermore, only a simple deflection routing algorithm [137] was considered. Most existing deflection routing algorithms have been evaluated using simulation and analytical methods. Experimental performance evaluation of deflection routing algorithms using larger testbeds remains an open research topic.

Chapter 4

Reinforcement Learning-Based Algorithms for Virtual Network Embedding

The VNE problem may be divided into two subproblems: Virtual Node Mapping (VNoM) and Virtual Link Mapping (VLiM). VNoM algorithms map virtual nodes onto substrate nodes while VLiM algorithms map virtual links onto substrate paths. Algorithms that have been proposed for solving VNE are categorized into three categories depending on the approaches taken to solve these two subproblems [63]. The *uncoordinated two-stage algorithms* first solve the VNoM problem and provide a node mapping to the VLiM solver. In these approaches, the VNoM and VLiM solvers operate independently without any coordination [118], [151]. The *coordinated two-stage algorithms* also first solve the VNoM problem. Unlike the uncoordinated solutions, these algorithms consider the virtual link mappings when solving the VNoM problem [48], [51], [69]. The *coordinated one-stage algorithms* solve the VNoM and VLiM problems simultaneously. When two virtual nodes are mapped, also mapped is the virtual link connecting the two nodes [150].

Most VNE algorithms proposed in the literature address the VNoM while solving the VLiM using the shortest-path algorithms (k-shortest path, Breadth-First Search (BFS), and Dijkstra) or the Multi-Commodity Flow (MCF) algorithm. Unlike the MCF algorithm, the shortest-path algorithms do not allow path splitting. Path splitting [151] enables a virtual link to be mapped onto multiple substrate paths.

The contributions of this Chapter are:

- We model the VNoM problem as a Markov Decision Process (MDP). MDPs decompose sequential decision-making problems into states, actions, transition probabilities between the states given the actions, and the received rewards for performing actions in given states. We have selected MDP to model VNE to show that the VNE is inher-

ently a decision making problem and, therefore, may be solved using reinforcement learning algorithms.

- We introduce two *M*onte C*a*rlo Tree Search-based *V*irtual Network *E*mbedd*i*n*g* algorithms: MaVEn-M and MaVEn-S. They are coordinated two-stage VNE algorithms that solve the proposed MDP for VNoM using the Monte Carlo Tree Search (MCTS) algorithm [53], [89]. In order to improve performance of the MaVEn algorithms, we parallelize them by employing the MCTS root parallelization technique [44], [45].

  Classical tree search algorithms such as A* [111] and IDA* [90] require an admissible heuristic function that is capable of accurately computing the value of tree nodes at any given depth. In many applications, such as VNE, identifying a heuristic function is infeasible. An advantage of MCTS is that it does not require accurate evaluation functions [122]. MCTS is a recent algorithm that has been successfully employed to solve NP-complete puzzles [123]. It has also revolutionized the computer *Go* game [54], [67] and is rapidly replacing classical search algorithms as the method of choice in challenging domains [68].

  The proposed MaVEn-M algorithm employs the MCF algorithm to coordinate VNoM and VLiM when solving the VNoM subproblem. It also employs MCF to solve the VLiM subproblem after it obtains the VNoM solution. MaVEn-S employs a simple BFS algorithm. A number of existing VNE algorithms find only one solution for virtual network mapping and they are unable to improve the solution even if additional execution time is available [48], [51], [69], [157]. One advantage of the proposed algorithms is that their runtime may be adjusted according to the VNR arrival rates. If the VNR arrival rate is low, their execution time may be increased to find more profitable embedding solutions.

- We develop a VNE simulator *VNE-Sim* written in C++. It is based on the *Discrete Event System Specification* (DEVS) framework [135] and employs the *Adevs* library [110]. For performance evaluation, we implement the proposed MaVEn-M and MaVEn-S algorithms, the existing MIP-based R-Vine and D-Vine algorithms [51], and the node-ranking-based GRC algorithm [69]. We also introduce *profitability* as a new metric for comparing VNE algorithms.

The VNE algorithm presented in this chapter are summarized in Table 4.1.

The remainder of this Chapter is organized as follows. In Section 4.1, we present the VNE problem and its objective function, establish its upper bound, and introduce profitability as a performance metric. The available Vine [51] and GRC [69] node mapping, a Breadth-First Search-based link mapping, and the MCF link mapping algorithms are presented in Section 4.2. An MDP formulation of the VNoM problem is proposed in Section 4.4. To solve the VNE problem, we then introduce two MaVEn algorithms that utilize MCTS

Table 4.1: Summary of the presented algorithms.

VNE algorithm	VNoM algorithm	VLiM algorithm
MaVEn-M	Based on MCTS	MCF
MaVEn-S	Based on MCTS	BFS-based shortest-path
Parallel MaVEn-M	Based on MCTS with root parallelization	MCF
Parallel Maven-S	Based on MCTS with root parallelization	BFS-based shortest-path
R-Vine	Based on MIP with randomized rounding	MCF
D-Vine	Based on MIP with deterministic rounding	MCF
GRC	Based on Node-Ranking	BFS-based shortest-path

for finding optimal action policies for the proposed MDP. In Section 4.5, performance of the MaVEn algorithms is compared to the existing VNE algorithms. We conclude this Chapter with discussions in Section 4.6.

4.1 Virtual Network Embedding Problem

Let $G^s(N^s, E^s)$ denote the substrate network graph, where $N^s = \{n_1^s, n_2^s, \ldots, n_j^s\}$ is the set of j substrate nodes (vertices) while $E^s = \{e_1^s, e_2^s, \ldots, e_k^s\}$ is the set of k substrate edges (links). We denote by $e^s(n_a^s, n_b^s)$ a substrate edge between substrate nodes n_a^s and n_b^s. Let the i^{th} VNR be denoted by a triplet $\Psi_i(G^{\Psi_i}, \omega^{\Psi_i}, \xi^{\Psi_i})$, where $G^{\Psi_i}(N^{\Psi_i}, E^{\Psi_i})$ is the virtual network graph with $N^{\Psi_i} = \{n_1^{\Psi_i}, n_2^{\Psi_i}, \ldots, n_\ell^{\Psi_i}\}$ and $E^{\Psi_i} = \{e_1^{\Psi_i}, e_2^{\Psi_i}, \ldots, e_m^{\Psi_i}\}$ denoting the set of ℓ virtual nodes and m virtual edges, respectively. Furthermore, ω^{Ψ_i} is the VNR arrival time and ξ^{Ψ_i} is its life-time according to distributions Ω and Ξ, respectively. We denote by $e^{\Psi_i}(n_a^{\Psi_i}, n_b^{\Psi_i})$ a virtual edge between virtual nodes $n_a^{\Psi_i}$ and $n_b^{\Psi_i}$. We assume that a substrate node $n^s \in N^s$ possesses resources such as residual CPU capacity $\mathcal{C}(n^s)$ while a VNR node $n^{\Psi_i} \in N^{\Psi_i}$ requires a CPU capacity $\mathcal{C}(n^{\Psi_i})$. Similarly, a substrate edge $e^s \in E^s$ possesses a set of properties such as its residual bandwidth $\mathcal{B}(e^s)$ while a virtual edge $e^{\Psi_i} \in E^{\Psi_i}$ is constrained by a set of requirements such as bandwidth $\mathcal{B}(e^{\Psi_i})$.

VNoM algorithms assign a virtual node n^{Ψ_i} to a substrate node n^s that satisfies requirements of the virtual node. We denote such mapping by a tuple (n^{Ψ_i}, n^s). A complete virtual node map for Ψ_i is denoted by π:

$$\pi = \{(n^{\Psi_i}, n^s) | \forall n^{\Psi_i} \in N^{\Psi_i}\} \tag{4.1}$$

VLiM algorithms establish a virtual link using one or more substrate links. If a virtual link e^{Ψ_i} is established using q substrate links $\{e_1^s, e_2^s, \ldots, e_q^s\}$, we denote such mapping by a tuple $(e^{\Psi_i}, \{e_1^s, e_2^s, \ldots, e_q^s\})$. The goal of VNoM and VLiM algorithms is to optimize an objective function.

We assume that substrate nodes possess residual CPU capacities $\mathcal{C}(n^s)$ and are located at coordinates $\mathcal{L}(n^s) = (x_{n^s}, y_{n^s})$. Virtual nodes require a CPU capacity $\mathcal{C}(n^{\Psi_i})$ and have a location preference $\mathcal{L}(n^{\Psi_i}) = (x_{\Psi_i}, y_{\Psi_i})$. The only assumed substrate link resource is

the residual link bandwidth $\mathcal{B}(e^s)$. Virtual links have bandwidth requirements $\mathcal{B}(e^{\Psi_i})$ [51]. Assuming that path splitting [151] is not permitted for link mapping, a substrate node is eligible to host a virtual node if:

$$\mathcal{C}(n^s) \geq \mathcal{C}(n^{\Psi_i}), \tag{4.2}$$

$$d(\mathcal{L}(n^s), \mathcal{L}(n^{\Psi_i})) \leq \delta^{\Psi_i}, \tag{4.3}$$

$$n^s : \forall\, e^{\Psi_i} \in E_{n^{\Psi_i}}\ \exists\, e^s \in E_{n^s}\ \mid\ \mathcal{B}(e^s) \geq B(e^{\Psi_i}), \tag{4.4}$$

where $d(.,.)$ is the Euclidean distance function, δ^{Ψ_i} is a predetermined maximum allowable distance for the node embeddings of Ψ_i, E_{n^s} is the set of all substrate links connected to a substrate node n^s, and $E_{n^{\Psi_i}}$ is the set of all virtual links connected to a virtual node n^{Ψ_i}. If path splitting is permitted, (4.4) becomes:

$$\sum_{e^s \in E_{n^s}} \mathcal{B}(e^s) \geq \sum_{e^{\Psi_i} \in E_{n^{\Psi_i}}} \mathcal{B}(e^{\Psi_i}). \tag{4.5}$$

The substrate nodes that satisfy (4.2) to (4.5) form the set of candidate nodes $N^s(n^{\Psi_i})$ for embedding the virtual node n^{Ψ_i}.

4.1.1 Objective of Virtual Network Embedding

Majority of the proposed VNE algorithms have the objective to maximize the profit of InPs [48], [51], [69], [157]. Embedding revenue, cost, and the VNR acceptance ratio are the three main contributing factors to the generated profit.

Revenue

InPs generate revenue by embedding VNRs. The revenue generated by embedding a VNR Ψ_i is calculated as a weighted sum of VNR resource requirements:

$$\mathbf{R}(G^{\Psi_i}) = w_c \sum_{n^{\Psi_i} \in N^{\Psi_i}} \mathcal{C}(n^{\Psi_i}) + w_b \sum_{e^{\Psi_i} \in E^{\Psi_i}} \mathcal{B}(e^{\Psi_i}), \tag{4.6}$$

where w_c and w_b are the weights for CPU and bandwidth requirements, respectively [157]. The network provider receives a revenue only if the virtual network request is accepted for embedding. No revenue is generated in the case the request is rejected. In this Chapter, we assume that requests are served one at a time [69], [157]. Furthermore, we are not considering future VNR arrivals. Hence, to maximize the revenue, the service provider should try to accept as many requests as possible. The generated revenue only depends on whether or not the request is accepted. If a request is accepted for embedding, the generated revenue (4.6) does not depend on the substrate resources used to serve the request.

Cost

For embedding a VNR Ψ_i, the InP incurs a cost based on the resources it allocates for embedding the VNR. The incurred cost is calculated as:

$$\mathbf{C}(G^{\Psi_i}) = \sum_{n^{\Psi_i} \in N^{\Psi_i}} \mathcal{C}(n^{\Psi_i}) + \sum_{e^{\Psi_i} \in E^{\Psi_i}} \sum_{e^s \in E^s} f_{e^s}^{e^{\Psi_i}}, \tag{4.7}$$

where $f_{e^s}^{e^{\Psi_i}}$ denotes the total bandwidth of the substrate edge e^s that is allocated for the virtual edge e^{Ψ_i} [51], [151]. Unlike the revenue function (4.6), the cost (4.7) depends on the embedding configuration. Hence, if a VNR Ψ_i is accepted, the cost $\mathbf{C}(G^{\Psi_i})$ values depend on the embedding configuration within the substrate network.

Acceptance Ratio

In a given time interval τ, the ratio of the number of accepted VNRs $|\Psi^a(\tau)|$ to the total number of VNRs that arrived $|\Psi(\tau)|$ defines the acceptance ratio or the probability of accepting a VNR:

$$p_a^\tau = \frac{|\Psi^a(\tau)|}{|\Psi(\tau)|}. \tag{4.8}$$

Node and Link Utilizations

Node utilization $\mathcal{U}(N^s)$ is defined as:

$$\mathcal{U}(N^s) = 1 - \frac{\sum_{n^s \in N^s} \mathcal{C}(n^s)}{\sum_{n^s \in N^s} \mathcal{C}_{max}(n^s)}, \tag{4.9}$$

where $\mathcal{C}(n^s)$ is the available CPU resources of a substrate node n^s while $\mathcal{C}_{max}(n^s)$ is the maximum CPU resources of the node. Similarly, link utilization $\mathcal{U}(E^s)$ is defined as:

$$\mathcal{U}(E^s) = 1 - \frac{\sum_{e^s \in E^s} \mathcal{B}(e^s)}{\sum_{e^s \in e^s} \mathcal{B}_{max}(e^s)}, \tag{4.10}$$

where $\mathcal{B}(e^s)$ is the available bandwidth of a substrate link e^s while $\mathcal{B}_{max}(e^s)$ is the maximum bandwidth of the link.

Similar to other proposed algorithms [48], [51], [69], we also aim to maximize the InP profit defined as the difference between the generated revenue and cost. Maximizing the revenue and acceptance ratio while minimizing the cost of VNR embeddings maximizes the generated profit of InPs. Therefore, we define the objective of embedding a VNR Ψ_i as

maximizing the following objective function:

$$\mathcal{F}(\Psi_i) = \begin{cases} \mathbf{R}(G^{\Psi_i}) - \mathbf{C}(G^{\Psi_i}) & \text{successful embeddings} \\ \Gamma & \text{otherwise} \end{cases}, \tag{4.11}$$

where Γ defines the greediness of the embedding. Assuming that $\mathbf{R}(G^{\Psi_i}) - \mathbf{C}(G^{\Psi_i}) \in [a, b]$, setting Γ to a value smaller than a results in a greedy VNR embedding because in this case Γ is the lower bound of $\mathcal{F}$. Hence, to maximize $\mathcal{F}$ (4.11), any successful embedding is better than rejecting the VNR. Assigning $\Gamma \in [a, b]$ introduces a preferential VNR acceptance and thus if

$$\mathbf{R}(G^{\Psi_i}) - \mathbf{C}(G^{\Psi_i}) < \Gamma, \tag{4.12}$$

rejecting the VNR Ψ_i maximizes $\mathcal{F}(\Psi_i)$. We only consider the greedy approach by assigning a large negative penalty for unsuccessful embeddings ($\Gamma \to -\infty$).

In order to define the upper bound of the objective function, let us consider the case where path splitting is not permitted. In this case, (4.7) becomes:

$$\mathbf{C}(G^{\Psi_i}) = \sum_{n^{\Psi_i} \in N^{\Psi_i}} \mathcal{C}(n^{\Psi_i}) + \sum_{e^{\Psi_i} \in E^{\Psi_i}} \eta_{e^{\Psi_i}} \mathcal{B}(e^{\Psi_i}), \tag{4.13}$$

where $\eta_{e^{\Psi_i}}$ denotes the length of the substrate path used to accommodate the virtual edge e^{Ψ_i}. Hence:

$$\begin{aligned} \mathbf{R}(G^{\Psi_i}) - \mathbf{C}(G^{\Psi_i}) &= w_c \sum_{n^{\Psi_i} \in N^{\Psi_i}} \mathcal{C}(n^{\Psi_i}) - \sum_{n^{\Psi_i} \in N^{\Psi_i}} \mathcal{C}(n^{\Psi_i}) \\ &\quad + w_b \sum_{e^{\Psi_i} \in E^{\Psi_i}} \mathcal{B}(e^{\Psi_i}) - \sum_{e^{\Psi_i} \in E^{\Psi_i}} \eta_{e^{\Psi_i}} \mathcal{B}(e^{\Psi_i}) \\ &= (w_c - 1) \sum_{n^{\Psi_i} \in N^{\Psi_i}} \mathcal{C}(n^{\Psi_i}) \\ &\quad + \sum_{e^{\Psi_i} \in E^{\Psi_i}} (w_b - \eta_{e^{\Psi_i}}) \mathcal{B}(e^{\Psi_i}). \end{aligned} \tag{4.14}$$

The substrate path lengths $\eta_{e^{\Psi_i}} \forall e^{\Psi_i} \in E^{\Psi_i}$ are the only parameters that depend on the embedding configuration. Therefore, (4.14) is maximized when the path lengths are minimized. The minimum substrate path length for embedding a virtual link is equal to 1. Hence:

$$\begin{aligned} \max\{\mathbf{R}(G^{\Psi_i}) - \mathbf{C}(G^{\Psi_i})\} &= (w_c - 1) \sum_{n^{\Psi_i} \in n^{\Psi_i}} \mathcal{C}(n^{\Psi_i}) \\ &\quad + (w_b - 1) \sum_{e^{\Psi_i} \in E^{\Psi_i}} \mathcal{B}(e^{\Psi_i}). \end{aligned} \tag{4.15}$$

In order to remove the influence of the weights on the calculations of the upper bound, we assume $w_c = w_b = 1$ [48], [51], [151]. Hence:

$$\max\{\mathbf{R}(G^{\Psi_i}) - \mathbf{C}(G^{\Psi_i})\} = 0. \tag{4.16}$$

The upper bound of the objective function is achieved when Ψ_i is successfully embedded (4.11). Therefore, we define the upper bound for the objective function as:

$$\begin{aligned} \max\{\mathcal{F}(\Psi_i)\} &= \max\{\mathbf{R}(G^{\Psi_i}) - \mathbf{C}(G^{\Psi_i})\} \\ &\triangleq \mathcal{F}^{ub}(\Psi_i). \end{aligned} \tag{4.17}$$

The same upper bound would be achieved if path splitting was permitted because the minimum substrate path length for embedding a virtual link might not be less than 1.

4.1.2 Virtual Network Embedding Performance Metrics

Acceptance ratio, revenue to cost ratio, and substrate network resource utilization are the main VNE performance metrics [51], [69], [157]. Considering acceptance and revenue to cost ratios independently does not adequately estimate performance of VNE algorithms. For example, high acceptance ratio when the average revenue to cost ratio is low is undesirable because it leaves the substrate resources underutilized [51]. The same applies to having a high revenue to cost ratio while having a low acceptance ratio. Therefore, acceptance and average revenue to cost ratios should be considered simultaneously. Hence, we introduce profitability θ as a new performance measure. The profitability θ in a time interval τ is calculated as a product of acceptance and revenue to cost ratios:

$$\theta = p_a^{\tau} \times \frac{\sum\limits_{\Psi^i \in \Psi^a(\tau)} \mathbf{R}(G^{\Psi^i})}{\sum\limits_{\Psi^i i \in \Psi^a(\tau)} \mathbf{C}(G^{\Psi^i})}, \tag{4.18}$$

where p_a^{τ} is the acceptance ratio during the interval τ (4.8) and $\Psi^a(\tau)$ is the set of all accepted VNRs in the interval τ. Since the CPU and bandwidth revenue weights are $w_c = w_b = 1$, the maximum profitability $\theta_{max} = 1$. Higher profitability implies that the algorithm has high acceptance and high revenue to cost ratios. Therefore, VNE algorithms are desired to have profitability values close to 1.

4.2 Available Virtual Network Embedding Algorithms

In this section, we first describe the node mapping procedures of R-Vine, D-Vine [51], and GRC [69] algorithms. We then describe the Breadth-First Search-Based (BFS) shortest-path and MCF algorithms that are often employed for solving the VLiM problem.

4.2.1 Virtual Node Mapping Algorithms

R-Vine and D-Vine Algorithms

In order to identify a viable virtual node embedding for a VNR Ψ_i, R-Vine and D-Vine algorithms first create an *augmented substrate graph* based on the location preferences of the virtual nodes. We first define a cluster $\Lambda(n^{\Psi_i})$ for every virtual node n^{Ψ_i}:

$$\Lambda(n^{\Psi_i}) = \{n^s \in N^s \mid d(\mathcal{L}(n^s), \mathcal{L}(n^{\Psi_i})) \leq \delta^{\Psi_i}\}. \tag{4.19}$$

Meta nodes $\nu(n^{\Psi_i})$ are then created $\forall n^{\Psi_i} \in N^{\Psi_i}$. A meta node $\nu(n_k^{\Psi_i})$ is then connected to all substrate nodes in $\Lambda(n_k^{\Psi_i})$ using *meta edges* of infinite bandwidth capacities. Hence, the augmented substrate graph $G^{s'}(N^{s'}, E^{s'})$ is generated, where

$$N^{s'} = N^s \cup \{\nu(n^{\Psi_i}) \mid n^{\Psi_i} \in N^{\Psi_i}\} \tag{4.20}$$

and

$$E^{s'} = E^s \cup \{e^{s'}(\nu(n^{\Psi_i}), n^s) \mid n^{\Psi_i} \in N^{\Psi_i}, n^s \in \Lambda(n^{\Psi_i})\}. \tag{4.21}$$

The VNE problem may be formulated as a mixed integer $|E^{\Psi_i}|$-commodity flow problem, where each virtual link $e^{\Psi_i} \in E^{\Psi_i}$ is a commodity with source $s_{e^{\Psi_i}}$ and destination $t_{e^{\Psi_i}}$. Sources and destinations of the flows are the meta nodes $\nu(n^{\Psi_i})$ $\forall n^{\Psi_i} \in N^{\Psi_i}$. Assuming $\epsilon << 1$, the **VNE MIP** is formulated as:

VNE MIP

Variables:

- $f^{e^{\Psi_i}}_{e^s(n_j^s, n_k^s)}$: flow variables that denote the bandwidth of the substrate link $e^s(n_j^s, n_k^s)$ that is allocated for the virtual link e^{Ψ_i}.
- $x_{n_j^s, n_k^s}$: binary variables that are set to "1" if $\sum_{e^{\Psi_i} \in E^{\Psi_i}} f^{e^{\Psi_i}}_{e^s(n_j^s, n_k^s)} + f^{e^{\Psi_i}}_{e^s(n_k^s, n_j^s)} > 0$.

Objective:

$$\text{minimize} \quad \sum_{e^s \in E^s} \frac{1}{\mathcal{B}(e^s) + \epsilon} \sum_{e^{\Psi_i} \in E^{\Psi_i}} f^{e^{\Psi_i}}_{e^s} + \sum_{n^s \in N^s} \frac{1}{\mathcal{C}(n^s) + \epsilon} \sum_{n^{s'} \in N^{s'} \setminus N^s} x_{n^s, n^{s'}} \mathcal{C}_{max}(n^{s'}). \tag{4.22}$$

Constraints:

Domain:

$$f^{e^{\Psi_i}}_{e^s(n^{s'}_j,n^{s'}_k)} \geq 0 \qquad \forall e^{\Psi_i} \in E^{\Psi_i},\ \forall n^{s'}_j,\ \forall n^{s'}_k \in N^{s'}; \tag{4.23}$$

$$x_{n^{s'}_j,n^{s'}_k} \in \{0,1\} \qquad \forall n^{s'}_j,\ \forall n^{s'}_k \in N^{s'}. \tag{4.24}$$

Meta and Binary:

$$\sum_{n^s \in \Lambda n^{\Psi_i}} x_{n^s,\nu n^{\Psi_i}} = 1 \qquad \forall n^{\Psi_i} \in N^{\Psi_i}; \tag{4.25}$$

$$\sum_{n^{\Psi_i} \in N^{\Psi_i}} x_{\nu n^{\Psi_i},n^s} \leq 1 \qquad \forall n^s \in N^s; \tag{4.26}$$

$$x_{n^{s'}_j,n^{s'}_k} = x_{n^{s'}_k,n^{s'}_j} \qquad \forall n^{s'}_j,\ \forall n^{s'}_k \in N^{s'}. \tag{4.27}$$

Capacity:

$$\sum_{e^{\Psi_i} \in E^{\Psi_i}} \left(f^{e^{\Psi_i}}_{e^s(n^s_j,n^s_k)} + f^{e^{\Psi_i}}_{e^s(n^s_k,n^s_j)} \right) \leq \mathcal{B}(e^s(n^s_j,n^s_k)) \cdot x_{n^s_j,n^s_k} \qquad \forall n^s_j, n^s_k \in N^{s'}; \tag{4.28}$$

$$\mathcal{C}(n^s_j) \geq x_{n^s_j,n^s_k} \mathcal{C}_{max}(n^s_k) \qquad \forall n^s_k \in N^{s'} \setminus N^s,\ \forall n^s_j \in N^s. \tag{4.29}$$

Flow Conservation:

$$\sum_{n^s_j \in N^s} f^{e^{\Psi_i}}_{e^s(n^s_j,n^s_k)} - \sum_{n^s_j \in N^s} f^{e^{\Psi_i}}_{e^s(n^s_k,n^s_j)} = 0 \qquad \forall e^{\Psi_i} \in E^{\Psi_i},\ \forall n^s_k \in N^s \setminus \{s_{e^{\Psi_i}}, t_{e^{\Psi_i}}\}. \tag{4.30}$$

Demand Satisfaction:

$$\sum_{n^s_j \in N^s} f^{e^{\Psi_i}}_{e^s(s_{e^{\Psi_i}},n^s_j)} - \sum_{n^s_j \in N^s} f^{e^{\Psi_i}}_{e^s(n^s_j,s_{e^{\Psi_i}})} = \mathcal{B}(e^{\Psi_i}) \qquad \forall e^{\Psi_i} \in E^{\Psi_i}; \tag{4.31}$$

$$\sum_{n^s_j \in N^s} f^{e^{\Psi_i}}_{e^s(t_{e^{\Psi_i}},n^s_j)} - \sum_{n^s_j \in N^s} f^{e^{\Psi_i}}_{e^s(n^s_j,t_{e^{\Psi_i}})} = -\mathcal{B}(e^{\Psi_i}) \qquad \forall e^{\Psi_i} \in E^{\Psi_i}. \tag{4.32}$$

The Domain Constraint (4.24) is the integer constraint that makes the problem an MIP. Finding solution of MIPs is $\mathcal{NP}$-hard and, therefore, computationally intractable [124]. Linear Programming is computationally simpler than MIP and may be solved in polynomial

time. Therefore, in order to solve MIPs, relaxation methods [124] are employed to obtain a linear program from the original MIP. When a solution for the corresponding linear program is found, rounding techniques are employed to translate the solution of the linear program to the solution of the original MIP. Vine algorithms solve **VNE LP**, which is obtained by relaxing the integer constraint (4.24):

VNE LP:

$$x_{n_j^{s'}, n_k^{s'}} \in [0,1] \qquad \forall n_j^{s'},\ \forall n_k^{s'} \in N^{s'}. \tag{4.33}$$

Let us define $flow(n^{s'}, n^s)$ as:

$$flow(n^{s'}, n^s) = \sum_{e^{\Psi_i} \in E^{\Psi_i}} \left(f^{e^{\Psi_i}}_{e^s(n^{s'}, n^s)} + f^{e^{\Psi_i}}_{e^s(n^s, n^{s'})} \right). \tag{4.34}$$

The pseudocode of R-Vine and D-Vine node mapping algorithms are listed in Algorithm 5 and Algorithm 6, respectively.

Algorithm 5 R-Vine: Randomized virtual network embedding algorithm

procedure R-VINE(Ψ_i)
  $\pi \leftarrow \emptyset$
  Create augmented substrate graph $G^{s'}$
  Solve **VNE LP**
  for all $n^s \in N^s$ **do**
    $\varphi(n^s) \leftarrow 0$
  end for
  for all $n^{\Psi_i} \in N^{\Psi_i}$ **do**
    if $\Lambda(n^{\Psi_i}) \setminus \{n^s \in N^s \mid \varphi(n^s) = 1\} = \emptyset$ **then**
      Reject Ψ_i
      return
    end if
    $\varrho_{max} \leftarrow 0$
    for all $n^s \in \Lambda(n^{\Psi_i})$ **do**
      $\varrho_{n^s} \leftarrow flow(\nu(n^{\Psi_i}), n^s) \cdot x_{\nu(n^{\Psi_i}) n^s}$
      $\varrho_{max} \leftarrow \varrho_{max} + \varrho_{n^s}$
    end for
    for all $n^s \in \Lambda(n^{\Psi_i})$ **do**
      $\varrho_{n^s} \leftarrow \varrho_{n^s} / \varrho_{max}$
    end for
    $n^s_{max} = \underset{n^s \in \Lambda(n^{\Psi_i})}{\text{argmax}} \{\varrho_{n^s} \mid \varphi(n^s) = 0\}$
    add (n_i^{Ψ}, n^s_{max}) to π with probability ϱ_{n^s}
    $\varphi(n^s_{max}) \leftarrow 1$ with probability ϱ_{n^s}
  end for
  return π
end procedure

Algorithm 6 D-Vine: Deterministic virtual network embedding algorithm

procedure D-VINE(Ψ_i)
  $\pi \leftarrow \emptyset$
  Create augmented substrate graph $G^{s'}$
  Solve **VNE LP**
  for all $n^s \in N^s$ **do**
    $\varphi(n^s) \leftarrow 0$
  end for
  for all $n^{\Psi_i} \in N^{\Psi_i}$ **do**
    if $\Lambda(n^{\Psi_i}) \setminus \{n^s \in N^s \mid \varphi(n^s) = 1\} = \emptyset$ **then**
      Reject Ψ_i
      return
    end if
    for all $n^s \in \Lambda(n^{\Psi_i})$ **do**
      $\varrho_{n^s} \leftarrow flow(\nu(n^{\Psi_i}), n^s) \cdot x_{\nu(n^{\Psi_i}) n^s}$
    end for
    $n^s_{max} = \underset{n^s \in \Lambda(n^{\Psi_i})}{\text{argmax}} \{\varrho_{n^s} \mid \varphi(n^s) = 0\}$
    add (n_i^{Ψ}, n^s_{max}) to π
    $\varphi(n^s_{max}) \leftarrow 1$
  end for
  return π
end procedure

The Global Resource Capacity Algorithm

Let $Adj(n_i^s)$ denote the set of substrate nodes adjacent to n_i^s and $e^s(n_i^s, n_j^s)$ denote the substrate link that connects the substrate nodes n_i^s and n_j^s. The GRC algorithm first calculates the embedding capacity $r(n_i^s)$ for a substrate node n_i^s as:

$$r(n_i^s) = (1-d)\hat{\mathcal{C}}(n_i^s) + d \sum_{n_j^s \in Adj(n_i^s)} \frac{\mathcal{B}(e^s(n_i^s, n_j^s)) \cdot r(n_j^s)}{\sum_{n_k^s \in Adj(n_j^s)} \mathcal{B}(e^s(n_j^s, n_k^s))}, \tag{4.35}$$

where $0 < d < 1$ is a constant damping factor and $\hat{\mathcal{C}}(n_i^s)$ is the normalized CPU resources of n_i^s:

$$\hat{\mathcal{C}}(n_i^s) = \frac{\mathcal{C}(n_i^s)}{\sum_{n^s \in N^s} \mathcal{C}(n^s)}. \tag{4.36}$$

Let us assume that a substrate network is composed of p nodes. The vector form of the embedding capacity for all substrate nodes in the substrate network is:

$$\mathbf{r} = (1-d)\mathbf{c} + d\mathbf{M}\mathbf{r}, \tag{4.37}$$

where $\mathbf{c} = \left(\hat{\mathcal{C}}(n_1^s), \ldots, \hat{\mathcal{C}}(n_p^s)\right)^T$, $\mathbf{r} = \left(r(n_1^s), \ldots, r(n_p^s)\right)^T$, and $\mathbf{M}$ is a $p \times p$ square matrix. The m_{ij} element of $\mathbf{M}$ is calculated as:

$$m_{ij} = \begin{cases} \dfrac{\mathcal{B}(e^s(n_i^s, n_j^s))}{\sum_{n_k^s \in Adj(n_j^s)} \mathcal{B}(e^s(n_j^s, n_k^s))} & e^s(n_i^s, n_j^s) \in E^s \\ 0 & \text{otherwise} \end{cases}. \tag{4.38}$$

The GRC algorithm iteratively calculates $\mathbf{r}$ by initially setting $\mathbf{r}_0 = \hat{\mathbf{c}}$ and calculating $\mathbf{r}_{k+1}$ as:

$$\mathbf{r}_{k+1} = (1-d)\mathbf{c} + d\mathbf{M}\mathbf{r}_k, \tag{4.39}$$

where $\mathbf{r}_0$ is the value of $\mathbf{r}$ after k iterations. The iterative process is terminated when:

$$|\mathbf{r}_{k+1} - \mathbf{r}_k| < \sigma, \tag{4.40}$$

where $\sigma << 1$ is a predefined stopping threshold. The procedure for calculating the GRC vector $\mathbf{r}$ is shown in Algorithm 7. When the embedding capacity vectors for substrate and virtual nodes are calculated, the nodes are ranked based on their capacity. The GRC algorithm then selects the substrate node with the highest rank to embed the virtual node with the highest rank, provided that the substrate node satisfies the virtual node CPU requirement. Otherwise, the substrate node with the second highest rank is selected. If no

Algorithm 7 GRC vector calculation: The input graph may be a substrate or a virtual network graph.

procedure GRCVecCalc($G(N, E)$,σ)
  Initialize $\mathbf{c}$ and $\mathbf{M}$
  $\mathbf{r}_0 = \mathbf{c}$
  $k = 0$
  $\Delta = \infty$
  while $\Delta \geqslant \sigma$ **do**
    $\mathbf{r}_{k+1} = (1-d)\mathbf{c} + d\mathbf{M}\mathbf{r}_k$
    $\Delta = \| \mathbf{r}_{k+1} - \mathbf{r}_k \|$
    $k = k + 1$
  end while
  return $\mathbf{r}_k$
end procedure

substrate node satisfies the virtual node CPU requirement, the VNR is denied. The GRC virtual node mapping is listed in Algorithm 8.

Algorithm 8 GRC Virtual Node Mapping

procedure GRCVNoM(G^s,Ψ_i)
  $\pi \leftarrow \emptyset$
  $\mathbf{r}^s \leftarrow$ GRCVecCalc(G^s,σ)
  $\mathbf{r}^{\Psi_i} \leftarrow$ GRCVecCalc(G^{Ψ_i},σ)
  Sort $\mathbf{r}^s$ in descending order to get $\mathbf{r}^s_{sort}$
  Sort $\mathbf{r}^{\Psi_i}$ in descending order to get $\mathbf{r}^{\Psi_i}_{sort}$
  for each virtual node n^{Ψ_i} in the order of $\mathbf{r}^{\Psi_i}_{sort}$ **do**
    for each unselected substrate node n^s in the order of $\mathbf{r}^s_{sort}$ **do**
      if $\mathcal{C}(n^s) \geq \mathcal{C}(n^{\Psi_i})$ **then**
        add (n_i^{Ψ}, n^s) to π
        mark n^s as *selected*
        $Count \leftarrow Count + 1$
        break
      end if
    end for
  end for
  if $Count < |N^{\Psi_i}|$ **then**
    Reject Ψ_i
  end if
  return π
end procedure

4.2.2 Virtual Link Mapping Algorithms

VLiM algorithms take as the input a virtual node mapping π and identify a virtual link mapping.

Breadth-First Search Algorithm

BFS is a graph traversal algorithm. Starting at a given node, BFS first visits all its neighbors. The visited nodes are marked and then all their neighbors that are not yet visited are traversed. BFS may be implemented using a *first-in-first-out* (FIFO) queue. An important property of the BFS algorithm is that the first encounter of a node marks the shortest path

of arriving at that node in terms of number of hops. The BFS-Based VLiM algorithm is listed in Algorithm 9.

