UNIVERSITÉ DE LORRAINE | BIBLIOTHÈQUES UNIVERSITAIRES

AVERTISSEMENT
DROIT D'AUTEUR

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document. Toute contrefaçon, plagiat ou reproduction illicite encourt une poursuite pénale.

Contact bibliothèque : ddoc-theses-contact@univ-lorraine.fr
(Cette adresse ne permet pas de contacter les auteurs)

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10

UNIVERSITÉ DE LORRAINE
34 COURS LÉOPOLD — CS 25233
54052 NANCY CEDEX
TÉL : 03 54 50 54 00
CONTACT@UNIV-LORRAINE.FR
WWW.UNIV-LORRAINE.FR

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

In-Network Placement of ACL Rules Under Constraints

THÈSE

présentée et soutenue publiquement le 25 septembre 2026

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Wafik ZAHWA

Composition du jury

Président :	Bernardetta Addis	Professeure à l'Université de Lorraine
Rapporteurs :	Stefano Secci	Professeur au CNAM, Paris
	Hind Castel-Taleb	Professeure à Télécom SudParis
Examinateurs :	Noura Limam	Research Assistant Professor, University of Waterloo, Canada
Encadrants :	Abdelkader Lahmadi	Professeur à l'Université de Lorraine
	Michaël Rusinowitch	Directeur de recherche Inria (émérite), Nancy-Grand Est
	Mondher Ayadi	Directeur Général, Numeryx, Paris

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Remerciements

First and foremost, I would like to express my deepest gratitude to my supervisors, Abdelkader Lahmadi and Michael Rusinowitch. This thesis would not have been possible without your guidance, trust, and continuous support. Beyond the scientific supervision, you have played an important role in my personal and academic development. I am sincerely grateful for the time you devoted to me, for the lessons you taught me, and for believing in me throughout this journey.

I would also like to thank the members of my Ph.D. committee, Stefano Secci, Hind Castel-Taleb, Noura Limam, and Bernardetta Addis, for accepting to evaluate this work and for taking part in this important milestone. I am grateful for your time, your thoughtful feedback, and your valuable comments, which contributed to improving the quality of this thesis.

My sincere thanks also go to the members of my Comité de Suivi, Bernardetta Addis and Ye-Qiong Song, for their advice, encouragement, and constructive suggestions throughout the different stages of my doctoral work. Their guidance helped me refine the direction of my research and move forward with greater clarity.

I am grateful to all my colleagues at Numeryx Technologies, and I would like to express special thanks to Dr. Mondher Ayadi, CEO of Numeryx Technologies, for giving me the opportunity to conduct this thesis and for making this experience possible. I also thank the members of the RESIST and PESTO research teams at Inria Nancy for their support and for the stimulating research environment they provided during my Ph.D. journey. I am equally thankful to my colleagues at ENSEM for the warm and supportive atmosphere.

To my family, I owe more than words can express. To my beloved parents, Ibrahim and Insaf, thank you for your unconditional love, sacrifices, patience, and belief in me. Everything I have achieved is deeply rooted in your endless support. To my sister Hadil and my brother Abbas, thank you for your encouragement, your presence, and for always being there when I needed you most.

I would also like to thank my dear friends Dani, Soha, Chahrour, Wail, Jana, Jaber, Lyne, and Franco. Despite the distance, your friendship has remained a constant source of strength and comfort. Your support, kindness, and presence have meant a great deal to me throughout this journey.

I would like to express my heartfelt gratitude to my friends in the Lebanese Red Cross, Hassan Saad, Hassan Badawi, and Dbouk, with whom I had the privilege of serving as a volunteer. Their dedication to humanity, compassion, and peace has always been a source of inspiration to me. Through their courage and selfless service, they remind us of the profound impact that kindness and commitment to others can have on the world.

I hold a special thought for Hassan Badawi, who lost his life during a rescue mission. May his soul rest in eternal peace. His courage, generosity, and sacrifice will remain forever in my heart and memory.

Finally, to my fiancée, Lara, I would like to express my deepest gratitude. Your love, patience, encouragement, and belief in me carried me through many difficult moments. Your presence gave me strength when I needed it most, and your support has meant more to me than words can express.

"On this earth is what makes,
life worth living."

— Mahmoud Darwish

Abstract

Modern networks must enforce increasingly large and complex security policies while operating under strict hardware limitations. Access Control Lists (ACLs) remain one of the most widely used mechanisms for traffic filtering and access control, but their deployment is constrained by the limited capacity, high cost, and energy consumption of Ternary Content Addressable Memory (TCAM). As ACL tables continue to grow, storing complete policies on individual switches becomes increasingly impractical. This thesis addresses this scalability challenge by adopting a split-and-distribution paradigm, in which an ACL is decomposed into smaller rule subsets and distributed across multiple network devices while preserving the semantics of the original centralized policy. Adopting this paradigm raises a central and non-trivial question: how should rule subsets be placed across the network so that policy correctness is preserved under TCAM memory constraints, routing variability, and additional operational requirements?

To address this question, we consider three complementary placement settings. First, we investigate distributed ACL placement under routing uncertainty, where traffic may follow multiple possible source-destination paths. We formulate the problem on single-source-destination directed acyclic graphs and propose complementary solutions, including an st-mincut-based method, a greedy heuristic, and reinforcement learning (RL). Second, we generalize the RL approaches to topologies of arbitrary size by combining graph neural networks (GNNs) with deep reinforcement learning (DQN). This graph-aware formulation allows the placement policy to exploit the structural properties of the network rather than being tied to a fixed topology. The framework is further extended to incorporate distance-aware placement constraints, demonstrating its flexibility in adapting to additional requirements. Finally, the thesis studies a more operational setting in which placement is restricted to selected routing paths and must account for practical constraints such as bandwidth efficiency and rule ordering. The proposed constraint-aware framework integrates these requirements into the placement process.

Extensive experiments on synthetic and real-world topologies, including comparisons against state-of-the-art baselines, demonstrate that the proposed framework consistently achieves high placement success rates and efficient TCAM utilization, while offering scalability, topology generalization, and adaptability to new operational constraints that classical heuristic and Integer Linear Programming (ILP)-based optimization methods cannot match.

Keywords: access control list, automated rule placement, software defined network, graph neural network, reinforcement learning, bandwidth constraint, ordering constraint.

Résumé

Les réseaux modernes doivent appliquer des politiques de sécurité volumineuses et complexes, tout en fonctionnant sous de fortes contraintes matérielles. Les Access Control Lists (ACLs), largement utilisés pour le filtrage du trafic et le contrôle d'accès, voient leur déploiement limité par la capacité réduite, le coût élevé et la consommation énergétique des Ternary Content Addressable Memories (TCAM). Quand la taille des ACLs augmente, le stockage de politiques complètes sur des commutateurs individuels devient impraticable. Cette thèse aborde ce défi de passage à l'échelle en adoptant un paradigme de découpage et de distribution, dans lequel une ACL est décomposée en sous-ensembles de règles plus petits et distribués sur plusieurs équipements réseau, tout en préservant la sémantique de la politique centralisée initiale. L'adoption de ce paradigme soulève une question centrale et non triviale: comment placer les sous-ensembles de règles dans le réseau afin que la correction de la politique soit préservée, et sous contraintes de mémoire TCAM, de variabilité du routage et d'exigences opérationnelles supplémentaires?

Pour répondre à cette question, nous considérons trois configurations de placement complémentaires. Premièrement, nous étudions le placement distribué des ACLs sous incertitude de routage, où le trafic peut suivre plusieurs chemins possibles entre une source et une destination. Nous formulons le problème sur des graphes orientés acycliques à source unique et destination unique, et proposons des solutions complémentaires, incluant une méthode basée sur le st-mincut, une heuristique gloutonne et l'apprentissage par renforcement (RL). Deuxièmement, nous généralisons les approches RL à des topologies de taille arbitraire en combinant les réseaux de neurones de graphes (GNNs) avec l'apprentissage profond par renforcement (DQN). Cette formulation sensible au graphe permet à la politique de placement d'exploiter les propriétés structurelles du réseau, plutôt que d'être liée à une topologie fixe. Elle est ensuite étendue afin d'intégrer des contraintes de placement liées à la distance, démontrant ainsi sa flexibilité pour s'adapter à des exigences supplémentaires. Enfin, cette thèse étudie la situation importante où le placement est restreint à des chemins de routage sélectionnés et doit tenir compte de contraintes pratiques telles que l'efficacité en bande passante et l'ordonnancement des règles. L'approche proposée, sensible aux contraintes, intègre ces exigences dans le processus de placement.

Des expérimentations approfondies menées sur des topologies synthétiques et réelles, ainsi que les comparaisons avec des approches de référence de l'état de l'art, confirment l'efficacité du cadre proposé. Les résultats montrent qu'il permet d'obtenir de manière stable des taux élevés de réussite du placement et une utilisation efficace de la mémoire TCAM. Ils mettent également en évidence sa scalabilité, sa généralisation à différentes topologies et son adaptabilité à de nouvelles contraintes opérationnelles, là où les méthodes heuristiques classiques et les approches fondées sur la programmation linéaire en nombres entiers (ILP) restent plus limitées.

Mots-clés: liste de contrôle d'accès, placement automatisé de règles, réseau défini par logiciel (SDN), réseau de neurones de graphes, apprentissage par renforcement, contrainte de bande passante, contrainte d'ordonnancement.