Algorithm 9 BFS-Based VLiM algorithm

```
1: procedure BFS(G^s, Ψ_i, π)
2:     LinkMap ← ∅
3:     CurrentPath ← ∅
4:     Queue ← ∅
5:     for all n^s ∈ N^s do
6:         predecessor[n^s] ← ∅
7:     end for
8:     for all e^{Ψ_i}(n_a^{Ψ_i}, n_b^{Ψ_i}) ∈ E^{Ψ_i} do
9:         n^s_source ← π(n_a^{Ψ_i})
10:        n^s_dest ← π(n_b^{Ψ_i})
11:        n^s_current ← n^s_source
12:        Queue.add(n^s_current)
13:        while Queue.size() > 0 do
14:            n^s_current ← Queue.pop()
15:            if n^s_current is not marked visited then
16:                mark n^s_current as visited
17:                for all do n^s ∈ Adj(n^s_current)
18:                    if B(e^s(n^s_current, n^s)) ≥ B(e^{Ψ_i}) then
19:                        Queue.add(n^s)
20:                        predecessor[n^s] = n^s_current
21:                    end if
22:                end for
23:            end if
24:        end while
25:        if predecessor[n^s_dest] = ∅ then
26:            Reject Ψ_i
27:        end if
28:        CurrentPath.add(n^s_dest)
29:        n^s_current ← predecessor[n^s_dest]
30:        while n^s_current ≠ n^s_source do
31:            CurrentPath.add(n^s_current)
32:            n^s_current ← predecessor[n^s_dest]
33:        end while
34:        LinkMap.add(CurrentPath)
35:    end for
36: end procedure
```

Multicommodity Flow Algorithm

Let us assume that the virtual nodes $n_a^{\Psi_i}$ and $n_b^{\Psi_i}$ are mapped by VNoM onto substrate nodes n_a^s and n_b^s, respectively. A virtual link $e^{\Psi_i}(n_a^{\Psi_i}, n_b^{\Psi_i})$ between the virtual nodes $n_a^{\Psi_i}$ and $n_b^{\Psi_i}$ is a flow between the substrate nodes n_a^s and n_b^s. In MCF terminology, a flow is a commodity $k_i = (s_i, t_i, d_i)$, where s_i, t_i, and d_i are the flow source, destination, and demand, respectively [24]. The virtual link $e^{\Psi_i}(n_a^{\Psi_i}, n_b^{\Psi_i})$ that connects $n_a^{\Psi_i}$ and $n_b^{\Psi_i}$ may be denoted as a commodity $k_{e^{\Psi_i}}(s_{e^{\Psi_i}} = n_a^s, t_{e^{\Psi_i}} = n_b^s, d_{e^{\Psi_i}} = \mathcal{B}(e^{\Psi_i}))$. Hence, assuming $\epsilon << 1$, MCF may be formulated as the following linear program:

Objective:

$$\text{minimize} \quad \sum_{e^s \in E^s} \frac{1}{\mathcal{B}(e^s) + \epsilon} \sum_{e^{\Psi_i} \in E^{\Psi_i}} f_{e^s}^{e^{\Psi_i}}. \tag{4.41}$$

Constraints:

Capacity:

$$\sum_{e^{\Psi_i} \in E^{\Psi_i}} (f^{e^{\Psi_i}}_{e^s(n^s_j, n^s_k)} + f^{e^{\Psi_i}}_{e^s(n^s_k, n^s_j)}) \leq \mathcal{B}(e^s(n^s_j, n^s_k)) \qquad \forall n^s_j,\ \forall n^s_k \in N^s. \tag{4.42}$$

Flow Conservation:

$$\sum_{n^s_j \in N^s} f^{e^{\Psi_i}}_{e^s(n^s_j, n^s_k)} - \sum_{n^s_j \in N^s} f^{e^{\Psi_i}}_{e^s(n^s_k, n^s_j)} = 0$$

$$\forall e^{\Psi_i} \in E^{\Psi_i},\ \ \forall n^s_k \in N^s \setminus \{s_{e^{\Psi_i}}, t_{e^{\Psi_i}}\}. \tag{4.43}$$

Demand Satisfaction:

$$\sum_{n^s_j \in N^s} f^{e^{\Psi_i}}_{e^s(s_{e^{\Psi_i}}, n^s_j)} - \sum_{n^s_j \in N^s} f^{e^{\Psi_i}}_{e^s(n^s_j, s_{e^{\Psi_i}})} = d_{e^{\Psi_i}} \qquad \forall e^{\Psi_i} \in E^{\Psi_i}, \tag{4.44}$$

$$\sum_{n^s_j \in N^s} f^{e^{\Psi_i}}_{e^s(t_{e^{\Psi_i}}, n^s_j)} - \sum_{n^s_j \in N^s} f^{e^{\Psi_i}}_{e^s(n^s_j, t_{e^{\Psi_i}})} = -d_{e^{\Psi_i}} \qquad \forall e^{\Psi_i} \in E^{\Psi_i}. \tag{4.45}$$

4.3 Virtual Network Embedding Algorithms and Data Center Networks

Various algorithms have been proposed to find profitable virtual network embeddings given arbitrary substrate and virtual network topologies [48], [51], [69]. Performance of these algorithms has been evaluated using synthetic topologies that resemble ISP networks. These algorithms were designed without assumptions regarding the structure of substrate and virtual networks. Therefore, they may be employed for any given substrate and virtual network topology. They are often complex because assuming a specific structure of substrate and virtual topologies may simplify the embedding process. A study that considered the impact of common ISP topologies (Ladder, Star, and Hub and Spoke) on VNE revealed that topological features significantly affect quality of the solution [100].

The advent of Software Defined Networking (SDN) has recently enabled cloud providers such as the Amazon Web Services (AWS) [1] to offer network virtualization services that requires embedding of the virtual networks in data center networks. Even though the defined topologies of data center network permit designing specialized VNE algorithms [28], [72], we employ the BCube [71] and Fat-Tree [20] data center topologies to compare the performance of the proposed algorithms.

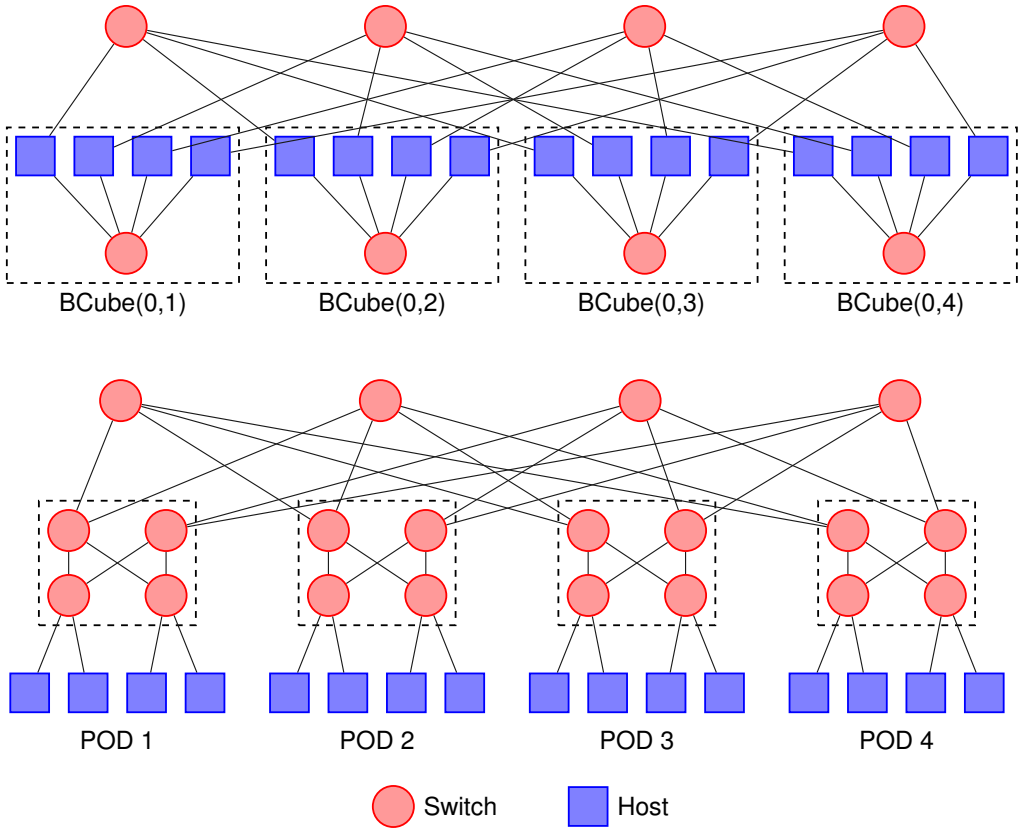

Figure 4.1: Examples of data center topologies: BCube$(2, 4)$ (top) and Fat-Tree$_4$ (bottom) network topologies.

4.3.1 Data Center Network Topologies

They are often designed to be scalable to a large number of servers and to have fault tolerance against various failures such as link or server rack failure. Furthermore, data center networks should provide high network capacity to support services with high bandwidth requirements [73]. Data center networks consist of hosts, switches, and links. Hosts may be used for virtual node embeddings while switches are used only for traffic forwarding.

We denote a BCube [71] topology with BCube(k, n) where k is the BCube level and n is the number of hosts in the level-0 BCube. This topology is recursively structured. BCube level-0 consists of n hosts connected to an n-port switch. In this topology, switches are not directly connected to each other and servers perform packet forwarding functions. The BCube$(2, 4)$ network topology is shown in Fig. 4.1 (top).

Fat-Tree topology is a special Clos architecture that was initially proposed to interconnect processors of parallel supercomputers [92]. It has been recently used for data center networks [20]. A Fat-Tree$_k$ topology is constructed using $(k/2)^2 + k^2$ k-port switches and supports $k^3/4$ hosts. An example of a Fat-Tree$_4$ is shown in Fig. 4.1 (bottom).

4.4 Virtual Network Embedding as a Markov Decision Process

The VNE problem may be modeled as a sequential decision-making problem where a decision-making agent (VNE algorithm) receives VNRs. The agent solves the VNoM and VLiM for each VNR Ψ_i and receives a reward $\mathcal{F}(\Psi_i)$ for these mappings. The agent's objective is to maximize this reward. (Throughout this chapter, we interchangeably use terms agent and decision-making agent.)

4.4.1 A Finite-Horizon Markov Decision Process Model for Coordinated Virtual Node Mapping

Let us consider a substrate network with j nodes and k links. At an arbitrary time instant ω^{Ψ_i}, the VNE solver receives for embedding a VNR Ψ_i that requires ℓ virtual nodes and m virtual links. We define the MDP for VNoM of Ψ_i as a finite-horizon MDP $\mathcal{M}^{\Psi_i}$. The decision-making agent consecutively selects ℓ substrate nodes for embedding nodes of Ψ_i, yielding to ℓ decision-making instances at discrete times t until the horizon $T = \ell$ is reached. If all nodes are successfully mapped, the process reaches its terminal state at $t = \ell + 1$.

The initial state of $\mathcal{M}^{\Psi_i}$ is defined by the tuple:

$$\phi_1^{\Psi_i} = (N_1^{\Psi_i}, N_1^s), \tag{4.46}$$

where $N_1^{\Psi_i}$ is the ordered set of all virtual nodes that are yet to be embedded. Similarly, N_1^s is the set of substrate nodes that are initially available for embedding virtual nodes. We assume that in a given state ϕ_q, the agent tries to identify a substrate node $n^s \in N_q^s$ for embedding the *first element* $n_q^{\Psi^i}$ of the set $N_q^{\Psi_i}$. In the initial state, no virtual node has been embedded and thus all substrate nodes are available for embedding the first virtual node. Hence, $N_1^{\Psi_i} = N^{\Psi_i}$ and $N_1^s = N^s$. The agent selects a substrate node $n^s \in \{N^s(n_1^{\Psi_i}) \cap N_1^s\}$. Therefore, the set of actions is:

$$\mathcal{A}_1^{\Psi_i} = \{\varepsilon\} \cup \{(n_1^{\Psi_i}, n^s) : \forall n^s \in \{N^s(n_1^{\Psi_i}) \cap N_1^s\}\}, \tag{4.47}$$

where ε denotes an arbitrary action that forces the transition to a terminal state. As the result of selecting a substrate node n_a^s for embedding the virtual node $n_1^{\Psi_i}$, the agent receives a reward and $\mathcal{M}^{\Psi_i}$ transits to state:

$$\phi_2^{\Psi_i} = (N_2^{\Psi_i} = N_1^{\Psi_i} \setminus \{n_1^{\Psi_i}\}, N_2^s = N_1^s \setminus \{n_a^s\}). \tag{4.48}$$

This state transition occurs because multiple virtual nodes cannot be embedded into one substrate node. Depending on the choice of the substrate node for embedding the first virtual node $n_1^{\Psi_i}$, there are $|\mathcal{A}_1^{\Psi_i}|$ possibilities for the next state, where $|\mathcal{A}_1^{\Psi_i}|$ denotes the

number of viable actions in state $\phi_1^{\Psi_i}$. The probability of the state transition is:

$$\Pr(\phi_2^{\Psi_i} \mid n_a^s, \phi_1^{\Psi_i}) = 1. \tag{4.49}$$

The second action is then selected from the set:

$$\mathcal{A}_2^{\Psi_i} = \{\varepsilon\} \cup \{(n_2^{\Psi_i}, n^s) : \forall n^s \in \{N^s(n_2^{\Psi_i}) \cap N_2^s\}\}. \tag{4.50}$$

This process continues until $\mathcal{M}^{\Psi_i}$ transits to the ℓ^{th} state:

$$\phi_\ell^{\Psi_i} = (N_\ell^{\Psi_i} = N_{\ell-1}^{\Psi_i} \setminus \{n_{\ell-1}^{\Psi_i}\}, N_\ell^s = N_{\ell-1}^s \setminus \{n_z^s\}), \tag{4.51}$$

where n_z^s denotes the substrate node selected for embedding the virtual node $n_{\ell-1}^{\Psi_i}$. The agent finally selects a substrate node for embedding the last virtual node from:

$$\mathcal{A}_\ell^{\Psi_i} = \{\varepsilon\} \cup \{(n_\ell^{\Psi_i}, n^s) : \forall n^s \in \{N^s(n_\ell^{\Psi_i}) \cap N_\ell^s\}\}. \tag{4.52}$$

The decision-making horizon has been now reached and $\mathcal{M}^{\Psi_i}$ transits to a terminal state where the reward R_ρ is calculated. We assume that the immediate rewards $R_t \, \forall t \leq \ell$ are zero and the agent receives a reward R_ρ only when it reaches a terminal state because partially mapping a virtual network does not necessarily lead to a successful full mapping. Reaching a terminal state when $t < \ell$ implies that for a virtual node n^{Ψ_i}, $N^s(n^{\Psi_i}) = \varnothing$. Therefore, ε that forces $\mathcal{M}^{\Psi_i}$ to its terminal state has been selected. This implies that the VNoM has been unsuccessful. Therefore, $R_\rho = \Gamma$ (4.11). On the contrary, reaching a terminal state when $t = \ell + 1$ implies that the VNoM has been successful. The agent may then proceed to solve VLiM. If VLiM is successful, the VNR is accepted for embedding and the agent receives the reward $R_\rho = \mathbf{R}(G^{\Psi_i}) - \mathbf{C}(G^{\Psi_i})$. Otherwise, the VNR is rejected and $R_\rho = \Gamma$ (4.11).

Link mappings are not considered at the intermediate states of $\mathcal{M}^{\Psi_i}$. However, the agent is unable to select an optimal action policy π^* without knowing the final reward R_ρ, which requires solving the VLiM problem. Hence, finding π^* for $\mathcal{M}^{\Psi_i}$ results a coordinated VNoM solution [63].

4.4.2 Monte Carlo Tree Search for Solving the Virtual Node Mapping

The number of state variables of the proposed MDP $\mathcal{M}^{\Psi_i}$ depends on the number of substrate and virtual nodes $|N^s|$ and $|N^{\Psi_i}|$, respectively. Consequently, the number of states of $\mathcal{M}^{\Psi_i}$ grows exponentially with $|N^s|$ and $|N^{\Psi_i}|$. Hence, finding an exact solution for $\mathcal{M}^{\Psi_i}$ is intractable for even a fairly small $|N^s|$ and $|N^{\Psi_i}|$.

We employ the MCTS algorithm [89] for solving the proposed MDP. Consider the search tree for solving $\mathcal{M}^{\Psi_i}$. Its depth is equal to the horizon T of $\mathcal{M}^{\Psi_i}$. The nodes and edges of

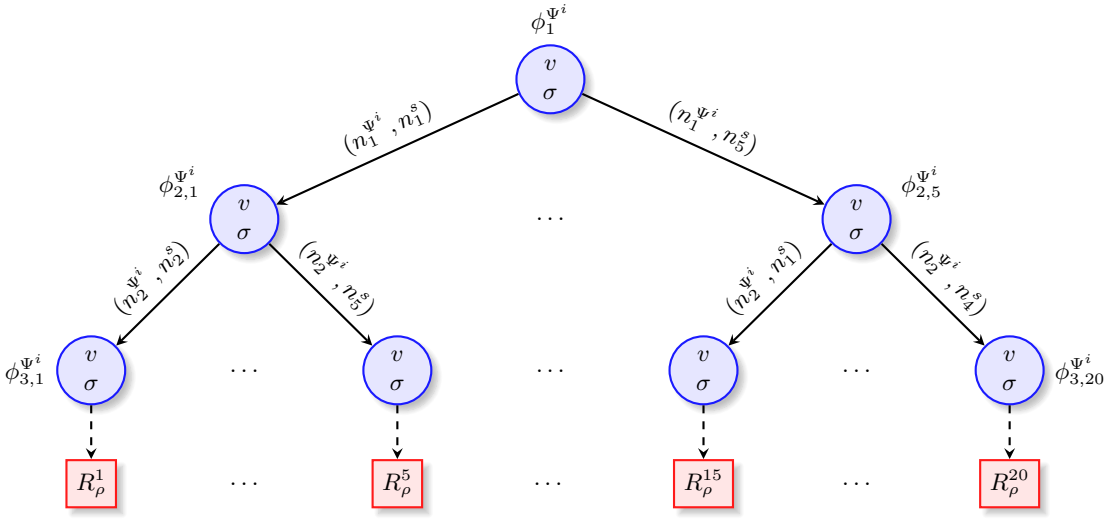

Figure 4.2: Example of a VNoM search tree for embedding a VNR Ψ^i with 3 nodes onto a substrate network with 5 nodes.

the tree correspond to states and actions, respectively. The root of the tree corresponds to the initial state $\phi_1^{\Psi_i}$ of $\mathcal{M}^{\Psi_i}$. Let $|\mathcal{A}_m^{\Psi_i}|$ be the number of available actions at a given state $\phi_m^{\Psi_i}$. The search tree node that corresponds to the state $\phi_m^{\Psi_i}$ is followed by $|\mathcal{A}_m^{\Psi_i}|$ nodes, each corresponding to a possible next state $\phi_{m+1}^{\Psi_i}$ that is encountered as a result of selecting an action $(n_m^{\Psi_i}, n^s) \in \mathcal{A}_m^{\Psi_i}$. Each tree node stores a *value* and a *visit count*. The value of a non-terminal tree node is the cumulative sum of the rewards that have been received in the discovered and reachable terminal nodes. A path from the root to a leaf node defines an action policy π. An example of a VNoM search tree for embedding a VNR with 3 nodes onto a substrate network with 5 nodes is shown in Fig. 4.2.

4.4.3 MaVEn Algorithms

The proposed MaVEn-M and MaVEn-S algorithms employ MCTS to solve the MDP $\mathcal{M}^{\Psi_i}$. Their pseudocode is listed in Algorithm 10. MaVEn-M uses MCF to solve VLiM and to calculate values of terminal states while MaVEn-S uses a breadth first search shortest path algorithm.

Two implementation details should be considered when implementing Algorithm 10: (They have been omitted from the pseudocode due to space constrains.)

Node Creation

When creating the root or creating new nodes (lines 3 and 45), the child array of the node should be initialized. It is important to note that these child nodes are not yet part of the tree. Therefore, although they should have value v and visit count σ, they should be

distinguishable from the nodes that are part of the tree (line 44). This is achieve by setting v and σ while not setting their states.

Child Initialization

When initializing the child array of a node, value v and visit count σ of these child nodes should be set. In this stage, we may rule out the actions that are not viable in the state that corresponds to the parent node. Let us assume that the parent node corresponds to a state $(N_x^{\Psi i}, N_x^s)$. In this state, the goal is to find a substrate node $n^s \in N_x^s$ for embedding the first element of $N_x^{\Psi_i}$ denoted by $n_{current}^{\Psi_i}$. Since the viable actions are $\{n^s : \forall n^s \in N^s(n_{current}^{\Psi_i})\}$, we may set $v = 0$ and $\sigma = 0$ for the child nodes that correspond to these substrate nodes while setting the v and σ of all other child nodes to large negative and positive values, respectively. This ensures that only substrate nodes in $N^s(n_{current}^{\Psi_i})$ are considered as viable choices for embedding. Hence, the UCT strategy (2.14) will not select actions that are not viable.

Algorithm 10 Pseudocode of MaVEn algorithm: MaVEn-M employs the MCF algorithm while MaVEn-S uses a breadth first search-based shortest path algorithm to solve VLiM (lines 21, 64, and 87). The *head* keyword (lines 56, 78) refers to the first element of the set N^{Ψ_i}.

1: **procedure** MaVEn($\Psi_i, G^s(N^s, E^s)$)
2: $\phi \leftarrow (N^{\Psi^i}, N^s)$
3: Create *Root* ($v = 0, \sigma = 0, State = \phi$)
4: $nodesMap[] \leftarrow \varnothing$
5: $vnI \leftarrow 1$
6: **do**
7: $snI \leftarrow$ MCTS (*Root*)
8: **if** $snI \neq \varepsilon$ **then**
9: $nodesMap.Add\ (n^{\Psi_i}_{vnI}, n^s_{snI})$
10: **if** $Root.child[snI].State$ is terminal **then**
11: $terminate \leftarrow true$
12: **else**
13: $vnI \leftarrow vnI + 1$
14: $Root \leftarrow Root.child[snI]$
15: **end if**
16: **else**
17: $terminate \leftarrow$ true
18: **end if**
19: **while** $terminate \neq$ true
20: **if** $nodesMap.Size = |N^{\Psi^i}|$ **then**
21: Solve VLiM given using $nodesMap[]$
22: **else**
23: Reject Ψ^i
24: **end if**
25: **end procedure**
26: **procedure** MCTS(Tree Node TN)
27: **while** $\beta > 0$ **do**
28: $Reward \leftarrow$ Simulate (TN)
29: **if** $Reward = \Gamma$ **then**
30: **return** ε
31: **end if**
32: $TN.v \leftarrow TN.v + Reward$
33: $TN.\sigma \leftarrow TN.\sigma + 1$
34: $\beta \leftarrow \beta - 1$
35: **end while**
36: **return** $\underset{i}{\text{argmax}} \left(\frac{TN.child[i].v}{TN.child[i].\sigma}\right)$
37: **end procedure**
38: **procedure** Simulate(Tree Node TN)
39: $snI = \underset{i}{\text{argmax}} \left(\frac{TN.child[i].v}{TN.child[i].\sigma} D \sqrt{\frac{\ln(TN.\sigma)}{TN.child[i].\sigma}}\right)$
40: $(\phi_{next}, Reward) \leftarrow$ SampleNextState (TN, snI)
41: **if** ϕ_{next} is a terminal state **then**
42: **return** *Reward*
43: **end if**
44: **if** $TN.child[snI]$ does not exist **then**
45: Create a Tree Node: $TN'(v = 0, \sigma = 0, State = \phi_{next})$
46: $TN.child[i] \leftarrow TN'$
47: $Reward \leftarrow$ Rollout ($TN.child[snI]$)
48: **else**
49: $Reward \leftarrow$ Simulate ($TN.child[snI]$)
50: **end if**
51: $TN.child[snI].v \leftarrow TN.child[snI].v + Reward$
52: $TN.child[snI].\sigma \leftarrow TN.child[snI].\sigma + 1$
53: **return** *Reward*
54: **end procedure**
55: **procedure** SampleNextState(Tree Node TN, snI)
56: $n^{\Psi_i}_{current} \leftarrow TN.State.N^{\Psi_i}.head$
57: **if** $n^s_{snI} \in N^s(n^{\Psi_i}_{current})$ **then**
58: $\phi_{next} \leftarrow TN.State$
59: $\phi_{next}.N^{\Psi_i} \setminus \{n^{\Psi_i}_{current}\}$
60: $\phi_{next}.N^s \setminus \{n^s_{snI}\}$
61: **if** ϕ_{next} is a terminal state **then**
62: Find the current policy π by traversing the tree from the root to TN
63: Add the action $(n^{\Psi_i}_{current}, n^s_{snI})$ to π
64: Solve VLiM (SP or MCF) using π
65: Calculate *Reward* based on π and the solution of VLiM
66: **return** $(\phi_{next}, Reward)$
67: **else**
68: **return** $(\phi_{next}, 0)$
69: **end if**
70: **else**
71: **return** (ρ, Γ)
72: **end if**
73: **end procedure**
74: **procedure** Rollout(Tree Node TN)
75: $\phi_{current} \leftarrow TN.State$
76: Find the current policy π by traversing the tree from the root to TN
77: **while** $\phi_{current}$ is not terminal **do**
78: $n^{\Psi_i}_{current} \leftarrow \phi_{current}.N^{\Psi_i}.head$
79: Select a random substrate node: $n^s_{current} \in N^s(n^{\Psi_i}_{current})$
80: **if** $n^s_{current} = \varepsilon$ **then**
81: **return** Γ
82: **end if**
83: Add the action $(n^{\Psi_i}_{current}, n^s_{current})$ to π
84: $\phi_{current}.N^{\Psi^i} \setminus \{n^{\Psi^i}_{current}\}$
85: $\phi_{next}.N^s \setminus \{n^s_{current}\}$
86: **end while**
87: Solve VLiM (SP or MCF) using π
88: Calculate *Reward* based on π and the solution of VLiM
89: **return** *Reward*
90: **end procedure**

4.4.4 Parallelization of MaVEn

MCTS is highly parallelisable [126] and parallelization improves its performance. There are various techniques to successfully parallelize the MCTS process and improve the tree search execution time [45]. Root parallelization is one of the most successful methods to parallelize MCTS [44], [45]. Therefore, we parallelize MaVEn using root parallelization. Let us assume we have p available processors for parallelization. Each processor will be assigned a unique integer number $rank \in \{0, \ldots, p\}$. We assume that the processor with $rank = 0$ is the master processor that is responsible for collecting the information from other processors and selecting the best action.

Unique seed numbers must be assigned to each processor in order to avoid processors from generating identical search trees. For simplicity, we assume that the seed number that is assigned to each processor is equal to its *rank*. The pseudocode of the parallel MaVEn algorithm is shown in Algorithm 11.

Algorithm 11 Pseudocode of parallelized MaVEn algorithm.

```
1: procedure PARALLEL_MAVEN(p, Ψ_i, G^s(N^s, E^s))
2:     if my_rank ∈ {0, ..., p} then
3:         φ ← (N^{Ψ^i}, N^s)
4:         Create Root (v = 0, σ = 0, State = φ)
5:         nodesMap[] ← ∅
6:         vnI ← 1
7:         do
8:             snI ← MCTS (Root)
9:             if snI ≠ ε then
10:                nodesMap.Add (n^{Ψ_i}_{vnI}, n^s_{snI})
11:                if Root.child[snI].State is terminal then
12:                    terminate ← true
13:                else
14:                    vnI ← vnI + 1
15:                    Root ← Root.child[snI]
16:                end if
17:            else
18:                terminate ← true
19:            end if
20:        while terminate ≠ true
21:        if nodesMap.Size = |N^{Ψ^i}| then
22:            linksMap[] ← VLiM (G^s, Ψ_i, nodesMap[])
23:            Calculate F(G^{Ψ_i}) using linksMap[] and nodesMap[])
24:            if my_rank ≠ 0 then
25:                Send my_rank and F(G^{Ψ_i}) to the master processor (rank = 0)
26:            else
27:                Receive F_m(G^{Ψ_i}) from processors rank = m  ∀m ∈ {1, ..., p}
28:                Identify the rank_max of the processor that has the highest F(G^{Ψ_i})
29:                Receive linksMap_max[] and nodesMap_max[] from the processor rank_max
30:                Send the linksMap_max[] and nodesMap_max[] to all other processors
31:            end if
32:        else
33:            Reject Ψ^i
34:        end if
35:    end if
36: end procedure
```

4.5 Performance Evaluation

In this Section, we present the simulation results that are used to compare the proposed and the existing VNE algorithms. Simulations were performed on a Dell Optiplex-790 with 16 GB memory and the Intel Core i7 2600 processor. The simulation scenarios are organized based on the topology of the substrate network:

1. We present the simulation scenarios that employ a substrate network resembling an ISP network topology. The parameters used in these scenarios are adopted from the literature [48], [51], [69]. In these scenarios, we first compare the algorithms by keeping the VNR traffic load constant while varying the MCTS computational budget β from 5 to 250 samples per node embedding. We then compare the algorithms by increasing the VNR traffic load. We also evaluate the Parallel MaVEn-M and MaVEn-S algorithms using 2, 4, 6, and 8 processes by increasing the VNR traffic load and compare their performance with the serial MaVEn-M and MaVEn-S algorithms.

2. We present the scenarios that employ the BCube and Fat-Tree data center network topologies. First we compare the performance of the algorithms. We then compare the two data center network topologies to determine the topology better suited for virtual network embeddings. These topologies have not been employed yet for comparing VNE algorithms. Therefore, for selecting the substrate network resource capacities and virtual network requests resource requirements, we use parameters that resemble a realistic scenario.

4.5.1 Simulation Environment

In order to evaluate performance of the proposed MaVEn-M and MaVEn-S algorithms, we developed a discrete event simulator *VNE-Sim* based on the DEVS framework [135]. VNE-Sim is written in C++ and provides the base classes and operations needed to simulate VNE algorithms. We implement MaVEn-M, MaVEn-S, D-Vine [51], R-Vine [51], and GRC [69] algorithms and compare their acceptance ratio (4.8), revenue to cost ratio, profitability (4.18), and average execution time per VNR embedding. In simulations, the exploration constant is set to $D = 0.5$ [122]. Parameters for Vine and GRC algorithms are adopted from [51] and [69], respectively. The implemented GRC algorithm is modified to consider the preference criteria of the virtual node location (4.3).

The GNU Scientific Library's random number generator [15] is used to generate random numbers and the necessary distributions. We use the MT19937 *Mersenne Twister* [103] random number generator with the seed value 0. The GNU Linear Programming Kit (GLPK) [14] is used for solving the MCF problem.

4.5.2 Internet Service Provider Substrate Network Topology

In simulations, we use topology, substrate resources, and VNR requirements that have been adopted in the literature [48], [51], [69]. The Boston University Representative Topology Generator (BRITE) [13] is used to generate the substrate and VNR network graphs. The substrate graph consists of 50 nodes that are randomly placed on a 25×25 Cartesian plane [51]. Connections between the nodes are generated based on the Waxman algorithm [140] with the parameter $\alpha = 0.5$ and the exponential parameter $\beta = 0.2$ [156]. Each substrate node is connected to a maximum of 5 nodes. The generated substrate network graph has 221 edges. The VNR graphs are generated using the same process. The number of nodes in VNR graphs is uniformly distributed between 3 and 10 [51]. Each virtual node is connected to a maximum of 3 virtual nodes.

The CPU capacity of substrate nodes and the bandwidth of substrate links are uniformly distributed between 50 and 100 units. The CPU requirements of the virtual nodes are uniformly distributed between 2 and 20 while the bandwidth requirements of the virtual links are uniformly distributed between 0 and 50 [51]. The maximum allowable distance δ for embedding VNR nodes is uniformly distributed between 15 and 25 distance units.

We assume that the VNRs arrive according to a Poisson process with a mean arrival rate of λ requests per unit time. Their life-times are exponentially distributed with a mean $\frac{1}{\mu}$ yielding to a VNR traffic of $\lambda \times \frac{1}{\mu}$ Erlangs. For simulation scenarios, we assume $\frac{1}{\mu} = 1,000$. The duration of each simulation scenario is 50,000 time units [48], [51], [69].

MaVEn Computational Budget Scenarios

In these simulation scenarios, we assume the constant VNR arrival rate of 2 requests per 100 time units yielding to traffic load of 20 Erlangs. We vary the computational budget β between 5 and 250 samples per node embedding. Acceptance ratio, revenue to cost ratio, and profitability of the algorithms are shown in Fig. 4.3, Fig. 4.4, and Fig. 4.5, respectively.

MaVEn algorithms are capable of finding VNE embeddings that are more profitable than those found by the existing algorithms. The proposed MaVEn-M algorithm achieves the highest acceptance ratio even with a small computational budget. It further improves the revenue to cost ratio as the computational budget β increases, which results in higher profitability. With the smallest computational budget $\beta = 5$, the VNE embeddings generated by MaVEn-M are 45% and 65% more profitable than those identified by Vine and GRC algorithms, respectively. The performance of MaVEn-S algorithm in terms of acceptance ratio is comparable to R-Vine and D-Vine algorithms. However, MaVEn-S finds embeddings with lower revenue to cost ratios, which results in higher profitability.

Node and link utilizations of the algorithms as functions of computation budged β are shown in Fig. 4.6 (top) and Fig. 4.6 (bottom), respectively. Even though high node

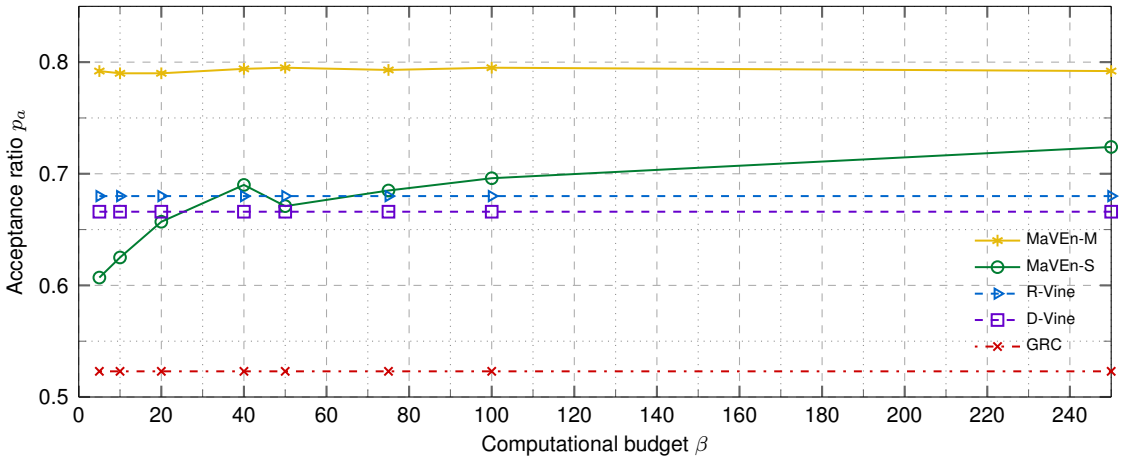

Figure 4.3: Comparison of the algorithms with a VNR traffic load of 20 Erlangs. Shown are the acceptance ratios as functions of computation budged β.

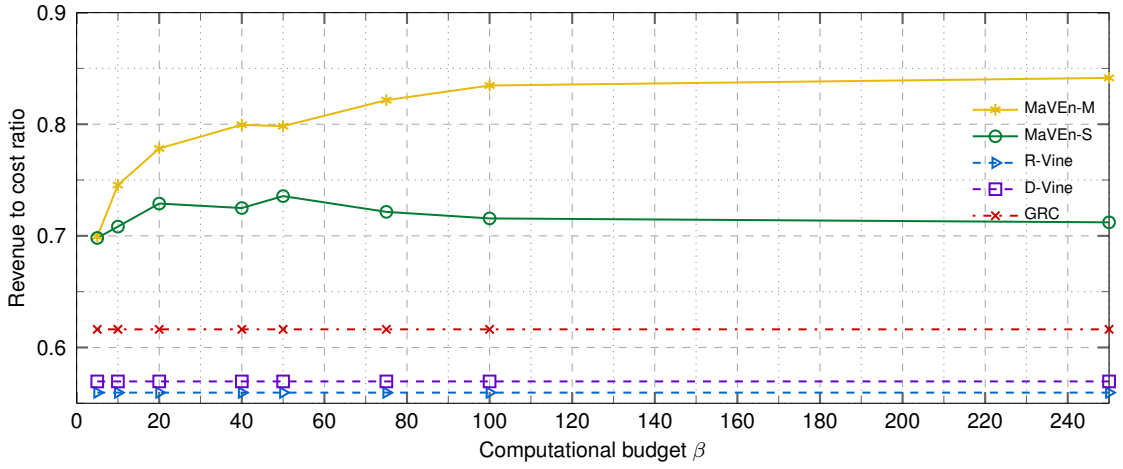

Figure 4.4: Comparison of the algorithms with a VNR traffic load of 20 Erlangs. Shown are the revenue to cost ratios (middle) as functions of computational budget β.

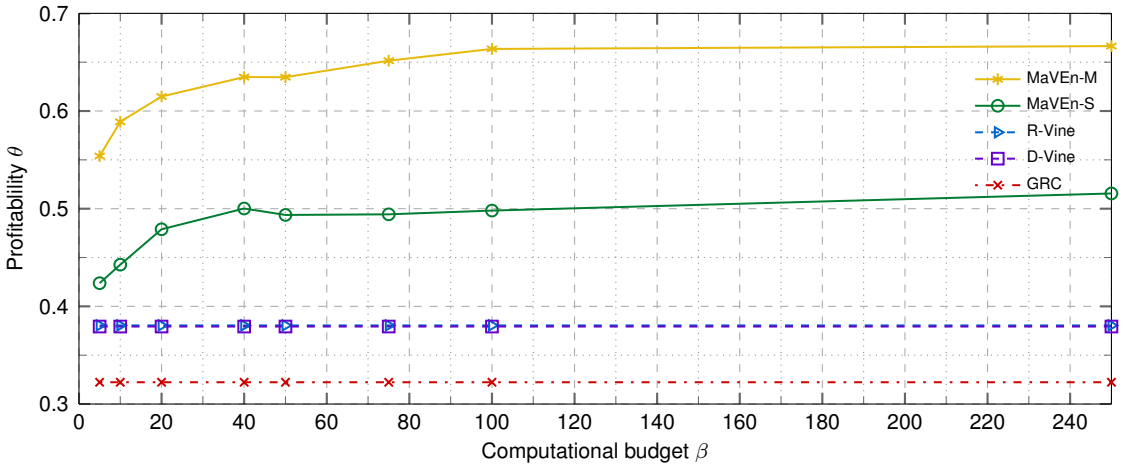

Figure 4.5: Comparison of the algorithms with a VNR traffic load of 20 Erlangs. Shown are the profitabilities as functions of computational budget β.

utilization is always desirable, the link utilization should be as small as possible. In the case of MaVEn-M, average node utilization is high as a result of the high acceptance ratio. As the computational budget is increased, MaVEn-M finds less costly embeddings thus reducing the link utilization.

The average processing time per VNR as a function of computational budget β is shown in Fig. 4.7. MaVEn-M requires considerably higher computational time compared to the other algorithms. The GRC algorithm achieves the best performance.

4.5.3 Variable Virtual Network Request Arrival Rate Scenarios

In these simulation scenarios, we assume that the computational budget β is 40 samples per node embedding while the VNR arrival rate is varied between 1 and 8 requests per 100 time units yielding to traffic loads of 10, 20, 30, 40, 50, 60, 70, and 80 Erlangs [69].

Acceptance ratio, revenue to cost ratio, and profitability of the algorithms are shown in Fig. 4.8, Fig. 4.9, and Fig. 4.10, respectively. The proposed algorithms achieve better performance compared to the existing algorithms. MaVEn-M has the best acceptance ratio in all scenarios while its revenue to cost ratio drops quickly below MaVEn-S at traffic load of 60 Erlangs. This leads to profitability comparable to MaVEn-S. Because the substrate network resources become scarce at higher VNR traffic loads, there is a higher likelihood of embedding virtual nodes onto substrate nodes that are further apart, which results in more costly embeddings [69]. Although at higher traffic loads the profitabilities of MaVEn-M and MaVEn-S are comparable, MaVEn-M may be preferred because it has higher acceptance ratio, which may result in higher customer satisfaction.

Node and link utilizations of the algorithms as functions of VNR traffic load are shown in Fig. 4.11 (top) and Fig. 4.11 (bottom), respectively. MaVEn-M achieves the highest node

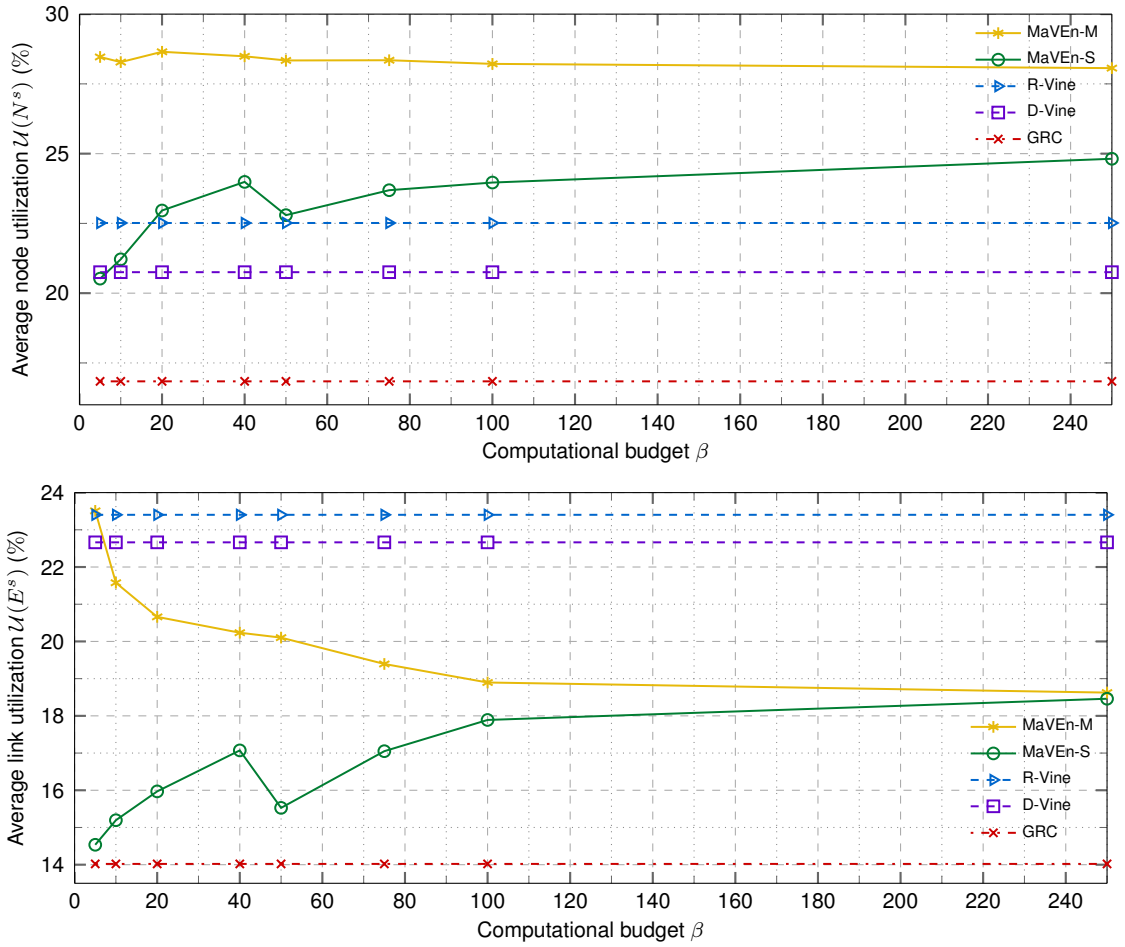

Figure 4.6: Comparison of the algorithms with a VNR traffic load of 20 Erlangs. Shown are node utilization (top) and link utilization (bottom) as functions of computational budget β.

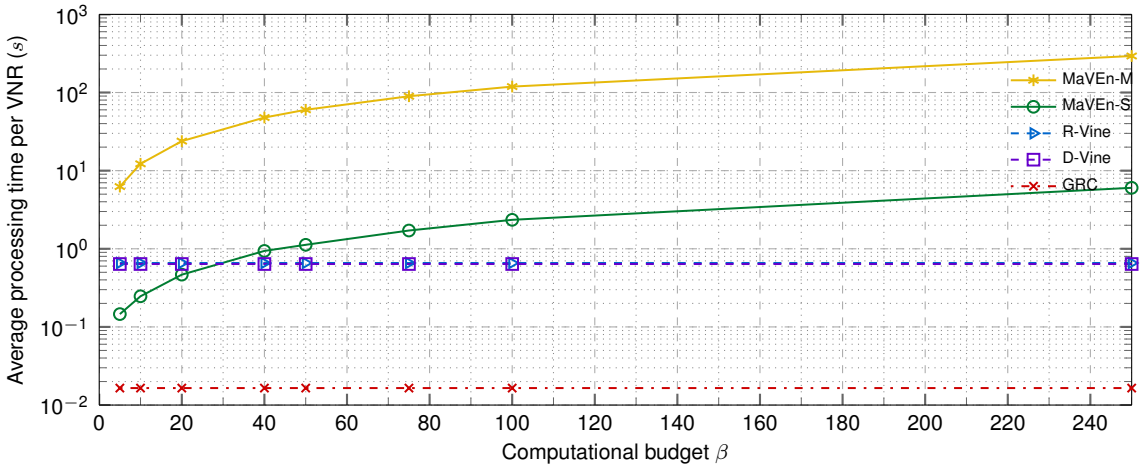

Figure 4.7: Comparison of the algorithms with a VNR traffic load of 20 Erlangs. Shown are the average processing times per VNR as functions of computational budget β.

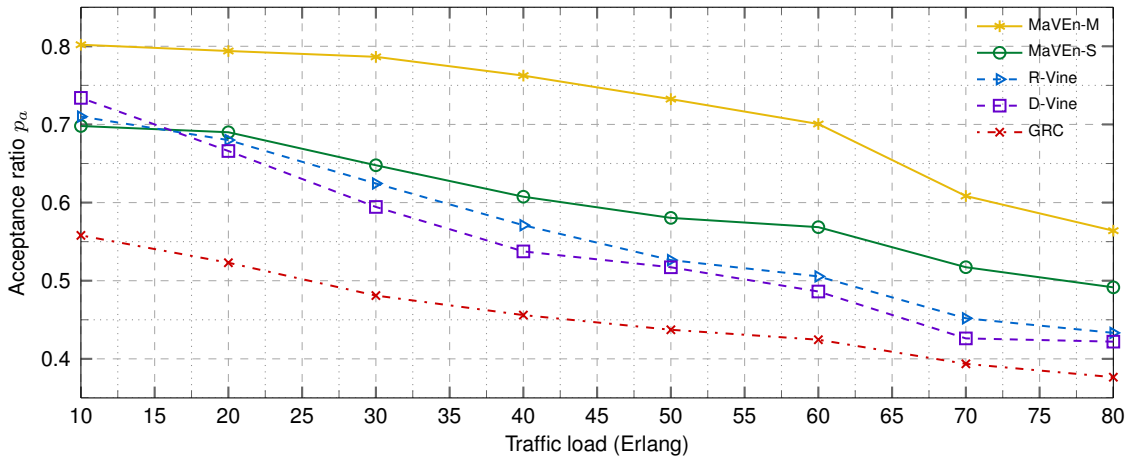

Figure 4.8: Performance of the algorithms with various VNR traffic loads. The MaVEn computational budget is $\beta = 40$ samples per virtual node embedding. Shown are the acceptance ratios as functions of VNR traffic load.

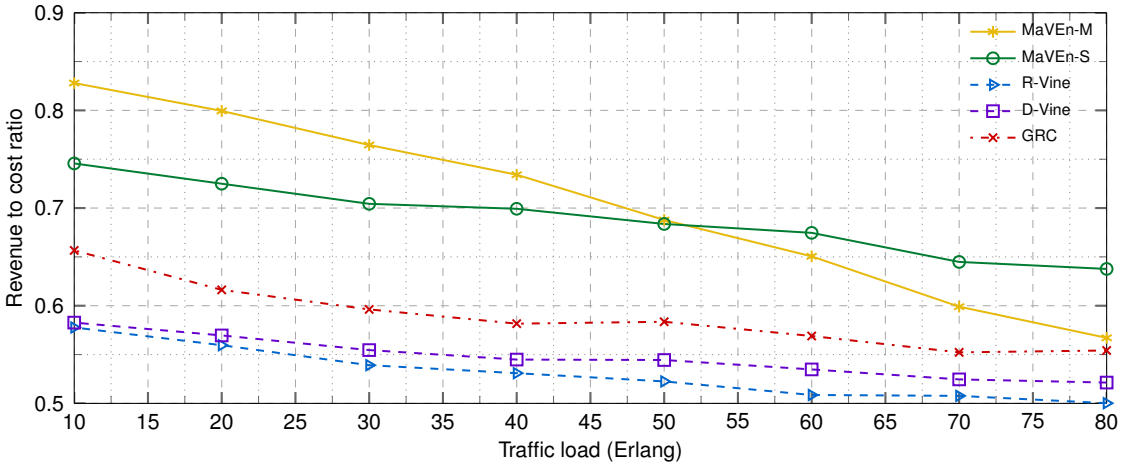

Figure 4.9: Performance of the algorithms with various VNR traffic loads. The MaVEn computational budget is $\beta = 40$ samples per virtual node embedding. Shown are revenue to cost ratios as functions of VNR traffic load.

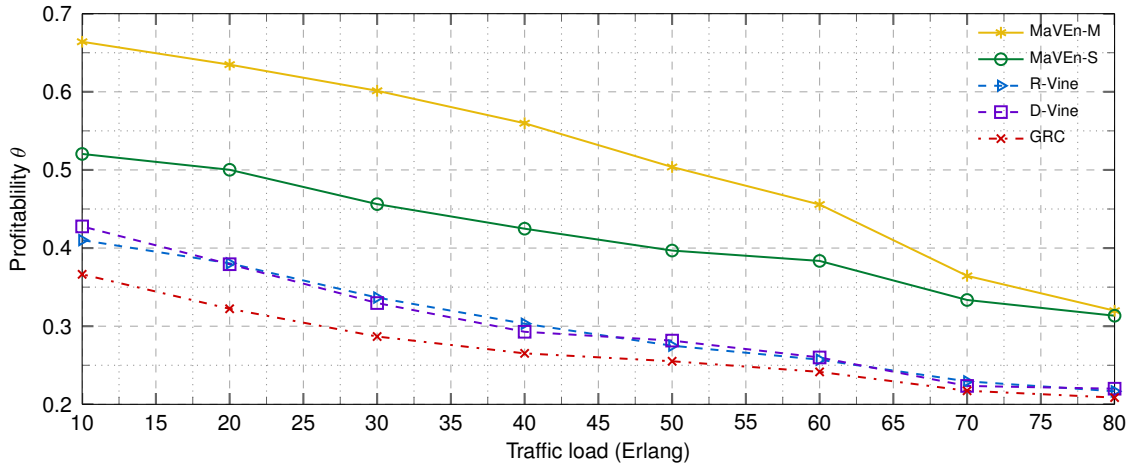

Figure 4.10: Performance of the algorithms with various VNR traffic loads. The MaVEn computational budget is $\beta = 40$ samples per virtual node embedding. Shown are profitabilities as functions of VNR traffic load.

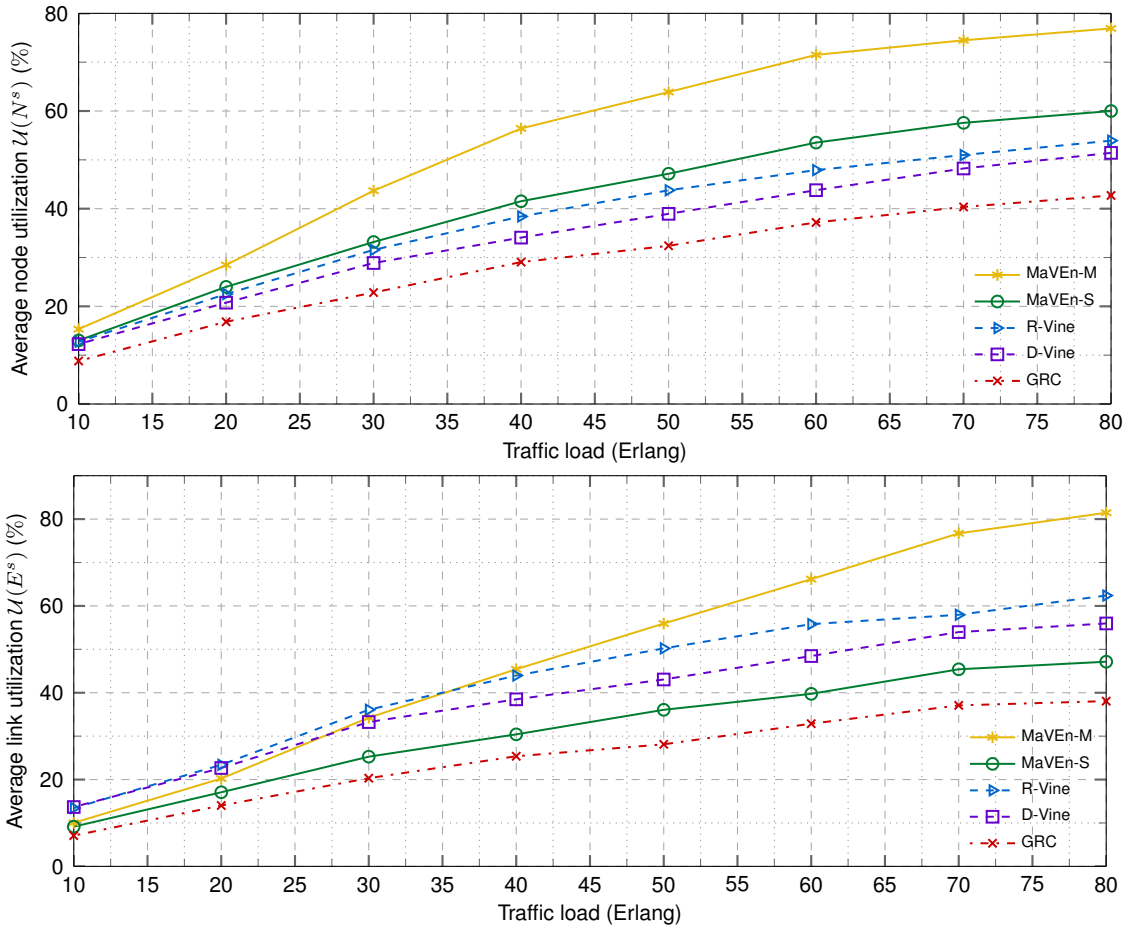

Figure 4.11: Performance of the algorithms with various VNR traffic loads. The MaVEn computational budget is $\beta = 40$ samples per virtual node embedding. Shown are node (top) and link (bottom) utilities as functions of VNR traffic load.

utilization in all scenarios. MaVEn-S utilizes the nodes better than R-Vine, D-Vine, and GRC algorithms. The small link utilization of GRC is attributed to its small node utilization. Therefore, in this case, small link utilization is not desirable because this situation leaves the resources under utilized. MaVEn-S achieves small link utilizations in all scenarios while achieving high node utilization. This is translated to large acceptance and revenue to cost ratios for MaVEn-S, as shown in Fig. 4.8 and Fig. 4.9, respectively.

The average execution time of the algorithms per VNR embedding is shown in Fig. 4.12. Although GRC achieves the smallest execution time, it does not utilize the substrate network resources efficiently. MaVEn-S achieves an average execution time comparable to the Vine algorithms while better utilizing the resources.

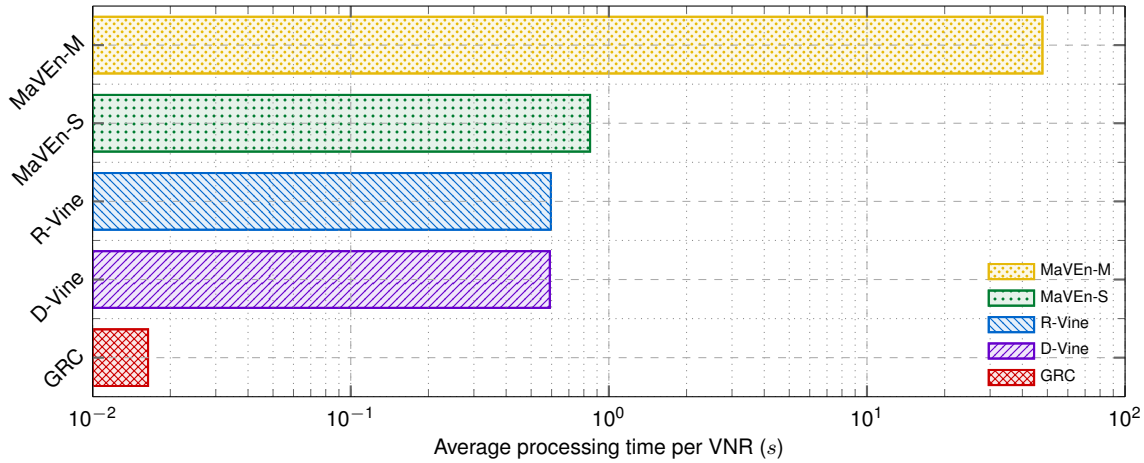

Figure 4.12: Average execution time of the algorithms per VNR embedding. Execution times are averaged for all VNR traffic load scenarios.

4.5.4 Parallel MaVEn Simulation Scenarios

In these simulation scenarios, we employ root parallelization and execute the MaVEn algorithms in parallel on 2, 4, 6, and 8 processors. We assume that the seed number assigned to each processor is equal to its *rank*. We employ the MPICH2 library [7] that implements the Message Passing Interface (MPI) standard [6]. All simulations were executed on a single Intel Core i7 2600 Quad-Core CPU that employs the Hyper-Threading technology enabling it to execute up to 8 parallel threads. The computational budget and VNR arrival rates are similar to the *Variable VNR Arrival Rate Simulation Scenarios* described in Subsection 4.5.3.

Performance of the parallel MaVEn-M algorithm is shown in Fig. 4.13 and Fig. 4.14. While the parallelization does not have a large impact on the acceptance ratio, it improves the revenue to cost ratio up to 10%. Hence, it results in identifying more profitable virtual network embeddings. The processing time of the algorithms is proportional to the number of used processors. Two factors contribute to the increase in the MaVEn-M processing time: disk I/O operations required by the MCF solver and the communication between processors. The I/O operations have the prevailing effect on the increase in processing time. Since the simulations were executed on a single machine, all processors share a common disk. Therefore, it is impossible to perform the disk I/O operations in parallel. As the number of processors increases, the likelihood of multiple processors accessing the disk at the same time increases, resulting in higher execution time.

Performance of the parallel MaVEn-S algorithm is shown in Fig. 4.15 and Fig. 4.16. Parallelization improves its acceptance and revenue to cost ratios resulting in increased profitability by up to 10%. The Intel Core i7 2600 Quad-Core CPU used for simulations employs hyper-threading where the operating system views each physical processor (core)

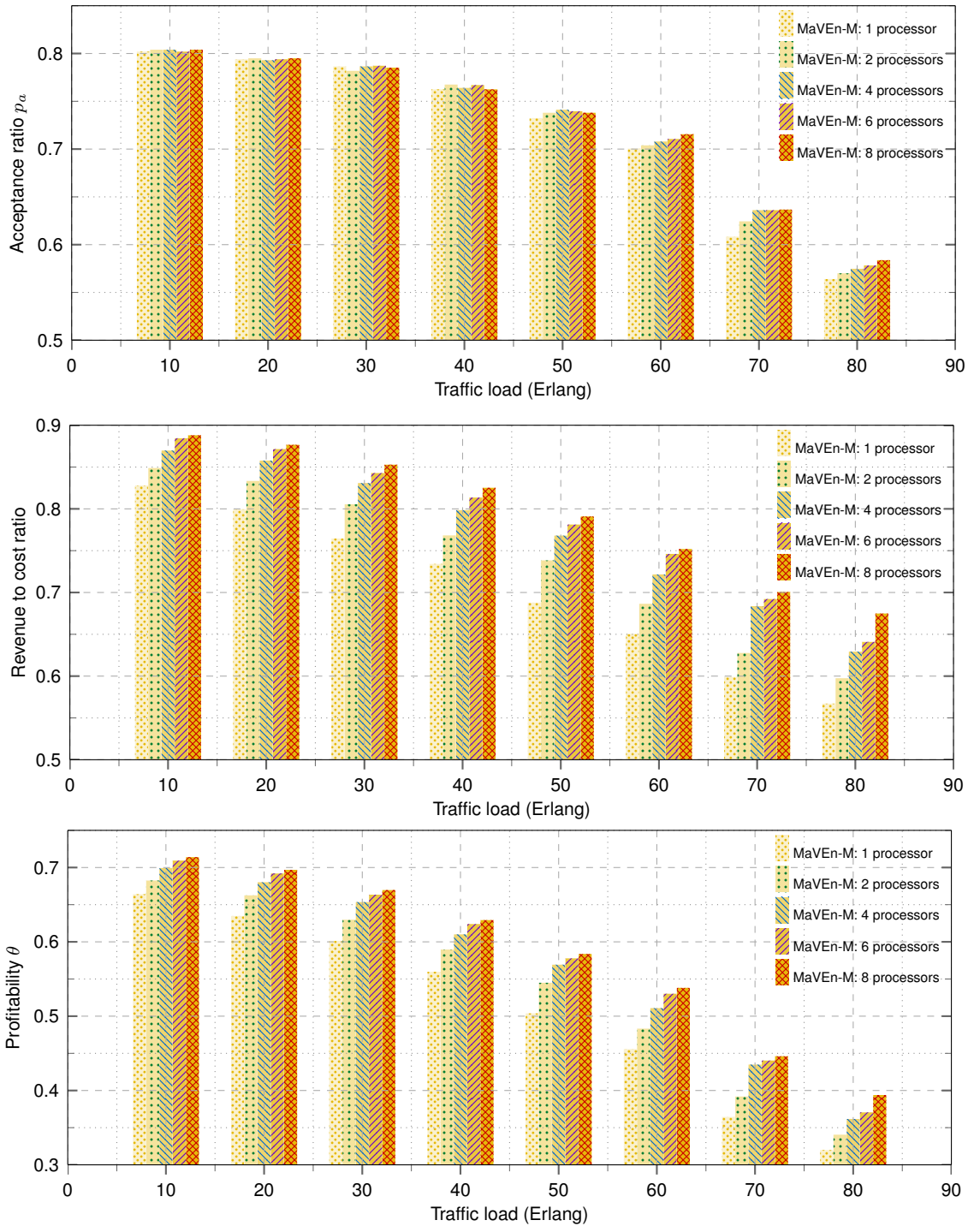

Figure 4.13: Acceptance ratio, revenue to cost ratio, and profitability of the parallel MaVEn-M algorithm with various VNR traffic loads using 1, 2, 4, 6, and 8 processors. The computation budget is $\beta = 40$ samples per virtual node embedding.

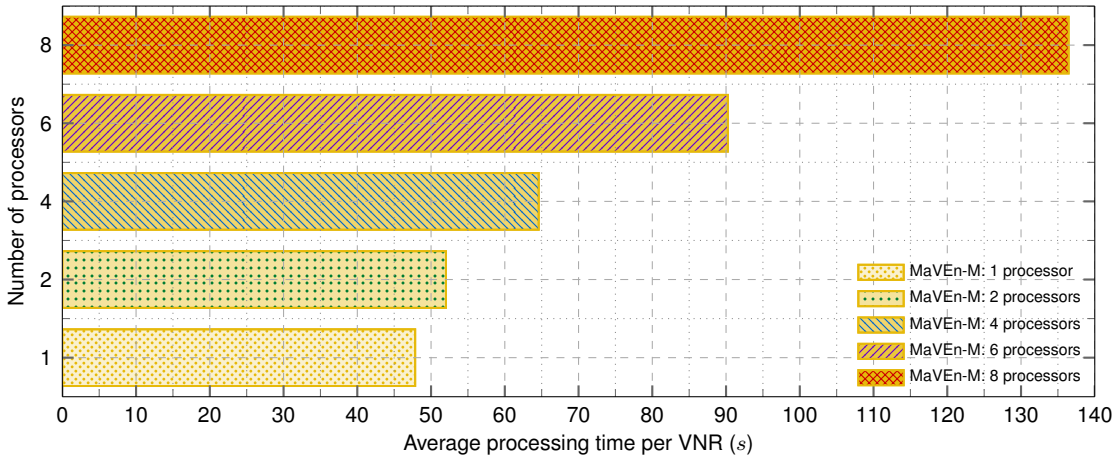

Figure 4.14: Average processing time of the parallel MaVEn-M algorithm per VNR embedding with various VNR traffic loads using 1, 2, 4, 6, and 8 processors. The computation budget is $\beta = 40$ samples per virtual node embedding.

as two logical processors. In a hyper-threaded architecture, physical execution resources are shared and the architecture state is duplicated [102]. Parallelization with MPI on a hyper-threaded architecture increases competition for accessing network and for the memory hierarchy resources [120]. Therefore, using up to 4 processors has no impact on the processing time of MaVEn-S because all threads are executed on physical processors. However, processing time of the algorithm increases when utilizing 6 or 8 processors because in these cases hyper-threading is employed.

4.5.5 Data Center Substrate Networks

Fast Network Simulation Setup (FNSS) [121] is used to generate the BCube and Fat-Tree substrate topologies. The BCube topology is a two-level ($k = 2$) topology with 4 hosts per BCube level-0 ($n = 4$) that consists of 64 hosts, 48 switches, and 192 link. The Fat-Tree substrate network graphs is generated using 6-port switches ($k = 6$). It consists of 54 hosts, 45 switches, and 162 links. The BCube and Fat-Tree topologies were selected to have comparable number of hosts, switches, and links. Because of the memory and simulation duration requirements, we have simulated data center topologies that are much smaller than deployed data centers. The VNR graphs are generated using the Boston University Representative Topology Generator (BRITE) [13]. Connections between the nodes in VNR graphs are generated based on the Waxman algorithm [140] with the parameter $\alpha = 0.5$ and the exponential parameter $\beta = 0.2$ [156]. The number of nodes is uniformly distributed between 3 and 10 [51]. Each virtual node is connected to a maximum of 3 virtual nodes.

The CPU capacity of all substrate hosts and the available bandwidth of all substrate links is initially set to 100 units. In our implementation, the substrate network switches have no CPU capacity because they may not be used for virtual node embedding. The CPU

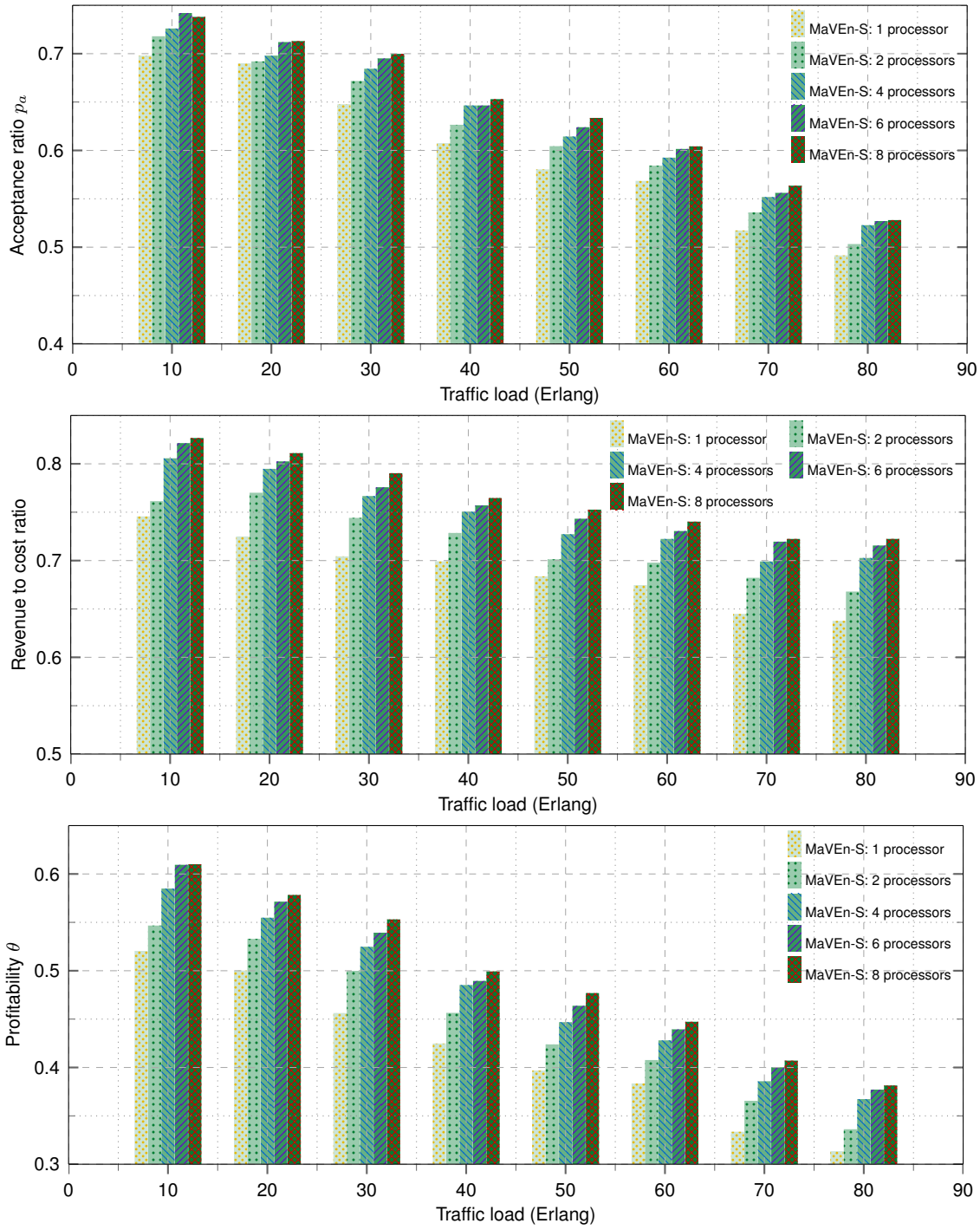

Figure 4.15: Acceptance ratio, revenue to cost ratio, and profitability of the parallel MaVEn-S algorithm with various VNR traffic loads using 1, 2, 4, 6, and 8 processors. The computation budget is $\beta = 40$ samples per virtual node embedding.

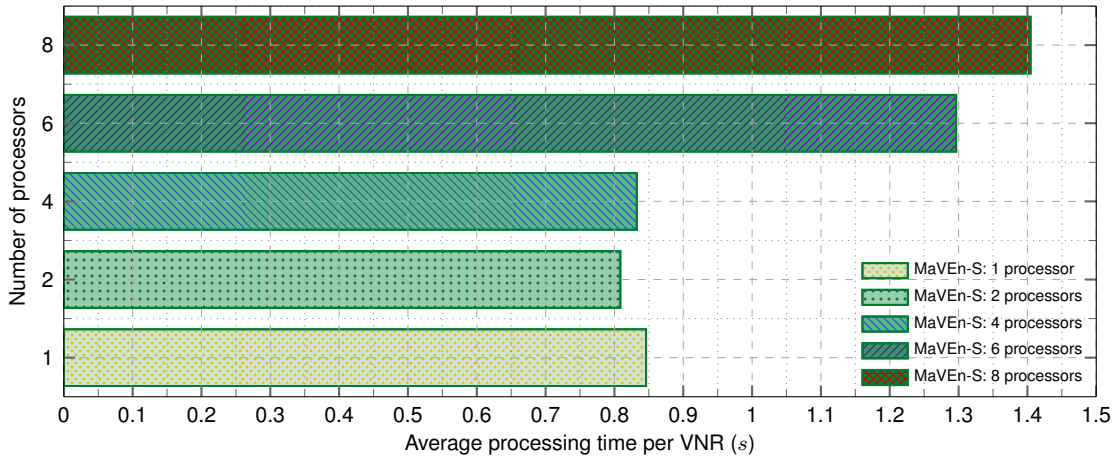

Figure 4.16: Average processing time of the parallel MaVEn-S algorithm per VNR embedding with various VNR traffic loads using 1, 2, 4, 6, and 8 processors. The computation budget is $\beta = 40$ samples per virtual node embedding.

requirements of the virtual nodes are uniformly distributed between 2 and 20 [51] while the bandwidth requirements of the virtual links are uniformly distributed between 1 and 10. The assigned bandwidth values are chosen to resemble a substrate network with 10 Gbps links and virtual networks with 100 Mbps to 1 Gbps links.

In these simulation scenarios, we assume that the computational budget β is 5 samples per node embedding while the VNR arrival rate is varied between 1 and 8 requests per 100 time units yielding to traffic loads of 10, 20, 30, 40, 50, 60, 70, and 80 Erlangs [69].

Acceptance ratio, revenue to cost ratio, profitability, and average execution time per VNR embedding of the algorithms are shown in Fig. 4.17, Fig. 4.18, Fig. 4.19, and Fig. 4.20, respectively. The MaVEn algorithms achieve superior performance in both BCube and Fat-Tree substrate networks. They achieve high acceptance and revenue to cost ratios yielding to high profitability. The margin of the performance improvement increases as the VNR traffic load is increased. For example, as shown in Fig. 4.17 (top), all algorithms achieve acceptance ratio close to 1 when VNR traffic load is 10 Erlangs. However, in the case of 60 Erlangs, MaVEn algorithms achieve acceptance ratios close to 0.95 while the acceptance ratios of other algorithms are below 0.7.

Comparing the BCube and Fat-Tree topologies reveals that in the case of Vine and GRC algorithms, the Fat-Tree topology results in up to 10% higher acceptance ratio as the VNR traffic load increases. This yields to up to 20% higher node utilization and up to 10% higher link utilization in the case of Fat-Tree topology. The revenue to cost ratio of the Fat-Tree topology is slightly lower than the BCube topology. Note that a desirable VNE should exhibit high acceptance ratio, substrate resource utilization, and revenue to cost ratio.

An explanation of the simulation results is that in BCube topologies, hosts perform traffic forwarding functions while in Fat-Tree topologies, traffic forwarding is only performed

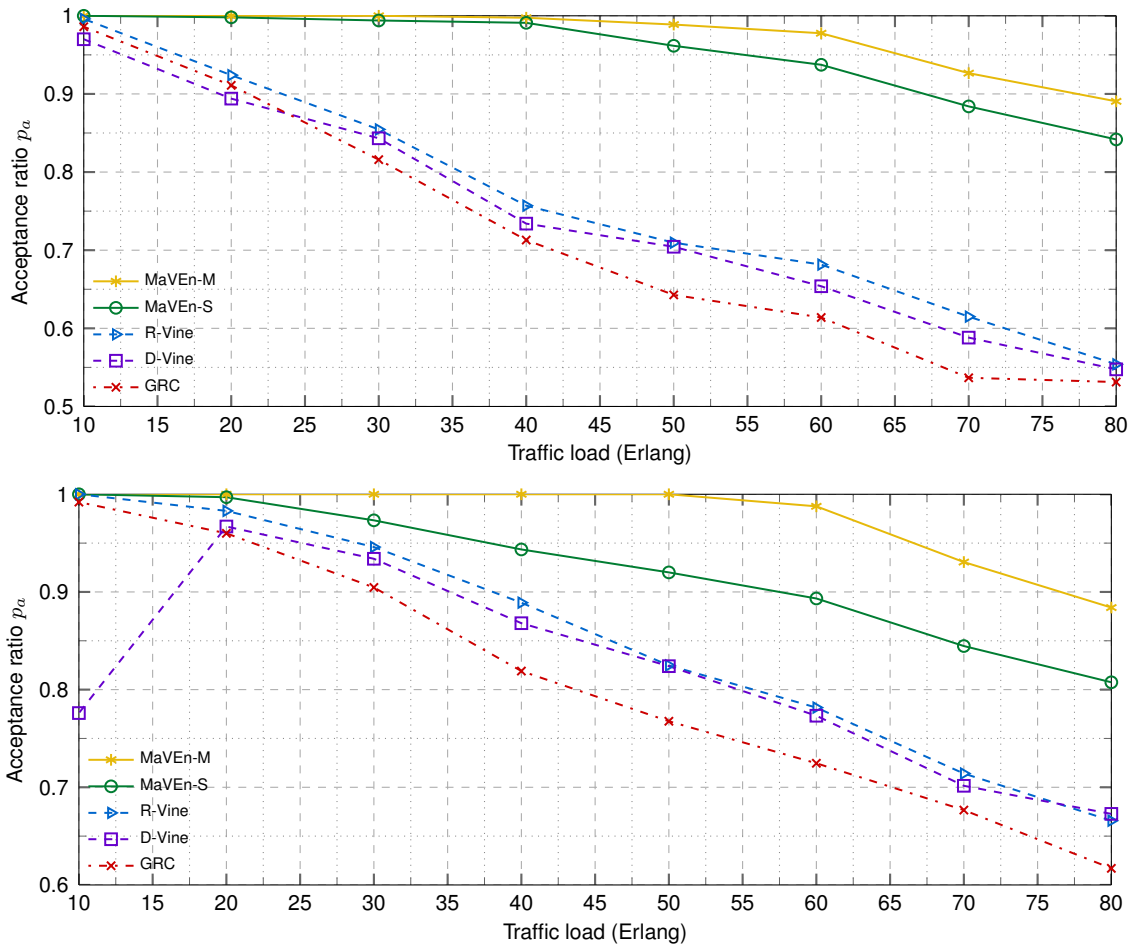

Figure 4.17: Performance of the algorithms with various VNR traffic loads. The MaVEn computational budget is $\beta = 5$ samples per virtual node embedding. Shown are the acceptance ratios as functions of VNR traffic load in the BCube (top) and Fat-Tree (bottom) scenarios.