Contents

1 General Introduction **1**

1.1 Context . . . 1
1.2 Problem Statement . . . 3
1.3 Contributions . . . 5
1.4 Thesis Organization . . . 6
1.5 Publications . . . 8

2 Background and State of the Art **9**

2.1 Introduction . . . 9
2.2 Background and Preliminaries . . . 10
2.2.1 Access Control Lists . . . 10
2.2.2 Reinforcement Learning . . . 13
2.2.3 Graph Neural Networks . . . 20
2.3 ACL Optimization and Placement Approaches . . . 23
2.3.1 Rule Compression . . . 23
2.3.2 Rule Eviction . . . 25
2.3.3 Split and Distribution . . . 27
2.4 Distributed ACL Placement Approaches . . . 31
2.4.1 Heuristic, Graph-Based, and Architectural Approaches . . . 31
2.4.2 Exact Optimization Approaches . . . 34
2.4.3 Machine Learning Approaches . . . 38
2.5 Summary . . . 41

3 Distributed ACL Placement Under Dynamic Routing Constraints **45**

3.1 Introduction . . . 45
3.2 Network Representation and Problem Statement . . . 46
3.3 Algorithms for Distributed Placement of ACL Rules . . . 49

3.3.1 st-mincut . . . 49
3.3.2 Greedy Heuristic . . . 52
3.3.3 Reinforcement Learning-Based Approach . . . 55
3.4 Experiments and Results . . . 57
3.4.1 Simulation Setup . . . 57
3.4.2 Results and Discussion . . . 59
3.5 Summary . . . 64

4 Topology-Aware Distributed Placement of ACLs Using GNN-DQN 65
4.1 Introduction . . . 65
4.2 Learning over Variable-Size Graphs . . . 66
4.2.1 Dimensional Normalization via Padding . . . 67
4.2.2 Topology-Aware Representation via GNNs . . . 68
4.3 ACL Rules Placement Using GNN-DQN . . . 69
4.3.1 Architecture and Learning Framework . . . 69
4.3.2 Agent State, Action, and Reward Design . . . 72
4.4 Experimental Setup and Methodology . . . 74
4.4.1 Experimental Configuration . . . 74
4.4.2 Synthetic Network Topology Generation . . . 75
4.5 Evaluation and Performance Analysis . . . 77
4.5.1 Training Behavior and Model Performance . . . 77
4.5.2 Generalization Across Graph Sizes and Rule Configurations . . . 79
4.6 Distance-Constrained ACL Rule Placement . . . 80
4.6.1 Motivation and Definition of Distance Constraints . . . 80
4.6.2 Adaptation of the GNN-DQN Framework . . . 83
4.6.3 Experimental Results . . . 84
4.7 Summary . . . 86

5 Constrained ACL Rule Placement over Selected Paths 89
5.1 Introduction . . . 89
5.2 Problem Formulation . . . 90
5.2.1 Motivating Example . . . 91
5.2.2 Network Representation and Problem Statement . . . 93
5.2.3 Rule Placement under Constraints . . . 95
5.3 ACL Rule Placement Framework . . . 96

5.3.1 Agent State, Action, and Reward Design . . . 97
5.3.2 Reward Shaping under Bandwidth Constraint . . . 98
5.3.3 Handling Rule Ordering Constraint . . . 98
5.4 Evaluation Methodology and Baseline Comparison . . . 101
5.4.1 Baseline Algorithms for ACL Rule Placement . . . 101
5.4.2 Experimental Setup . . . 102
5.5 Experimental Results and Performance Analysis . . . 103
5.5.1 Performance With Underutilized Capacity . . . 103
5.5.2 Performance under Near Maximum Capacity . . . 106
5.5.3 Effect of the Bandwidth Constraint . . . 108
5.5.4 Generalization to Real-World Network Topologies . . . 109
5.6 Discussion and Practical Deployment Considerations . . . 112
5.6.1 Mininet Deployment . . . 112
5.6.2 Discussion . . . 113
5.7 Summary . . . 115

6 General Conclusion **117**
6.1 Achievements . . . 117
6.2 Limitations . . . 119
6.3 Future Work . . . 121

Bibliography **123**

List of Figures **135**

List of Tables **137**

Acronyms **139**

Appendix **141**

A Résumé de la thèse en français **141**
A.1 Contexte . . . 141
A.2 Problématique . . . 144
A.3 Notre contribution . . . 146
A.3.1 Placement distribué des ACLs sous incertitude de routage (Chapitre 3). 147
A.3.2 Placement scalable sensible à la topologie via GNN-DQN (Chapitre 4). 147

A.3.3 Placement des ACLs sensible aux contraintes sur des chemins sélectionnés (Chapitre 5). 148
A.4 Limitations 149
A.5 Travaux futurs 150

Chapter 1

General Introduction

Contents

1.1 Context 1
1.2 Problem Statement 3
1.3 Contributions 5
1.4 Thesis Organization 6
1.5 Publications 8

1.1 Context

Networks today carry an enormous and constantly growing variety of traffic. For example, hospitals exchange sensitive patient data with remote physicians, factories connect thousands of sensors to centralized control systems, banks process millions of transactions every second, and video platforms stream high-definition content to millions of users worldwide. This diversity is not accidental; it reflects how much networks have evolved over the past two decades, both in how they are built and how they are used. This complexity is also visible inside enterprise networks. A typical corporate network today may connect around 35,000 devices across nearly 80 different types, with about one-third of them operating outside direct IT control [1]. These unmanaged devices include IoT equipment such as smart TVs and sensors, as well as personal laptops and smartphones brought by employees. As a result, networks are becoming more heterogeneous and harder to control. Each of these services comes with specific communication requirements that must be enforced within the network. Sensitive data should only be accessible to authorized users, industrial systems must remain isolated from external traffic, financial operations need strong protection against unauthorized access, and multimedia applications require efficient traffic handling without compromising security. These requirements, known as *security policies*, define how network traffic should be controlled and protected. As networks continue to grow, these policies become larger and more complex, making them increasingly difficult to manage and enforce. Understanding this growing complexity and its implications is the starting point of this thesis.

In practice, security policies are defined as high-level requirements that describe how network traffic should be controlled. To be enforced by network devices, these policies are translated into concrete rules or configurations, most commonly in the form of Access Control Lists (ACLs). An ACL can be understood as a list of instructions placed at the entry point of a network device, such as a router or a switch, that determines how incoming packets should be handled. Each instruction, called a rule, specifies a condition, for example, that a packet originates from a given IP address and targets a specific service or port, and an associated action, such as allowing the packet, blocking it, or applying a particular treatment.

The size of ACLs has grown significantly due to two main factors. First, the rapid expansion and diversification of connected devices require increasingly fine-grained security policies. In 2021, around 13.8 billion devices were connected to the Internet worldwide [2], a number expected to reach 39 billion by 2030 [3]. These include smartphones, industrial sensors, medical devices, vehicles, and smart appliances, all with different communication patterns and security requirements. Each category needs dedicated rules specifying which systems it can access and under what conditions, directly increasing the size and complexity of ACLs. This trend is further amplified by cloud computing and multi-tenant environments, where multiple customers (tenants) share the same physical infrastructure while running independent applications and services. To ensure security and privacy, each tenant must be strictly isolated from others, meaning their traffic must be controlled and separated through dedicated ACL rules. As the number of tenants grows, this requirement leads to a significant increase in the number of rules [4].

Second, the evolving cybersecurity landscape continuously drives the expansion of ACL policies. Security rules must be regularly updated to respond to new threats, vulnerabilities, and malicious actors. Today's attacks are diverse and increasingly sophisticated. For example, ransomware attacks-where attackers block access to data and demand payment to restore it-have affected more than 72% of organisations worldwide in 2023 [5]. Distributed Denial of Service (DDoS) attacks, which consist in overwhelming systems with massive traffic to make them unavailable, increased by 46% in the first half of 2024 compared to 2023 [6]. More broadly, cyber-attacks increased by 30% in 2024, reaching an average of 1,636 weekly attacks per organisation [7]. Attackers also exploit software vulnerabilities, sometimes using artificial intelligence tools [8], and use techniques such as sending deceptive emails with malicious content (phishing), compromising trusted external partners to gain access (supply chain attacks), or stealing legitimate user credentials. A key difficulty is that many attacks originate from IP addresses that are identified as malicious only after they are observed. As a result, new filtering rules must be continuously added. More than 5,000 new malicious IP addresses are discovered every day [9], and some rule sets already contain over 250,000 entries dedicated to blocking known threats. In parallel, regulatory requirements further contribute to this growth. More than 170 data protection laws were introduced worldwide in 2023-2024 [10], each imposing constraints on how sensitive data must be handled. Translating these requirements into enforceable ACL rules adds yet another layer of complexity to network security policies.

At this point, a natural question arises: if networks require more rules, why not simply store them all on each networking device? The answer lies in the hardware used to enforce ACLs. Network devices must inspect every packet and decide, within a tiny frac-

tion of a second, which rule applies. At modern line rates, where millions of packets are processed per second, rules cannot be checked sequentially. Instead, networking devices rely on Ternary Content Addressable Memory (TCAM) [11, 12], a specialized memory that compares packets against all rules simultaneously in parallel in a single operation. TCAM supports ternary values (0, 1, and "*" for "any"), making it the standard and nearly universal memory for ACL enforcement in modern routers and switches.

However, this performance comes at a cost. TCAM is significantly more expensive to manufacture than traditional memory, consumes a large amount of power (up to 60% of a switch's total energy), and occupies a substantial chip area [13, 14, 15]. As a result, its capacity is inherently limited. In practice, while the size of ACLs can reach up to 100,000 rules as early as 2022 [16], this number is expected to grow rapidly in the coming years due to the continuous expansion of connected devices, cloud services, and security requirements discussed earlier. In contrast, modern switch platforms provide concrete evidence that ACL enforcement remains constrained by finite TCAM resources. For example, Cisco Catalyst 9000 series switches support between approximately 1,000 and 18,000 IPv4 ACL entries, depending on the model [17]. The effective capacity is even lower for IPv6, as each rule may consume multiple TCAM entries (typically two) due to hardware-level expansion. Similar constraints are observed in newer architectures: Cisco Silicon One Q200-based switches provide around 8K IPv4 or 4K IPv6 ACL entries [18]. Although some modern high-performance platforms, such as NVIDIA Spectrum SN4000 [19], advertise much larger shared or external TCAM capacities, these resources remain finite and are often shared among ACLs, routes, MAC entries, tunnels, and other forwarding features.

Therefore, despite ongoing hardware improvements, TCAM capacity remains a fundamental bottleneck. The continuous growth of ACL rule sets, combined with limited and shared hardware resources, creates a persistent scalability challenge for network operators. This limitation is consistently highlighted in recent studies [20, 21, 22], and directly motivates the need for more efficient approaches to ACL management. In this thesis, our primary focus will be to address this problem.

1.2 Problem Statement

The growing complexity of ACL policies, combined with the limited, shared capacity of TCAM, poses a fundamental scalability challenge. In practice, it is often impossible to store the entire ACL table on a single switch while maintaining correct and efficient packet processing. As a result, ensuring that security policies are correctly applied across the network becomes increasingly difficult, especially as infrastructures scale and operational constraints multiply. Among the different approaches proposed in the literature, the split-and-distribution paradigm [23, 24, 25] offers the most structurally sound direction: the original ACL is decomposed into smaller rule subsets, which are then installed across multiple switches so that any packet traversing the network encounters all rules applicable to it along its path. This approach directly addresses the TCAM bottleneck without relying on approximation or reactive replacement.