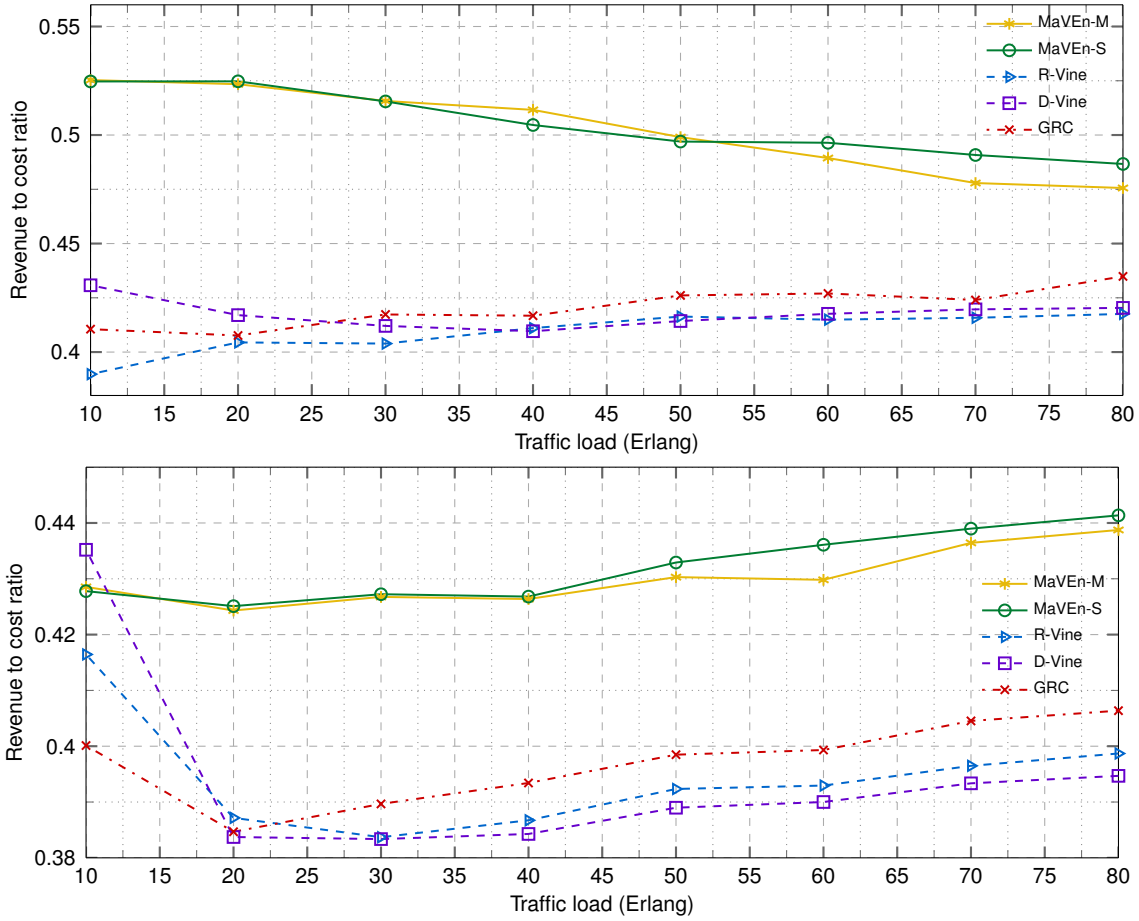

Figure 4.18: Performance of the algorithms with various VNR traffic loads. The MaVEn computational budget is $\beta = 5$ samples per virtual node embedding. Shown are the revenue to cost ratios as functions of VNR traffic load in the BCube (top) and Fat-Tree (bottom) scenarios.

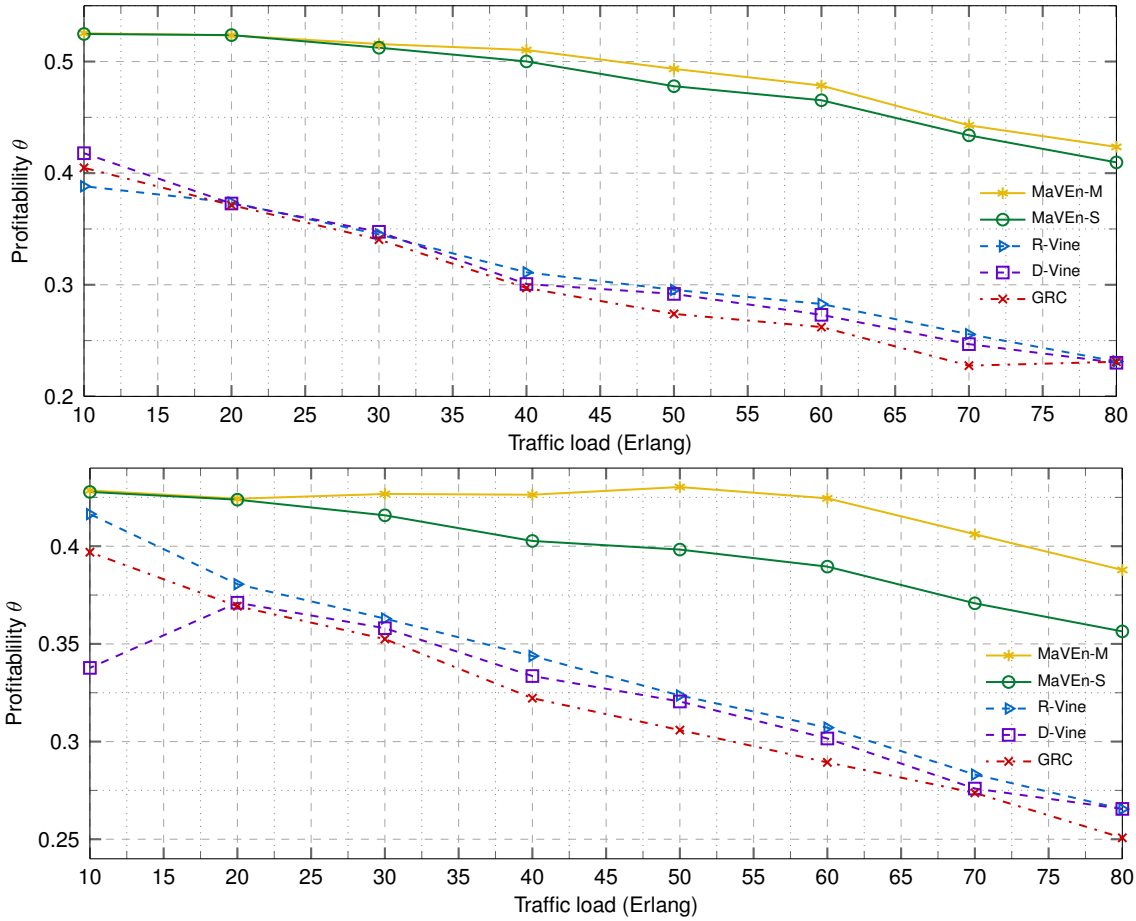

Figure 4.19: Performance of the algorithms with various VNR traffic loads. The MaVEn computational budget is $\beta = 5$ samples per virtual node embedding. Shown are the profitabilities as functions of VNR traffic load in the BCube (top) and Fat-Tree (bottom) scenarios.

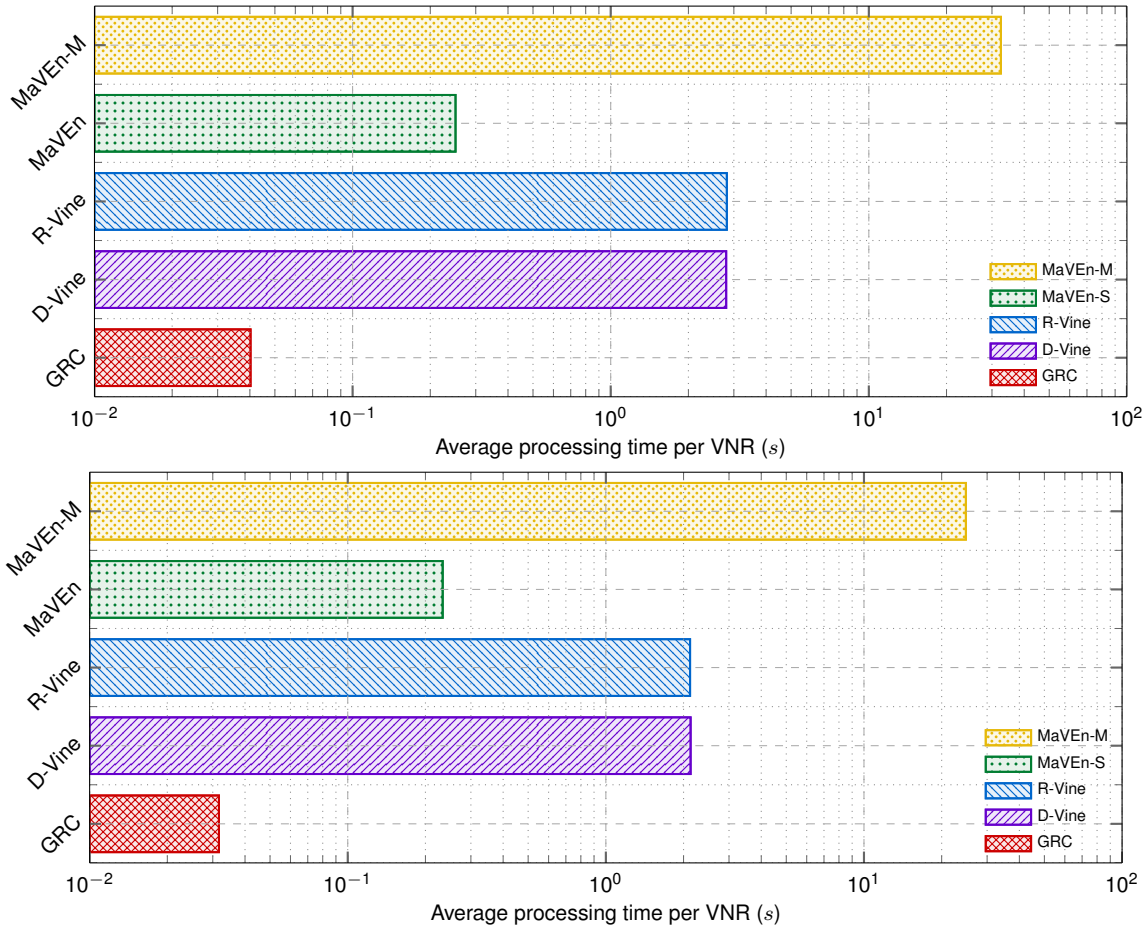

Figure 4.20: Average execution time of the algorithms per VNR embedding in the BCube (top) and Fat-Tree (bottom) scenarios. Execution times are averaged for all VNR traffic load scenarios.

by the switches. In the case of BCube topology, this introduces additional traffic over the links that are connected to the hosts. These links are important for the virtual network embeddings especially for embedding virtual nodes that require multiple connections to other virtual nodes. Therefore, performing traffic forwarding using only the core switches instead of the hosts could potentially lead to higher VNR acceptance ratio. The generated Fat-Tree topology has higher switch to host ratio (0.84) compared to the BCube topology (0.75). This provides additional paths between the hosts that, in turn, may result in higher acceptance ratio.

4.6 Discussion

In this Chapter, we modeled the Virtual Network Embedding (VNE) problem as a Markov Decision Process (MDP) and proposed two new VNE algorithms (MaVEn-M and MaVEn-S) that solve the proposed MDP by utilizing the Monte Carlo Tree Search. We developed a discrete event simulator for simulating VNE algorithms and implemented MaVEn-M, MaVEn-S, R-Vine, D-Vine, and GRC algorithms for comparisons. The simulation results show that MaVEn-M and MaVEn-S exhibit promising performance. The advantage of the proposed algorithms is that, time permitting, they search for more profitable embeddings compared to the available algorithms that identify only a single embedding solution.

The MaVEn-S and GRC algorithms use a shortest-path-based algorithm without path splitting to solve VLiM, which is stricter than the MCF algorithm that enables path splitting utilized by MaVEn-M and Vine algorithms. For example, consider a virtual node $n_x^{\Psi_i}$ attached to a virtual link that requires 5 units of bandwidth and a substrate node n_y^s attached to two substrate links with available bandwidths of 4 and 1. While utilizing MCF permits embedding $n_x^{\Psi_i}$ onto n_y^s, such embedding is infeasible without path splitting.

Superior performance of the proposed algorithms comes at the cost of higher execution time. The GRC algorithm has the lowest execution time. However, the solutions found are less profitable compared to other algorithms. The stochastic processes governing the VNRs arrival and life-time distributions have not been well investigated [51]. Hence, it is difficult to estimate a reasonable trade-off between execution time and profitability. A five-year traffic analysis (2002 to 2007) of the research-based ProtoGENI project [9] identified an average of 4 VNR arrivals per hour [148]. Based on this low arrival rate, we may assume that allocating a few minutes of processing time is feasible. Even though we have not considered varying the computational budget during simulations, InP operators may adjust the execution time of the MaVEn algorithms according to VNR arrival rates and the size of VNR graphs to avoid long queuing delays. Furthermore, MCTS is highly parallelisable [126] and, therefore, larger number of processors may be employed to achieve scalability in larger network topologies.

We have used the GLPK to solve the MCF problem, which currently does not support distributed computing. Furthermore, our implementation of the shortest-path algorithm relies only on the random access memory while the MCF implementation relies on slow disk I/O operations. Parallelizing the MCF and eliminating the disk I/O improves the performance of MaVEn-M and Vine algorithms.

MCTS parallelization improves performance of the MaVEn algorithms. We simulated the parallel MaVEn algorithms using multiple threads of a single CPU. However, the parallelized algorithms may be executed on clusters that are highly optimized for parallel computing and comprise large number of processors.

We simulated data center topologies that are much smaller than the currently deployed data center networks. Larger data center topologies possess the same structure as employed in simulations. However, we did not evaluate the effects of topology size on performance of VNE algorithms.

The VNE algorithms considered in this dissertation are designed without assuming any specific structure of substrate and virtual networks that may simplify the embedding process. The complexity of these algorithms imposes a barrier on their scalability. For example, it has been reported that the execution time of Vine algorithms for embedding virtual networks in a BCube network topology with 512 servers is prohibitively high [72]. Therefore, considering network structure will help achieve scalability. The MCTS algorithm, employed by the MaVEn algorithms, may easily be extended to consider network structure (domain knowledge). Employing domain knowledge enables MCTS to direct the search to promising subtrees without affecting asymptotic convergence [126] and, hence, achieve better performance.

Chapter 5

VNE-Sim: A Virtual Network Embedding Simulator

VNE-Sim is a discrete event simulator written in C++. It is based on the 2011 C++ standard (C++11) that introduces smart pointers and lambda functions. These new facilities are extensively employed in the VNE-Sim. The hierarchy of the VNE-Sim directories is shown in Fig. 5.1.

The CMake build system [3] is employed for compiling the simulator. The *CMakeLists.txt* file that resides in the root directory includes the build instructions such as selecting the compiler, setting the correct build paths, and searching for the required external libraries.

VNE-Sim relies on several external libraries. While some required libraries are downloaded and installed automatically by CMake during the build process, some required external libraries are expected to be installed on the system prior to initiating the build process.

The libraries that are automatically installed by CMake are:

- Fast Network Simulation Setup (FNSS) [4] used for generating data center network topologies;
- Hiberlite library [5] used for writing the simulation results to disk;
- The Adevs library [110] employed for modeling the virtual network embedding process as a discrete event system.

The libraries that must exist on the system prior to compilation are:

- Boost File System, Log, Thread, and Unit Test Framework libraries [2] are mainly used in the core classes for file handling, logging, testing, and debugging. The test cases and experiments are written using the Unit Test Framework.
- GNU Scientific Library [15] is used for generating random numbers. The required random numbers and distributions are generated using this library.

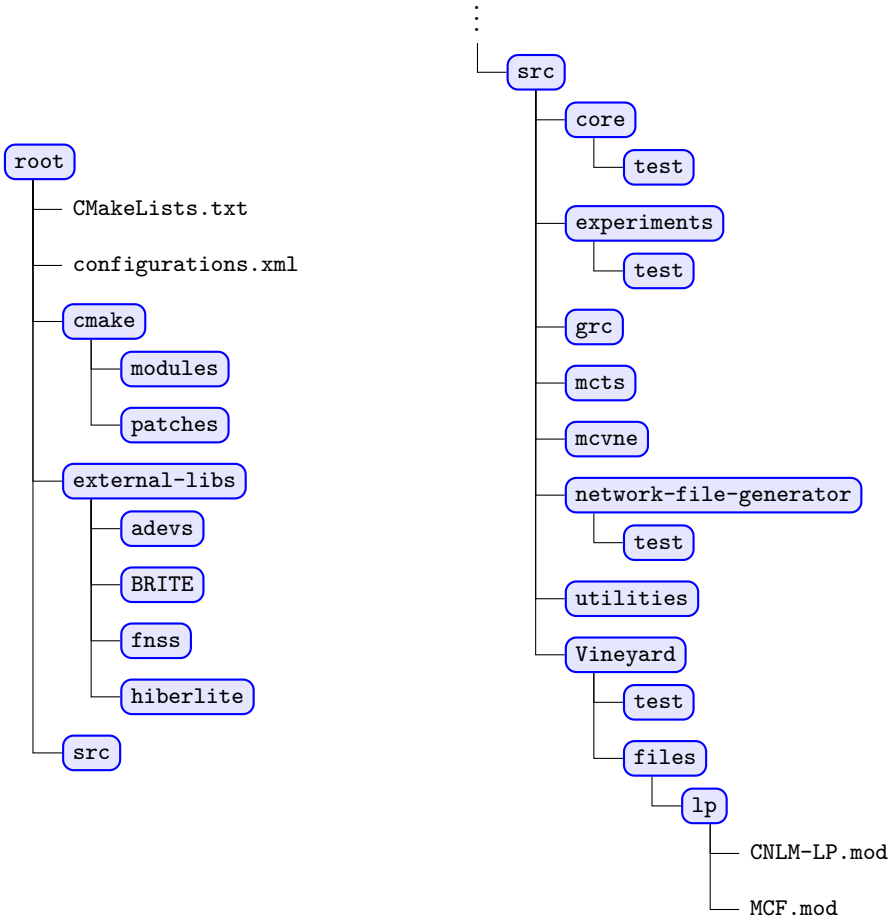

Figure 5.1: Hierarchy of the VNE-Sim directory.

- GNU Linear Programming Kit (GLPK) [14] is used for solving the MCF problem. It is also employed by the Vine algorithms.
- The Message Passing Interface (MPI) standard is used for MCTS parallelization. Therefore, either the OpenMPI [8] or MPICH [7] libraries should be installed the system for compilation of the MCTS in a parallel mode.
- SQLite3 library [10] is used for handling simulation results. The simulation results are exported automatically as SQLite3 databases.

CMake uses scripts to search for the libraries. Certain scripts are not included in the default installation of CMake. They reside in the *cmake/modules* directory.

VNE-Sim uses a modified versions of the Hiberlite and FNSS libraries. Therefore, CMake applies the required modifications after downloading the libraries. The patches that CMake uses for these two libraries reside in *cmake/patches*. VNE-Sim also contains a modified version of the Boston University Representative Topology Generator (BRITE) [13]

that is compiled as a part of the simulator. External libraries that are installed by CMake and the BRITE library reside in the *external-libs* directory.

The *src* directory contains the simulator source code. Various components of VNE-Sim and their dependencies are shown in Fig. 5.2.

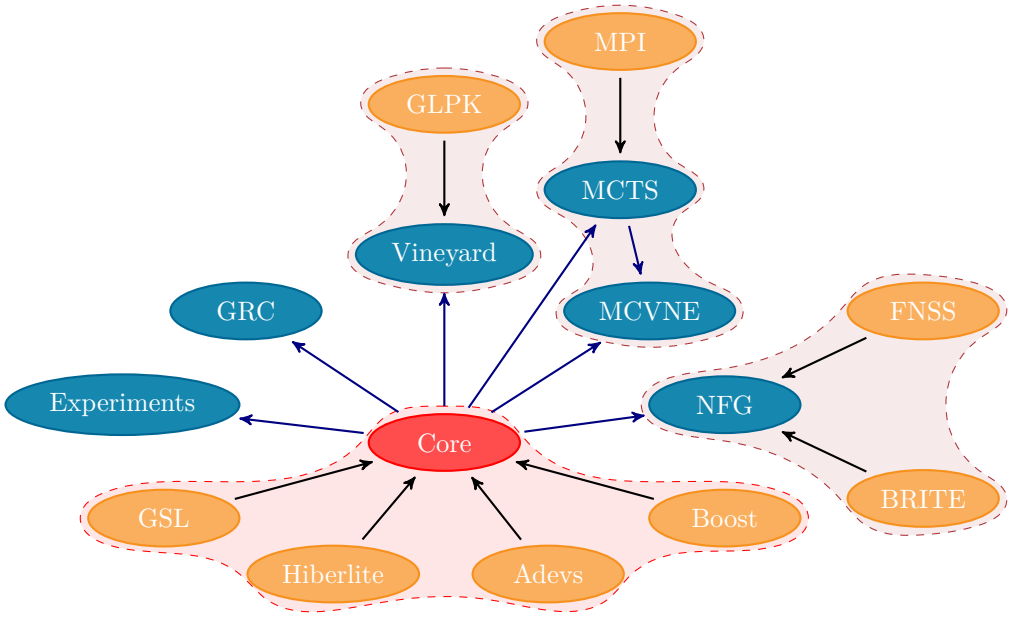

Figure 5.2: The dependencies of VNE-Sim components.

The components are located in various subdirectories of *src*:

- *src/core* contains the classes and interfaces required for implementing various virtual network embedding algorithms. It also contains the discrete event simulation system that models the process of embedding virtual networks as a discrete event system.
- *src/experiments* contains various simulation scenarios presented in Chapter 4.
- *src/grc* contains the implementation of the Global Resource Capacity (GRC) algorithm [69].
- *src/mcts* contains a modular implementation of the Monte Carlo Tree Search (MCTS) algorithm [53], [89]. We have adopted a similar structure to an available MCTS simulator [126].
- *src/mcvne* contains the implementation of the MaVEn algorithms.
- *src/network-file-generator* contains the network file generator package. This package may be used for generating various network topologies and virtual network requests. This package employs the BRITE [13] and FNSS [4] libraries for generating network topologies. It also uses the GNU Scientific Library [15] for generating distributions of VNR arrival times, life-times, and resource requirements.

- *src/utilities* contains the logging system and the Unit Test Framework initializer.
- *src/Vineyard* contains the implementation of R-Vine and D-Vine algorithms [51].

In Section 5.1, we describe the implementation of the core classes of the simulator and their usage.

5.1 The Simulator Core: *src/core*

The content of the core directory is shown in Fig. 5.3.

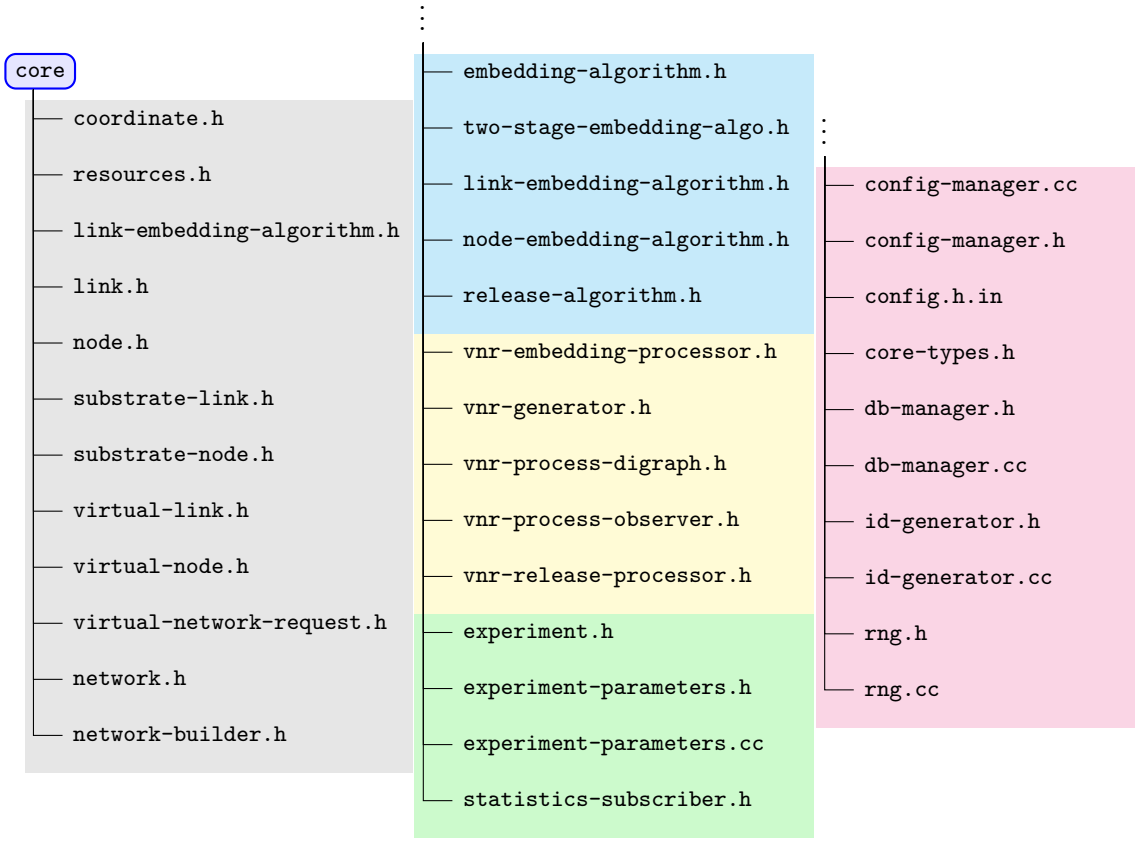

Figure 5.3: Content of the core directory.

The facilities required to simulate VNE algorithms are implemented as abstract template classes in the *core* directory. They may be divided into five categories:

1. *Network Component Classes* provide a framework for defining substrate and virtual nodes and links, networks, and virtual network requests. The files that are in this groups are listed in the gray box shown in Fig. 5.3.
2. *Virtual Network Embedding Classes* define an interface for embedding algorithms. They are listed in the cyan box shown in Fig. 5.3.

3. *Discrete Event Simulation Classes* model the virtual network embedding process as a discrete event system. All these classes are derived from Adevs library [110]. The files corresponding to these classes are listed in the yellow box shown in Fig. 5.3.

4. *Experiment and Result Collection Classes* connect various components to define simulation scenarios that the user requires. They create SQL database tables for simulation parameters and presenting simulation results. The corresponding files are listed in the green box shown in Fig. 5.3.

5. *Operation Related Classes* provide the basic required functionalities such as managing the configuration file, database access, type definitions, and random number generation. These classes are listed in the red box shown in Fig. 5.3.

5.1.1 Network Component Classes

Substrate and Virtual Nodes and Links

Substrate and virtual nodes and links are derived from the node (*core/node.h*) and link (*core/link.h*) base template classes, respectively. They are implemented as C++ *variadic* templates where the resources of the substrate and virtual elements are defined by the template arguments. For example, the `SubstrateNode<typename...>` template class is listed in Listing 5.1

Listing 5.1: Implementation of the SubstrateNode base class.

```
1 template<typename... NODERES>
2 class SubstrateNode: public Node<NODERES...>
3 {
4         ...
5 };
```

A substrate node that possesses CPU capacity of type `double` and memory of type `int` is defined by a `<double,int>` specialization of the `SubstrateNode` template class. The `VYSubstrateNode<>` template class (*src/Vineyard/vy-substrate-node.h*) is an example of a class that is derived from a `<double>` specialization of the `SubstrateNode` template class. The `SubstrateLink<typename...>`, *VirtualNode<typename...>*, and `VirtualLink<typename...>` template classes are similar to `SubstrateNode<typename...>`.

Networks and Virtual Network Requests

In VNE-Sim, a network is defined by the `Network<typename,typename>` template class (*core/network.h*). Its definition is shown in Listing 5.2:

Listing 5.2: Definition of the Network base class.

```
template<typename... NodeT, template<typename...> class NodeC,
    typename... LinkT, template <typename...> class LinkC>
class Network<NodeC<NodeT...>, LinkC<LinkT...>>
{
        ...
};
```

The network template class requires two arguments: a Node class and a Link class. This enables implementing substrate and virtual networks using the same template class. For example, in the Vineyard package (*src/Vineyard/*), a substrate network is defined by specializing the `Network` class with template arguments: `VYSubstrateNode<>` and `VYSubstrateLink<>`. Similarly, a virtual network is defined by using `VYVirtualNode<>` and `VYVirtualLink<>` template arguments. Declaration of pointers to substrate and virtual networks is shown in Listing 5.3.

Listing 5.3: Declaration of pointers to substrate and virtual networks.

```
std::shared_ptr<Network<VYSubstrateNode<>, VYSubstrateLink<>>> substrate_net;
std::shared_ptr<Network<VYVirtualNode<>, VYVirtualLink<>>> virtual_net;
```

VNRs are derived from the `VirtualNetworkRequest<typename>` template class (*core/virtual-network-request.h*). This class maintains a pointer to a virtual network. Its definition is shown in Listing 5.4. This class requires a template argument that defines the type of the virtual networks that it entails.

Listing 5.4: Definition of the VirtualNetworkRequest base class.

```
template<typename... NODERES, template <typename...> class NODECLASS,
            typename... LINKRES, template <typename...> class LINKCLASS>
class VirtualNetworkRequest<Network<NODECLASS<NODERES...>, LINKCLASS<LINKRES...>>>
{
        ...
};
```

5.1.2 Virtual Network Embedding Classes

These classes define an interface for implementing VNE algorithms. The backbone of these classes is the `EmbeddingAlgorithm<typename, typename>` abstract template class (*src/embedding-algorithm.h*). The first template argument is a specialization of the `Network<typename, typename>` while the second argument is a specialization of the VirtualNetworkRequest template class. The classes that are derived from `EmbeddingAlgorithm` should implement the `embedVNR` virtual function. The definition of this template is shown in Listing 5.5.

Listing 5.5: Definition of the EmbeddingAlgorithm template class.

```
template<typename,typename> class EmbeddingAlgorithm;

template <typename... SNODERES, template <typename...> class SNODECLASS,
          typename... SLINKRES, template <typename...> class SLINKCLASS,
          typename... VNODERES, template <typename...> class VNODECLASS,
          typename... VLINKRES, template <typename...> class VLINKCLASS,
          template <typename> class VNRCLASS>
class EmbeddingAlgorithm<
    Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES...>>,
    VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>> {
public:
    typedef VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>
        VNR_TYPE;
    typedef Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES...>>
        SUBSTRATE_TYPE;

    virtual Embedding_Result embedVNR (std::shared_ptr<VNR_TYPE> vnr) = 0;

protected:
    EmbeddingAlgorithm (NetworkBuilder<SUBSTRATE_TYPE>& _sb) : substrate_network(
        _sb.getNetwork()) {};
    EmbeddingAlgorithm (std::shared_ptr<SUBSTRATE_TYPE> _sn) : substrate_network(
        _sn) {};
    std::shared_ptr<SUBSTRATE_TYPE> substrate_network;
};
```

Two stage VNE algorithms employ a node mapping algorithm to map the virtual nodes onto substrate nodes. They also employ a link mapping algorithm for mapping virtual links onto substrate paths. Two stage embedding algorithms may be implemented in VNE-Sim using the `TwoStageEmbeddingAlgo<typename,typename>` template class (*core/two-stage-embedidng-algo.h*). Since `TwoStageEmbeddingAlgo` is derived from `EmbeddingAlgorithm<typename, typename>`, its first and second template arguments identify the types of substrate network and virtual network requests that it recognizes for operation, respectively. `NodeEmbeddingAlgorithm<typename,typename>` (*core/node-embedidng-algorithm.h*) and `LinkEmbeddingAlgorithm<typename, typename>` (*core/link-embedidng-algorithm.h*) are interfaces for virtual node and link mapping algorithms, respectively. The constructor of specializations of the `TwoStageEmbeddingAlgo<typename,typename>` requires pointers to a `NodeEmbeddingAlgorithm<typename,typename>` and a `LinkEmbeddingAlgorithm<typename,typename>`. The template arguments of the node and link embedding algorithms are similar to those of the two stage embedding algorithm. The implementations of TwoStageEmbeddingAlgo, NodeEmbeddingAlgorithm, and LinkEmbeddingAlgo are presented in Appendix B Listings B.1, B.2, and B.3, respectively.

5.1.3 Discrete Event Simulation Classes

The VNR embedding process may be modeled using three discrete event processes. The VNRs are first *generated* based on an arrival process. The arrived VNRs are then passed

to an embedding algorithm. During the *embedding processes*, the VNRs are subjected to queuing and processing delays. Based on the outcome of the embedding process, a VNR may either be accepted or rejected for embedding. Finally, when the life-time of an embedded VNR expires, the *release process* begins, where an algorithm release the resources occupied by the VNR.

The template class `VNRGenerator<typename>` is used to generate virtual network requests. It implements VNR generation process. The required template argument is a specialization of the `VirtualNetworkRequest` template class. `VYVNRGenerator<>` is a specialization of the `VNRGenerator` template class. It may be found in the *Vineyard* package. The embedding and release processes are implemented using the `VNREmbeddingProcessor<typename,typename>` and `VNRReleaseProcessor<typename,typename>` template classes, respectively. Their first template argument is a specialization of the `Network` class that represents a substrate network while the second template argument is a `VirtualNetworkRequest` specialization.

The `VNRProcessObserver` implements the Observer design pattern [65]. It is mainly used for logging, collecting statistics, and recording the embedding outcomes. The events that occur in the three VNR embedding phases are transparent to the Observer. The `VNRProcessDigraph` connects the three main processes and the observer. It requires four template arguments. They are specializations of a generator, embedding, release, and observer template classes.

5.1.4 Experiment and Result Collection Classes

A simulation scenario in VNE-Sim is defined using the `Experiment<typename>` template class. The required template argument is a specialization of `VNRProcessDigraph` template class. The Experiment class also maintains statistics that are collected during a simulation and writes them to disk when the simulation is completed. Various examples of specializations of the Experiment template class may be found in *src/experiment* directory.

5.1.5 Operation Related Classes

The `ConfigManager` class reads the configuration file (*root/configurations.xml*) and maintains a structured *Boost Property Tree* of the configurations. This class is a singleton [65]. Therefore, a pointer to its instance must be acquired first. After obtaining the pointer, a property value may be acquired by calling the `ConfigManager::getConfig` function. For example, acquiring the *core.dbPath* property from configuration file is demonstrated in Listing 5.6.

Listing 5.6: Example of ConfigManger usage.

```
1 std::string dbPath = ConfigManager::Instance()->
2                          getConfig<std::string>("core.dbPath");
```

The `DBManager` class creates instances of `hiberlite::Database` class [5] and maintains pointers to the created databases in a *map* data structure. This class is also implemented as

singleton and a pointer to its instance should be acquired first using the `Instance()` function. Its interface contains two main functions: `createDB` and `getDB`. The `createDB` function requires a name as an `std::string` argument. It creates a database using the given name and returns a pointer to the created database. Pointers to the created databases may also be acquired using the `getDB` function, which requires the name of a database as an `std::string` argument.

All network component classes require identification numbers for operation. Classes that have similar types should not possess identical identification numbers. The `IdGenerator` class produces such type-based unique numbers. It is also singleton and acquisition of a pointer to its instance is similar to the previously described singleton classes. The generated identification numbers are naturally ordered. The `IdGenerator` interface consists of two template functions: the `getId<typename>` function that generates a unique ID for a class type that is given as its template argument, and the `peekId<typename>` function that returns the next ID that will be generated for the type given as its template argument. For example, the constructor of `VYVirtualNode<>`, where a unique ID for this class is acquired is shown in Listing 5.7.

Listing 5.7: Example of ConfigManger usage.

```
1 typedef VYVirtualNode<> this_t;
2
3 template<>
4 VYVirtualNode<>::VYVirtualNode (double cpu, int _x, int _y) :
5 VirtualNode<double>(cpu,true),
6 coordinate(VYCoordinate(_x, _y))
7 {
8      this->id  = vne::IdGenerator::Instance()->getId<this_t>(this);
9 }
```

The `RNG` class employs the GNU Scientific Library [15]. It may be used to generate random numbers and various probability distributions. The seed and type of the random number generator are defined by *core.rngSeed* and *core.rngType* fields in the configuration file, respectively. Classes using `RNG` may either use a general purpose random number generator by calling the `getGeneralRNG` function or they may subscribe to `RNG` to get their own specific random number generator. Only the derivatives of `RNGSubscriber` may subscribe to `RNG` and possess their specific random number generator.

Chapter 6

Conclusion

In this Dissertation, we investigated applications of reinforcement learning algorithms to two research areas in computer networks: deflection routing in Optical Burst-Switched (OBS) networks and solving the Virtual Network Embedding (VNE) problem. The deflection routing problem is time-sensitive and decisions should be identified in real-time while a VNE algorithm is permitted to have additional processing time for finding a profitable embedding. This difference has a major impact on the employed learning algorithms and the design choices.

The first application we considered was deflection routing in buffer-less networks. We proposed a Predictive Q-learning-based Deflection Routing (PQDR) algorithm to improve performance of the existing Reinforcement Learning-Based Deflection Routing Scheme (RLDRS), which employs Q-learning and whose complexity depends on the size of the network. Even though employing Predictive Q-learning instead of Q-learning improved the burst loss probability in high traffic loads, it had no large impact on the networks with low to moderate loads. We attribute this behaviour to the nature of the Q-learning algorithm, which inefficiently processes the feedback (reinforcement) signals. Even though more efficient reinforcement learning algorithms have been proposed, higher efficiency comes at the cost of higher processing time. Therefore, the dependence of PQDR and RLDRS complexity on the size of the network (number of nodes and links) quickly becomes a bottleneck in larger network topologies. In order to be able to employ more efficient reinforcement learning algorithms, we proposed the Node Degree Dependent (NDD) signalling algorithm whose complexity depends on a node degree rather than the network size. We then proposed a three-layer feed-forward neural network for generating deflection decisions and introduced two deflection routing algorithms named NN-NDD and ENN-NDD that employ the designed neural network. The NN-NDD and ENN-NDD algorithms discard fewer bursts than the PQDR algorithm and RLDRS in low to moderate traffic loads because the underlying neural network is able to process the reinforcement signals more efficiently. Performance evalua-

tions showed promising performance using a single hidden-layer neural network. Therefore, we did not consider networks with multiple hidden layer. Experimenting with larger neural networks may be considered for future work. We also did not consider burst aggregation algorithms in simulation scenarios. Future work may include investigating various aggregation algorithms and their effect on performance of deflection routing algorithms.

The second application we considered was modeling VNE as a reinforcement learning process using the Markov Decision Process (MDP) framework. We also showed that the complexity of finding an optimal policy for the proposed MDP is exponential in the size of the substrate network and in the size of requested virtual networks. Hence, we introduced two MaVEn algorithms that employ Monte Carlo Tree Search (MCTS) for finding near-optimal solutions. We improved the performance of the MaVEn algorithms by employing MCTS root parallelization. Performance of the proposed algorithms was evaluated using a synthesized network topology that resembled an Internet Service Provider network topology. We also considered BCube and Fat-Tree data center network topologies. Simulation results indicated that the MaVEn algorithms have superior performance compared to the Vine and Global Resource Capacity (GRC) VNE algorithms. Future work may include improving the performance of MaVEn-M algorithm by parallelizing the Multicommodity Flow (MCF) algorithm and eliminating the disk I/O operations. Furthermore, executing the parallelized MaVEn algorithms on large clusters that are highly optimized for parallel computing will better reveal the dependence of the their performance on the number of processors. We simulated data center topologies that are much smaller than the currently deployed data center networks because of the memory and execution time limitations. Using path selection algorithms that are specifically designed for data center network topologies will improve the performance of the MaVEn algorithms. This will, in turn, enable simulating larger and more realistic data center network topologies. The size and the structure of the requested virtual network topologies also affect scalability and performance of the algorithms. Evaluation of these effects are also of interest and may be topic of future research.

We also developed *VNE-Sim* for simulating VNE algorithms. *VNE-Sim* formalizes the VNE process as a discrete event system based on the DEVS framework and uses the *Adevs* library. The modular and scalable design of *VNE-Sim* enables researchers to seamlessly implement new VNE algorithms and evaluate their performance. The initial development phase of *VNE-Sim* has been completed. The future work includes implementing other existing VNE algorithms, implementing a source code documentation system such as Doxygen, in depth comparison of *VNE-Sim* and *Alevin* in terms of memory usage and capabilities, and implementing a scripting infrastructure for visualization of results.

While reinforcement learning algorithms have found applications in various fields, they are still not widely employed for computer networking applications. The results and algorithms presented in this Dissertation have demonstrated that learning algorithms provide viable solutions and may be implemented in computer networks.

Bibliography

[1] (2015, Sept.) Amazon Web Services [Online]. Available: https://aws.amazon.com/.

[2] (2015, Dec.) Boost C++ Libraries [Online]. Available: http://www.boost.org/.

[3] (2015, Dec.) Cmake Build System [Online]. Available: https://cmake.org/.

[4] (2015, Dec.) Fast Network Simulation Setup [Online]. Available: http://fnss.github.io/.

[5] (2015, Dec.) Hiberlite Library [Online]. Available: https://github.com/paulftw/hiberlite/.

[6] (2015, Dec.) Message Passing Interface Forum [Online]. Available: http://www.mpi-forum.org/.

[7] (2015, Dec.) MPICH: High-Performance Portable MPI [Online]. Available: https://www.mpich.org/.

[8] (2015, Dec.) Open MPI: Open Source High Performance Computing [Online]. Available: http://www.open-mpi.org/.

[9] (2015, July) ProtoGENI [Online]. Available: http://www.protogeni.net/.

[10] (2015, Dec.) SQLite: Small. Fast. Reliable. Choose any three [Online]. Available: https://www.sqlite.org/.

[11] (2016, Jan.) A special report: a brief history of NSF and the Internet [Online]. Available: http://www.nsf.gov/news/special_reports/cyber/internet.jsp/.

[12] (2016, Jan.) Autonomous system numbers [Online]. Available: http://www.iana.org/assignments/as-numbers/.

[13] (2016, Jan.) Boston University Representative Internet Topology Generator. [Online] Available: http://www.cs.bu.edu/brite/.

[14] (2016, Jan.) GLPK–GNU Linear Programming Kit [Online]. Available: http://www.gnu.org/software/glpk/.

[15] (2016, Jan.) GSL–GNU Scientific Library [Online]. Available: https://www.gnu.org/software/gsl/.

[16] (2016, Jan.) iDef ns-3 implementation repository [Online]. Available: http://bitbucket.org/shaeri/hmm-deflection/.

[17] (2016, Jan.) The ns-3 network simulator [Online]. Available: http://www.nsnam.org/.

[18] A. S. Acampora and S. I. A. Shah, "Multihop lightwave networks: a comparison of store-and-forward and hot-potato routing," in *Proc. IEEE INFOCOM*, vol. 1, Bal Harbour, FL, USA, Apr. 1991, pp. 10–19.

[19] R. G. Addie, T. D. Neame, and M. Zukerman, "Performance evaluation of a queue fed by a Poisson Pareto burst process," *Comput. Netw.*, vol. 40, no. 3, pp. 377–397, Oct. 2002.

[20] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, Oct. 2008.

[21] N. Al-Rousan, S. Haeri, and Lj. Trajković, "Feature selection for classification of BGP anomalies using bayesian models," in *Proc. ICMLC 2012*, Xi'an, China, July 2012, pp. 140–147.

[22] N. Al-Rousan and Lj. Trajković, "Machine learning models for classification of BGP anomalies," in *Proc. IEEE Conf. High Performance Switching and Routing*, Belgrade, Serbia, June 2012, pp. 103–108.

[23] E. Alpaydin, *Introduction to Machine Learning*, 2nd ed., ser. Adaptive Computation and Machine Learning series. Cambridge, MA, USA: MIT Press, 2010.

[24] D. G. Andersen, "Theoretical approaches to node assignment," Dec. 2002, Unpublished Manuscript [Online]. Available: http://repository.cmu.edu/compsci/86/.

[25] T. Anderson, L. Peterson, S. Shenker, and J. S. Turner, "Overcoming the Internet impasse through virtualization," *Computer*, vol. 38, no. 4, pp. 34–41, Apr. 2005.

[26] H. Baier and M. H. M. Winands, "Nested Monte-Carlo Tree Search for online planning in large MDPs," in *Proc. 20th European Conf. Artificial Intelligence*, Montpellier, France, Aug. 2012, pp. 109–114.

[27] I. Baldine, G. N. Rouskas, H. G. Perros, and D. Stevenson, "Jumpstart: a just-in-time signaling architecture for wdm burst-switched networks," *IEEE Commun. Mag.*, vol. 40, no. 2, pp. 82–89, Feb. 2002.

[28] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 242–253, Oct. 2011.

[29] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, Oct. 1999.

[30] P. Baran, "On distributed communications networks," *IEEE Trans. Commun. Syst.*, vol. CS–12, no. 1, pp. 1–9, Mar. 1964.

[31] A. G. Barto and T. G. Dietterich, "Reinforcement learning and its relationship to supervised learning," in *Handbook of Learning and Approximate Dynamic Programming*, J. Si, A. G. Barto, W. Powell, and D. Wunsch, Eds. Piscataway, NJ, USA: Wiley-IEEE Press, 2004, pp. 45–63.

[32] B. G. Bathula and V. M. Vokkarane, "QoS-based manycasting over optical burst-switched (OBS) networks," *IEEE/ACM Trans. Netw.*, vol. 18, no. 1, pp. 271–283, Feb. 2010.

[33] A. Belbekkouche, A. Hafid, and M. Gendreau, "Novel reinforcement learning-based approaches to reduce loss probability in buffer-less OBS networks," *Comput. Netw.*, vol. 53, no. 12, pp. 2091–2105, Aug. 2009.

[34] A. Belbekkouche, A. Hafid, M. Tagmouti, and M. Gendreau, "Topology-aware wavelength partitioning for DWDM OBS networks: a novel approach for absolute QoS provisioning," *Computer Networks*, vol. 54, no. 18, pp. 3264–3279, Dec. 2010.

[35] A. Belbekkouche, M. M. Hasan, and A. Karmouch, "Resource discovery and allocation in network virtualization," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 4, pp. 1114–1128, 2012.

[36] R. Bellman, *Dynamic Programming.* Princeton, NJ, USA: Princeton University Press, 1957.

[37] D. P. Bertsekas, *Dynamic Programming: Deterministic and Stochastic Models.* Englewood Cliffs, NJ, USA: Prentice-Hall, 1987.

[38] F. Borgonovo, *Routing in Communications Networks.* NJ, USA: Prentice-Hall, 1995.

[39] F. Borgonovo, L. Fratta, and J. Bannister, "Unslotted deflection routing in all-optical networks," in *Proc. IEEE GLOBECOM*, Houston, TX, USA, Dec. 1993, pp. 119–125.

[40] J. F. Botero, X. Hesselbach, M. Duelli, D. Schlosser, A. Fischer, and H. de Meer, "Energy efficient virtual network embedding," *IEEE Commun. Lett.*, vol. 16, no. 5, pp. 756–759, May 2012.

[41] J. A. Boyan and M. L. Littman, "Packet routing in dynamically changing networks: a reinforcement learning approach," in *Advances in Neural Inform. Process. Syst.*, J. Jack, D. Cowan, G. Tesauro, and J. Alspector, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers, 1994, vol. 6, pp. 671–678.

[42] S. Bregni, A. Caruso, and A. Pattavina, "Buffering-deflection tradeoffs in optical burst switching," *Photon. Netw. Commun.*, vol. 20, no. 2, pp. 193–200, Aug. 2010.

[43] K. L. Calvert, M. B. Dora, and E. W. Zegura, "Modeling Internet topology," *IEEE Commun. Mag.*, vol. 35, no. 6, pp. 160–163, June 1997.

[44] T. Cazenave and N. Jouandeau, "On the parallelization of UCT," in *Proc. Computer Games Workshop 2007 (CGW 2007)*, H. J. van den Herik, J. W. Uiterwijk, and M. H. Winands, Eds. Universiteit Maastricht, 2007, vol. 5131, pp. 93–101.

[45] G. Chaslot, M. Winands, and H. J. Herik, "Parallel Monte-Carlo tree search," in *Lecture Notes in Computer Science: Computers and Games*, H. J. Herik, X. Xu, Z. Ma, and M. Winands, Eds. Springer, 2008, vol. 5131, pp. 60–71.

[46] Q. Chen, H. Chang, R. Govindan, S. Jamin, S. Shenker, and W. Willinger, "The origin of power laws in Internet topologies revisited," in *Proc. INFOCOM*, New York, NY, USA, Apr. 2002, pp. 608–617.

[47] Y. Chen, C. Qiao, and X. Yu, "Optical burst switching: a new area in optical networking research," *IEEE Netw.*, vol. 18, no. 3, pp. 16–23, June 2004.

[48] X. Cheng, S. Su, Z. Zhang, H. Wang, F. Yang, Y. Luo, and J. Wang, "Virtual network embedding through topology-aware node ranking," *Comput. Commun. Rev.*, vol. 41, pp. 38–47, Apr. 2011.

[49] T. Chich, J. Cohen, and P. Fraigniaud, "Unslotted deflection routing: a practical and efficient protocol for multihop optical networks," *IEEE/ACM Trans. Netw.*, vol. 9, no. 1, pp. 47–59, Feb. 2001.

[50] S. P. M. Choi and D. Y. Yeung, "Predictive q-routing: a memory-based reinforcement learning approach to adaptive traffic control," in *Advances in Neural Inform. Process. Syst.*, D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, Eds. Cambridge, MA, USA: The MIT Press, 1996, vol. 8, pp. 945–951.

[51] M. Chowdhury, M. R. Rahman, and R. Boutaba, "ViNEYard: Virtual network embedding algorithms with coordinated node and link mapping," *IEEE/ACM Trans. Netw.*, vol. 20, no. 1, pp. 206–219, Feb. 2012.

[52] N. M. M. K. Chowdhury and R. Boutaba, "Network virtualization: state of the art and research challenges," *IEEE Commun. Mag.*, vol. 47, no. 7, pp. 20–26, July 2009.

[53] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo Tree Search," in *Proc. 5th Int. Conf. Comput. and Games (CG'06)*, Turin, Italy, May 2006, pp. 72–83.

[54] R. Coulom, "Computing Elo ratings of move patterns in the game of Go," *Int. Comput. Games Association Journal*, vol. 30, no. 4, pp. 198–208, Dec. 2007.

[55] M. E. Crovella and A. Bestavros, "Self-Similarity in World Wide Web traffic: Evidence and possible causes," *IEEE/ACM Trans. Netw.*, vol. 5, no. 6, pp. 835–846, Dec. 1997.

[56] F. D'Epenoux, "A probabilistic production and inventory problem," *Management Science*, vol. 10, pp. 98–108, 1963.

[57] A. I. A. El-Rahman, S. I. Rabia, and H. M. H. Shalaby, "Mac-layer performance enhancement using control packet buffering in optical burst-switched networks," *J. Lightwave Technol.*, vol. 30, no. 11, pp. 1578–1586, June 2012.

[58] P. Erdös and A. Reńyi, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci.*, vol. 5, pp. 17–61, 1960.

[59] I. Fajjari, N. Aitsaadi, G. Pujolle, and H. Zimmermann, "VNE-AC: Virtual network embedding algorithm based on ant colony metaheuristic," in *Proc. IEEE ICC 2011*, Kyoto, Japan, June 2011, pp. 1–6.

[60] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the Internet topology," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 251–262, Aug. 1999.

[61] P. Faratin, D. Clark, S. Bauer, W. Lehr, P. Gilmore, and A. Berger, "The growing complexity of Internet interconnection," *Communications and Strategies*, no. 72, pp. 51–71, Dec. 2008.

[62] N. Feamster, L. Gao, and J. Rexford, "How to lease the Internet in your spare time," *Comput. Commun. Rev.*, vol. 37, no. 1, pp. 61–64, Jan. 2007.

[63] A. Fischer, J. F. Botero, M. T. Beck, H. de Meer, and X. Hesselbach, "Virtual network embedding: a survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 1888–1906, 2013.

[64] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and C. Diot, "Packet-level traffic measurements from the Sprint IP backbone," *IEEE Netw.*, vol. 17, no. 6, pp. 6–16, Dec. 2003.

[65] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed., ser. Addison-Wesley Professional Computing Series. Boston, MA, USA: Addison-Wesley, 1994.

[66] X. Gao and M. Bassiouni, "Improving fairness with novel adaptive routing in optical burst-switched networks," *J. Lightw. Technol.*, vol. 27, no. 20, pp. 4480–4492, Oct. 2009.

[67] S. Gelly and D. Silver, "Achieving master level play in 9 x 9 computer Go," in *Proc. 23rd Conf. Artificial Intelligence*, Chicago, IL, USA, July 2008, pp. 1537–1540.

[68] S. Gelly and D. Silver, "Monte-Carlo tree search and rapid action value estimation in computer Go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, July 2011.

[69] L. Gong, Y. Wen, Z. Zhu, and T. Lee, "Toward profit-seeking virtual network embedding algorithm via global resource capacity," in *Proc. IEEE INFOCOM*, Toronto, ON, Canada, Apr. 2014, pp. 1–9.

[70] A. Greenberg and B. Hajek, "Deflection routing in hypercube networks," *IEEE Trans. Commun.*, vol. 40, no. 6, pp. 1070–1081, June 1992.

[71] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: A high performance, server-centric network architecture for modular data centers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 63–74, Oct. 2009.

[72] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "SecondNet: a data center network virtualization architecture with bandwidth guarantees," in *Proc. ACM CoNEXT 2010*, Philadelphia, PA, USA, Dec. 2010, p. 15.

[73] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: a scalable and fault-tolerant network structure for data centers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 75–86, Oct. 2008.

[74] S. Haeri, M. Arianezhad, and Lj. Trajković, "A predictive q-learning-based algorithm for deflection routing in buffer-less networks," in *Proc. IEEE Int. Conf. Syst., Man, and Cybern.*, Manchester, UK, Oct. 2013, pp. 764–769.

[75] S. Haeri, W. W.-K. Thong, G. Chen, and Lj. Trajković, "A reinforcement learning-based algorithm for deflection routing in optical burst-switched networks," in *Proc. The 14th IEEE Int. Conf. Inform. Reuse and Integration (IRI 2013)*, San Francisco, USA, Aug. 2013, pp. 474–481.

[76] S. Haeri and Lj. Trajković, "Deflection routing in complex networks," in *Proc. IEEE Int. Symp. Circuits and Systems*, Melbourne, Australia, June 2014, pp. 2217–2220.

[77] S. Haeri and Lj. Trajković, "Deflection routing in complex networks," in *Complex Systems and Networks*, J. Lu, X. Yu, G. Chen, and W. Yu, Eds. Berlin: Springer, 2015, pp. 395–422.

[78] S. Haeri and Lj. Trajković, "Intelligent deflection routing in buffer-less networks," *IEEE Tran. Cybern.*, vol. 45, no. 2, pp. 316–327, Feb. 2015.

[79] I. Houidi, W. Louati, W. Ben Ameur, and D. Zeghlache, "Virtual network provisioning across multiple substrate networks," *Computer Networks*, vol. 55, no. 4, pp. 1011–1023, Mar. 2011.

[80] R. A. Howard, *Dynamic Programming and Markov Processes.* Cambridge, Massachusetts, USA: MIT Press, 1960.

[81] G.-B. Huang and H. A. Babri, "Upper bounds on the number of hidden neurons in feedforward networks with arbitrary bounded nonlinear activation functions," *IEEE Trans. Neural Netw.*, vol. 9, no. 1, pp. 224–229, Jan. 1998.

[82] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: a new learning scheme of feedforward neural networks," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, Budapest, Hungary, July 2004, pp. 985–990.

[83] S. Iyer, S. Bhattacharyya, N. Taft, and C. Diot, "An approach to alleviate link overload as observed on an IP backbone," in *Proc. IEEE INFOCOM*, vol. 1, Stanford, CA, USA, Mar. 2003, pp. 406–416.

[84] M. Izal and J. Aracil, "On the influence of self-similarity on optical burst switching traffic," in *Proc. IEEE GLOBECOM*, vol. 3, Taipei, Taiwan, Nov. 2002, pp. 2308–2312.

[85] A. Jayaraj, T. Venkatesh, and C. S. R. Murthy, "Loss classification in optical burst switching networks using machine learning techniques: improving the performance of tcp," *IEEE J. Sel. Areas Commun.*, vol. 26, no. 6, pp. 45–54, Aug. 2008.

[86] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: a survey," *J. Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[87] M. Kearns, Y. Mansour, and A. Y. Ng, "A sparse sampling algorithm for near-optimal planning in large Markov decision processes," *Machine Learning*, vol. 49, no. 2, pp. 193–208, 2002.

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

[89] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Lecture Notes in Computer Science: Proc. 17th European Conference on Machine Learning (ECML)*, J. Furnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Springer, 2006, vol. 4212, pp. 282–293.

[90] R. E. Korf, "Depth-first iterative deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, no. 1, pp. 97–109, Sept. 1985.

[91] T. Legrand, H. Nakajima, P. Gavignet, and B. Cousin, "Labelled OBS test bed for contention resolution study," in *Proc. 5th Int. Conf. Broadband Communications, Networks and Systems*, London, UK, 2008, pp. 82–87.

[92] C. E. Leiserson, "Fat-Trees: universal networks for hardware-efficient supercomputing," *IEEE Trans. Comput.*, vol. 30, no. 10, pp. 892–901, Oct. 1985.

[93] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, "On the self-similar nature of ethernet traffic (extended version)," *IEEE/ACM Trans. Netw.*, vol. 2, no. 1, pp. 1–15, Feb. 1994.

[94] M. Levesque, H. Elbiaze, and W. Aly, "Adaptive threshold-based decision for efficient hybrid deflection and retransmission scheme in obs networks," in *Proc. 13th Int. Conf. Optical Network Design and Modeling*, Braunschweig, Germany, Feb. 2009, pp. 55–60.

[95] S. Li, M. Wang, E. W. M. Wong, V. Abramov, and M. Zukerman, "Bounds of the overflow priority classification for blocking probability approximation in obs networks," *J. Opt. Commun. Netw.*, vol. 5, no. 4, pp. 378–393, Apr. 2013.

[96] J. Lischka and H. Karl, "A virtual network mapping algorithm based on subgraph isomorphism detection," in *Proc. ACM VISA*, Barcelona, Spain, Aug. 2009, pp. 81–88.

[97] M. L. Littman, T. L. Dean, and L. P. Kaelbling, "On the complexity of solving Markov decision problems," in *Proc. Eleventh Conf. Uncertainty in Artificial Intell.*, Montreal, QU, Canada, Aug. 1995, pp. 394–402.

[98] H.-L. Liu, B. Zhang, and S.-L. Shi, "A novel contention resolution scheme of hybrid shared wavelength conversion for optical packet switching," *J. Lightwave Technol.*, vol. 30, no. 2, pp. 222–228, Jan. 2012.

[99] J. Lü, G. Chen, M. Ogorzalek, and Lj. Trajković, "Theories and applications of complex networks: advances and challenges," in *Proc. IEEE Int. Symp. Circuits and Syst.*, Beijing, China, May 2013, pp. 2291–2294.

[100] M. C. Luizelli, L. R. Bays, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary, "Characterizing the impact of network substrate topologies on virtual network embedding," in *Proc. 9th Int. Conf. Netw. and Service Manag. (CNSM 2013)*, Zurich, Switzerland, Oct. 2013, pp. 42–50.

[101] J. G. March, "Exploration and exploitation in organizational learning," *Organization Science*, vol. 2, no. 1, pp. 71–87, Feb. 1991.

[102] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading technology architecture and microarchitecture," *Intel Technol. J.*, vol. 6, no. 1, pp. 4–15, Feb. 2002.

[103] M. Matsumoto and T. Nishimura, "Mersenne Twister: a 623-dimensionally equidistributed uniform pseudorandom number generator," *ACM Trans. Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, Jan. 1998.

[104] N. F. Maxemchuk, "Comparison of deflection and store and forward techniques in the manhattan street and shuffle exchange networks," in *Proc. IEEE INFOCOM*, vol. 3, Ottawa, ON, Canada, Apr. 1989, pp. 800–809.

[105] Merit/NSFNET Information Services, "The technology timetable," *Link Letter*, vol. 7, no. 1, pp. 8–11, July 1994.

[106] M. Mirza, J. Sommers, P. Barford, and X. Zhu, "A machine learning approach to TCP throughput prediction," *IEEE/ACM Trans. Netw.*, vol. 18, no. 4, pp. 1026–1039, Aug. 2010.

[107] X. Mountrouidou and H. Perros, "On the departure process of the burst aggregation algorithms in optical burst switching," *J. of Comput. Netw.*, vol. 53, no. 3, pp. 247–264, Feb. 2009.

[108] M. Najiminaini, L. Subedi, and Lj. Trajković, "Spectral analysis of Internet topology graphs," in *Proc. IEEE Int. Symp. Circuits and Syst.*, Taipei, Taiwan, May 2009, pp. 1697–1700.

[109] A. Nowe, K. Steenhaut, M. Fakir, and K. Verbeeck, "Q-learning for adaptive load based routing," in *Proc. IEEE Int. Conf. Syst., Man, and Cybern.*, vol. 4, San Diego, CA, USA, Oct. 1998, pp. 3965–3970.

[110] J. J. Nutaro, *Building Software for Simulation: Theory and Algorithms, with Applications in C++*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2010.

[111] N. J. N. P. E. Hart and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst., Science, and Cybern.*, vol. 4, no. 2, pp. 100–107, July 1968.

[112] V. Paxson and S. Floyd, "Wide-area traffic: The failure of poisson modeling," *IEEE/ACM Trans. Netw.*, vol. 3, no. 3, pp. 226–244, June 1995.

[113] J. Perelló, F. Agraz, S. Spadaro, J. Comellas, and G. Junyent, "Using updated neighbor state information for efficient contention avoidance in OBS networks," *Comput. Commun.*, vol. 33, no. 1, pp. 65–72, Jan. 2010.

[114] H. G. Perros, *Connection-Oriented Networks: SONET/SDH, ATM, MPLS and Optical Networks*. Chichester, UK: John Wiley & Sons, Inc., 2005.

[115] L. Peshkin and V. Savova, "Reinforcement learning for adaptive routing," in *Proc. Int. Joint Conf. Neural Netw.*, vol. 2, Honolulu, HI, USA, May 2002, pp. 1825–1830.

[116] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, ser. Wiley Series in Probability and Statistics. Hoboken, NJ, USA: John Wiley & Sons, Inc., 1994.

[117] C. Qiao and M. Yoo, "Optical burst switching (OBS)—a new paradigm for an optical Internet," *J. of High Speed Netw.*, vol. 8, no. 1, pp. 69–84, Mar. 1999.

[118] A. Razzaq and M. Rathore, "An approach towards resource efficient virtual network embedding," in *Proc. INTERNET 2010*, Valencia, Spain, Sept. 2010, pp. 68–73.

[119] S. Russell and P. Norvig, *Artificial Intelligence: a Modern Approach*, 2nd ed., ser. Prentice Hall Series in Artificial Intelligence. Upper Saddle River, NJ, USA: Pearson Education, Inc., 2002.

[120] S. Saini, H. Jin, R. Hood, D. Barker, P. Mehrotra, and R. Biswas, "The impact of hyper-threading on processor resource utilization in production applications," in *Proc. 18th Int. Conf. High Performance Computing (HiPC 2011)*, Bengaluru, India, Dec. 2011, pp. 18–21.

[121] L. Saino, C. Cocora, and G. Pavlou, "A toolchain for simplifying network simulation setup," in *Proc. 6th Int. ICST Conf. Simulation Tools and Techniques (SIMUTools 2013)*, Cannes, French Riviera, France, Mar. 2013, pp. 82–91.

[122] M. P. D. Schadd, M. H. M. Winands, M. J. W. Tak, and J. W. H. M. Uiterwijk, "Single-player Monte-Carlo tree search for SameGame," *Knowledge-Based Systems*, vol. 34, pp. 3–11, Oct. 2012.

[123] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, and H. Aldewereld, "Addressing NP-complete puzzles with Monte-Carlo methods," in *Proc. AISB Symposium on Logic and the Simulation of Interaction and Reasoning*, vol. 9, Aberdeen, Scotland, Apr. 2008, pp. 55–61.

[124] A. Schrijver, *Theory of Linear and Integer Programming*, 1st ed. Chichester, UK: John Wiley & Sons, Inc., 1998.

[125] G. Siganos, M. Faloutsos, P. Faloutsos, and C. Faloutsos, "Power-laws and the AS-level Internet topology," *IEEE/ACM Trans. Networking*, vol. 11, no. 4, pp. 514–524, Aug. 2003.

[126] D. Silver and J. Veness, "Monte-Carlo planning in large POMDPs," in *Advances in Neural Inform. Process. Syst. 23: 24th Annual Conference on Neural Information Processing Systems*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, Eds. Curran Associates, Inc., 2010, vol. 3, pp. 2164–2172.

[127] L. Subedi and Lj. Trajković, "Spectral analysis of Internet topology graphs," in *Proc. IEEE Int. Symp. Circuits and Syst.*, Paris, France, June 2010, pp. 1803–1806.

[128] Y. Sun, T. Hashiguchi, V. Q. Minh, X. Wang, H. Morikawa, and T. Aoyama, "Design and implementation of an optical burst-switched network testbed," *IEEE Communions Magazine*, vol. 43, no. 11, pp. 48–55, Nov. 2005.

[129] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1998.

[130] C. Szepesvári, "Reinforcement learning algorithms for MDPs—a survey," Department of Computing Science, University of Alberta, Tech. Rep. TR09-13, 2009.

[131] W. W.-K. Thong and G. Chen, "Jittering performance of random deflection routing in packet networks," *Communications in Nonlinear Science and Numerical Simulation*, vol. 18, no. 3, pp. 616–624, Mar. 2013.

[132] W. W.-K. Thong, G. Chen, and Lj. Trajković, "Red-f routing protocol for complex networks," in *Proc. IEEE Int. Symp. Circuits and Systems*, Seoul, Korea, May 2012, pp. 1644–1647.

[133] Lj. Trajković, "Analysis of Internet topologies," *IEEE Circuits and Syst. Mag.*, vol. 10, no. 3, pp. 48–54, July 2010.

[134] J. S. Turner and D. E. Taylor, "Diversifying the Internet," in *Proc. IEEE GLOBECOM 2005*, vol. 2, St. Louis, MO, USA, Dec. 2005, pp. 755–760.

[135] A. M. Uhrmacher, "Dynamic structures in modeling and simulation: a reflective approach," *ACM Trans. Modeling and Computer Simulation*, vol. 11, no. 2, pp. 206–232, Apr. 2001.

[136] X. Wang, X. Jiang, and A. Pattavina, "Efficient designs of optical lifo buffer with switches and fiber delay lines," *IEEE Trans. Commun.*, vol. 59, no. 12, pp. 3430–3439, Dec. 2011.

[137] X. Wang, H. Morikawa, and T. Aoyama, "Burst optical deflection routing protocol for wavelength routing WDM networks," *Opt. Netw. Mag.*, vol. 3, no. 6, pp. 12–19, Nov. 2002.

[138] C. J. C. H. Watkins and P. Dayan, "Technical note, Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992.

[139] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-world networks," *Nature*, vol. 393, pp. 440–442, June 1998.

[140] B. M. Waxman, "Routing of multipoint connections," *IEEE J. Sel. Areas Commun.*, vol. 6, no. 9, pp. 1617–1622, Dec. 1988.

[141] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3, pp. 229–256, May 1992.

[142] R. J. Williams and L. C. Baird, "Tight performance bounds on greedy policies based on imperfect value functions," College of Computer Science, Northeastern University, Tech. Rep. NU-CCS-93-13, 1993.

[143] R. J. Williams and J. Peng, "Function optimization using connectionist reinforcement learning algorithms," *Connection Science*, vol. 3, no. 3, pp. 241–268, 1991.

[144] E. W. M. Wong, J. Baliga, M. Zukerman, A. Zalesky, and G. Raskutti, "A new method for blocking probability evaluation in obs/ops networks with deflection routing," *J. Lightwave Technol.*, vol. 27, no. 23, pp. 5335–5347, Dec. 2009.

[145] G. Wu, W. Dai, X. Li, and J. Chen, "A maximum-efficiency-first multi-path route selection strategy for optical burst switching networks," *Optik—Int. J. Light and Electron Optics*, vol. 125, no. 10, pp. 2229–2233, May 2014.

[146] Y. Xiong, M. Vandenhoute, and H. C. Cankaya, "Control architecture in optical burst-switched WDM networks," *IEEE J. Sel. Areas Commun.*, vol. 18, no. 10, pp. 1838–1851, Aug. 2000.

[147] X. Yang and D. Wetherall, "Source selectable path diversity via routing deflections," in *Proc. ACM SIGCOMM*, New York, NY, USA, Oct. 2006, pp. 159–170.

[148] Q. Yin and T. Roscoe, "VF2x: Fast, efficient virtual network mapping for real testbed workloads," in *Proc. 8th Int. ICST TridentCom 2012*, Thessaloniki, Greece, June 2012, pp. 271–286.

[149] M. Yoo, C. Qiao, and S. Dixit, "Comparative study of contention resolution policies in optical burst-switched WDM networks," in *Proc. SPIE*, vol. 4123, Boston, MA, USA, Oct. 2000, pp. 124–135.

[150] H. Yu, V. Anand, C. Qiao, H. Di, and X. Wei, "A cost efficient design of virtual infrastructures with joint node and link mapping," *J. Netw. and Syst. Management*, vol. 20, no. 1, pp. 97–115, Sept. 2012.

[151] M. Yu, Y. Yi, J. Rexford, and M. Chiang, "Rethinking virtual network embedding: substrate support for path splitting and migration," *Comput. Commun. Rev.*, vol. 38, no. 2, pp. 19–29, Mar. 2008.

[152] X. Yu, J. Li, X. Cao, Y. Chen, and C. Qiao, "Traffic statistics and performance evaluation in optical burst switched networks," *J. Lightw. Technol.*, vol. 22, no. 12, pp. 2722–2738, Dec. 2004.

[153] A. Zalesky, H. Vu, Z. Rosberg, E. W. M. Wong, and M. Zukerman, "Modelling and performance evaluation of optical burst switched networks with deflection routing and wavelength reservation," in *Proc. IEEE INFOCOM*, vol. 3, Hong Kong SAR, China, Mar. 2004, pp. 1864–1871.

[154] A. Zalesky, H. Vu, Z. Rosberg, E. W. M. Wong, and M. Zukerman, "Obs contention resolution performance," *Perform. Eval.*, vol. 64, no. 4, pp. 357–373, May 2007.

[155] A. Zalesky, H. Vu, Z. Rosberg, E. W. M. Wong, and M. Zukerman, "Stabilizing deflection routing in optical burst switched networks," *IEEE J. Sel. Areas Commun.*, vol. 25, no. 6, pp. 3–19, Aug. 2007.

[156] E. W. Zegura, K. L. Calvert, and M. J. Donahoo, "A quantitative comparison of graph-based models for Internet topology," *IIEEE/ACM Trans. Netw.*, vol. 5, no. 6, pp. 770–783, Dec. 1997.

[157] S. Zhang, Y. Qian, J. Wu, and S. Lu, "An opportunistic resource sharing and topology-aware mapping framework for virtual networks," in *Proc. IEEE INFOCOM*, Orlando, FL, USA, Mar. 2012, pp. 2408–2416.

[158] Z. Zhang, X. Cheng, S. Su, Y. Wang, K. Shuang, and Y. Luo, "A unified enhanced particle swarm optimization-based virtual network embedding algorithm," *Int. J. Commun. Syst.*, vol. 26, no. 8, pp. 1054–1073, Aug. 2012.

[159] Y. Zhu and M. Ammar, "Algorithms for assigning substrate network resources to virtual network components," in *Proc. IEEE INFOCOM*, Barcelona, Spain, Apr. 2006, pp. 1–12.

[160] M. Zukerman, T. D. Neame, and R. G. Addie, "Internet traffic modeling and future technology implications," in *Proc. IEEE INFOCOM*, vol. 1, San Francisco, CA, USA, Mar. 2003, pp. 587–596.