However, adopting this paradigm introduces a new and non-trivial challenge: determining where each rule subset should be placed so that the distributed system collectively

enforces the same policy as the original centralized ACL. This problem is particularly challenging for several reasons. First, traffic between a source and a destination may traverse multiple possible paths, especially in dynamic or load-balanced networks, which requires ensuring that at least one valid filtering point exists along each path. Second, switch capacities are limited and heterogeneous, making the placement decisions interdependent. Third, rule dependencies and ordering constraints must be preserved to guarantee policy correctness. Existing solutions fail to address these challenges jointly; exact optimization approaches become computationally expensive as the network size and constraint set grow, and heuristic approaches are problem-specific and do not adapt when constraints evolve.

Addressing these challenges requires designing scalable and adaptive placement strategies that go beyond classical exact formulations. Therefore, the objective of this thesis is to address the ACL management problem by relying on a smaller TCAM size, splitting the initial ACL table into smaller rule sets, and distributing their part over the network in such a way that the same control functionality between endpoints is maintained. Formally, the problem addressed in this thesis can be stated as follows: given a network topology, a set of routing paths, and a decomposed ACL, how should rule subsets be assigned to switches such that (i) ACL semantic is preserved, i.e., all packets are correctly filtered according to the original policy, (ii) switch memory constraints are respected, and (iii) additional operational constraints, such as routing variability, bandwidth limitations, or rule dependencies, are satisfied?

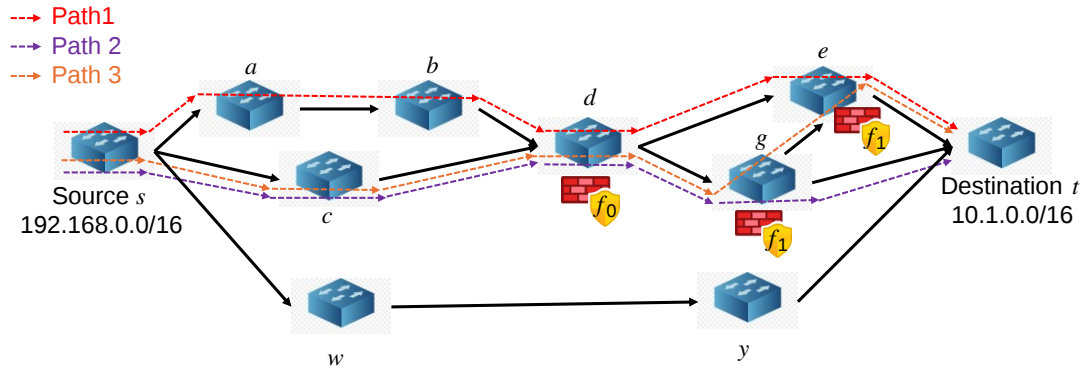


Figure 1.1 – A network topology for a placement scenario with 2 rules f_0 and f_1 to be placed.

Consider the network illustrated in Fig. 1.1, consisting of a source node s, a destination node t, and intermediate switches a–g, as well as nodes w and y, each with a capacity of one rule. Let the initial ACL be decomposed into two rule subsets, denoted by f_0 and f_1. Under the basic requirement of covering all routing paths in the set $\mathcal{P} = \{\text{Path 1, Path 2, Path 3}\}$ while minimizing the number of installed rules, a feasible placement consists of assigning f_0 to node d and f_1 to nodes e and g. However, if f_1 has higher priority than f_0, meaning that it must be evaluated first to preserve the original ACL semantics, the placement must be adjusted accordingly. In this case, f_1 is assigned to node d, while f_0 is placed at nodes e and g. Furthermore, suppose that f_1 is primarily

composed of drop rules. To reduce unnecessary bandwidth consumption, it is desirable to apply such rules as early as possible along the traffic path. Under this constraint, f_1 is moved closer to the ingress, namely to nodes a and c, while f_0 remains at node d. Finally, in a dynamic routing environment where the exact path taken by traffic cannot be predicted in advance, all possible paths between s and t must be covered. This requirement leads to an extension of the placement, with f_0 assigned to node w and f_1 to node y. This example illustrates how ACL rule placement must adapt to multiple and evolving constraints, including path coverage, rule ordering, and bandwidth considerations, thereby highlighting the need for flexible and constraint-aware placement strategies.

Based on the challenges identified, this thesis aims to answer the following research questions related to the scalable and efficient deployment of distributed ACL policies:

RQ1: *How can ACL rule subsets be optimally placed across a network of memory-constrained switches such that complete policy enforcement is guaranteed over* all *possible source-to-destination paths, independently of routing changes?* Most existing approaches assume static routing or a single predetermined path, while in modern networks, traffic between a source and a destination may follow multiple possible paths due to routing dynamics, load balancing, or failures. This raises the question of how to ensure that every possible path is properly filtered, while avoiding unnecessary rule replication and respecting switch memory constraints.

RQ2: *How can distributed ACL placement be made scalable and adaptable to large and dynamic network topologies?* Exact optimization approaches become computationally expensive as the network size, rule set, and constraint space grow. This question investigates whether learning-based and graph-aware methods can provide efficient and generalizable placement strategies across different network configurations.

RQ3: *How can practical operational constraints be incorporated into the ACL placement problem?* In real deployments, placement decisions must account for additional constraints such as limited bandwidth, rule dependencies, and predefined routing policies. This question focuses on how to integrate such constraints into the placement process while preserving policy correctness and scalability.

1.3 Contributions

This thesis addresses the problem of scalable and correct deployment of ACLs under hardware constraints, focusing on the split-and-distribution paradigm. The main contributions of this work are summarized as follows:

- **C1: Distributed ACL Placement under Routing Uncertainty.** We investigate the problem of placing ACL rule subsets in environments where traffic may traverse multiple possible paths. Unlike traditional approaches that assume fixed routing, we propose a placement model that guarantees policy enforcement across all feasible paths. This work highlights the impact of routing variability on rule distribution and introduces methods to ensure correctness while minimizing resource usage.
- **C2: Scalable and Topology-Aware Placement using Learning-Based Methods.** To address the scalability limitations of exact optimization approaches, we

propose a novel learning-based framework for distributed ACL placement. In particular, we leverage graph-based representations of network topologies combined with reinforcement learning techniques to learn efficient placement strategies. The proposed approach adapts to different network structures and significantly reduces computation time while maintaining high-quality solutions.

- **C3: Constraint-Aware ACL Placement under Practical Network Conditions.** We extend the placement problem by incorporating realistic operational constraints, including memory limitations, placement on selected routing paths, rule dependencies, and bandwidth considerations. We propose models and algorithms that integrate these constraints into the placement process, enabling more practical and deployable solutions in real-world networks.

Overall, this thesis contributes to advancing the state of the art in distributed ACL management by proposing scalable, adaptive, and constraint-aware placement solutions that address the limitations of existing approaches.

1.4 Thesis Organization

This thesis is organized into six chapters, each of the core ones (3,4,5) addressing a specific aspect of the distributed ACL placement problem under realistic network constraints. The three research questions formulated in Section 1.2 are answered progressively across Chapters 3, 4, and 5, which constitute the scientific core of the work. The remaining chapters provide the context, foundations, and conclusions necessary to frame these contributions coherently. Figure 1.2 illustrates the overall structure of the thesis and the mapping between each research question, its corresponding contribution, and the chapter in which it is addressed.

Chapter 2 establishes the technical foundations on which the thesis builds and positions the contributions with respect to the existing literature. It begins by introducing the three conceptual pillars of the thesis: Access Control Lists as the security-policy model enforced in networks, Reinforcement Learning as the sequential decision-making framework used to address the placement problem, and Graph Neural Networks as the topology-aware representation learning technique. It then surveys the principal families of approaches proposed to manage ACL scalability under TCAM constraints, namely rule compression, rule eviction, and split-and-distribution, and identifies why the latter is the most structurally sound direction. The chapter concludes with a systematic review of the state of the art in distributed ACL placement, covering heuristic and graph-based methods, exact optimization approaches, and learning-based techniques, and distils the research gaps that directly motivate the three contributions of this thesis.

Chapter 3 addresses RQ1 by investigating how ACL rule subsets can be correctly placed in networks where traffic may follow any of several possible paths between source and destination. We introduce three complementary placement strategies: an exact polynomial-time algorithm based on the st-mincut formulation; a greedy heuristic that iteratively maximizes path coverage; and RL agents that learn adaptive placement policies in constrained settings. Chapter 4 addresses RQ2 by tackling the generalization challenge identified at the end of Chapter 3. It proposes a learning-based framework that combines Graph Neural

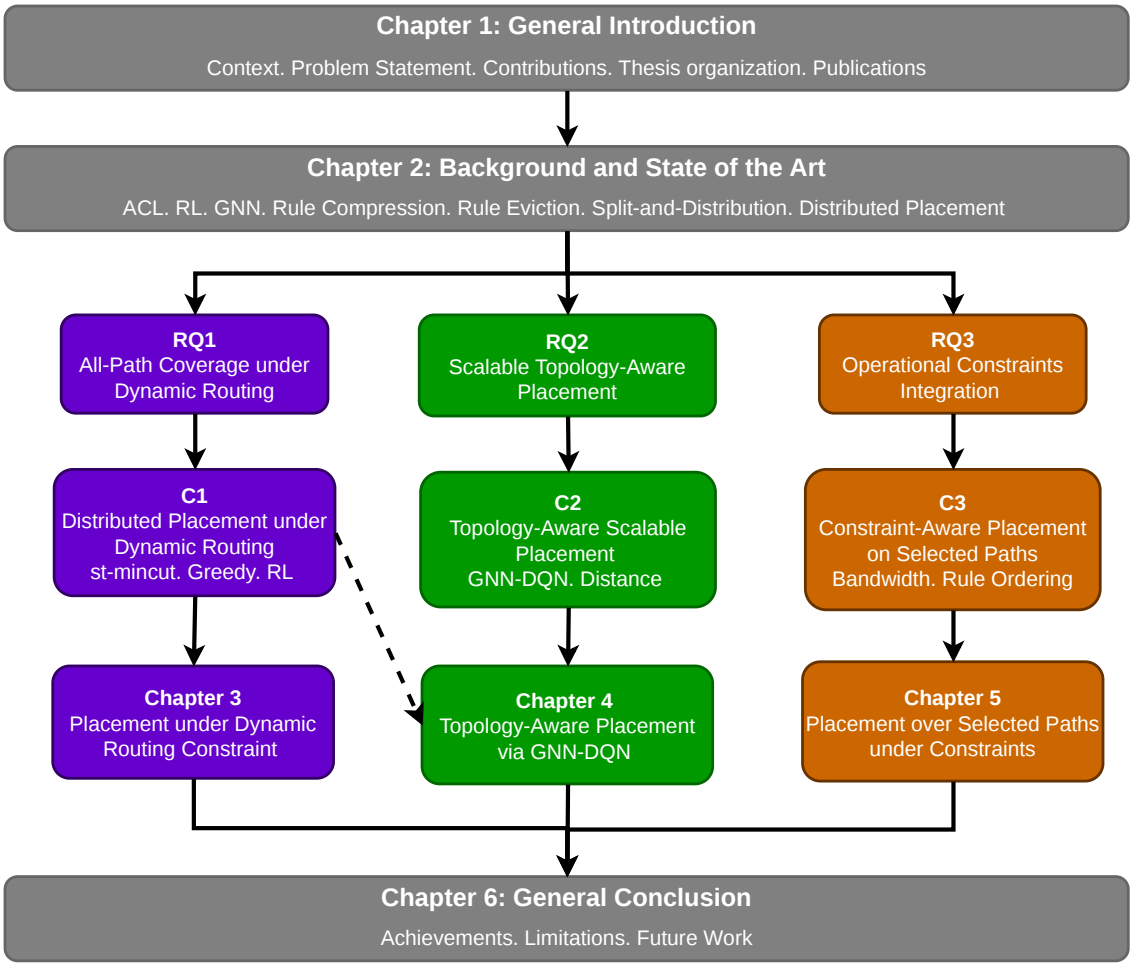


Figure 1.2 – Overview of the thesis organization, illustrating the relationships between research questions, proposed contributions, and corresponding chapters.

Networks with Deep Q-Networks, enabling the agent to capture the structural properties of the network graph and learn placement policies that transfer across topologies of different sizes and structures without retraining. The chapter further extends the framework to handle distance-constrained placement, where an additional operational constraint limits how far a rule may be placed from the traffic source or destination, demonstrating the flexibility of the architecture to accommodate new requirements through reward shaping alone, without modifying the underlying model. Chapter 5 addresses RQ3 by extending the placement framework to the most realistic and operationally demanding setting considered in this thesis. Rather than requiring coverage over all possible paths, we restrict placement to a predefined subset of routing paths, reflecting practical network routing policies such as shortest-path or traffic-engineering strategies. Two additional operational constraints are incorporated: a bandwidth wastage constraint, which enforces the early placement of filtering rules to minimize unnecessary resource consumption along traffic

paths, and a rule ordering constraint, which ensures semantic consistency when dependencies exist between rule subsets. Extensive experimental evaluation against state-of-the-art baselines across both synthetic and real-world topologies demonstrates that the proposed approach consistently achieves high success rates and efficient resource utilization, while maintaining the scalability and adaptability that classical methods cannot provide. Finally, Chapter 6 concludes the thesis by summarizing the main contributions and discussing potential directions for future research.

1.5 Publications

The contributions of this thesis have been published in several international conferences as detailed below:

- Wafik Zahwa, Abdelkader Lahmadi, Michael Rusinowitch, and Mondher Ayadi. "Deep Reinforcement Learning for In-Network Placement of ACL Rules Under Constraints". In 2025 21st International Conference on Network and Service Management (CNSM), pp. 1-7. IEEE, 2025. [26]
- Wafik Zahwa, Abdelkader Lahmadi, Michael Rusinowitch, and Mondher Ayadi. "In-network ACL rules placement using deep reinforcement learning". In 2024 IEEE International Mediterranean Conference on Communications and Networking (MeditCom), pp. 341-346. IEEE, 2024. [27]
- Wafik Zahwa, Abdelkader Lahmadi, Michael Rusinowitch, and Mondher Ayadi. "Automated placement of in-network ACL rules". In 2023 IEEE 9th International Conference on Network Softwarization (NetSoft), pp. 486-491. IEEE, 2023.[28]

Chapter 2

Background and State of the Art

Contents

2.1 Introduction 9
2.2 Background and Preliminaries 10
2.2.1 Access Control Lists 10
2.2.2 Reinforcement Learning 13
2.2.3 Graph Neural Networks 20
2.3 ACL Optimization and Placement Approaches 23
2.3.1 Rule Compression 23
2.3.2 Rule Eviction 25
2.3.3 Split and Distribution 27
2.4 Distributed ACL Placement Approaches 31
2.4.1 Heuristic, Graph-Based, and Architectural Approaches 31
2.4.2 Exact Optimization Approaches 34
2.4.3 Machine Learning Approaches 38
2.5 Summary 41

2.1 Introduction

ACLs remain one of the most widely deployed mechanisms for enforcing traffic filtering and security policies in modern networks. They provide a practical mechanism for translating security and traffic-control policies into enforceable filtering rules within network devices. As discussed in Chapter 1, the continuous growth of network infrastructures, cloud platforms, multi-tenant environments, and security requirements has led to a significant expansion of ACL policies. Since ACL rules are commonly stored in TCAM, which provides high-speed parallel lookup and wildcard matching [11, 12], their deployment is constrained by the limited capacity, high cost, and energy consumption of this memory technology. Consequently, scalable ACL management has become a central challenge in modern networks.

Several families of approaches have been proposed to address this issue. Compression techniques aim to reduce the number of installed entries by merging or transforming semantically equivalent rules. Eviction and caching mechanisms maintain only a subset of rules in fast memory while offloading less frequently used entries to slower storage. Split-and-distribution approaches exploit the aggregate memory resources of multiple switches by decomposing an ACL into smaller rule subsets and distributing them across the network. This last paradigm is particularly relevant to this thesis, as it directly addresses the TCAM bottleneck while preserving the semantics of the original centralized policy. Among these alternatives, the split-and-distribution paradigm is particularly attractive because it addresses the scalability problem at the network level rather than only reducing or dynamically managing the rule set within a single device. Exploiting the aggregate memory resources of multiple switches, it enables large ACL policies to be deployed collectively while preserving the semantics of the original centralized policy. Unlike compression, its effectiveness does not depend on the presence of rule redundancies, and unlike eviction, it does not rely on reactive replacement mechanisms or traffic-dependent rule residency.

This thesis focuses on this paradigm and investigates the problem of efficiently placing ACL rule subsets over a network topology under practical constraints. The objective is to ensure correct policy enforcement while respecting device limitations and operational requirements.

This chapter provides the technical background and literature review required to position the contributions of the thesis. Section 2.2 introduces the main concepts used in the following chapters, namely ACLs, Reinforcement Learning, and Graph Neural Networks. Section 2.3 reviews the principal approaches proposed for ACL memory optimization, including compression, eviction, and split-and-distribution. Section 2.4 focuses on distributed ACL placement methods and discusses their strengths and limitations. Finally, Section 2.5 summarizes the main observations, identifies the remaining research gaps, and motivates the approaches developed in this thesis.

2.2 Background and Preliminaries

This section introduces the principal concepts and technical tools required to understand the remainder of the thesis. It first presents ACLs, which constitute the security-policy model considered throughout this work. It then provides a concise overview of Reinforcement Learning (RL) and Graph Neural Networks (GNNs), which form the methodological foundations of the learning-based placement approaches developed in later chapters.

2.2.1 Access Control Lists

ACLs constitute one of the most established and widely deployed mechanisms for enforcing network security policies. They are commonly used in routers, switches, firewalls, and SDN environments to regulate packet forwarding decisions by specifying which traffic flows are permitted, denied, redirected, or subjected to additional treatment according to predefined administrative policies [29, 30, 31]. Because ACLs operate directly in the

packet-processing path, they remain a fundamental first line of defense for traffic filtering, segmentation, and access control in modern networks. From a functional perspective, an ACL can be viewed as an ordered set of matching rules evaluated sequentially against incoming packets. Each rule typically consists of three components: (i) a set of matching predicates over packet-header fields, (ii) an associated action, and (iii) a priority level used to resolve overlaps between rules. Common matching fields include source and destination IP addresses, transport-layer ports, protocol type, VLAN identifiers, and interface metadata. The associated action usually specifies whether matching packets should be permitted or denied, although more advanced systems may support logging, mirroring, rate-limiting, redirection, or traffic marking [32, 33]. Formally, a rule can be represented as:

$$rule_i = \langle match_i, action_i, priority_i \rangle$$

where $match_i$ denotes the multidimensional matching condition, $action_i$ the operation applied to matching packets, and $priority_i$ the precedence of the rule.

Priority	IP Src	IP Dst	Protocol	Src Port	Dst Port	Action
4	192.168.100.0/24	19.153.234.163/32	TCP	[1,65534]	[1,65534]	Permit
3	201.162.23.191/32	10.133.202.145/32	TCP	[1,65534]	[1,65534]	Deny
2	184.166.27.94/31	98.85.58.204/31	UDP	[1,65534]	[1,65534]	Deny + Log
1	192.168.100.2/30	19.153.234.0/24	TCP	[1,65534]	[1,65534]	Rate-limit
0	21.79.252.56/30	49.50.220.0/22	ICMP	[1,65534]	[1,65534]	Redirect

Table 2.1 – Example of an ACL rule table.

ACLs are generally processed according to a *first-match* or *highest-priority match* semantics. In practice, when a packet arrives at a switch or firewall interface, its header fields are compared against the installed ACL rules. The first rule whose matching predicates are satisfied determines the final action applied to the packet. This ordering is critical because two syntactically valid rules may overlap while prescribing different actions. Consequently, ACL correctness depends not only on the content of the rules, but also on their relative order [33, 34]. In many operational deployments, an implicit default deny rule is placed at the end of the ACL so that any traffic not explicitly authorized is automatically blocked. Table 2.1 shows a rule table containing 5 ACL rules. Each rule has a priority between 0 and 4, a source and destination address, and an action field. A packet with an IP source 192.168.100.1 and an IP destination 19.153.234.163 matches two rules, one with priority 4 and the other with priority 1. In this case, the action of rule with the highest priority, accept in this case, is applied.

Historically, ACLs first emerged in traditional IP routers as interface-based packet filters. Early implementations mainly supported simple permit/deny decisions based on source or destination addresses. Later, so-called extended ACLs introduced richer multi-field matching over Layer-3 and Layer-4 headers, including protocol type and transport ports, thereby significantly increasing policy expressiveness. This evolution enabled ACLs to support more advanced security objectives such as service-level filtering, network segmentation, access restriction to management services, tenant isolation, and traffic steering

[30]. In modern programmable networks, ACL semantics may also be extended to trigger additional actions such as logging suspicious traffic, redirecting flows to middleboxes, rate-limiting selected traffic classes, or tagging packets for downstream processing [23, 35, 36].