Appendix A

iDef: Selected Code Sections

Listing A.1: deflector.h: Deflector Base Class

```
1  /* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
2  /*
3   * Copyright (c) 2011 Centre for Chaos and Complex Networks,
4   *                    Department of Electronic Engineering,
5   *                    City University of Hong Kong;
6   *                    Communication Networks Laboratory,
7   *                    School of Engineering Science,
8   *                    Simon Fraser University, Burnaby, BC, Canada
9   *
10  * This program is free software; you can redistribute it and/or modify
11  * it under the terms of the GNU General Public License version 2 as
12  * published by the Free Software Foundation;
13  *
14  * This program is distributed in the hope that it will be useful,
15  * but WITHOUT ANY WARRANTY; without even the implied warranty of
16  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17  * GNU General Public License for more details.
18  *
19  * You should have received a copy of the GNU General Public License
20  * along with this program; if not, write to the Free Software
21  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
22  *
23  * Author: Wilson Wang-Kit Thong <wilsonwk@ee.cityu.edu.hk>
24  *         Soroush Haeri <soroosh.haeri@me.com>
25  */
26 #ifndef DEFLECTOR_H
27 #define DEFLECTOR_H
28
29 #include "ns3/ipv4.h"
30 #include "ns3/ipv4-l3-protocol.h"
31 #include "ns3/packet.h"
32 #include "ns3/net-device.h"
33 #include "ns3/uinteger.h"
34
35 namespace ns3 {
36
37 class Ipv4DeflectionRouting;
38
39 class Deflector : public Object {
40 public:
41   static TypeId GetTypeId(void);
42
43   Deflector();
44
```

```
45    /**
46     * Set the node for which this PomcpDeflector serves
47     */
48    void SetNode(Ptr<Node> node);
49    virtual void SetIpv4DeflectionRouting(Ptr<Ipv4DeflectionRouting> routing) = 0;
50    Ptr<Ipv4> GetIpv4() const;
51    /*
52     * this method must be implemented in any instance of
53     * class derived from deflector
54     */
55    virtual Ptr<Ipv4Route> GetDeflectionRoute(Ptr<const Packet> p,
56                                              const Ipv4Header &header,
57                                              Ptr<Ipv4Route> defaultRoute,
58                                              Ptr<const NetDevice> idev) = 0;
59    virtual void ProcessPacketAfterArrivalCallback(const Ipv4Header &header,
60                                                   Ptr<const Packet> p,
61                                                   uint32_t rxif) = 0;
62    virtual void DropCallback(const Ipv4Header &header, Ptr<const Packet> _p,
63                              Ipv4L3Protocol::DropReason reason, Ptr<Ipv4> ipv4,
64                              uint32_t dropif) = 0;
65    virtual void PacketTxCallback(Ptr<const Packet> _p, Ptr<Ipv4> ipv4,
66                                  uint32_t txif) = 0;
67    virtual void PacketRxCallback(Ptr<const Packet> _p, Ptr<Ipv4> ipv4,
68                                  uint32_t rxif) = 0;
69
70  private:
71    Ptr<Ipv4> m_ipv4;
72  };
73
74  } /* namespace ns3 */
75  #endif /* DEFLECTOR_H */
```

Listing A.2: rl-deflector.h: Deflection Manager Base Class

```
1   /* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
2   /*
3    * Copyright (c) 2012 Communication Networks Laboratory,
4    *                    School of Engineering Science,
5    *                    Simon Fraser University, Burnaby, BC, Canada
6    *
7    * This program is free software; you can redistribute it and/or modify
8    * it under the terms of the GNU General Public License version 2 as
9    * published by the Free Software Foundation;
10   *
11   * This program is distributed in the hope that it will be useful,
12   * but WITHOUT ANY WARRANTY; without even the implied warranty of
13   * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
14   * GNU General Public License for more details.
15   *
16   * You should have received a copy of the GNU General Public License
17   * along with this program; if not, write to the Free Software
18   * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
19   *
20   * Author:  Wilson Wang-Kit Thong <wilsonwk@ee.cityu.edu.hk>
21   *          Soroush Haeri <soroosh.heri@me.com>
22   *
23   */
24  #ifndef RL_DEFLECTOR_H_
25  #define RL_DEFLECTOR_H_
26
27  #include "ns3/deflector.h"
28  #include "ns3/signal-manager.h"
29  #include "ns3/object-base.h"
30  #include "ns3/node.h"
31  #include "ns3/pointer.h"
```

```
32 #include "ns3/ipv4-route.h"
33 #include "ns3/packet.h"
34 #include "ns3/ipv4-header.h"
35 #include "ns3/net-device.h"
36 #include "ns3/socket.h"
37 #include "ns3/rl-header.h"
38 #include "ns3/rl-tag.h"
39 #include "ns3/decision-maker-abstract.h"
40 #include "ns3/rl-mappings.h"
41 #include "ns3/ipv4-l3-protocol.h"
42 #include "ns3/inet-socket-address.h"
43 #include "ns3/pseudo-obs-net-device.h"
44 #include <vector>
45
46 namespace ns3 {
47 class DecisionMaker;
48 class SignalManager;
49 class RLMappings;
50
51 class RLDeflector : public Deflector {
52 private:
53   typedef Callback<Ptr<Ipv4Route>, Ptr<const Packet>, const Ipv4Header &,
54                    Ptr<Ipv4Route>, Ptr<const NetDevice>> DeflectionCallback_t;
55
56 public:
57   static const uint32_t RL_PORT;
58
59   static TypeId GetTypeId(void);
60
61   RLDeflector();
62
63   /**
64    * Set the decision maker process
65    * and set the update callback from within
66    */
67   void SetDecisionMaker(Ptr<DecisionMaker> DM);
68   /**
69   * Set the signal manager
70   * and set the signal manager updateDM callback from within
71   */
72   void SetSignalManager(Ptr<SignalManager> SM);
73   /**
74   * Set the signal manager
75   * and set the signal manager updateDM callback from within
76   */
77   void SetMappings(Ptr<RLMappings> mapping);
78   /**
79     * Set the callback function to return an alternate
80     * route (deflect) when the default route is blocked
81     *
82     * \param cb the callback to handle deflection
83     */
84   void SetDeflectionCallback(DeflectionCallback_t cb);
85   void SetIpv4DeflectionRouting(Ptr<Ipv4DeflectionRouting> routing);
86   /**
87      * Given an input net-device and a default route, return
88      * a route to deflect packet away from the default route
89      *
90      * \param p the packet being deflected
91      * \param header the IPv4 header of the deflected packet
92      * \param defaultRoute the default route which is not to be used
93      * \param dev the net-device from which the packet comes to
94      * the node
95      * \return the deflected route for the packet to leave the node
96      */
97   Ptr<Ipv4Route> GetDeflectionRoute(Ptr<const Packet> p,
98                                     const Ipv4Header &header,
```

```
99                                      Ptr<Ipv4Route> defaultRoute,
100                                     Ptr<const NetDevice> idev);
101   Ptr<DecisionMaker> GetDecisionMaker() const;
102   Ptr<SignalManager> GetSignalManager() const;
103   Ptr<RLMappings> GetMappings() const;
104   Ptr<Node> GetNode() const;
105
106   /**
107     * Send notification packet back to the first deflector when
108     * the packet is dropped, if it has been deflected at all
109     *
110     * \param header the header of the dropped packet
111     * \param p the dropped packet itself
112     * \param reason the reason of why the packet is dropped
113     * \param ipv4 the node from which the packet is dropped
114     * \param dropif the IPv4 interface index from which the packet is dropped
115     */
116   void DropCallback(const Ipv4Header &header, Ptr<const Packet> p,
117                     Ipv4L3Protocol::DropReason reason, Ptr<Ipv4> ipv4,
118                     uint32_t dropif);
119   /**
120     * When a packet is transmitted this function is called
121     * it is used to update the nodes statistics
122     *
123     * \param p the dropped packet itself
124     * \param ipv4 the node from which the packet is dropped
125     * \param txif the IPv4 interface index from which the packet is transmitted
126     */
127   void PacketTxCallback(Ptr<const Packet> _p, Ptr<Ipv4> ipv4, uint32_t txif);
128   /**
129     * When a packet is received this function is called
130     * it is used to update the nodes statistics or send back
131     * feedbacks.
132     *
133     * \param p the dropped packet itself
134     * \param ipv4 the node from which the packet is dropped
135     * \param rxif the IPv4 interface index from which the packet is recieved
136     */
137   void PacketRxCallback(Ptr<const Packet> _p, Ptr<Ipv4> ipv4, uint32_t rxif);
138   /**
139    * Used to clean PacketTags
140    *
141    * \param header the header of the arrival packet
142    * \param p the arrived packet
143    * \param rxif the IPv4 interface index from at which the packet arrives
144    */
145   void ProcessPacketAfterArrivalCallback(const Ipv4Header &header,
146                                          Ptr<const Packet> p, uint32_t rxif);
147   /**
148      * \return true if the route is blocked by an busy net-device; false
149    * otherwise
150      */
151   bool IsBlocked(Ptr<Ipv4Route> route);
152
153   Ptr<Socket> GetSocket();
154
155 private:
156   /*
157    * TODO:
158    * this is the compatibility checking map to make sure
159    * the selected signal manager, decision maker and state mappings are
160    * compatible.
161    * it is to be implemented.
162    */
163   void ReceiveControl(Ptr<Socket> socket);
164
165   /**
```

```
166     * Setup UDP socket for sending and receiving control packets
167     */
168   void RLSocketSetup();
169
170   /// For sending and receiving control packets
171   Ptr<Socket> m_socket;
172
173   /*
174    * an instance of the decision maker.
175    */
176   Ptr<DecisionMaker> m_decisionMaker;
177   Ptr<SignalManager> m_signalManager;
178   Ptr<RLMappings> m_mappings;
179
180   /*
181    * this is the call back set for updating the decision maker
182    * it is set to call the decision maker's update function.
183    * this call back is set upon initializing the decision maker of
184    * the deflector.
185    */
186
187   // this callback is set by the signal manager
188   // before each deflection this callback is set
189   DeflectionCallback_t DeflectionCallback;
190 };
191 }
192
193 #endif /* RL_DEFLECTOR_H_ */
```

Listing A.3: decision-making-abstract.h: Decision Maker Module Base Class

```
1  /* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
2  /*
3   * Copyright (c) 2012 Communication Networks Laboratory,
4   *                    School of Engineering Science,
5   *                    Simon Fraser University, Burnaby, BC, Canada
6   *
7   * This program is free software; you can redistribute it and/or modify
8   * it under the terms of the GNU General Public License version 2 as
9   * published by the Free Software Foundation;
10  *
11  * This program is distributed in the hope that it will be useful,
12  * but WITHOUT ANY WARRANTY; without even the implied warranty of
13  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
14  * GNU General Public License for more details.
15  *
16  * You should have received a copy of the GNU General Public License
17  * along with this program; if not, write to the Free Software
18  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
19  *
20  * Author: Soroush Haeri <soroosh.heri@me.com>
21  *
22  */
23 #ifndef DECISION_MAKER_ABSTRACT_H_
24 #define DECISION_MAKER_ABSTRACT_H_
25
26 #include <tr1/tuple>
27
28 //#include "generative-model-abstract.h"
29 #include "ns3/pointer.h"
30 #include "ns3/node.h"
31 #include "ns3/rl-deflector.h"
32
33 using namespace std;
34 namespace ns3 {
```

```
35
36 class RLDeflector;
37 /*
38  * This is the class abstraction for implementation of
39  * any decision making class.
40  * the deflection routing module may use any of the
41  * Implementations of this class abstraction.
42  */
43
44 class DecisionMaker : public Object {
45
46 public:
47   // constructor
48   DecisionMaker();
49   // DecisionMaker (uint32_t numActions_, uint32_t numStates_, uint32_t dmType_)
50   //:numActions(numActions_),
51   // numStates(numStates_),
52   // dmType(dmType_)
53   //{
54   //}
55   // destructor
56   // virtual ~DecisionMaker();
57   static TypeId GetTypeId(void);
58   /*
59    * \brief any implementation of the DecisionMaker
60    * must have select action. selectAction gets the
61    * current states of a node and returns the best
62    * action selected by the underlying Reinforcement Learning algorithm.
63    *
64    * \param state current state of the node.
65    * \return the action chosen by the underlying RL algorithm.
66 *
67   */
68   virtual uint32_t selectAction(uint64_t state) = 0;
69   /*
70    * \brief this function updates the decision maker after an action
71    * has been performed and the result of performing the action is
72    * seen on the channel.
73    *
74    * \param state the state of the node at which decision has been made if state
75    * is -1.
76    * \param action the action that was prescribed by the decision maker.
77    * \param reward the reward recieved as the result of performing the action.
78    */
79   virtual void update(uint64_t state, uint32_t action, double reward) = 0;
80   /*
81    * \brief this function updates an associative learning  decision makers
82    * weight matrices
83    * after an action has been performed and the result of performing the action
84    * is
85    * seen on the channel.
86    *
87    * \param reward the reward recieved as the result of performing the action.
88    */
89   // virtual void update (double reward);
90   /**
91    * Set ns3::Node to this simulator. It must be called after this simulator is
92    * constructed.
93    *
94    * \param node the ns3::Node to which this simulator attaches
95    */
96   // virtual void SetNode (Ptr<Node> node);
97   bool IsBusy();
98   void SetMaster(Ptr<RLDeflector> _master);
99   Ptr<RLDeflector> GetMaster();
100  /*any necessary operation that is needed to be done to make
101  * the decision maker ready to operate must be implemented in this function
```

```
102   * it is callsed from set master when the decision maker knows to which node
103   * it belongs it may initialize the required matrices or variables.
104   */
105   virtual void Initialize() = 0;
106   /*
107    * \brief this function may be implemented by Q learning based algorithms
108    *                    where Q value needs to be passed out to other modules.
109    *                    see RldrsSignalManager and RldrsQlearningDecisionMaker.
110    * \param state the state the Q value of which is required.
111    * \param action the action the Q value of which is required.
112    *
113    * \return double the query result. (Q value)
114    */
115   virtual double GetLastReward(uint64_t state, uint32_t action);
116
117 protected:
118   Ptr<RLDeflector> master;
119   /**
120    * \return Count the number of devices in specific type attached to the ns-3
121    * node
122    */
123   template <typename Device>
124   uint32_t GetNumberOfNeighbors(Ptr<Node> node) const;
125   /*
126    * \param numStates maximum number of states that a node may find itself at.
127    * \param numActions maximum number of actions available to a node.
128    * \param dmType is an integer defining the type of the decision making entity
129    *               this variable is used to determine the behavior of
130    *               the update function and if a simulator is needed
131    *               0: refers to class of associative learning algorithms such as
132    *               neural network algorithm where update function only requires one
       argument.
133    * \param nodeId keeps the id of the node this decision maker belongs to.
134    * \param numNeighbors keeps the number of neighbors of the node it is set at
135    * the initialization time.
136    */
137   uint64_t numStates;
138   // const uint32_t dmType;
139   uint32_t numNeighbors;
140   uint32_t nodeId;
141   /**
142 *\param reward the reward generated by Hmm
143 *\param nodeId the node ID from which the reward is found
144 */
145   // TracedCallback<double, uint32_t> m_rewardGeneratedTrace;
146   /**
147    * \param obs the observation generated by Hmm
148    * \param nodeId the node ID from which the observation is found
149    */
150   // TracedCallback<uint32_t, uint32_t> m_observationGeneratedTrace;
151   bool isBusy;
152 };
153 } // namespace ns3
154
155 #endif /* DECISION_MAKER_ABSTRACT_H_ */
```

Listing A.4: signal-manager.h: Signalling Module Base Class

```
1  /* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
2  /*
3   * Copyright (c) 2012 Communication Networks Laboratory,
4   *                    School of Engineering Science,
5   *                    Simon Fraser University, Burnaby, BC, Canada
6   *
7   * This program is free software; you can redistribute it and/or modify
```

```
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Author: Soroush Haeri <soroosh.heri@me.com>
 *
 */
#ifndef SIGNAL_MANAGER_H_
#define SIGNAL_MANAGER_H_

#include "ns3/rl-header.h"
//#include "ns3/decision-maker-abstract.h"
//#include "ns3/rl-mappings.h"
#include "ns3/reward-code-book.h"
#include "ns3/rl-deflector.h"
#include "ns3/object.h"
#include "ns3/ipv4.h"
#include "ns3/packet.h"
#include "ns3/node.h"
#include "ns3/ptr.h"
#include "ns3/ipv4-l3-protocol.h"

namespace ns3 {
class DecisionMaker;
class RLDeflector;

class SignalManager : public Object {
public:
  typedef Callback<void, uint64_t, uint32_t, double> UpdateDM_t;

public:
  SignalManager();
  static TypeId GetTypeId(void);
  virtual ~SignalManager();
  void setUpdateDMCallBack(UpdateDM_t _updateDM);
  void SetMaster(Ptr<RLDeflector> _master);
  Ptr<RLDeflector> GetMaster();
  /*
   * called from recieve control it is the function that processes feedback
   * messages
   * received from RL socket
   */
  virtual void PostProcessFeedBack(Ptr<Packet> p) = 0;
  virtual void PreProcessPacketTags(Ptr<Packet> p) = 0;
  virtual void ProcessDrop(const Ipv4Header &header, Ptr<const Packet> _p,
                           Ipv4L3Protocol::DropReason reason, Ptr<Ipv4> ipv4,
                           uint32_t dropif) = 0;
  virtual void ProcessPacketAfterArrival(Ptr<const Packet> p) = 0;
  virtual void ProcessTransmit(Ptr<const Packet> _p, Ptr<Ipv4> ipv4,
                               uint32_t txif) = 0;
  virtual void ProcessReceive(Ptr<const Packet> _p, Ptr<Ipv4> ipv4,
                              uint32_t rxif) = 0;

protected:
  Ptr<RLDeflector> master;
  /*
   * Callback to update decision maker
   */
  UpdateDM_t updateDM;
```

```
75
76 private:
77 };
78 }
79
80 #endif /* SIGNAL_MANAGER_H_ */
```

Listing A.5: rl-mappings.h: Mapping Module Base Class

```
1  /* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
2  /*
3   * Copyright (c) 2012 Communication Networks Laboratory,
4   *                    School of Engineering Science,
5   *                    Simon Fraser University, Burnaby, BC, Canada
6   *
7   * This program is free software; you can redistribute it and/or modify
8   * it under the terms of the GNU General Public License version 2 as
9   * published by the Free Software Foundation;
10  *
11  * This program is distributed in the hope that it will be useful,
12  * but WITHOUT ANY WARRANTY; without even the implied warranty of
13  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
14  * GNU General Public License for more details.
15  *
16  * You should have received a copy of the GNU General Public License
17  * along with this program; if not, write to the Free Software
18  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
19  *
20  * Author: Soroush Haeri <soroosh.heri@me.com>
21  *
22  */
23 #ifndef RL_MAPPINGS_H_
24 #define RL_MAPPINGS_H_
25 #include "ns3/rl-deflector.h"
26 #include "ns3/object.h"
27 #include "ns3/ptr.h"
28 #include "ns3/integer.h"
29 #include "ns3/node.h"
30 #include "ns3/ipv4-route.h"
31 #include "ns3/packet.h"
32 #include "ns3/ipv4-header.h"
33 #include "ns3/net-device.h"
34 #include "ns3/ipv4-address.h"
35 #include <map>
36 #include <tr1/tuple>
37 /*
38  * This class is used to define states for a reinforcement-learning
39  * agent that is making deflection decisions.
40  * For example one may decide to define the state as the destination of a
41  * packet while another define the state as the nodes link state.
42  * The decision making agent however does not need to know the definition of the
43  * state but only a number representing a state.
44  * All decision making agents must have a state-integer mapping that is derived
45  * from this class.
46  * we assume 5 arguments maximum however the user may pass any number of
47  * arguments up to 4
48  * and the may handle it in their Getstate() function implemention.
49  */
50 namespace ns3 {
51
52 class RLDeflector;
53
54 class RLMappings : public Object {
55 public:
56   RLMappings();
```

```
57    static TypeId GetTypeId(void);
58    virtual ~RLMappings();
59    virtual uint64_t GetState(Ptr<const Packet> p, const Ipv4Header &header,
60                              Ptr<Ipv4Route> defaultRoute,
61                              Ptr<const NetDevice> idev);
62    virtual uint64_t GetState(Ipv4Address destAddr);
63    Ptr<NetDevice> GetNetDeviceFromActionIndex(uint32_t action);
64    uint32_t GetActionIndexFromNetDevice(Ptr<NetDevice> net);
65    uint32_t GetNActions();
66    void SetMaster(Ptr<RLDeflector>);
67    Ptr<RLDeflector> GetMaster();
68
69  protected:
70    Ptr<RLDeflector> master;
71    void Initialize();
72  };
73  }
74  #endif
```

Appendix B

VNE-Sim: Selected Code Sections

Listing B.1: two-stage-embedding-algo.h: Two stage embedding algorithm interface implementation.

```
1  /**
2   * @file two-stage-embedding-algo.h
3   * @author Soroush Haeri <soroosh.haeri@me.com>
4   * @date Jun 12, 2014
5   *
6   * @copyright Copyright (c) Jun 12, 2014                    SOROUSH HAERI
7   *            All Rights Reserved
8   *
9   *     Permission to use, copy, modify, and distribute this software and its
10  *     documentation for any purpose and without fee is hereby granted, provided
11  *     that the above copyright notice appear in all copies and that both that
12  *     copyright notice and this permission notice appear in supporting
13  *     documentation, and that the name of the author not be used in advertising
14  *     or publicity pertaining to distribution of the software without specific,
15  *     written prior permission.
16  *
17  *     THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
18  *     ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
19  *     AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
20  *     DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
21  *     AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
22  *     OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
23  **/
24
25 #ifndef TWO_STAGE_EMBEDDING_ALGO_H_
26 #define TWO_STAGE_EMBEDDING_ALGO_H_
27
28 #include "embedding-algorithm.h"
29 #include "node-embedding-algorithm.h"
30 #include "link-embedding-algorithm.h"
31
32 namespace vne {
33     template<typename SUBNET, typename VNR>
34     class TwoStageEmbeddingAlgo : public EmbeddingAlgorithm<SUBNET, VNR>
35     {
36     public:
37         virtual Embedding_Result embeddVNR (std::shared_ptr<typename
                EmbeddingAlgorithm<SUBNET, VNR>::VNR_TYPE> vnr);
38         virtual ~TwoStageEmbeddingAlgo () {};
39
```

```
40      protected:
41          TwoStageEmbeddingAlgo (NetworkBuilder<typename EmbeddingAlgorithm<SUBNET, VNR
                >::SUBSTRATE_TYPE>& _sb,
42                                 std::shared_ptr<NodeEmbeddingAlgorithm<SUBNET, VNR>>
                                       _node_embedding_algo,
43                                 std::shared_ptr<LinkEmbeddingAlgorithm<SUBNET, VNR>>
                                       _link_embedding_algo);
44          TwoStageEmbeddingAlgo (std::shared_ptr<typename EmbeddingAlgorithm<SUBNET, VNR
                >::SUBSTRATE_TYPE> _sn,
45                                 std::shared_ptr<NodeEmbeddingAlgorithm<SUBNET, VNR>>
                                       _node_embedding_algo,
46                                 std::shared_ptr<LinkEmbeddingAlgorithm<SUBNET, VNR>>
                                       _link_embedding_algo);
47          std::shared_ptr<NodeEmbeddingAlgorithm<SUBNET, VNR>> node_embedding_algo;
48          std::shared_ptr<LinkEmbeddingAlgorithm<SUBNET, VNR>> link_embedding_algo;
49
50      };
51      template<typename SUBNET, typename VNR>
52      TwoStageEmbeddingAlgo<SUBNET,VNR>::TwoStageEmbeddingAlgo (NetworkBuilder<typename
            EmbeddingAlgorithm<SUBNET, VNR>::SUBSTRATE_TYPE>& _sb,
53          std::shared_ptr<NodeEmbeddingAlgorithm<SUBNET, VNR>> _node_embedding_algo,
54          std::shared_ptr<LinkEmbeddingAlgorithm<SUBNET, VNR>> _link_embedding_algo):
55      EmbeddingAlgorithm<SUBNET, VNR>::EmbeddingAlgorithm(_sb),
56      node_embedding_algo(_node_embedding_algo),
57      link_embedding_algo(_link_embedding_algo)
58      {
59      }
60      template<typename SUBNET, typename VNR>
61      TwoStageEmbeddingAlgo<SUBNET,VNR>::TwoStageEmbeddingAlgo (std::shared_ptr<typename
             EmbeddingAlgorithm<SUBNET, VNR>::SUBSTRATE_TYPE> _sn,
62          std::shared_ptr<NodeEmbeddingAlgorithm<SUBNET, VNR>> _node_embedding_algo,
63          std::shared_ptr<LinkEmbeddingAlgorithm<SUBNET, VNR>> _link_embedding_algo):
64      EmbeddingAlgorithm<SUBNET, VNR>::EmbeddingAlgorithm(_sn),
65      node_embedding_algo(_node_embedding_algo),
66      link_embedding_algo(_link_embedding_algo)
67      {
68      }
69      template<typename SUBNET, typename VNR>
70      Embedding_Result TwoStageEmbeddingAlgo<SUBNET,VNR>
71      ::embeddVNR(std::shared_ptr<typename EmbeddingAlgorithm<SUBNET, VNR>::VNR_TYPE>
            vnr)
72      {
73          if (node_embedding_algo->embeddVNRNodes (this->substrate_network, vnr) ==
                Embedding_Result::SUCCESSFUL_EMBEDDING
74              &&
75             link_embedding_algo->embeddVNRLinks (this->substrate_network, vnr) ==
                    Embedding_Result::SUCCESSFUL_EMBEDDING)
76          {
77              //finalize the node mappings
78
79              for (auto it = vnr->getNodeMap()->begin(); it !=  vnr->getNodeMap()->end()
                    ; it++)
80              {
81                  assert(this->substrate_network->getNode (it->second)->embedNode(vnr->
                        getVN()->getNode(it->first)) == Embedding_Result::
                        SUCCESSFUL_EMBEDDING && "Trying to map to a more than you have
                        resources!!");
82              }
83              //finalize the link mappings
84              for(auto it1 = vnr->getLinkMap()->begin(); it1 != vnr->getLinkMap()->end()
                    ; it1++)
85              {
86                  for(auto it2 = it1->second.begin(); it2 != it1->second.end() ;it2++)
87                  {
88                      assert(this->substrate_network->getLink(it2->first)->embedLink (
                            vnr->getVN()->getLink(it1->first), it2->second)==
```

```
                        Embedding_Result::SUCCESSFUL_EMBEDDING && "Trying to map to a
                        link more than you have resources!!");
                }
            }
            return Embedding_Result::SUCCESSFUL_EMBEDDING;
        }
        return Embedding_Result::NOT_ENOUGH_SUBSTRATE_RESOURCES;
    }
}
#endif
```

Listing B.2: node-embedding-algorithm.h: Node embedding algorithm interface implementation.

```
/**
 * @file node-embedding-algorithm.h
 * @author Soroush Haeri <soroosh.haeri@me.com>
 * @date Jun 12, 2014
 *
 * @copyright Copyright (c) Jun 12, 2014                    SOROUSH HAERI
 *      All Rights Reserved
 *
 *      Permission to use, copy, modify, and distribute this software and its
 *      documentation for any purpose and without fee is hereby granted, provided
 *      that the above copyright notice appear in all copies and that both that
 *      copyright notice and this permission notice appear in supporting
 *      documentation, and that the name of the author not be used in advertising
 *      or publicity pertaining to distribution of the software without specific,
 *      written prior permission.
 *
 *      THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
 *      ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
 *      AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
 *      DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
 *      AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 *      OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 **/

#ifndef NODE_EMBEDDING_ALGORITHM_H_
#define NODE_EMBEDDING_ALGORITHM_H_

#include "network.h"
#include "substrate-link.h"
#include "substrate-node.h"
#include "virtual-network-request.h"
#include "network-builder.h"
#include "config-manager.h"

namespace vne {
    template<typename,typename> class NodeEmbeddingAlgorithm;

    template<
    typename ... SNODERES, template <typename ...> class SNODECLASS,
    typename... SLINKRES, template <typename...> class SLINKCLASS,
    typename ... VNODERES, template <typename ...> class VNODECLASS,
    typename... VLINKRES, template <typename...> class VLINKCLASS,
    template<typename> class VNRCLASS>
    class NodeEmbeddingAlgorithm<Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES
        ...>>,
    VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>>
    {
        static_assert (std::is_base_of<SubstrateNode<SNODERES...>, SNODECLASS<SNODERES
            ...>>::value,
                       "Template arguments are not correctly set.");
```

```
49          static_assert (std::is_base_of<VirtualNode<VNODERES...>, VNODECLASS<VNODERES
                ...>>::value,
50                         "Template arguments are not correctly set.");
51          static_assert (std::is_base_of<SubstrateLink<SLINKRES...>, SLINKCLASS<SLINKRES
                ...>>::value,
52                         "Template arguments are not correctly set.");
53          static_assert (std::is_base_of<VirtualLink<VLINKRES...>, VLINKCLASS<VLINKRES
                ...>>::value,
54                         "Template arguments are not correctly set.");
55          static_assert (std::is_base_of<VirtualNetworkRequest<Network<VNODECLASS<
                VNODERES...>, VLINKCLASS<VLINKRES...>>>,
56                         VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES
                               ...>>>>::value, "Template arguments are not correctly set."
                               );
57
58      public:
59          const static bool IgnoreLocationConstrain();
60
61          typedef VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>
                VNR_TYPE;
62          typedef Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES...>>
                SUBSTRATE_TYPE;
63
64          virtual Embedding_Result embeddVNRNodes (std::shared_ptr<SUBSTRATE_TYPE>
                substrate_net, std::shared_ptr<VNR_TYPE> vnr) = 0;
65
66      protected:
67          NodeEmbeddingAlgorithm () {};
68          static int ignoreLocationConstrain;
69      };
70
71      template<
72      typename ... SNODERES, template <typename ...> class SNODECLASS,
73      typename... SLINKRES, template <typename...> class SLINKCLASS,
74      typename ... VNODERES, template <typename ...> class VNODECLASS,
75      typename... VLINKRES, template <typename...> class VLINKCLASS,
76      template<typename> class VNRCLASS>
77      int NodeEmbeddingAlgorithm<Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES
            ...>>,
78      VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>>::
            ignoreLocationConstrain = -1;
79
80      template<
81      typename ... SNODERES, template <typename ...> class SNODECLASS,
82      typename... SLINKRES, template <typename...> class SLINKCLASS,
83      typename ... VNODERES, template <typename ...> class VNODECLASS,
84      typename... VLINKRES, template <typename...> class VLINKCLASS,
85      template<typename> class VNRCLASS>
86      const bool NodeEmbeddingAlgorithm<Network<SNODECLASS<SNODERES...>, SLINKCLASS<
            SLINKRES...>>,
87      VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>>::
            IgnoreLocationConstrain()
88      {
89              if (ignoreLocationConstrain == -1)
90                ignoreLocationConstrain  = (int) ConfigManager::Instance()->getConfig<
                      bool>("core.ignoreLocationConstrain");
91              return (bool) ignoreLocationConstrain;
92      }
93  }
94
95  #endif
```

Listing B.3: link-embedding-algorithm.h: Link embedding algorithm interface implementation.

```
/**
 * @file link-embedding-algorithm.h
 * @author Soroush Haeri <soroosh.haeri@me.com>
 * @date Jun 12, 2014
 *
 * @copyright Copyright (c) Jun 12, 2014                    SOROUSH HAERI
 *            All Rights Reserved
 *
 *     Permission to use, copy, modify, and distribute this software and its
 *     documentation for any purpose and without fee is hereby granted, provided
 *     that the above copyright notice appear in all copies and that both that
 *     copyright notice and this permission notice appear in supporting
 *     documentation, and that the name of the author not be used in advertising
 *     or publicity pertaining to distribution of the software without specific,
 *     written prior permission.
 *
 *     THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
 *     ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
 *     AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
 *     DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
 *     AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 *     OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 **/

#ifndef LINK_EMBEDDING_ALGORITHM_H_
#define LINK_EMBEDDING_ALGORITHM_H_

#include <list>

#include "network.h"
#include "substrate-link.h"
#include "substrate-node.h"
#include "virtual-network-request.h"
#include "network-builder.h"

namespace vne {
    template<typename,typename> class LinkEmbeddingAlgorithm;

    template<
    typename ... SNODERES, template <typename ...> class SNODECLASS,
    typename... SLINKRES, template <typename...> class SLINKCLASS,
    typename ... VNODERES, template <typename ...> class VNODECLASS,
    typename... VLINKRES, template <typename...> class VLINKCLASS,
    template<typename> class VNRCLASS>
    class LinkEmbeddingAlgorithm<Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES
        ...>>,
    VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>>
    {
        static_assert (std::is_base_of<SubstrateNode<SNODERES...>, SNODECLASS<SNODERES
            ...>>::value,
                       "Template arguments are not correctly set.");
        static_assert (std::is_base_of<VirtualNode<VNODERES...>, VNODECLASS<VNODERES
            ...>>::value,
                       "Template arguments are not correctly set.");
        static_assert (std::is_base_of<SubstrateLink<SLINKRES...>, SLINKCLASS<SLINKRES
            ...>>::value,
                       "Template arguments are not correctly set.");
        static_assert (std::is_base_of<VirtualLink<VLINKRES...>, VLINKCLASS<VLINKRES
            ...>>::value,
                       "Template arguments are not correctly set.");
        static_assert (std::is_base_of<VirtualNetworkRequest<Network<VNODECLASS<
            VNODERES...>, VLINKCLASS<VLINKRES...>>>,
                       VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES
                           ...>>>>::value, "Template arguments are not correctly set."
                           );
        //TwoStageEmbeddingAlgo is a friend
        template<typename SUBNET, typename VNR>
```

```
60          friend class TwoStageEmbeddingAlgo;
61      public:
62          typedef VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>
                VNR_TYPE;
63          typedef Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES...>>
                SUBSTRATE_TYPE;
64
65          virtual Embedding_Result embeddVNRLinks (std::shared_ptr<SUBSTRATE_TYPE>
                substrate_net, std::shared_ptr<VNR_TYPE> vnr) = 0;
66          //implementing this funciton is optional. However, for using MCTS this
                function must be implemented.
67          //This function works on the given id sets insetad of the vnr default id sets.
68          virtual Embedding_Result embeddVNRLinksForIdSets (std::shared_ptr<
                SUBSTRATE_TYPE> substrate_net, std::shared_ptr<VNR_TYPE> vnr,
69              const std::map<int,int>* nodeIdMap,
70              std::map<int,std::list<std::pair<int, std::shared_ptr<Resources<SLINKRES
                    ...>>>>>* linkMap)
71          {return embeddVNRLinks (substrate_net,vnr);};
72
73          Link_Embedding_Algo_Types getType () const {return type;};
74
75      protected:
76          LinkEmbeddingAlgorithm (Link_Embedding_Algo_Types t) : type (t) {};
77          Link_Embedding_Algo_Types type;
78
79      };
80  }
81
82  #endif
```

Listing B.4: mcvne-node-embedding-algo.h: MaVEn Node Mapping Algorithm Header File

```
1   /**
2    * @file mcvne-node-embedding-algo.h
3    * @author Soroush Haeri <soroosh.haeri@me.com>
4    * @date 7/16/14
5    *
6    * @copyright Copyright (c) 7/16/14                        SOROUSH HAERI
7    *      All Rights Reserved
8    *
9    *      Permission to use, copy, modify, and distribute this software and its
10   *      documentation for any purpose and without fee is hereby granted, provided
11   *      that the above copyright notice appear in all copies and that both that
12   *      copyright notice and this permission notice appear in supporting
13   *      documentation, and that the name of the author not be used in advertising or
14   *      publicity pertaining to distribution of the software without specific,
15   *      written prior permission.
16   *
17   *      THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
18   *      ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
19   *      AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
20   *      DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
21   *      AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
22   *      OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
23   **/
24
25  #ifndef MCVNE_MCVNE_NODE_EMBEDDING_ALGO_
26  #define MCVNE_MCVNE_NODE_EMBEDDING_ALGO_
27
28  #include "mcvne/mcvne-simulator.h"
29
30  #include "core/node-embedding-algorithm.h"
31
32  using namespace vne::vineyard;
33
```

```
34 namespace vne {
35     namespace mcvne {
36         template<typename = Network<VYSubstrateNode<>,VYSubstrateLink<>> ,
37         typename = VYVirtualNetRequest<>>
38         class MCVNENodeEmbeddingAlgo : public NodeEmbeddingAlgorithm
39                                        <Network<VYSubstrateNode<>,VYSubstrateLink<>>,
                                             VYVirtualNetRequest<>>
40         {
41         public:
42             MCVNENodeEmbeddingAlgo ();
43             ~MCVNENodeEmbeddingAlgo ();
44             virtual Embedding_Result  embeddVNRNodes (std::shared_ptr<SUBSTRATE_TYPE>
                    substrate_network, std::shared_ptr<VNR_TYPE> vnr);
45         private:
46             std::shared_ptr<LinkEmbeddingAlgorithm<SUBSTRATE_TYPE, VNR_TYPE>>
                    link_embedder;
47         };
48     }
49 }
50 #endif /* defined(__vne_mcts__mcvne_node_embedding_algo__) */
```

Listing B.5: mcvne-node-embedding-algo.h: MaVEn Node Mapping Algorithm Source File

```
1  /**
2   * @file mcvne-node-embedding-algo.cc
3   * @author Soroush Haeri <soroosh.haeri@me.com>
4   * @date 7/16/14
5   *
6   * @copyright Copyright (c) 7/16/14                    SOROUSH HAERI
7   *      All Rights Reserved
8   *
9   *      Permission to use, copy, modify, and distribute this software and its
10  *      documentation for any purpose and without fee is hereby granted, provided
11  *      that the above copyright notice appear in all copies and that both that
12  *      copyright notice and this permission notice appear in supporting
13  *      documentation, and that the name of the author not be used in advertising or
14  *      publicity pertaining to distribution of the software without specific,
15  *      written prior permission.
16  *
17  *      THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
18  *      ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
19  *      AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
20  *      DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
21  *      AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
22  *      OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
23  **/
24
25 #include "mcvne-node-embedding-algo.h"
26 #include "mcvne-bfs-link-embedding-algo.h"
27
28 #include "mcts/mcts.h"
29 #include "Vineyard/vy-vine-link-embedding-algo.h"
30 #include "core/config-manager.h"
31
32 namespace vne {
33 namespace mcvne {
34
35 template <>
36 MCVNENodeEmbeddingAlgo<>::MCVNENodeEmbeddingAlgo()
37     : NodeEmbeddingAlgorithm<Network<VYSubstrateNode<>, VYSubstrateLink<>>,
38                              VYVirtualNetRequest<>>() {
39   std::string linkEmbedderName =
40       ConfigManager::Instance()->getConfig<std::string>(
41           "MCVNE.NodeEmbeddingAlgo.LinkEmbedder");
42
```

```
  if (linkEmbedderName.compare("MCF") == 0)
    link_embedder = std::shared_ptr<VYVineLinkEmbeddingAlgo<>>(
        new VYVineLinkEmbeddingAlgo<>());
  else if (linkEmbedderName.compare("BFS-SP") == 0)
    link_embedder = std::shared_ptr<MCVNEBFSLinkEmbeddingAlgo<>>(
        new MCVNEBFSLinkEmbeddingAlgo<>());
  else
    link_embedder = std::shared_ptr<VYVineLinkEmbeddingAlgo<>>(
        new VYVineLinkEmbeddingAlgo<>());
};

template <> MCVNENodeEmbeddingAlgo<>::~MCVNENodeEmbeddingAlgo(){};

template <>
Embedding_Result MCVNENodeEmbeddingAlgo<>::embeddVNRNodes(
    std::shared_ptr<SUBSTRATE_TYPE> substrate_network,
    std::shared_ptr<VNR_TYPE> vnr) {
  std::shared_ptr<MCVNESimulator<>> sim(
      new MCVNESimulator<>(substrate_network, vnr, link_embedder));
  MCTS mcts(sim);

  std::shared_ptr<VNENMState> st =
      std::static_pointer_cast<VNENMState>(sim->createStartState());

  bool terminate;
  double reward;
  int action;
  do {
    action = mcts.selectAction();
    terminate = sim->step(st, action, reward);
    if (reward <= -Infinity)
      terminate = true;
    else {
      if (st->getNodeMap()->size() < vnr->getVN()->getNumNodes())
        mcts.update(action, reward);
    }
  } while (!terminate);

#ifdef ENABLE_MPI
  if (ConfigManager::Instance()->getConfig<int>(
          "MCTS.MCTSParameters.ParallelizationType") == 1) {
    struct {
      double val;
      int rank;
    } ObjectiveValueIn, ObjectiveValueOut;
    ObjectiveValueIn.val = reward;
    ObjectiveValueIn.rank = MPI::COMM_WORLD.Get_rank();
    MPI::COMM_WORLD.Allreduce(&ObjectiveValueIn, &ObjectiveValueOut, 1,
                              MPI::DOUBLE_INT, MPI::MAXLOC);

    int nodeMapSize;
    if (ObjectiveValueOut.rank == ObjectiveValueIn.rank) {
      nodeMapSize = (int)st->getNodeMap()->size();
    }

    MPI::COMM_WORLD.Bcast(&nodeMapSize, 1, MPI::INT, ObjectiveValueOut.rank);

    if (nodeMapSize != vnr->getVN()->getNumNodes())
      return Embedding_Result::NOT_ENOUGH_SUBSTRATE_RESOURCES;

    struct {
      int sNodeId;
      int vNodeId;
    } nodeMap[nodeMapSize];

    if (ObjectiveValueOut.rank == ObjectiveValueIn.rank) {
      int count = 0;
```

```
      for (auto it = st->getNodeMap()->begin(); it != st->getNodeMap()->end();
           it++) {
        nodeMap[count].sNodeId = it->second;
        nodeMap[count].vNodeId = it->first;
        count++;
      }
    }

    MPI::COMM_WORLD.Bcast(nodeMap, nodeMapSize, MPI_2INT,
                          ObjectiveValueOut.rank);

    for (int i = 0; i < nodeMapSize; i++) {
      vnr->addNodeMapping(nodeMap[i].sNodeId, nodeMap[i].vNodeId);
    }
  } else {
    if (st->getNodeMap()->size() != vnr->getVN()->getNumNodes())
      return Embedding_Result::NOT_ENOUGH_SUBSTRATE_RESOURCES;
    for (auto it = st->getNodeMap()->begin(); it != st->getNodeMap()->end();
         it++) {
      vnr->addNodeMapping(it->second, it->first);
    }
  }
#else
  if (st->getNodeMap()->size() != vnr->getVN()->getNumNodes())
    return Embedding_Result::NOT_ENOUGH_SUBSTRATE_RESOURCES;
  for (auto it = st->getNodeMap()->begin(); it != st->getNodeMap()->end();
       it++) {
    vnr->addNodeMapping(it->second, it->first);
  }
#endif
  return Embedding_Result::SUCCESSFUL_EMBEDDING;
}
}
}
```