From an implementation viewpoint, ACLs are commonly stored in high-speed lookup memories TCAM [12]. TCAM is particularly attractive because it performs parallel lookups in constant time and directly supports ternary symbols $\{0, 1, *\}$, where the wildcard symbol "*" allows prefix and masked matching required by ACL rules. This capability makes TCAM especially suitable for multi-field packet classification, where rules may contain exact matches, prefixes, or wildcard fields. By contrast, conventional SRAM and DRAM provide higher storage density and lower cost, but require sequential or tree-based lookup algorithms whose latency increases with rule-set size. Standard binary CAM enables fast exact matching, but does not naturally support wildcard expressions required by ACL semantics. For these reasons, TCAM has become the preferred hardware technology for implementing line-rate ACL filtering in routers and switches [11, 12, 37]. TCAM offers the lookup speed and wildcard expressiveness required for line-rate ACL enforcement. However, its limited capacity means that large rule sets cannot always be stored entirely on a single device. This practical constraint becomes particularly relevant in modern programmable networks and motivates the scalable deployment approaches discussed in the following sections.

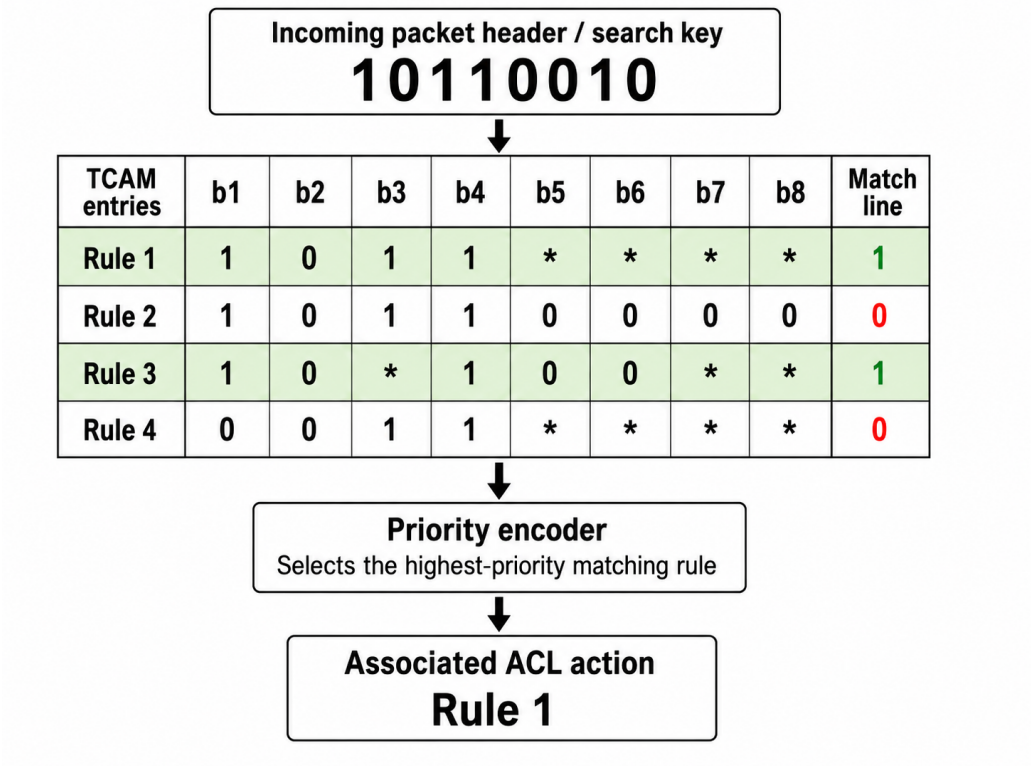


Figure 2.1 – Illustration of the TCAM matching process for ACL rule lookup.

TCAM matching process.
The TCAM matching process relies on a hardware-level parallel comparison between an incoming packet-header key and all entries stored in the table. The search key is extracted from the packet header, while each TCAM row represents one ACL rule encoded as a ternary pattern over the values $\{0, 1, *\}$. During a lookup, the bits of the search key are broadcast simultaneously to all TCAM rows. Each cell then compares the corresponding key bit with its stored value: a stored 0 matches only an input 0, a stored 1 matches only an input 1, and the wildcard symbol $*$ matches either value. A row is declared as matching only when all its non-wildcard bits match the corresponding bits of the search key. Since all rows are evaluated in parallel, multiple rules may match the same packet at the same time. In this case, a priority encoder selects the highest priority matching entry, and the action associated with that entry is applied to the packet. Figure 2.1 illustrates this mechanism using a simplified ACL-oriented example, where the search key $K = 10110010$ is compared in parallel with several ternary entries. In the example, both Rule 1 and Rule 3 match the key, but Rule 1 is selected because it has the highest priority.

2.2.2 Reinforcement Learning

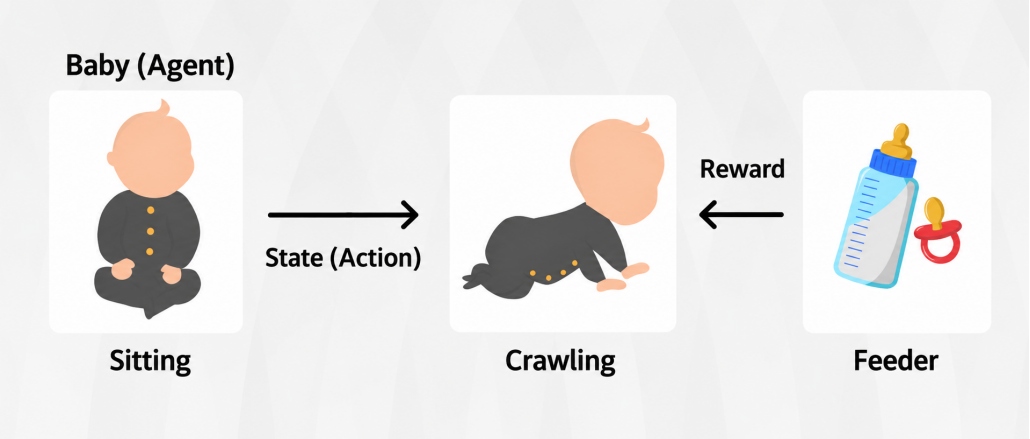


Figure 2.2 – Illustrative example of reinforcement learning, in which an agent improves its behavior through interaction with the environment and evaluative reward feedback [38].

Reinforcement Learning (RL) is a learning paradigm inspired by behavioral observations of humans and other animals, wherein agents adapt their actions through interaction with an environment based on evaluative feedback in the form of rewards and penalties. Figure 2.2 illustrates this principle through the intuitive example of a baby who gradually learns to crawl by associating its actions with rewarding outcomes. Sutton and Barto [39] define RL as the problem of mapping situations to actions so as to maximize a cumulative numerical reward signal, while other perspectives describe RL more broadly as learning through trial-and-error interaction. Despite variations in terminology across the literature, these definitions share a common emphasis on sequential decision-making driven by reward feedback.

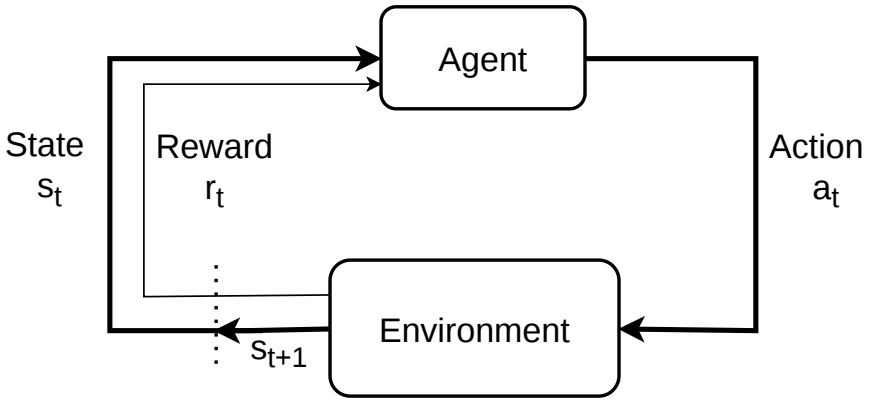


Figure 2.3 – Overview of the main RL elements: agent, environment, states, actions, and rewards.

Formally, and as illustrated in Figure 2.3, RL considers an *agent* that learns to make decisions by interacting directly with an *environment*. At every discrete time step t, the agent observes the current state $s_t \in \mathcal{S}$, selects an action $a_t \in \mathcal{A}$ according to a policy, and receives from the environment both a scalar reward $r_t \in \mathbb{R}$ and a transition to a new state s_{t+1}. The reward signal provides an evaluative measure of the quality of the agent's action, and learning emerges through repeated trial-and-error interactions. This closed-loop agent-environment interaction constitutes the core of RL and fundamentally distinguishes it from supervised and unsupervised learning, where explicit target labels or static data distributions are assumed.

The interaction process is typically organized into *episodes*, each corresponding to a complete trial in which the agent begins from an initial state and interacts with the environment until a terminal condition or a predefined horizon is reached. The agent's behavior is governed by a policy, which is considered optimal if it maximizes the expected cumulative reward over the episode.

This sequential decision-making problem is commonly modeled using the Markov Decision Process (MDP) framework [40], which extends the theory of Markov processes [41]. An MDP is defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ and $\mathcal{A}$ denote the state and action spaces, respectively, $\mathcal{P}(s' \mid s, a)$ represents the transition probability to state s' after taking action a in state s, $\mathcal{R}(s, a)$ is the expected immediate reward, and $\gamma \in [0, 1]$ is the discount factor that determines the relative importance of future rewards. The Markov property assumes that both transitions and rewards depend solely on the current state and action, and not on the full history of past interactions. This framework, rooted in Bellman's dynamic programming principle, underpins the majority of modern RL algorithms [39].

A *trajectory* τ denotes a sequence of states, actions, and rewards generated by the agent-environment interaction, i.e., $\tau = (s_0, a_0, r_1, s_1, a_1, \ldots)$. The objective of the agent is to maximize the expected cumulative discounted reward, referred to as the *return*. For a finite-horizon episode of length N, the return from time step t along a trajectory τ is

defined as

$$R(\tau) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{N} \gamma^k r_{r+k+1} \tag{2.1}$$

The agent's decision-making strategy is captured by a (possibly stochastic) policy $\pi(a \mid s)$, which defines a mapping from states to actions. The goal of RL is to identify an optimal policy π^* that maximizes the expected return over all trajectories induced by the policy, i.e.,

$$\pi^* = \arg\max_{\pi} \ \mathbb{E}_{\tau\sim\pi} \left[R(\tau)\right]. \tag{2.2}$$

Different classes of RL algorithms vary in how this objective is achieved, either by estimating value functions, as in value-based methods, or by directly optimizing the policy, as in policy-gradient approaches.

Model-based vs Model-free RL

RL methods are commonly categorized into *model-based* and *model-free* approaches. Model-based methods [42, 39] aim to explicitly approximate the environment dynamics by learning the state transition function $\mathcal{P}$ and the reward function $\mathcal{R}$. When accurate models of these functions are available, the resulting knowledge can be exploited to compute optimal policies by solving the underlying MDP. However, in many real-world applications, accurately modeling complex, high-dimensional, or stochastic environments is challenging, which significantly limits the practical applicability of model-based approaches.

Model-free methods [39] address these limitations by learning decision-making strategies directly from interaction with the environment, without explicitly estimating $\mathcal{P}$ or $\mathcal{R}$. As a result, model-free approaches are widely adopted in practice and form the basis of many modern RL algorithms. These methods can be broadly divided into two main classes: value-based and policy-based approaches.

- Value-based methods, such as Q-learning [43] and Deep Q-Networks (DQN)[44], focus on estimating value functions that quantify the long-term utility of actions taken in given states. Optimal policies are then derived implicitly by selecting actions that maximize the learned value estimates.
- Policy-based methods, including Proximal Policy Optimization (PPO) [45], directly optimize a parameterized policy that defines a probability distribution over actions conditioned on the current state. These methods aim to maximize the expected return without relying on explicit action-value maximization.

Q-learning

Q-learning (QL) [43] is a fundamental value-based RL algorithm that aims to compute an optimal policy by learning the optimal action-value function through direct interaction with the environment, without requiring an explicit model of the system dynamics. Originally proposed by Watkins, QL belongs to the class of *off-policy* methods, as it estimates the optimal policy independently of the policy used to generate exploratory behavior during training.

The central quantity in Q-learning is the action-value function $Q(s, a)$, which represents the expected cumulative discounted return obtained by executing action a in state s and subsequently following the optimal policy. The learning objective is to approximate the optimal action-value function $Q^*(s, a)$, which satisfies the Bellman optimality equation, Equation 2.3. This equation recursively characterizes optimal behavior by decomposing the expected return into an immediate reward and the discounted value of successor states.

$$Q^*(s, a) = \mathbb{E}\left[r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q^*(s_{t+1}, a') \;\middle|\; s_t = s,\ a_t = a \right]. \tag{2.3}$$

In practice, QL does not compute Q^* explicitly. Instead, it incrementally approximates this fixed point using observed state transitions. The algorithm maintains a tabular representation of the action-value function, where each entry corresponds to a state-action pair. At each time step t, upon observing a transition $(s_t, a_t, r_{t+1}, s_{t+1})$, the Q-value is updated according to the temporal-difference learning rule

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha \Big[r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q_t(s_{t+1}, a') - Q_t(s_t, a_t) \Big], \tag{2.4}$$

where $\alpha \in (0, 1]$ denotes the learning rate and $\gamma \in [0, 1]$ is the discount factor. The term inside the brackets corresponds to the temporal-difference (TD) error, which measures the mismatch between the current estimate and the one-step Bellman target.

Under standard assumptions on the learning rate and sufficient exploration of the state-action space, QL is guaranteed to converge to the optimal action-value function Q^* for finite MDPs. An optimal policy π^* can then be obtained by selecting, in each state, the action that maximizes the learned Q-value,

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q^*(s, a). \tag{2.5}$$

Algorithm 1 summarizes the standard tabular Q-learning procedure used to iteratively approximate the optimal action-value function. During learning, the agent must balance between *exploration*, i.e., selecting actions to discover new state-action outcomes, and *exploitation*, i.e., selecting actions that are expected to yield high returns based on current value estimates. In this work, this trade-off is handled using an ϵ-greedy strategy with a decaying exploration rate, which encourages extensive exploration during early training while progressively favoring exploitation of the learned action-value estimates. The terminal flag `done` ensures proper episode termination, while the decay of ϵ enables a smooth transition from exploration to policy refinement as learning progresses. Although alternative exploration mechanisms exist, such as softmax action selection or upper-confidence-based methods, the ϵ-greedy strategy is adopted in this work due to its simplicity and widespread use in classical QL.

Despite its theoretical guarantees, classical Q-learning suffers from scalability limitations when applied to problems with large or continuous state and action spaces, as it relies on tabular representations of the action-value function. To illustrate this limitation, consider a problem in which the state is represented by a binary matrix of size $n \times f$, where each entry encodes the presence or absence of a given feature. The total number

Algorithm 1: Tabular Q-learning algorithm

Input: State space $\mathcal{S}$, action space $\mathcal{A}$, learning rate α, discount factor γ, initial exploration rate ϵ_0, minimum exploration rate $\epsilon_{\min}$, decay factor λ
Output: Approximate optimal action-value function Q
Algorithm:
Initialize $Q(s,a) \leftarrow 0$ for all $(s,a) \in \mathcal{S} \times \mathcal{A}$;
Initialize exploration rate $\epsilon \leftarrow \epsilon_0$;
for *each episode* **do**
  Initialize state s;
  Set `done` $\leftarrow$ `false`;
  Set $t \leftarrow 0$;
  while *done* $=$ *false* ***and*** $t <$ *cutoff* **do**
    Select action a according to ϵ-greedy policy:

$$a = \begin{cases} \text{random action from } \mathcal{A}, & \text{with probability } \epsilon, \\ \arg\max_{a' \in \mathcal{A}} Q(s,a'), & \text{otherwise;} \end{cases}$$

    Execute action a, observe reward r, next state s', and terminal flag `done`;
    Update the action-value function:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma \max_{a' \in \mathcal{A}} Q(s',a') - Q(s,a)\big)$$

    Set $s \leftarrow s'$;
    Set $t \leftarrow t+1$;
  end
  Update exploration rate:

$$\epsilon \leftarrow \max(\epsilon_{\min}, \lambda\epsilon)$$

end

of possible states is then 2^{nf}. Even for moderate values of n and f, the state space grows exponentially, making it infeasible to store or update a Q-table containing one value for each state-action pair. In such scenarios, classical tabular QL becomes computationally impractical, as both memory requirements and sample complexity quickly exceed realistic limits. This exponential growth of the state space, commonly referred to as the *curse of dimensionality*, motivates the use of function approximation techniques, such as deep neural networks, to generalize value estimates across similar states. Such approaches form the basis of deep Q-learning, which is discussed in the next subsection.

Deep Q-learning

Deep Q-learning (DQN) [44] extends classical QL to problems with large or high-dimensional state spaces by replacing the tabular representation of the action-value function with a parameterized function approximator, typically a neural network (NNs) [46].

Instead of storing a distinct Q-value for each state-action pair, DQN approximates the action-value function as $Q(s, a; \theta)$, where θ denotes the network parameters. This formulation enables generalization across states and alleviates the scalability limitations inherent to tabular QL.

The learning objective in DQN remains unchanged: the goal is to approximate the optimal action-value function that satisfies the Bellman optimality equation. However, the use of nonlinear function approximation introduces significant stability challenges, as naively applying the QL update with a neural network can lead to divergence due to correlated samples and non-stationary targets.

To address these issues, DQN introduces two key mechanisms. First, an *experience replay* buffer is employed to store past transitions, from which mini-batches are sampled uniformly during training. This breaks temporal correlations between consecutive updates and improves data efficiency. Second, a *target network* with parameters θ^- is used to compute the target Q-values, while the main network parameters (online network) θ are updated. The target network parameters are periodically synchronized with the online network, thereby stabilizing the learning process. Algorithm 2 summarizes the main steps of the DQN approach.

Formally, the network parameters are optimized by minimizing the mean-squared temporal-difference loss

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')}\left[\left(r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right], \tag{2.6}$$

where the target values are computed using the fixed target network. Action selection during training typically follows the same exploration strategy adopted in tabular QL, such as an ϵ-greedy policy, while the learned network implicitly captures complex state representations.

By combining value-based learning with deep function approximation, DQN enables QL to scale to problems with large and continuous state spaces. Nevertheless, DQN remains fundamentally a value-based method and inherits some limitations of QL, including sensitivity to hyperparameters and challenges in environments with highly stochastic dynamics or large action spaces. These limitations have motivated the development of alternative approaches, including policy-gradient and actor-critic methods, which are discussed in the following sections.

Algorithm 2: Deep Q-Network (DQN) algorithm

Input: Action space $\mathcal{A}$, discount factor γ, learning rate α, replay capacity N, minibatch size B, target update period C, maximum episodes M, maximum steps per episode T, initial exploration rate ϵ_0, minimum exploration rate $\epsilon_{\min}$, decay factor λ

Output: Online network parameters θ approximating $Q(s, a; \theta)$

Initialize replay buffer $\mathcal{D}$ with capacity N;
Initialize online network parameters θ randomly;
Initialize target network parameters $\theta^- \leftarrow \theta$;
Initialize exploration rate $\epsilon \leftarrow \epsilon_0$;
Initialize global step counter $g \leftarrow 0$;
for *episode* $= 1$ **to** M **do**
  Reset environment and observe initial state s;
  Set `done` $\leftarrow$ `false` and $t \leftarrow 0$;
  while *`done`* $=$ *`false`* ***and*** $t < T$ **do**
    Select action a according to ϵ-greedy policy:

$$a = \begin{cases} \text{random action from } \mathcal{A}, & \text{with probability } \epsilon, \\ \arg\max_{a' \in \mathcal{A}} Q(s, a'; \theta), & \text{otherwise.} \end{cases}$$

    Execute action a, observe reward r, next state s', and terminal flag `done`;
    Store transition $(s, a, r, s', \texttt{done})$ in $\mathcal{D}$;
    Set $s \leftarrow s'$;
    if $|\mathcal{D}| \geq B$ **then**
      Sample a minibatch $\{(s_i, a_i, r_i, s'_i, \texttt{done}_i)\}_{i=1}^{B}$ uniformly from $\mathcal{D}$;
      Compute targets:

$$y_i = \begin{cases} r_i, & \text{if } \texttt{done}_i = \texttt{true}, \\ r_i + \gamma \max_{a' \in \mathcal{A}} Q(s'_i, a'; \theta^-), & \text{otherwise;} \end{cases}$$

      Minimize the mean-squared TD loss:

$$\mathcal{L}(\theta) = \frac{1}{B} \sum_{i=1}^{B} \left(y_i - Q(s_i, a_i; \theta) \right)^2$$

      and update θ using stochastic gradient descent with step size α;
    end
    if $g \bmod C = 0$ **then**
      Update target network: $\theta^- \leftarrow \theta$;
    end
    Update exploration rate:

$$\epsilon \leftarrow \max(\epsilon_{\min}, \lambda \epsilon)$$

    Set $t \leftarrow t + 1$ and $g \leftarrow g + 1$;
  end
end

2.2.3 Graph Neural Networks

Graph-structured data arise naturally in numerous domains where entities and their pairwise relationships must be explicitly modeled, including social and communication networks, molecular interaction graphs, transportation systems, and large-scale infrastructure networks. In such contexts, the relational structure among entities is fundamental to understanding system behavior. However, classical deep learning architectures, including convolutional neural networks (CNNs) and recurrent neural networks (RNNs), are designed for data defined on regular Euclidean domains, including grids and sequences. These models inherently assume fixed-size inputs and structured neighborhoods, making them ill-suited to represent the irregular connectivity, variable size, and permutation invariance properties that characterize graph data.