Listing B.6: mcts.h: MCTS Header File

```
/**
 * @file mcts.h
 * @author Soroush Haeri <soroosh.haeri@me.com>
 * @date 7/16/14
 *
 * @note This class is based on the POMCP implementation by David Silver and
 *       Joel Veness.
 * POMCP is published in NIPS 2010:
 *      "Online Monte-Carlo Plannin/g in Large POMDPs".
 *
 * @copyright Copyright (c) 7/16/14                      SOROUSH HAERI
 *     All Rights Reserved
 *
 *     Permission to use, copy, modify, and distribute this software and its
 *     documentation for any purpose and without fee is hereby granted, provided
 *     that the above copyright notice appear in all copies and that both that
 *     copyright notice and this permission notice appear in supporting
 *     documentation, and that the name of the author not be used in advertising
 *     or publicity pertaining to distribution of the software without specific,
 *     written prior permission.
 *
 *     THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
 *     ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
 *     AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
 *     DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
 *     AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 *     OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 **/
```

```
29
30 #ifndef MCTS_MCTS_
31 #define MCTS_MCTS_
32
33 #include "mcts/tree-node.h"
34
35 #include "mcts/mcts-simulator.h"
36 #include "mcts/mcts-statistics.h"
37
38 #if ENABLE_MPI
39 #include <mpi.h>
40 #endif
41
42 namespace vne {
43 namespace mcts {
44
45 class MCTS {
46 public:
47   struct Parameters {
48     Parameters();
49
50     int MaxDepth;
51     int NumSimulations;
52     /** How many nodes to add at each expansion step */
53     int ExpandCount;
54     bool AutoExploration;
55     /** One option is to set Exploration Constant c = R_{hi}-R_{lo}*/
56     double ExplorationConstant;
57     bool UseRave;
58     double RaveDiscount;
59     double RaveConstant;
60     /** When set, the baseline rollout algorithm is run. */
61     bool DisableTree;
62     bool UseSinglePlayerMCTS;
63     double SPMCTSConstant;
64
65     // 0: Root Parallelization with synchronizaiton after each action
66     // 1: Root Parallelization without synch
67     int ParalleizationType;
68   };
69
70   MCTS(std::shared_ptr<MCTSSimulator> sim);
71   ~MCTS();
72
73   int selectAction();
74   bool update(int action, double reward);
75
76   void UCTSearch();
77   void rolloutSearch();
78
79   double rollout(std::shared_ptr<State> st);
80
81   const MCTSSimulator::Status &getStatus() const { return status; }
82
83   static void initFastUCB(double exploration);
84
85 private:
86   const std::shared_ptr<MCTSSimulator> simulator;
87   std::vector<int> history;
88   int treeDepth, peakTreeDepth;
89   Parameters params;
90   std::shared_ptr<TreeNode> root;
91   MCTSSimulator::Status status;
92
93   void clearStatistics();
94   MCTSStatistics statTreeDepth;
95   MCTSStatistics statRolloutDepth;
```

```
96    MCTSStatistics statTotalReward;
97
98    int greedyUCB(std::shared_ptr<TreeNode> node, bool ucb) const;
99    int selectRandom() const;
100   double simulateNode(std::shared_ptr<TreeNode> node);
101   void addRave(std::shared_ptr<TreeNode> node, double totalReward);
102   std::shared_ptr<TreeNode> expandNode(const std::shared_ptr<State> state);
103
104   // Fast lookup table for UCB
105   static const int UCB_N = 10000, UCB_n = 100;
106   static double UCB[UCB_N][UCB_n];
107   static bool initialisedFastUCB;
108
109   double fastUCB(int N, int n, double logN) const;
110 };
111 }
112 }
113 #if ENABLE_MPI
114 void sumFunction(const void *input, void *inoutput, int len,
115                  const MPI::Datatype &datatype);
116 #endif
117 #endif
```

Listing B.7: mcts.cc: MCTS Source File

```
1  /**
2   * @file mcts.cc
3   * @author Soroush Haeri <soroosh.haeri@me.com>
4   * @date 7/16/14
5   *
6   * @note This class is based on the POMCP implementation by David Silver and
7   *       Joel Veness.
8   * POMCP is published in NIPS 2010:
9   *      "Online Monte-Carlo Plannin/g in Large POMDPs".
10  *
11  * @copyright Copyright (c) 7/16/14                    SOROUSH HAERI
12  *     All Rights Reserved
13  *
14  *     Permission to use, copy, modify, and distribute this software and its
15  *     documentation for any purpose and without fee is hereby granted, provided
16  *     that the above copyright notice appear in all copies and that both that
17  *     copyright notice and this permission notice appear in supporting
18  *     documentation, and that the name of the author not be used in advertising
19  *     or publicity pertaining to distribution of the software without specific,
20  *     written prior permission.
21  *
22  *     THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
23  *     ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
24  *     AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
25  *     DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
26  *     AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
27  *     OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
28  **/
29
30 #include "mcts.h"
31 #include "core/config-manager.h"
32 #include "utilities/logger.h"
33
34 #include <math.h>
35 #include <algorithm>
36
37 using namespace std;
38
39 namespace vne {
40 namespace mcts {
```

```
41 MCTS::Parameters::Parameters()
42     : MaxDepth(ConfigManager::Instance()->getConfig<int>(
43           "MCTS.MCTSParameters.MaxDepth")),
44       NumSimulations(ConfigManager::Instance()->getConfig<int>(
45           "MCTS.MCTSParameters.NumSimulations")),
46       ExpandCount(ConfigManager::Instance()->getConfig<int>(
47           "MCTS.MCTSParameters.ExpandCount")),
48       AutoExploration(ConfigManager::Instance()->getConfig<bool>(
49           "MCTS.MCTSParameters.AutoExploration")),
50       ExplorationConstant(ConfigManager::Instance()->getConfig<double>(
51           "MCTS.MCTSParameters.ExplorationConstant")),
52       UseRave(ConfigManager::Instance()->getConfig<bool>(
53           "MCTS.MCTSParameters.UseRave")),
54       RaveDiscount(ConfigManager::Instance()->getConfig<double>(
55           "MCTS.MCTSParameters.RaveDiscount")),
56       RaveConstant(ConfigManager::Instance()->getConfig<double>(
57           "MCTS.MCTSParameters.RaveConstant")),
58       DisableTree(ConfigManager::Instance()->getConfig<bool>(
59           "MCTS.MCTSParameters.DisableTree")),
60       UseSinglePlayerMCTS(ConfigManager::Instance()->getConfig<bool>(
61           "MCTS.MCTSParameters.UseSinglePlayerMCTS")),
62       SPMCTSConstant(ConfigManager::Instance()->getConfig<double>(
63           "MCTS.MCTSParameters.SPMCTSConstant")),
64       ParalleizationType(ConfigManager::Instance()->getConfig<int>(
65           "MCTS.MCTSParameters.ParallelizationType")) {}
66 MCTS::MCTS(const std::shared_ptr<MCTSSimulator> sim)
67     : simulator(sim), params(Parameters()), treeDepth(0),
68       history(vector<int>()) {
69   if (params.AutoExploration) {
70     if (params.UseRave) {
71       params.ExplorationConstant = 0;
72     } else
73       params.ExplorationConstant = simulator->getRewardRange();
74   }
75   initFastUCB(params.ExplorationConstant);
76   TreeNode::NumChildren = simulator->getNumActions();
77   root = expandNode(simulator->createStartState());
78 }
79
80 MCTS::~MCTS() {}
81
82 bool MCTS::update(int action, double reward) {
83   BOOST_LOG_TRIVIAL(debug) << "Roots value: " << root->value.getValue()
84                            << std::endl;
85   history.push_back(action);
86   // Find matching vnode from the rest of the tree
87   std::shared_ptr<TreeNode> child_node = root->child(action);
88   BOOST_LOG_TRIVIAL(debug) << "Child's node value: "
89                            << child_node->value.getValue() << std::endl;
90
91   // Delete old tree and create new root
92   if (child_node->getState() == nullptr) {
93     const std::shared_ptr<State> st = root->getState()->getCopy();
94     double dummyReward;
95     bool terminal = simulator->step(st, action, dummyReward);
96     if (!terminal) {
97       *child_node = *(expandNode(st));
98     }
99   }
100  *root = *child_node;
101  return true;
102 }
103
104 int MCTS::selectAction() {
105   if (params.DisableTree)
106     rolloutSearch();
107   else {
```

```
108     UCTSearch();
109 #ifdef ENABLE_MPI
110     if (params.ParalleizationType == 0) {
111       int numChildren = (int)simulator->getNumActions();
112       int childrenCounts[numChildren];
113       double childrenValues[numChildren];
114       int childrenCountsGlobal[numChildren];
115       double childrenValuesGlobal[numChildren];
116 
117       for (int i = 0; i < numChildren; i++) {
118         childrenCounts[i] = root->child(i)->value.getCount();
119         childrenValues[i] = root->child(i)->value.getValue();
120       }
121 
122       MPI::Op customSumOp;
123       customSumOp.Init(&sumFunction, true);
124 
125       MPI::COMM_WORLD.Allreduce(childrenCounts, childrenCountsGlobal,
126                                 numChildren, MPI::INT, customSumOp);
127       MPI::COMM_WORLD.Allreduce(childrenValues, childrenValuesGlobal,
128                                 numChildren, MPI::DOUBLE, customSumOp);
129 
130           for (int i = 0; i < numChildren; i++) {
131         root->child(i)->value.set(childrenCountsGlobal[i],
132                                   childrenValuesGlobal[i] /
133                                       (double)childrenCountsGlobal[i]);
134       }
135     }
136 #endif
137   }
138 #ifdef ENABLE_MPI
139   // this is to make sure all processes are on the same page
140   // because sometimes UCB might return different values if there are more
141   // than one actions with maximum average value
142   if (params.ParalleizationType == 0) {
143     int action = -1;
144     int rank = MPI::COMM_WORLD.Get_rank();
145     if (rank == 0)
146       action = greedyUCB(root, false);
147     MPI::COMM_WORLD.Bcast(&action, 1, MPI::INT, 0);
148     assert(action != -1);
149     return action;
150   } else
151     return greedyUCB(root, false);
152 #else
153   return greedyUCB(root, false);
154 #endif
155 }
156 
157 void MCTS::rolloutSearch() {
158   int historyDepth = (int)history.size();
159   std::vector<int> legal;
160   assert(root->getState() != nullptr);
161   simulator->generateLegal(root->getState(), history, legal, status);
162   random_shuffle(legal.begin(), legal.end());
163 
164   for (int i = 0; i < params.NumSimulations; i++) {
165     int action = legal[i % legal.size()];
166     std::shared_ptr<State> st = root->getState()->getCopy();
167     simulator->validate(st);
168 
169     double immediateReward, delayedReward, totalReward;
170     bool terminal = simulator->step(st, action, immediateReward);
171 
172     std::shared_ptr<TreeNode> node = root->child(action);
173     if (node->getState() == nullptr && !terminal) {
174       *node = *(expandNode(st));
```

```
175       }
176
177       history.push_back(action);
178       delayedReward = rollout(st->getCopy());
179       totalReward = immediateReward + (simulator->getDiscount() * delayedReward);
180       root->child(action)->value.add(totalReward);
181       st.reset();
182       history.resize(historyDepth);
183     }
184   }
185
186   void MCTS::UCTSearch() {
187     clearStatistics();
188
189     int historyDepth = (int)history.size();
190
191     for (int n = 0; n < params.NumSimulations; n++) {
192       status.Phase = MCTSSimulator::Status::TREE;
193
194       treeDepth = 0;
195       peakTreeDepth = 0;
196
197       double totalReward = simulateNode(root);
198       root->value.add(totalReward);
199
200       // addRave(root, totalReward);
201       statTotalReward.Add(totalReward);
202       statTreeDepth.Add(peakTreeDepth);
203
204       history.resize(historyDepth);
205     }
206   }
207
208   double MCTS::simulateNode(std::shared_ptr<TreeNode> node) {
209     double immediateReward, delayedReward = 0;
210
211     int action = greedyUCB(node, true);
212
213     peakTreeDepth = treeDepth;
214
215     if (treeDepth >= params.MaxDepth) // search horizon reached
216       return 0;
217
218     std::shared_ptr<State> st = node->getState()->getCopy();
219     bool terminal = simulator->step(st, action, immediateReward);
220     history.push_back(action);
221     std::shared_ptr<TreeNode> child_node = node->child(action);
222
223     if (child_node->getState() == nullptr && !terminal &&
224         node->value.getCount() >= params.ExpandCount)
225       *child_node = *(expandNode(st));
226
227     if (!terminal) {
228       treeDepth++;
229       if (!(child_node->getState() == nullptr))
230         delayedReward = simulateNode(child_node);
231       else
232         delayedReward = rollout(st->getCopy());
233       treeDepth--;
234     }
235     double totalReward =
236         immediateReward + simulator->getDiscount() * delayedReward;
237
238     child_node->value.add(totalReward);
239     addRave(node, totalReward);
240     return totalReward;
241   }
```

```
242
243 void MCTS::addRave(std::shared_ptr<TreeNode> node, double totalReward) {
244   double totalDiscount = 1.0;
245   for (int t = treeDepth; t < history.size(); ++t) {
246     std::shared_ptr<TreeNode> child_node = node->child(history[t]);
247     child_node->AMAF.add(totalReward, totalDiscount);
248     totalDiscount *= params.RaveDiscount;
249   }
250 }
251
252 std::shared_ptr<TreeNode> MCTS::expandNode(const std::shared_ptr<State> state) {
253   std::shared_ptr<TreeNode> node(new TreeNode(state));
254   node->value.set(0, 0);
255   simulator->prior(node, history, status);
256   return node;
257 }
258
259 int MCTS::greedyUCB(std::shared_ptr<TreeNode> node, bool ucb) const {
260   static vector<int> besta;
261   besta.clear();
262   double bestq = -Infinity;
263   int N = node->value.getCount();
264   double logN = log(N + 1);
265
266   // these values will only change if partitioning is enabled;
267   int mystart = 0;
268   int myend = simulator->getNumActions();
269   for (int action = mystart; action < myend; action++) {
270     double q;
271     int n;
272
273     std::shared_ptr<TreeNode> child_node = node->child(action);
274     q = child_node->value.getValue();
275     n = child_node->value.getCount();
276
277     if (params.UseRave && child_node->AMAF.getCount() > 0) {
278       double n2 = child_node->AMAF.getCount();
279       double beta = n2 / (n + n2 + params.RaveConstant * n * n2);
280       q = (1.0 - beta) * q + beta * child_node->AMAF.getValue();
281     }
282
283     if (ucb)
284       q += fastUCB(N, n, logN);
285
286     if (params.UseSinglePlayerMCTS && child_node->value.getCount() > 0 &&
287         child_node->value.getValue() > -Infinity)
288       q += sqrt(
289           (child_node->value.getSumSquaredValue() -
290            n * child_node->value.getValue() * child_node->value.getValue() +
291            params.SPMCTSConstant) /
292           n);
293
294     if (q >= bestq) {
295       if (q > bestq)
296         besta.clear();
297       bestq = q;
298       besta.push_back(action);
299     }
300   }
301
302   assert(!besta.empty());
303   int randomIndex =
304       (int)gsl_rng_uniform_int(RNG::Instance()->getGeneralRNG(), besta.size());
305   return besta[randomIndex];
306 }
307
308 double MCTS::rollout(std::shared_ptr<State> state) {
```

```
309   status.Phase = MCTSSimulator::Status::ROLLOUT;
310
311   double totalReward = 0.0;
312   double discount = 1.0;
313   bool terminal = false;
314   int numSteps;
315   for (numSteps = 0; numSteps + treeDepth < params.MaxDepth && !terminal;
316        ++numSteps) {
317     double reward;
318
319     int action = simulator->selectRandom(state, history, status);
320     terminal = simulator->step(state, action, reward);
321
322     totalReward += reward * discount;
323     discount *= simulator->getDiscount();
324   }
325
326   statRolloutDepth.Add(numSteps);
327
328   return totalReward;
329 }
330
331 double MCTS::UCB[UCB_N][UCB_n];
332 bool MCTS::initialisedFastUCB = false;
333
334 void MCTS::initFastUCB(double exploration) {
335   for (int N = 0; N < UCB_N; ++N)
336     for (int n = 0; n < UCB_n; ++n)
337       if (n == 0)
338         UCB[N][n] = Infinity;
339       else
340         UCB[N][n] = exploration * sqrt(log(N + 1) / n);
341   initialisedFastUCB = true;
342 }
343
344 inline double MCTS::fastUCB(int N, int n, double logN) const {
345   if (initialisedFastUCB && N < UCB_N && n < UCB_n)
346     return UCB[N][n];
347
348   if (n == 0)
349     return Infinity;
350   else
351     return params.ExplorationConstant * sqrt(logN / n);
352 }
353
354 void MCTS::clearStatistics() {
355   statTreeDepth.Clear();
356   statRolloutDepth.Clear();
357   statTotalReward.Clear();
358 }
359 }
360 }
361 #if ENABLE_MPI
362 void sumFunction(const void *input, void *inoutput, int len,
363                  const MPI::Datatype &datatype) {
364   for (int i = 0; i < len; i++) {
365     if (datatype == MPI::INT) {
366
367       int *currentInPtr = (int *)input;
368       currentInPtr += i;
369       int *currentOutPtr = (int *)inoutput;
370       currentOutPtr += i;
371       if ((*currentInPtr < LargeInteger) && (*currentOutPtr < LargeInteger)) {
372         *(currentOutPtr) = *currentInPtr + (*currentOutPtr);
373       } else if (*currentInPtr < LargeInteger &&
374                  (*currentOutPtr >= LargeInteger)) {
375         *(currentOutPtr) = *(currentInPtr);
```

```
376       }
377     } else if (datatype == MPI::DOUBLE) {
378       double *currentInPtr = (double *)input;
379       currentInPtr += i;
380       double *currentOutPtr = (double *)inoutput;
381       currentOutPtr += i;
382       if ((*currentInPtr > -Infinity) && (*currentOutPtr > -Infinity)) {
383         *(currentOutPtr) = *currentInPtr + (*currentOutPtr);
384       } else if (*currentInPtr > -Infinity && (*currentOutPtr <= -Infinity)) {
385         *(currentOutPtr) = *(currentInPtr);
386       }
387     }
388   }
389 };
390 #endif
```

Listing B.8: tree-node.h: MCTS Tree Node Class Header File

```
1  /**
2   * @file tree-node.h
3   * @author Soroush Haeri <soroosh.haeri@me.com>
4   * @date 7/16/14
5   *
6   * @note This class is based on the POMCP implementation by David Silver and
7   *       Joel Veness.
8   * POMCP is published in NIPS 2010:
9   *      "Online Monte-Carlo Plannin/g in Large POMDPs".
10  *
11  * @copyright Copyright (c) 7/16/14                      SOROUSH HAERI
12  *     All Rights Reserved
13  *
14  *     Permission to use, copy, modify, and distribute this software and its
15  *     documentation for any purpose and without fee is hereby granted, provided
16  *     that the above copyright notice appear in all copies and that both that
17  *     copyright notice and this permission notice appear in supporting
18  *     documentation, and that the name of the author not be used in advertising
19  *     or publicity pertaining to distribution of the software without specific,
20  *     written prior permission.
21  *
22  *     THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
23  *     ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
24  *     AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
25  *     DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
26  *     AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
27  *     OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
28  **/
29
30 #ifndef MCTS_TREE_NODE_
31 #define MCTS_TREE_NODE_
32
33 #include <memory>
34 #include <vector>
35
36 #include "core/core-types.h"
37
38 #include <mcts/state.h>
39
40 namespace vne {
41 namespace mcts {
42 template <class COUNT> class Value {
43 public:
44   void set(COUNT count, double value) {
45     Count = count;
46     Total = value * count;
47     // Total = value;
```

```
48      SumSquaredValue = count * value * value;
49    }
50
51    void add(double totalReward) {
52      Count += 1.0;
53      Total += totalReward;
54      // if (Total == 0)
55      //    Total = totalReward;
56      // else if (totalReward > Total)
57      //    Total = totalReward;
58      SumSquaredValue += totalReward * totalReward;
59    }
60
61    void add(double totalReward, COUNT weight) {
62      Count += weight;
63      Total += totalReward * weight;
64      SumSquaredValue += totalReward * totalReward * weight;
65    }
66
67    double getValue() const {
68      return Count == 0 ? Total : Total / Count;
69      // return Total;
70    }
71
72    double getSumSquaredValue() const { return SumSquaredValue; }
73
74    COUNT getCount() const { return Count; }
75
76  private:
77    COUNT Count;
78    double Total;
79    long double SumSquaredValue;
80  };
81
82  class TreeNode {
83  public:
84    Value<int> value;
85    Value<double> AMAF;
86
87    void setChildren(int count, double value);
88    std::shared_ptr<TreeNode> child(int c);
89
90    const std::shared_ptr<State> getState() const;
91
92    static int NumChildren;
93
94    ~TreeNode();
95    TreeNode(std::shared_ptr<State> st);
96
97  protected:
98    TreeNode();
99    void Initialize();
100   std::shared_ptr<State> state;
101   std::vector<std::shared_ptr<TreeNode>> Children;
102 };
103 }
104 }
105 #endif
```

Listing B.9: tree-node.cc: MCTS Tree Node Source Header File

```
1  /**
2   * @file tree-node.cc
3   * @author Soroush Haeri <soroosh.haeri@me.com>
4   * @date 7/16/14
```

```
 *
 * @note This class is based on the POMCP implementation by David Silver and
 *       Joel Veness.
 * POMCP is published in NIPS 2010:
 *      "Online Monte-Carlo Plannin/g in Large POMDPs".
 *
 * @copyright Copyright (c) 7/16/14                       SOROUSH HAERI
 *     All Rights Reserved
 *
 *     Permission to use, copy, modify, and distribute this software and its
 *     documentation for any purpose and without fee is hereby granted, provided
 *     that the above copyright notice appear in all copies and that both that
 *     copyright notice and this permission notice appear in supporting
 *     documentation, and that the name of the author not be used in advertising
 *     or publicity pertaining to distribution of the software without specific,
 *     written prior permission.
 *
 *     THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING
 *     ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
 *     AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
 *     DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
 *     AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 *     OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 **/
#include "tree-node.h"

#include <assert.h>
#include <iostream>

namespace vne {
namespace mcts {
int TreeNode::NumChildren = 0;
TreeNode::TreeNode(std::shared_ptr<State> st) : state(std::move(st)) {
  assert(NumChildren);
  Children.resize(TreeNode::NumChildren);
  Initialize();
}

TreeNode::TreeNode() : state(nullptr) {
  assert(NumChildren);
  Children.resize(TreeNode::NumChildren);
}

void TreeNode::Initialize() {
  for (int action = 0; action < Children.size(); action++) {
    Children[action] = std::shared_ptr<TreeNode>(new TreeNode());
  }
}

TreeNode::~TreeNode() {}

const std::shared_ptr<State> TreeNode::getState() const { return state; }

void TreeNode::setChildren(int count, double value) {
  for (int action = 0; action < NumChildren; action++) {
    std::shared_ptr<TreeNode> node = Children[action];
    node->value.set(count, value);
    node->AMAF.set(count, value);
  }
}
std::shared_ptr<TreeNode> TreeNode::child(int c) { return Children[c]; }
}
}
```

Listing B.10: mcts-simulator.h: MCTS Simulator Base Class Header File

```
/**
 * @file mcts-simulator.h
 * @author Soroush Haeri <soroosh.haeri@me.com>
 * @date 7/16/14
 *
 * @note This class is based on the POMCP implementation by David Silver and
 *       Joel Veness.
 * POMCP is published in NIPS 2010:
 *      "Online Monte-Carlo Plannin/g in Large POMDPs".
 *
 * @copyright Copyright (c) 7/16/14                        SOROUSH HAERI
 *     All Rights Reserved
 *
 *     Permission to use, copy, modify, and distribute this software and its
 *     documentation for any purpose and without fee is hereby granted, provided
 *     that the above copyright notice appear in all copies and that both that
 *     copyright notice and this permission notice appear in supporting
 *     documentation, and that the name of the author not be used in advertising
 *     or publicity pertaining to distribution of the software without specific,
 *     written prior permission.
 *
 *     THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
 *     ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
 *     AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
 *     DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
 *     AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 *     OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 **/

#ifndef MCTS_MCTS_SIMULATOR_
#define MCTS_MCTS_SIMULATOR_

#include "mcts/tree-node.h"
#include "mcts/state.h"

#include "core/rng.h"

#include "core/network.h"
#include "core/substrate-link.h"
#include "core/substrate-node.h"
#include "core/virtual-network-request.h"

#include "core/core-types.h"

namespace vne {
namespace mcts {
class MCTSSimulator : public RNGSubscriber {
public:
  struct Knowledge {
    enum { PURE, LEGAL, SMART, NUM_LEVELS };

    Knowledge();

    int RolloutLevel;
    int TreeLevel;
    int SmartTreeCount;
    double SmartTreeValue;

    int Level(int phase) const {
      assert(phase < Status::NUM_PHASES);
      if (phase == Status::TREE)
        return TreeLevel;
      else
        return RolloutLevel;
    }
```

```
66    };
67
68    struct Status {
69      Status();
70
71      enum { TREE, ROLLOUT, NUM_PHASES };
72
73      int Phase;
74    };
75
76    virtual std::shared_ptr<State> createStartState() const = 0;
77
78    // Update state according to action, and get observation and reward.
79    // Return value of true indicates termination of episode (if episodic)
80    // Action is the ID of the substrate node that will be used to host the
81    // current VNR node.
82    virtual bool step(std::shared_ptr<State> state, int action,
83                      double &reward) const = 0;
84
85    // Sanity check
86    virtual bool validate(const std::shared_ptr<State> state) const {
87      return true;
88    };
89
90    // Modify state stochastically to some related state
91    virtual bool localMove(std::shared_ptr<State> state,
92                           const std::vector<int> &history,
93                           const Status &status) const {
94      return true;
95    };
96
97    // Use domain knowledge to assign prior value and confidence to actions
98    // Should only use fully observable state variables
99    void prior(std::shared_ptr<TreeNode> node, const std::vector<int> &history,
100              const Status &status) const;
101
102   // Use domain knowledge to select actions stochastically during rollouts
103   int selectRandom(const std::shared_ptr<State> state,
104                    const std::vector<int> &history, const Status &status) const;
105
106   // Generate set of legal actions
107   virtual void generateLegal(const std::shared_ptr<State> state,
108                              const std::vector<int> &history,
109                              std::vector<int> &actions,
110                              const Status &status) const;
111
112   // Generate set of preferred actions
113   virtual void generatePreferred(const std::shared_ptr<State> state,
114                                  const std::vector<int> &history,
115                                  std::vector<int> &actions,
116                                  const Status &status) const;
117
118   virtual ~MCTSSimulator();
119
120   int getNumActions() { return numActions; };
121   double getDiscount() const { return discount; }
122   double getRewardRange() const { return rewardRange; }
123   double getHorizon(double accuracy, int undiscountedHorizon = 100) const;
124
125 protected:
126   MCTSSimulator();
127   MCTSSimulator(int numActions);
128
129   int numActions;
130   Knowledge knowledge;
131   double discount, rewardRange;
132   inline int Random(int max) const;
```

```
};
}
}
#endif
```

Listing B.11: mcts-simulator.cc: MCTS Simulator Base Class Source File

```
/**
 * @file mcts-simulator.cc
 * @author Soroush Haeri <soroosh.haeri@me.com>
 * @date 7/16/14
 *
 * @note This class is based on the POMCP implementation by David Silver and
 *       Joel Veness.
 * POMCP is published in NIPS 2010:
 *      "Online Monte-Carlo Plannin/g in Large POMDPs".
 *
 * @copyright Copyright (c) 7/16/14                      SOROUSH HAERI
 *     All Rights Reserved
 *
 *     Permission to use, copy, modify, and distribute this software and its
 *     documentation for any purpose and without fee is hereby granted, provided
 *     that the above copyright notice appear in all copies and that both that
 *     copyright notice and this permission notice appear in supporting
 *     documentation, and that the name of the author not be used in advertising
 *     or publicity pertaining to distribution of the software without specific,
 *     written prior permission.
 *
 *     THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
 *     ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
 *     AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
 *     DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
 *     AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 *     OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 **/

#include "mcts-simulator.h"
#include "core/config-manager.h"

#include <math.h>

namespace vne {
namespace mcts {

MCTSSimulator::Knowledge::Knowledge()
    :

      TreeLevel(ConfigManager::Instance()->getConfig<int>(
          "MCTS.Simulator.Knowledge.TreeLevel")),
      RolloutLevel(ConfigManager::Instance()->getConfig<int>(
          "MCTS.Simulator.Knowledge.RolloutLevel")),
      SmartTreeCount(ConfigManager::Instance()->getConfig<int>(
          "MCTS.Simulator.Knowledge.SmartTreeCount")),
      SmartTreeValue(ConfigManager::Instance()->getConfig<double>(
          "MCTS.Simulator.Knowledge.SmartTreeValue")) {}

MCTSSimulator::Status::Status()
    :

      Phase(TREE) {}

MCTSSimulator::MCTSSimulator()
    :

      knowledge(Knowledge()),
```

```
59        discount(ConfigManager::Instance()->getConfig<double>(
60            "MCTS.Simulator.discount")),
61        rewardRange(ConfigManager::Instance()->getConfig<double>(
62            "MCTS.Simulator.rewardRange")) {
63    assert(discount > 0 && discount <= 1);
64  }
65
66  MCTSSimulator::MCTSSimulator(int _numActions)
67      :
68
69        numActions(_numActions),
70        knowledge(Knowledge()),
71        discount(ConfigManager::Instance()->getConfig<double>(
72            "MCTS.Simulator.discount")),
73        rewardRange(ConfigManager::Instance()->getConfig<double>(
74            "MCTS.Simulator.rewardRange")) {
75    assert(discount > 0 && discount <= 1);
76  }
77
78  MCTSSimulator::~MCTSSimulator() {}
79
80  inline int MCTSSimulator::Random(int max) const {
81    return (int)gsl_rng_uniform_int(RNG::Instance()->getGeneralRNG(), max);
82  }
83
84  void MCTSSimulator::generateLegal(const std::shared_ptr<State> state,
85                                    const std::vector<int> &history,
86                                    std::vector<int> &actions,
87                                    const Status &status) const {
88    for (int a = 0; a < numActions; ++a)
89      actions.push_back(a);
90  }
91
92  void MCTSSimulator::generatePreferred(const std::shared_ptr<State> state,
93                                        const std::vector<int> &history,
94                                        std::vector<int> &actions,
95                                        const Status &status) const {}
96
97  int MCTSSimulator::selectRandom(const std::shared_ptr<State> state,
98                                  const std::vector<int> &history,
99                                  const Status &status) const {
100   static std::vector<int> actions;
101
102   if (knowledge.RolloutLevel >= Knowledge::SMART) {
103     actions.clear();
104     generatePreferred(state, history, actions, status);
105     if (!actions.empty())
106       return actions[Random((int)actions.size())];
107   }
108
109   if (knowledge.RolloutLevel >= Knowledge::LEGAL) {
110     actions.clear();
111     generateLegal(state, history, actions, status);
112     if (!actions.empty())
113       return actions[Random((int)actions.size())];
114   }
115
116   return Random(numActions);
117 }
118
119 void MCTSSimulator::prior(std::shared_ptr<TreeNode> node,
120                           const std::vector<int> &history,
121                           const Status &status) const {
122   static std::vector<int> actions;
123   if (knowledge.TreeLevel == Knowledge::PURE || node->getState() == nullptr) {
124     node->setChildren(0, 0);
125     return;
```

```
126     } else {
127       node->setChildren(+LargeInteger, -Infinity);
128     }
129
130     if (knowledge.TreeLevel >= Knowledge::LEGAL) {
131       actions.clear();
132       generateLegal(node->getState(), history, actions, status);
133
134       for (auto i_action = actions.begin(); i_action != actions.end();
135            ++i_action) {
136         int a = *i_action;
137         std::shared_ptr<TreeNode> node_child = node->child(a);
138         node_child->value.set(0, 0);
139         node_child->AMAF.set(0, 0);
140       }
141     }
142
143     if (knowledge.TreeLevel >= Knowledge::SMART) {
144       actions.clear();
145       generatePreferred(node->getState(), history, actions, status);
146
147       for (auto i_action = actions.begin(); i_action != actions.end();
148            ++i_action) {
149         int a = *i_action;
150         std::shared_ptr<TreeNode> node_child = node.get()->child(a);
151         node_child->value.set(knowledge.SmartTreeCount, knowledge.SmartTreeValue);
152         node_child->AMAF.set(knowledge.SmartTreeCount, knowledge.SmartTreeValue);
153       }
154     }
155   }
156
157   double MCTSSimulator::getHorizon(double accuracy,
158                                    int undiscountedHorizon) const {
159     if (discount == 1)
160       return undiscountedHorizon;
161     return log(accuracy) / log(discount);
162   }
163   }
164   }
```