Graph Neural Networks (GNNs) [47, 48] were introduced to overcome these limitations by enabling learning directly on graph-structured inputs. At a high level, GNNs learn latent representations (embeddings) for nodes, edges, or entire graphs by leveraging the relational information encoded in the graph topology [49, 50]. These learned embeddings can then be used for downstream tasks such as node classification, link prediction, or graph-level inference.

Formally, given a graph $G = (V, E)$, where V denotes the set of nodes and $E \subseteq V \times V$ represents the set of edges, a GNN constructs vector representations for graph elements by explicitly exploiting the structural dependencies defined by E. Unlike multilayer perceptrons (MLPs) or CNNs, which operate on fixed-dimensional Euclidean inputs, GNNs are tailored to non-Euclidean domains in which the number of nodes, their connectivity patterns, and their ordering may vary across instances [51]. By incorporating structural inductive biases that respect permutation invariance and locality, GNNs enable principled learning over variable-size relational systems.

The core computational paradigm underlying most GNN architectures is the *message passing* mechanism. In this framework, node representations are computed through iterative exchanges of information between neighboring nodes. Each node maintains a hidden representation that summarizes both its intrinsic features and the information received from its adjacent nodes. At each message-passing layer, corresponding to one iteration of neighborhood information exchange, every node aggregates the messages received from its neighbors using a permutation-invariant function, such as summation, averaging, or maximization, and then updates its internal representation through a learnable transformation. After L layers, the embedding of a node incorporates structural information from nodes up to L hops away. This iterative message passing process enables the network to capture local connectivity patterns while progressively integrating broader topological context.

Several GNN variants have been proposed, differing mainly in their aggregation and update mechanisms. Graph Convolutional Networks (GCN) [48] introduce a normalized convolution operator inspired by spectral graph theory. Graph Attention Networks (GAT) [52] incorporate attention mechanisms that assign learnable importance weights to neighboring nodes. GraphSAGE [53] adopts an inductive neighborhood sampling strategy to improve scalability to large graphs. Graph Isomorphism Networks (GIN) [54] employ injective aggregation functions to enhance representational expressiveness.

Importantly, the Message Passing Neural Network (MPNN) framework [55] provides a unified formulation that generalizes most of these architectures. In this view, GCN, GAT, GraphSAGE, and GIN can be interpreted as specific instantiations of the same message passing principle with different choices of aggregation and update functions. The MPNN formulation, therefore, constitutes a foundational milestone in the development of graph neural architectures and serves for developing the models adopted in this thesis.

Message Passing Neural Network

The Message Passing Neural Network (MPNN) framework, introduced by Gilmer et al. [55], provides a general and unified formulation of graph neural architectures. At its core, an MPNN computes node representations through an iterative message passing mechanism, whereby nodes exchange information with their neighbors in order to progressively encode local and multi-hop structural dependencies in a principled manner.

Let $G = (V, E)$ be a graph where each node $v \in V$ is associated with an initial feature vector $\mathbf{x}_v \in \mathbb{R}^F$, and optionally each edge $(u, v) \in E$ is associated with an edge feature vector $\mathbf{e}_{uv}$. The MPNN operates in two main stages: a message passing phase performed over L layers, followed by an optional readout phase that produces a graph-level representation. The functioning of the MPNN can be described as follows.

1. **Initialization**:
   Each node v is first associated with a feature vector $\mathbf{x}_v$, whose components are selected according to the information relevant to the problem under consideration. These features summarize the attributes that characterize the role or state of the node in the system. For example, in a social network graph, a node may represent a person whose feature vector includes attributes such as age, profession, or interests. These features describe the individual independently of their social connections. Similarly, in a communication network, a node may represent a router characterized by attributes such as its current load, available capacity, or degree (i.e., number of directly connected neighbors). In both cases, these initial features describe each node before relational information from neighboring nodes is incorporated through the message-passing process.

   The initial embedding of each node is then obtained through a non-linear transformation function $I(\cdot)$:
   $$\mathbf{h}_v^{(0)} = I(\mathbf{x}_v), \quad \forall v \in V. \tag{2.7}$$

2. **Message Passing Phase:**
   The message passing phase constitutes the core mechanism of the MPNN. At each layer, the embedding of every node is updated by receiving, aggregating, and transforming information from its neighboring nodes. Through this iterative procedure, node representations progressively integrate structural context. Figure 2.4 illustrates this process for a target node, showing how information from its local neighborhood is aggregated and then used to update its embedding. For each layer $l = 0, \ldots, L-1$, information propagation along the edges of the graph is performed through three successive steps: message computation, aggregation, and update.

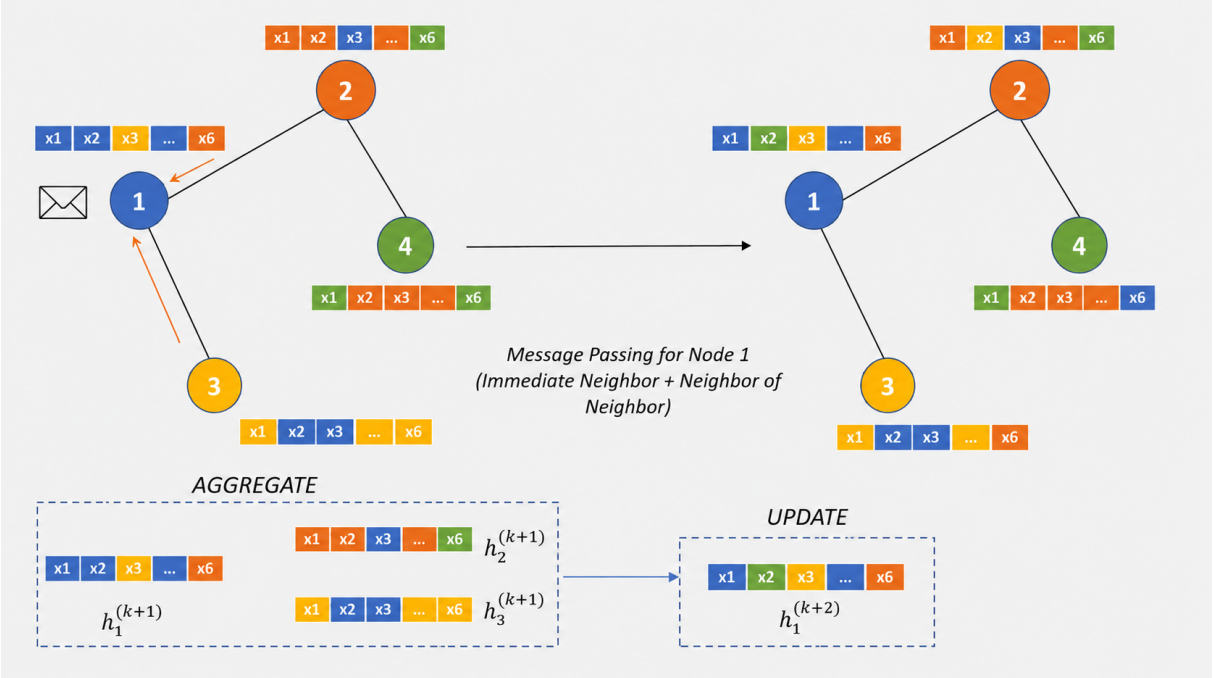


Figure 2.4 – Message-passing mechanism in an MPNN for updating a target node representation.

- **Message computation**: In this step, each neighbor u of node v computes a message that reflects the information it sends to v. Formally,

$$\mathbf{m}_{u\to v}^{(k)} = M^{(k)}\big(\mathbf{h}_u^{(k)}, \mathbf{h}_v^{(k)}, \mathbf{e}_{uv}\big), \tag{2.8}$$

  where $M^{(l)}(\cdot)$ is a learnable function that defines how node u contributes information to node v at layer k.
- **Aggregation:** Once messages from all neighboring nodes are computed, node v aggregates them into a single summarized message:

$$\mathbf{m}_v^{(k)} = \mathrm{AGG}^{(k)}\left(\{\mathbf{m}_{u\to v}^{(k)} : u \in \mathcal{N}(v)\}\right), \tag{2.9}$$

  where $\mathcal{N}(v)$ denotes the set of neighbors of node v, and $\mathrm{AGG}^{(l)}(\cdot)$ is a permutation-invariant operator (e.g., sum, mean, or max). The permutation invariance property ensures that the representation does not depend on the ordering of neighboring nodes.
- **Update:** Finally, node v updates its embedding by combining its previous representation with the aggregated message:

$$\mathbf{h}_v^{(k+1)} = U^{(k)}\big(\mathbf{h}_v^{(k)}, \mathbf{m}_v^{(k)}\big), \tag{2.10}$$

  where $U^{(k)}(\cdot)$ is a learnable update function, typically implemented as a multilayer perceptron.

After L layers, the node embedding $\mathbf{h}_v^{(L)}$ incorporates structural information from nodes up to L hops away in the graph. The depth L therefore controls the receptive field of each node representation.

3. **Readout Phase :**
When a graph-level representation is required, a permutation-invariant readout function aggregates node embeddings:

$$\mathbf{h}_G = R\left(\left\{\mathbf{h}_v^{(L)} : v \in V\right\}\right),$$

where $R(\cdot)$ may be a summation, mean pooling, or a learnable aggregation mechanism.

The MPNN formulation generalizes a broad family of GNN architectures. In particular, GCN, GAT, GraphSAGE, and GIN can all be interpreted as specific instantiations obtained through different choices of the message function M, the aggregation operator AGG, and the update function U.