Listing B.12: vne-mcts-simulator.h: General VNE Simulator Base Class Source File

```
1   /**
2    * @file vne-mcts-simulator.h
3    * @author Soroush Haeri <soroosh.haeri@me.com>
4    * @date 7/16/14
5    *
6    * @copyright Copyright (c) 7/16/14                    SOROUSH HAERI
7    *      All Rights Reserved
8    *
9    *      Permission to use, copy, modify, and distribute this software and its
10   *      documentation for any purpose and without fee is hereby granted, provided
11   *      that the above copyright notice appear in all copies and that both that
12   *      copyright notice and this permission notice appear in supporting
13   *      documentation, and that the name of the author not be used in advertising or
14   *      publicity pertaining to distribution of the software without specific,
15   *      written prior permission.
16   *
17   *      THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
18   *      ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
19   *      AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
20   *      DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
21   *      AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
22   *      OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
23   **/
```

```
24
25 #ifndef MCVNE_VNE_MCTS_SIMULATOR_
26 #define MCVNE_VNE_MCTS_SIMULATOR_
27
28 #include "mcts/tree-node.h"
29 #include "mcts/mcts-simulator.h"
30 #include "mcvne/vne-nm-state.h"
31
32 #include "core/rng.h"
33
34 #include "core/network.h"
35 #include "core/substrate-link.h"
36 #include "core/substrate-node.h"
37 #include "core/virtual-network-request.h"
38 #include "core/config-manager.h"
39 #include "core/link-embedding-algorithm.h"
40
41 using namespace vne::mcts;
42
43 namespace vne {
44 namespace mcvne {
45
46 template <typename, typename> class VNEMCTSSimulator;
47
48 template <typename... SNODERES, template <typename...> class SNODECLASS,
49           typename... SLINKRES, template <typename...> class SLINKCLASS,
50           typename... VNODERES, template <typename...> class VNODECLASS,
51           typename... VLINKRES, template <typename...> class VLINKCLASS,
52           template <typename> class VNRCLASS>
53 class VNEMCTSSimulator<
54     Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES...>>,
55     VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>>
56     : public MCTSSimulator {
57   static_assert(std::is_base_of<SubstrateNode<SNODERES...>,
58                                 SNODECLASS<SNODERES...>>::value,
59                 "Template arguments are not correctly set.");
60   static_assert(
61       std::is_base_of<VirtualNode<VNODERES...>, VNODECLASS<VNODERES...>>::value,
62       "Template arguments are not correctly set.");
63   static_assert(std::is_base_of<SubstrateLink<SLINKRES...>,
64                                 SLINKCLASS<SLINKRES...>>::value,
65                 "Template arguments are not correctly set.");
66   static_assert(
67       std::is_base_of<VirtualLink<VLINKRES...>, VLINKCLASS<VLINKRES...>>::value,
68       "Template arguments are not correctly set.");
69   static_assert(
70       std::is_base_of<VirtualNetworkRequest<Network<VNODECLASS<VNODERES...>,
71                                                     VLINKCLASS<VLINKRES...>>>,
72                       VNRCLASS<Network<VNODECLASS<VNODERES...>,
73                                        VLINKCLASS<VLINKRES...>>>>::value,
74       "Template arguments are not correctly set.");
75
76 public:
77   typedef VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>
78       VNR_TYPE;
79   typedef Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES...>>
80       SUBSTRATE_TYPE;
81
82   virtual ~VNEMCTSSimulator(){};
83
84   virtual std::shared_ptr<State> createStartState() const override;
85
86   // Update state according to action, and get observation and reward.
87   // Return value of true indicates termination of episode (if episodic)
88   // Action is the ID of the substrate node that will be used to host the
89   // current VNR node.
90   virtual bool step(std::shared_ptr<State> state, int action,
```

```
                    double &reward) const override;

  // Sanity check
  virtual bool validate(const std::shared_ptr<State> state) const override;

  // Generate set of legal actions
  virtual void generateLegal(const std::shared_ptr<State> state,
                             const std::vector<int> &history,
                             std::vector<int> &actions,
                             const Status &status) const override;

  // Generate set of preferred actions
  virtual void generatePreferred(const std::shared_ptr<State> state,
                                 const std::vector<int> &history,
                                 std::vector<int> &actions,
                                 const Status &status) const override;

protected:
  VNEMCTSSimulator(
      std::shared_ptr<SUBSTRATE_TYPE> subs_net, std::shared_ptr<VNR_TYPE> vnr,
      std::shared_ptr<LinkEmbeddingAlgorithm<SUBSTRATE_TYPE, VNR_TYPE>>
          _link_embedder);

  virtual std::shared_ptr<std::set<int>> getValidSubstrateNodeIdSetForVNNodeId(
      int vn_id, const std::shared_ptr<std::set<int>> used_sn_ids) const = 0;
  virtual double calculateImmediateReward(std::shared_ptr<VNENMState> st,
                                          int action) const = 0;
  virtual double calculateFinalReward(
      std::shared_ptr<VNENMState> st,
      const std::map<
          int,
          std::list<std::pair<int, std::shared_ptr<Resources<SLINKRES...>>>>>
          *linkMap) const = 0;
  std::shared_ptr<LinkEmbeddingAlgorithm<SUBSTRATE_TYPE, VNR_TYPE>>
      link_embedder;
  std::shared_ptr<SUBSTRATE_TYPE> substrate_net;
  std::shared_ptr<VNR_TYPE> vnr;

  bool setAlpha;
  bool setBeta;

private:
  bool isActionLegal(int action, std::shared_ptr<VNENMState> st) const;
};

template <typename... SNODERES, template <typename...> class SNODECLASS,
          typename... SLINKRES, template <typename...> class SLINKCLASS,
          typename... VNODERES, template <typename...> class VNODECLASS,
          typename... VLINKRES, template <typename...> class VLINKCLASS,
          template <typename> class VNRCLASS>
inline bool VNEMCTSSimulator<
    Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES...>>,
    VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>>::
    isActionLegal(int action, std::shared_ptr<VNENMState> st) const {
  std::shared_ptr<std::set<int>> validActionSet =
      getValidSubstrateNodeIdSetForVNNodeId(st->getCurrentVNId(),
                                            st->getUsedSNIds());
  bool ret = validActionSet->find(action) != validActionSet->end();
  return ret;
}

template <typename... SNODERES, template <typename...> class SNODECLASS,
          typename... SLINKRES, template <typename...> class SLINKCLASS,
          typename... VNODERES, template <typename...> class VNODECLASS,
          typename... VLINKRES, template <typename...> class VLINKCLASS,
          template <typename> class VNRCLASS>
VNEMCTSSimulator<
```

```
158     Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES...>>,
159     VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>>::
160     VNEMCTSSimulator(
161         std::shared_ptr<VNEMCTSSimulator::SUBSTRATE_TYPE> _substrate_net,
162         std::shared_ptr<VNEMCTSSimulator::VNR_TYPE> _vnr,
163         std::shared_ptr<LinkEmbeddingAlgorithm<SUBSTRATE_TYPE, VNR_TYPE>>
164             _link_embedder)
165     : MCTSSimulator(_substrate_net->getNumNodes()),
166       substrate_net(_substrate_net), vnr(_vnr), link_embedder(_link_embedder),
167       setAlpha(ConfigManager::Instance()->getConfig<bool>(
168           "MCVNE.VNEMCTSSimulator.setAlpha")),
169       setBeta(ConfigManager::Instance()->getConfig<bool>(
170           "MCVNE.VNEMCTSSimulator.setBeta")) {}
171
172 template <typename... SNODERES, template <typename...> class SNODECLASS,
173           typename... SLINKRES, template <typename...> class SLINKCLASS,
174           typename... VNODERES, template <typename...> class VNODECLASS,
175           typename... VLINKRES, template <typename...> class VLINKCLASS,
176           template <typename> class VNRCLASS>
177 std::shared_ptr<State> VNEMCTSSimulator<
178     Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES...>>,
179     VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>>::
180     createStartState() const {
181   const std::shared_ptr<std::set<int>> VNNodeIdSet =
182       vnr->getVN()->getNodeIdSet();
183   std::shared_ptr<VNENMState> st(
184       new VNENMState(VNNodeIdSet, vnr->getId(), vnr->getNodeMap()));
185   return st;
186 }
187
188 template <typename... SNODERES, template <typename...> class SNODECLASS,
189           typename... SLINKRES, template <typename...> class SLINKCLASS,
190           typename... VNODERES, template <typename...> class VNODECLASS,
191           typename... VLINKRES, template <typename...> class VLINKCLASS,
192           template <typename> class VNRCLASS>
193 bool VNEMCTSSimulator<
194     Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES...>>,
195     VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>>::
196     step(std::shared_ptr<State> state, int action, double &reward) const {
197   std::shared_ptr<VNENMState> st = std::static_pointer_cast<VNENMState>(state);
198
199   if (!isActionLegal(action, st)) {
200     reward = -Infinity;
201     return true;
202   }
203
204   st->addNodeMapping(action);
205
206   if (st->isTreminal()) {
207     std::map<int,
208              std::list<std::pair<int, std::shared_ptr<Resources<SLINKRES...>>>>>
209         linkMap;
210     Embedding_Result result = link_embedder->embeddVNRLinksForIdSets(
211         substrate_net, vnr, st->getNodeMap(), &linkMap);
212     if (result != Embedding_Result::SUCCESSFUL_EMBEDDING)
213       reward = -Infinity;
214     else
215       reward = calculateFinalReward(st, &linkMap);
216     return true;
217   } else
218     reward += calculateImmediateReward(st, action);
219   return false;
220 }
221
222 template <typename... SNODERES, template <typename...> class SNODECLASS,
223           typename... SLINKRES, template <typename...> class SLINKCLASS,
224           typename... VNODERES, template <typename...> class VNODECLASS,
```

```
225           typename... VLINKRES, template <typename...> class VLINKCLASS,
226           template <typename> class VNRCLASS>
227 bool VNEMCTSSimulator<
228     Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES...>>,
229     VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>>::
230     validate(const std::shared_ptr<State> state) const {
231   std::shared_ptr<VNENMState> st = std::static_pointer_cast<VNENMState>(state);
232   return (vnr->getId() == st->getVNRId());
233 }
234
235 template <typename... SNODERES, template <typename...> class SNODECLASS,
236           typename... SLINKRES, template <typename...> class SLINKCLASS,
237           typename... VNODERES, template <typename...> class VNODECLASS,
238           typename... VLINKRES, template <typename...> class VLINKCLASS,
239           template <typename> class VNRCLASS>
240 void VNEMCTSSimulator<
241     Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES...>>,
242     VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>>::
243     generateLegal(const std::shared_ptr<State> state,
244                   const std::vector<int> &history, std::vector<int> &actions,
245                   const Status &status) const {
246   std::shared_ptr<VNENMState> st = std::static_pointer_cast<VNENMState>(state);
247   std::shared_ptr<std::set<int>> validActionSet =
248       getValidSubstrateNodeIdSetForVNNodeId(st->getCurrentVNId(),
249                                             st->getUsedSNIds());
250   actions = std::vector<int>(validActionSet->begin(), validActionSet->end());
251 }
252
253 template <typename... SNODERES, template <typename...> class SNODECLASS,
254           typename... SLINKRES, template <typename...> class SLINKCLASS,
255           typename... VNODERES, template <typename...> class VNODECLASS,
256           typename... VLINKRES, template <typename...> class VLINKCLASS,
257           template <typename> class VNRCLASS>
258 void VNEMCTSSimulator<
259     Network<SNODECLASS<SNODERES...>, SLINKCLASS<SLINKRES...>>,
260     VNRCLASS<Network<VNODECLASS<VNODERES...>, VLINKCLASS<VLINKRES...>>>>::
261     generatePreferred(const std::shared_ptr<State> state,
262                       const std::vector<int> &history,
263                       std::vector<int> &actions, const Status &status) const {
264   generateLegal(state, history, actions, status);
265 }
266 }
267 }
268 #endif
```

Listing B.13: mcvne-simulator.h: Our VNE Simulator Implementation Header File

```
1  /**
2   * @file mcvne-simulator.h
3   * @author Soroush Haeri <soroosh.haeri@me.com>
4   * @date 7/16/14
5   *
6   * @copyright Copyright (c) 7/16/14                        SOROUSH HAERI
7   *     All Rights Reserved
8   *
9   *     Permission to use, copy, modify, and distribute this software and its
10  *     documentation for any purpose and without fee is hereby granted, provided
11  *     that the above copyright notice appear in all copies and that both that
12  *     copyright notice and this permission notice appear in supporting
13  *     documentation, and that the name of the author not be used in advertising or
14  *     publicity pertaining to distribution of the software without specific,
15  *     written prior permission.
16  *
17  *     THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
18  *     ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
```

```
 *     AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
 *     DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
 *     AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 *     OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 **/

#ifndef MCVNE_MCVNE_SIMULATOR_
#define MCVNE_MCVNE_SIMULATOR_

#include "mcvne/vne-mcts-simulator.h"
#include "core/node-embedding-algorithm.h"

#include "Vineyard/vy-substrate-link.h"
#include "Vineyard/vy-substrate-node.h"
#include "Vineyard/vy-virtual-net-request.h"

#include "core/network.h"

using namespace vne::vineyard;

namespace vne {
namespace mcvne {
template <typename = Network<VYSubstrateNode<>, VYSubstrateLink<>>,
          typename = VYVirtualNetRequest<>>
class MCVNESimulator
    : public VNEMCTSSimulator<Network<VYSubstrateNode<>, VYSubstrateLink<>>,
                              VYVirtualNetRequest<>> {
public:
  MCVNESimulator(
      std::shared_ptr<SUBSTRATE_TYPE> subs_net, std::shared_ptr<VNR_TYPE> vnr,
      std::shared_ptr<LinkEmbeddingAlgorithm<SUBSTRATE_TYPE, VNR_TYPE>>
          _link_embedder);

  ~MCVNESimulator();

protected:
  virtual std::shared_ptr<std::set<int>> getValidSubstrateNodeIdSetForVNNodeId(
      int vn_id, std::shared_ptr<std::set<int>> used_sn_ids) const override;
  virtual double calculateImmediateReward(std::shared_ptr<VNENMState> st,
                                          int action) const override;
  virtual double calculateFinalReward(
      std::shared_ptr<VNENMState> st,
      const std::map<
          int, std::list<std::pair<int, std::shared_ptr<Resources<double>>>>>
          *linkMap) const override;

private:
  struct ReachabilityConditionWithPathSpliting {
    bool operator()(const std::shared_ptr<const VYSubstrateNode<>> lhs,
                    const std::shared_ptr<
                        const std::vector<std::shared_ptr<VYSubstrateLink<>>>>
                        linksConnectedToLhs,
                    const std::shared_ptr<const VYVirtualNode<>> rhs,
                    const std::shared_ptr<const std::vector<
                        std::shared_ptr<VYVirtualLink<>>>> linksConnectedToRhs,
                    double maxD,
                    std::shared_ptr<std::set<int>> used_sn_ids) const {
      double sum_sn_link_bw = 0.0;
      double sum_vn_link_bw = 0.0;
      for (auto it = linksConnectedToLhs->begin();
           it != linksConnectedToLhs->end(); it++) {
        sum_sn_link_bw += (*it)->getBandwidth();
      }
      for (auto it = linksConnectedToRhs->begin();
           it != linksConnectedToRhs->end(); it++) {
        sum_vn_link_bw += (*it)->getBandwidth();
      }
```

```
86        if (NodeEmbeddingAlgorithm<
87                Network<VYSubstrateNode<>, VYSubstrateLink<>>,
88                VYVirtualNetRequest<>>::IgnoreLocationConstrain())
89          return (sum_sn_link_bw >= sum_vn_link_bw &&
90                  used_sn_ids->find(lhs->getId()) == used_sn_ids->end() &&
91                  lhs->getCPU() >= rhs->getCPU());
92        return (sum_sn_link_bw >= sum_vn_link_bw &&
93                used_sn_ids->find(lhs->getId()) == used_sn_ids->end() &&
94                lhs->getCoordinates().distanceFrom(rhs->getCoordinates()) <=
95                    maxD &&
96                lhs->getCPU() >= rhs->getCPU());
97      }
98    };
99    struct ReachabilityConditionNoPathSpliting {
100     bool operator()(const std::shared_ptr<const VYSubstrateNode<>> lhs,
101                     const std::shared_ptr<
102                         const std::vector<std::shared_ptr<VYSubstrateLink<>>>>
103                         linksConnectedToLhs,
104                     const std::shared_ptr<const VYVirtualNode<>> rhs,
105                     const std::shared_ptr<const std::vector<
106                         std::shared_ptr<VYVirtualLink<>>>> linksConnectedToRhs,
107                     double maxD,
108                     std::shared_ptr<std::set<int>> used_sn_ids) const {
109       std::map<int, double> usedSLBW;
110       for (auto itr = linksConnectedToRhs->begin();
111            itr != linksConnectedToRhs->end(); itr++) {
112         int count = 0;
113         for (auto itl = linksConnectedToLhs->begin();
114              itl != linksConnectedToLhs->end(); itl++) {
115           if ((*itr)->getBandwidth() <=
116               ((*itl)->getBandwidth() - usedSLBW[(*itl)->getId()])) {
117             count++;
118             usedSLBW[(*itl)->getId()] += (*itr)->getBandwidth();
119             break;
120           }
121         }
122         if (count == 0)
123           return false;
124       }
125       if (NodeEmbeddingAlgorithm<
126               Network<VYSubstrateNode<>, VYSubstrateLink<>>,
127               VYVirtualNetRequest<>>::IgnoreLocationConstrain())
128         return (used_sn_ids->find(lhs->getId()) == used_sn_ids->end() &&
129                 lhs->getCPU() >= rhs->getCPU());
130       return (used_sn_ids->find(lhs->getId()) == used_sn_ids->end() &&
131               lhs->getCoordinates().distanceFrom(rhs->getCoordinates()) <=
132                   maxD &&
133               lhs->getCPU() >= rhs->getCPU());
134     }
135   };
136 };
137 }
138 }
139 #endif /* defined(__vne_mcts__mcvne_simulator__) */
```

Listing B.14: mcvne-simulator.h: Our VNE Simulator Implementation Header File

```
1  /**
2   * @file mcvne-simulator.cc
3   * @author Soroush Haeri <soroosh.haeri@me.com>
4   * @date 7/16/14
5   *
6   * @copyright Copyright (c) 7/16/14                    SOROUSH HAERI
7   *     All Rights Reserved
8   *
```

```
 *     Permission to use, copy, modify, and distribute this software and its
 *     documentation for any purpose and without fee is hereby granted, provided
 *     that the above copyright notice appear in all copies and that both that
 *     copyright notice and this permission notice appear in supporting
 *     documentation, and that the name of the author not be used in advertising or
 *     publicity pertaining to distribution of the software without specific,
 *     written prior permission.
 *
 *     THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
 *     ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
 *     AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
 *     DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
 *     AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 *     OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 **/

#include "mcvne-simulator.h"

namespace vne {
namespace mcvne {
template <>
MCVNESimulator<>::MCVNESimulator(
    std::shared_ptr<SUBSTRATE_TYPE> subs_net, std::shared_ptr<VNR_TYPE> vnr,
    std::shared_ptr<LinkEmbeddingAlgorithm<SUBSTRATE_TYPE, VNR_TYPE>>
        _link_embedder)
    : VNEMCTSSimulator<Network<VYSubstrateNode<>, VYSubstrateLink<>>,
                       VYVirtualNetRequest<>>(subs_net, vnr, _link_embedder) {}

template <> MCVNESimulator<>::~MCVNESimulator() {}

template <>
std::shared_ptr<std::set<int>>
MCVNESimulator<>::getValidSubstrateNodeIdSetForVNNodeId(
    int vn_id, std::shared_ptr<std::set<int>> used_sn_ids) const {
  double distance = vnr->getMaxDistance();
  if (link_embedder->getType() ==
      Link_Embedding_Algo_Types::WITH_PATH_SPLITTING)
    return substrate_net
        ->getNodesIDsWithConditions<ReachabilityConditionWithPathSpliting>(
            vnr->getVN()->getNode(vn_id),
            vnr->getVN()->getLinksForNodeId(vn_id), distance, used_sn_ids);
  return substrate_net
      ->getNodesIDsWithConditions<ReachabilityConditionNoPathSpliting>(
          vnr->getVN()->getNode(vn_id), vnr->getVN()->getLinksForNodeId(vn_id),
          distance, used_sn_ids);
}

template <>
double
MCVNESimulator<>::calculateImmediateReward(std::shared_ptr<VNENMState> st,
                                           int action) const {
  return 0;
}

template <>
double MCVNESimulator<>::calculateFinalReward(
    std::shared_ptr<VNENMState> st,
    const std::map<
        int, std::list<std::pair<int, std::shared_ptr<Resources<double>>>>>
        *linkMap) const {

  double revenue = 0.0;
  const std::shared_ptr<std::vector<std::shared_ptr<VYVirtualNode<>>>>
      vnr_node_vec = vnr->getVN()->getAllNodes();
  const std::shared_ptr<std::vector<std::shared_ptr<VYVirtualLink<>>>>
      vnr_link_vec = vnr->getVN()->getAllLinks();
  for (int i = 0; i < vnr_node_vec->size(); i++) {
```

```
76      revenue += vnr_node_vec->at(i)->getCPU();
77    }
78    for (int i = 0; i < vnr_link_vec->size(); i++) {
79      revenue += vnr_link_vec->at(i)->getBandwidth();
80    }
81    BOOST_LOG_TRIVIAL(debug)
82        << " ============= in calculateFinalReward =================="
83        << std::endl;
84    double cost = 0.0;
85    int count = 0;
86    for (auto it = linkMap->begin(); it != linkMap->end(); it++) {
87      for (auto it2 = it->second.begin(); it2 != it->second.end(); it2++) {
88        double flow_bw = std::get<0>(*(it2->second));
89        cost += flow_bw;
90        BOOST_LOG_TRIVIAL(debug) << "vnr link ID: " << count
91                                 << " SUB BW: " << std::get<0>(*it2->second)
92                                 << " using substrate link ID: " << it2->first
93                                 << std::endl;
94      }
95      count++;
96    }
97    BOOST_LOG_TRIVIAL(debug) << "total link cost: " << cost << std::endl;
98
99    for (auto it = st->getNodeMap()->begin(); it != st->getNodeMap()->end();
100        it++) {
101     double vn_cpu = vnr->getVN()->getNode(it->first)->getCPU();
102     cost += vn_cpu;
103   }
104   BOOST_LOG_TRIVIAL(debug) << "total cost: " << cost << std::endl;
105   BOOST_LOG_TRIVIAL(debug) << "total reward: " << revenue - cost << std::endl;
106   return (1000 + revenue - cost);
107 }
108 }
109 }
```

Listing B.15: state.h: MCTS Tree Node State Base Class

```
1  /**
2   * @file state.h
3   * @author Soroush Haeri <soroosh.haeri@me.com>
4   * @date 7/16/14
5   *
6   * @copyright Copyright (c) 7/16/14                        SOROUSH HAERI
7   *      All Rights Reserved
8   *
9   *      Permission to use, copy, modify, and distribute this software and its
10  *      documentation for any purpose and without fee is hereby granted, provided
11  *      that the above copyright notice appear in all copies and that both that
12  *      copyright notice and this permission notice appear in supporting
13  *      documentation, and that the name of the author not be used in advertising
14  *      or publicity pertaining to distribution of the software without specific,
15  *      written prior permission.
16  *
17  *      THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
18  *      ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
19  *      AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
20  *      DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
21  *      AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
22  *      OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
23  **/
24
25 #ifndef MCTS_STATE_
26 #define MCTS_STATE_
27
28 #include <memory>
```

```
29
30 namespace vne {
31 namespace mcts {
32 class State {
33 public:
34   virtual std::shared_ptr<State> getCopy() const = 0;
35   virtual ~State(){};
36
37 protected:
38   State(){};
39 };
40 }
41 }
42 #endif
```

Listing B.16: vne-nm-state.h: VNE Node Mapping State Header File

```
1  /**
2   * @file vne-nm-state.h
3   * @author Soroush Haeri <soroosh.haeri@me.com>
4   * @date 7/16/14
5   *
6   * @copyright Copyright (c) 7/16/14                    SOROUSH HAERI
7   *     All Rights Reserved
8   *
9   *     Permission to use, copy, modify, and distribute this software and its
10  *     documentation for any purpose and without fee is hereby granted, provided
11  *     that the above copyright notice appear in all copies and that both that
12  *     copyright notice and this permission notice appear in supporting
13  *     documentation, and that the name of the author not be used in advertising or
14  *     publicity pertaining to distribution of the software without specific,
15  *     written prior permission.
16  *
17  *     THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
18  *     ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
19  *     AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
20  *     DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
21  *     AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
22  *     OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
23  **/
24
25 #ifndef MCVNE_VNE_NM_STATE_
26 #define MCVNE_VNE_NM_STATE_
27
28 #include "mcts/state.h"
29
30 #include "core/network.h"
31 #include "core/virtual-network-request.h"
32
33 using namespace vne::mcts;
34
35 namespace vne {
36 namespace mcvne {
37
38 class VNENMState : public State {
39 public:
40   VNENMState(std::shared_ptr<std::set<int>> _VNRNodeIdSet, int _vnrid,
41              const std::map<int, int> *mappedNodes);
42   virtual ~VNENMState();
43
44   virtual std::shared_ptr<State> getCopy() const override;
45
46   void addNodeMapping(int sNodeId);
47   const std::map<int, int> *getNodeMap() const;
48
```

```
49    int getPreviousVNId();
50    int getCurrentVNId() const;
51    int getNextVNId();
52
53    int getVNRId() const;
54
55    std::shared_ptr<std::set<int>> getUsedSNIds();
56
57    bool isTreminal() const;
58    bool isStartState() const;
59
60  protected:
61    VNENMState();
62
63  private:
64    int vnrID;
65    std::set<int>::const_iterator VNIdSetIterator;
66    std::shared_ptr<std::set<int>> VNNodeIdSet;
67    // keeps the valid choices for embedding of the current VN.
68    // nodeMap<VirtualnodeId, SubstrateNodeId>;
69    std::map<int, int> nodeMap;
70  };
71  }
72  }
73  #endif
```

Listing B.17: vne-nm-state.cc: VNE Node Mapping State Source File

```
1   /**
2    * @file vne-nm-state.cc
3    * @author Soroush Haeri <soroosh.haeri@me.com>
4    * @date 7/16/14
5    *
6    * @copyright Copyright (c) 7/16/14                    SOROUSH HAERI
7    *      All Rights Reserved
8    *
9    *      Permission to use, copy, modify, and distribute this software and its
10   *      documentation for any purpose and without fee is hereby granted, provided
11   *      that the above copyright notice appear in all copies and that both that
12   *      copyright notice and this permission notice appear in supporting
13   *      documentation, and that the name of the author not be used in advertising or
14   *      publicity pertaining to distribution of the software without specific,
15   *      written prior permission.
16   *
17   *      THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
18   *      ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
19   *      AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
20   *      DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
21   *      AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
22   *      OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
23   **/
24
25  #include "vne-nm-state.h"
26
27  namespace vne {
28  namespace mcvne {
29  VNENMState::VNENMState(std::shared_ptr<std::set<int>> _idSet, int vnrid,
30                         const std::map<int, int> *mappedNodes)
31      : VNNodeIdSet(_idSet), vnrID(vnrid) {
32    if (mappedNodes->size() > 0) {
33      for (auto it = mappedNodes->begin(); it != mappedNodes->end(); it++) {
34        auto it2 = VNNodeIdSet->find(it->first);
35        assert(it2 != VNNodeIdSet->end());
36        nodeMap.insert(std::make_pair(it->first, it->second));
37        VNNodeIdSet->erase(it2);
```

```
38      }
39    }
40    VNIdSetIterator = VNNodeIdSet->begin();
41  };
42
43  VNENMState::VNENMState(){};
44
45  VNENMState::~VNENMState(){};
46
47  const std::map<int, int> *VNENMState::getNodeMap() const { return &nodeMap; };
48
49  int VNENMState::getPreviousVNId() {
50    if (VNIdSetIterator == VNNodeIdSet->begin())
51      return -1;
52    VNIdSetIterator--;
53    int nextId = *(VNIdSetIterator);
54    VNIdSetIterator++;
55
56    return nextId;
57  };
58
59  int VNENMState::getCurrentVNId() const {
60    if (VNIdSetIterator == VNNodeIdSet->end())
61      return -1;
62
63    return *(VNIdSetIterator);
64  };
65
66  int VNENMState::getNextVNId() {
67    VNIdSetIterator++;
68    if (VNIdSetIterator == VNNodeIdSet->end())
69      return -1;
70    int nextId = *(VNIdSetIterator);
71    VNIdSetIterator--;
72    return nextId;
73  };
74
75  int VNENMState::getVNRId() const { return vnrID; }
76
77  bool VNENMState::isTreminal() const {
78    return (VNIdSetIterator == VNNodeIdSet->end());
79  };
80
81  bool VNENMState::isStartState() const { return nodeMap.empty(); }
82
83  std::shared_ptr<State> VNENMState::getCopy() const {
84    std::shared_ptr<VNENMState> newSt(new VNENMState());
85    *newSt = *this;
86    return newSt;
87  }
88
89  std::shared_ptr<std::set<int>> VNENMState::getUsedSNIds() {
90    std::shared_ptr<std::set<int>> outSet(new std::set<int>());
91    for (auto it = nodeMap.begin(); it != nodeMap.end(); it++) {
92      outSet->insert(it->second);
93    }
94    return outSet;
95  }
96
97  void VNENMState::addNodeMapping(int sNodeId) {
98    assert(nodeMap.find(*VNIdSetIterator) == nodeMap.end());
99    nodeMap.insert(std::make_pair(*VNIdSetIterator, sNodeId));
100   VNIdSetIterator++;
101 }
102 }
103 }
```

Listing B.18: mcvne-bfs-link-embedding-algo.h: Breadth-First Search-Based Virtual Link Embedding Algorithm Header File

```
/**
 * @file mcvne-bfs-link-embedding-algo.h
 * @author Soroush Haeri <soroosh.haeri@me.com>
 * @date 7/16/14
 *
 * @copyright Copyright (c) 7/16/14                        SOROUSH HAERI
 *      All Rights Reserved
 *
 *      Permission to use, copy, modify, and distribute this software and its
 *      documentation for any purpose and without fee is hereby granted, provided
 *      that the above copyright notice appear in all copies and that both that
 *      copyright notice and this permission notice appear in supporting
 *      documentation, and that the name of the author not be used in advertising or
 *      publicity pertaining to distribution of the software without specific,
 *      written prior permission.
 *
 *      THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
 *      ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
 *      AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
 *      DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
 *      AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 *      OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 **/

#ifndef MCVNE_BFS_LINK_EMBEDDING_ALGO_
#define MCVNE_BFS_LINK_EMBEDDING_ALGO_

#include "glpk.h"

#include "core/network.h"
#include "core/link-embedding-algorithm.h"

#include "Vineyard/vy-substrate-node.h"
#include "Vineyard/vy-virtual-net-request.h"

using namespace vne::vineyard;

namespace vne {
namespace mcvne {

template <typename = Network<VYSubstrateNode<>, VYSubstrateLink<>>,
          typename = VYVirtualNetRequest<>>
class MCVNEBFSLinkEmbeddingAlgo
    : public LinkEmbeddingAlgorithm<
          Network<VYSubstrateNode<>, VYSubstrateLink<>>,
          VYVirtualNetRequest<>> {
public:
  MCVNEBFSLinkEmbeddingAlgo();
  ~MCVNEBFSLinkEmbeddingAlgo();
  virtual Embedding_Result
  embeddVNRLinks(std::shared_ptr<SUBSTRATE_TYPE> substrate_network,
                 std::shared_ptr<VNR_TYPE> vnr);
  virtual Embedding_Result embeddVNRLinksForIdSets(
      std::shared_ptr<SUBSTRATE_TYPE> substrate_network,
      std::shared_ptr<VNR_TYPE> vnr, const std::map<int, int> *nodeIdMap,
      std::map<int,
               std::list<std::pair<int, std::shared_ptr<Resources<double>>>>>
          *linkMap) override;

private:
  std::shared_ptr<const std::set<int>> substrateNodeIdSet;
  std::shared_ptr<const std::set<int>> substrateLinkIdSet;
  std::shared_ptr<const std::set<int>> virtualNodeIdSet;
  std::shared_ptr<const std::set<int>> virtualLinkIdSet;
```

```
65
66    std::vector<int> allNodeIds;
67
68    bool setAlpha;
69    bool setBeta;
70
71    inline Embedding_Result embeddLinks(
72        std::shared_ptr<SUBSTRATE_TYPE> substrate_network,
73        std::shared_ptr<VNR_TYPE> vnr,
74        const std::map<int, int> *nodeIdMap = nullptr,
75        std::map<int,
76                 std::list<std::pair<int, std::shared_ptr<Resources<double>>>>>
77            *linkMap = nullptr);
78 };
79 }
80 }
81 #endif
```

Listing B.19: mcvne-bfs-link-embedding-algo.cc: Breadth-First Search-Based Virtual Link Embedding Algorithm Source File

```
1  /**
2   * @file mcvne-bfs-link-embedding-algo.cc
3   * @author Soroush Haeri <soroosh.haeri@me.com>
4   * @date 7/16/14
5   *
6   * @copyright Copyright (c) 7/16/14                        SOROUSH HAERI
7   *      All Rights Reserved
8   *
9   *      Permission to use, copy, modify, and distribute this software and its
10  *      documentation for any purpose and without fee is hereby granted, provided
11  *      that the above copyright notice appear in all copies and that both that
12  *      copyright notice and this permission notice appear in supporting
13  *      documentation, and that the name of the author not be used in advertising or
14  *      publicity pertaining to distribution of the software without specific,
15  *      written prior permission.
16  *
17  *      THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
18  *      ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS; IN NO EVENT SHALL
19  *      AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
20  *      DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
21  *      AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
22  *      OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
23  **/
24
25 #include "mcvne-bfs-link-embedding-algo.h"
26 #include "core/config-manager.h"
27
28 namespace vne {
29 namespace mcvne {
30 template <>
31 MCVNEBFSLinkEmbeddingAlgo<>::MCVNEBFSLinkEmbeddingAlgo()
32     : LinkEmbeddingAlgorithm<Network<VYSubstrateNode<>, VYSubstrateLink<>>,
33                              VYVirtualNetRequest<>>(
34           Link_Embedding_Algo_Types::NO_PATH_SPLITTING),
35       substrateLinkIdSet(nullptr), substrateNodeIdSet(nullptr) {}
36 template <> MCVNEBFSLinkEmbeddingAlgo<>::~MCVNEBFSLinkEmbeddingAlgo() {}
37
38 template <>
39 inline Embedding_Result MCVNEBFSLinkEmbeddingAlgo<>::embeddLinks(
40     std::shared_ptr<SUBSTRATE_TYPE> substrate_network,
41     std::shared_ptr<VNR_TYPE> vnr, const std::map<int, int> *nodeIdMap,
42     std::map<int, std::list<std::pair<int, std::shared_ptr<Resources<double>>>>>
43         *linkMap) {
44   virtualNodeIdSet = vnr->getVN()->getNodeIdSet();
```

```
45    virtualLinkIdSet = vnr->getVN()->getLinkIdSet();
46
47    if (nodeIdMap == nullptr)
48      nodeIdMap = vnr->getNodeMap();
49
50    std::map<int, double> usedSLBW;
51    for (auto it = virtualLinkIdSet->begin(); it != virtualLinkIdSet->end();
52         it++) {
53      const std::shared_ptr<VYVirtualLink<>> currentVL =
54          vnr->getVN()->getLink(*it);
55
56      int VNfromId = currentVL->getNodeFromId();
57      int VNtoId = currentVL->getNodeToId();
58
59      double vlBW = currentVL->getBandwidth();
60
61      int SNfromId = nodeIdMap->find(VNfromId)->second;
62      int SNtoId = nodeIdMap->find(VNtoId)->second;
63
64      std::set<int> visitedSubstrateNodesIds;
65
66      // Create a queue for BFS
67      // It holds the the last visited substrate node id and a list of substrate
68      // link ids
69      // used to get there
70      std::list<std::pair<int, std::list<int>>> queue;
71
72      visitedSubstrateNodesIds.insert(SNfromId);
73      queue.push_back(std::make_pair(SNfromId, std::list<int>()));
74
75      bool foundSNtoId = false;
76      std::list<int> shortestPath;
77
78      while (!queue.empty() && !foundSNtoId) {
79        int currentSNId = queue.front().first;
80        std::list<int> currentPath = queue.front().second;
81        queue.pop_front();
82
83        const std::shared_ptr<const std::vector<
84            std::shared_ptr<VYSubstrateLink<>>>> slAttachedToCurrentSN =
85            substrate_network->getLinksForNodeId(currentSNId);
86
87        for (int i = 0; i < slAttachedToCurrentSN->size(); i++) {
88          // check if the link meets bandwidth constrains otherwise do not
89          // consider the link
90          std::shared_ptr<VYSubstrateLink<>> currentSL =
91              slAttachedToCurrentSN->at(i);
92          double residualBW = currentSL->getBandwidth();
93          std::list<int> currentPathCopy = currentPath;
94          if (usedSLBW.find(currentSL->getId()) != usedSLBW.end())
95            residualBW -= usedSLBW[currentSL->getId()];
96          if (residualBW >= vlBW) {
97            int nextSNId = -1;
98            if (slAttachedToCurrentSN->at(i)->getNodeToId() != currentSNId)
99              nextSNId = slAttachedToCurrentSN->at(i)->getNodeToId();
100           else
101             nextSNId = slAttachedToCurrentSN->at(i)->getNodeFromId();
102
103           // if the nextnode is destination:
104           if (nextSNId == SNtoId) {
105             foundSNtoId = true;
106             currentPathCopy.push_back(currentSL->getId());
107             usedSLBW[currentSL->getId()] += vlBW;
108             shortestPath = currentPathCopy;
109             break;
110           }
111           // if the destination node is not visited
```

```
112             if (visitedSubstrateNodesIds.find(nextSNId) ==
113                 visitedSubstrateNodesIds.end()) {
114               visitedSubstrateNodesIds.insert(nextSNId);
115               currentPathCopy.push_back(slAttachedToCurrentSN->at(i)->getId());
116               usedSLBW[currentSL->getId()] += vlBW;
117               queue.push_back(std::make_pair(nextSNId, currentPathCopy));
118             }
119           }
120         }
121       }
122       if (queue.empty() && !foundSNtoId)
123         return Embedding_Result::NOT_ENOUGH_SUBSTRATE_RESOURCES;
124
125       std::shared_ptr<Resources<double>> _res(new Resources<double>(vlBW));
126       for (auto it = shortestPath.begin(); it != shortestPath.end(); it++) {
127         if (linkMap == nullptr)
128           vnr->addLinkMapping(*it, currentVL->getId(), _res);
129         else
130           (*linkMap)[currentVL->getId()].push_back(std::make_pair(*it, _res));
131       }
132     }
133
134     if ((vnr->getLinkMap()->size() == virtualLinkIdSet->size() ||
135          (linkMap != nullptr && (linkMap->size() == virtualLinkIdSet->size()))))
136       return Embedding_Result::SUCCESSFUL_EMBEDDING;
137
138     return Embedding_Result::NOT_ENOUGH_SUBSTRATE_RESOURCES;
139   }
140
141   template <>
142   Embedding_Result MCVNEBFSLinkEmbeddingAlgo<>::embeddVNRLinks(
143       std::shared_ptr<SUBSTRATE_TYPE> substrate_network,
144       std::shared_ptr<VNR_TYPE> vnr) {
145     return (embeddLinks(substrate_network, vnr));
146   }
147
148   template <>
149   Embedding_Result MCVNEBFSLinkEmbeddingAlgo<>::embeddVNRLinksForIdSets(
150       std::shared_ptr<SUBSTRATE_TYPE> substrate_network,
151       std::shared_ptr<VNR_TYPE> vnr, const std::map<int, int> *nodeIdMap,
152       std::map<int, std::list<std::pair<int, std::shared_ptr<Resources<double>>>>>
153           *linkMap) {
154     return (embeddLinks(substrate_network, vnr, nodeIdMap, linkMap));
155   }
156   }
157   }
```