2.3 ACL Optimization and Placement Approaches

ACL management addresses a core bottleneck: the mismatch between (i) the volume and dynamism of per-flow or fine-grained policy rules and (ii) the small capacity of fast hardware tables [56]. Traditional approaches, where each switch maintains a complete copy of the ACL, quickly become infeasible in large-scale and dynamic environments due to memory constraints and energy consumption issues. From there, the research community has proposed several strategies to optimize the utilization of TCAM resources. These approaches can be broadly classified into three main categories [24], which have emerged progressively over time:

1. *Rule compression:* reduces the number of entries by merging or minimizing rules while preserving semantics.
2. *Rule eviction:* accepts that the table cannot hold everything; it keeps "valuable" entries resident and removes or offloads others as traffic changes.
3. *Rule splitting and distribution:* treats the network as a distributed enforcement substrate: a single switch need not hold all ACL logic, as long as packets traverse the necessary "pieces" somewhere across the topology.

In the following, we review these approaches, highlighting their key contributions, strengths, and limitations. This analysis will motivate the adoption of rule splitting and distribution as the most promising direction for scalable ACL management.

2.3.1 Rule Compression

Rule compression, also referred to as aggregation or classifier minimization, constitutes one of the earliest approaches proposed to address TCAM capacity limitations. The main objective is to reduce the number of rules stored in ACL table while preserving the exact semantics implemented in it. This is achieved by exploiting redundancies among rules and

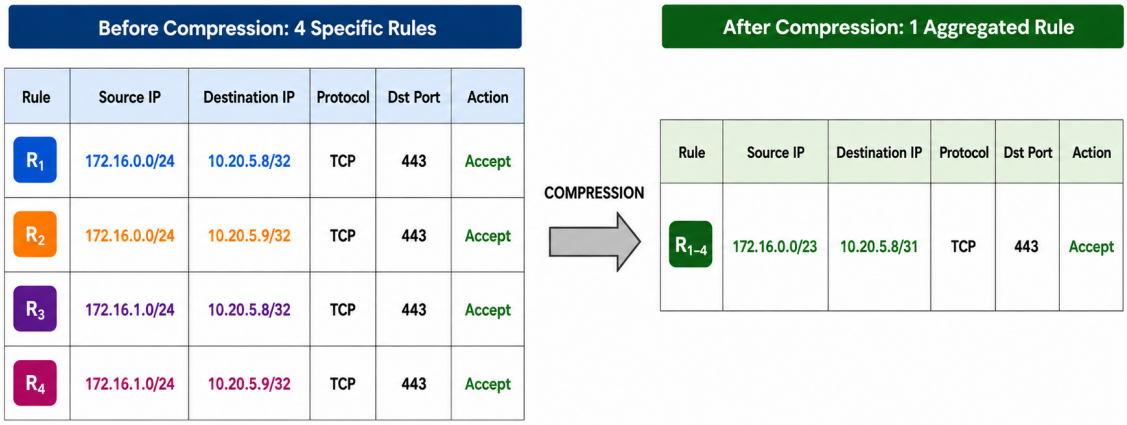


Figure 2.5 – Example of ACL rule compression through prefix aggregation.

replacing multiple specific entries with more general wildcard rules that cover equivalent traffic patterns, as illustrated in Figure 2.5.

A first line of work focuses on *range encoding techniques*, which aim to mitigate the rule expansion problem caused by representing multi-dimensional packet header fields (e.g., IP addresses and port ranges) in TCAM. In these approaches, rule matching patterns are typically represented as bit vectors within a multidimensional space, where each dimension corresponds to a header field [57, 25]. Early contributions, such as the scheme proposed by Liu [58], associate each rule with an additional key vector to encode ranges more compactly, at the expense of extra storage bits. Subsequently, Bremler-Barr *et al.* [59] introduced SRGE, a Gray-code-based encoding technique that bounds the number of prefixes required to represent a range, thereby improving worst-case expansion compared to naive prefix-based methods. At a higher level, topological transformation approaches [60] reorganize the rule space through domain compression and prefix alignment to eliminate redundancies prior to TCAM mapping. More recently, Abboud *et al.* [61] proposed the Double Mask representation, which generalizes CIDR encoding to efficiently handle complex ranges with exceptions, achieving significant compression gains on real-world ACL datasets.

A second line of research addresses compression at the *classifier level*, where the ACL is modeled as a prioritized Boolean classifier and minimized accordingly. Early works by Dong *et al.* [62] and Liu *et al.* [63] leverage rule trimming, merging, and decision-diagram-based techniques to eliminate redundant or shadowed rules while preserving policy correctness. Similarly, TCAM-specific optimizations such as TCAM Razor [64] and Bit Weaving [13] exploit the ternary nature of TCAM to merge entries that differ only in a limited number of bits, thereby reducing the overall rule count without requiring hardware modifications. Complementary approaches, such as the geometric formulation introduced by Applegate *et al.* [65], model rule spaces as multidimensional regions and perform minimization through geometric merging operations.

With the advent of Software Defined Networking (SDN), compression techniques have been extended to dynamic and programmable environments. In this context, approaches such as MINNIE [66] and incremental aggregation methods [67] dynamically compress flow

tables by merging compatible rules while preserving forwarding semantics during network operation. Additional works, including wildcard-based compression strategies [68], further enhance storage efficiency by exploiting the flexibility of ternary matching in OpenFlow based switches.

Overall, compression-based approaches provide an effective means of mitigating TCAM capacity limitations by reducing the number of rules while preserving policy semantics. They are particularly attractive due to their compatibility with existing hardware and their ability to operate without modifying the network architecture. Despite these advantages, compression techniques remain inherently limited in addressing the scalability challenges of modern SDN environments. As highlighted in the literature [56, 24], even after aggressive minimization, switches with constrained TCAM capacities may still be unable to accommodate all compressed rules, particularly in large-scale ACLs or multi-tenant scenarios. Furthermore, compression can negatively impact per-flow visibility. By aggregating multiple flows into generalized wildcard rules, these approaches make it more difficult to perform fine-grained monitoring, accounting, or rate-limiting, as operations on individual flows may inadvertently affect others. This limitation is particularly critical in security-sensitive environments where precise control over traffic is required. From a practical perspective, compression techniques often rely on complex preprocessing steps and may require recomputation when policies evolve, thereby limiting their efficiency in dynamic network conditions [67]. In addition, compression alone cannot address scenarios where rules must be enforced across multiple network paths, nor situations in which switch capacities remain insufficient even after minimization, thus highlighting the need for complementary strategies [56]. Consequently, while compression improves memory utilization, it does not fully resolve the challenges of scalability and adaptability in modern networks. To overcome these limitations, subsequent research has explored more dynamic approaches, notably rule eviction and caching strategies, which are discussed in the following section.

2.3.2 Rule Eviction

Eviction-based approaches address the scalability limitations of ACL tables by dynamically removing or offloading selected rules from constrained high-speed memory to auxiliary storage or secondary processing layers. Unlike compression techniques, which seek to reduce the size of the rule set itself, eviction assumes that the overall policy remains inherently large and instead focuses on managing scarce hardware resources through selective rule residency. In practice, eviction decisions are commonly driven by traffic statistics, whereby infrequently matched rules are removed from the primary ACL table and handled through alternative mechanisms such as software processing or secondary lookup structures. This allows frequently accessed rules to remain in fast memory, thereby preserving high performance for dominant traffic patterns under strict capacity constraints, as illustrated in Figure 2.6..

In the literature, this concept has been implemented through several classes of mechanisms. Classical replacement strategies include Least Recently Used (LRU), Least Frequently Used (LFU), First In First Out (FIFO), and random replacement, which respectively evict rules according to recency of access, match frequency, insertion order,

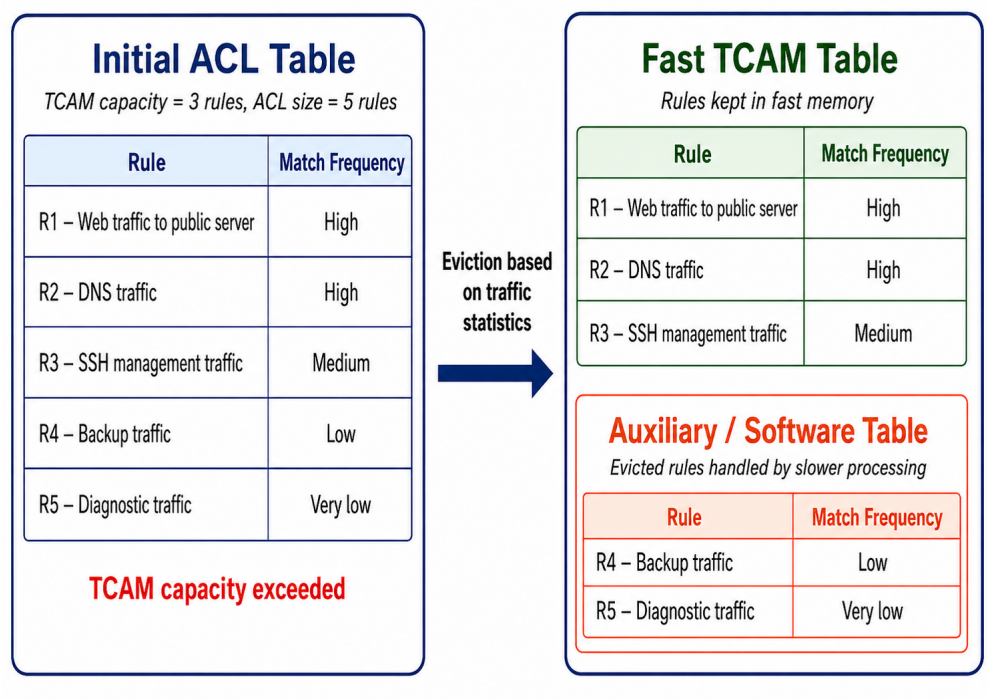


Figure 2.6 – Example of eviction-based ACL management.

or probabilistic selection [69, 70]. Other approaches rely on state-based eviction, where traffic behavior or protocol state is exploited to identify inactive or low-priority rules [71]. Timeout-based mechanisms have also been proposed, in which rules are removed after a fixed or adaptive idle period in order to reclaim memory resources dynamically [72, 73, 74, 75]. Although these strategies differ in their decision criteria, they share the common objective of ensuring that frequently used or operationally critical rules remain resident in the high-speed table, while less relevant entries are progressively displaced.

Compared with compression techniques, eviction offers several practical advantages. First, it does not attempt to rewrite, merge, or transform existing rules. Consequently, it avoids the semantic complexity and verification challenges often associated with compression, while preserving rule ordering, priority relationships, and conflict semantics. This significantly reduces the risk of unintended policy modifications. Second, eviction directly targets the fundamental hardware limitation of restricted fast-memory capacity. Whereas the benefits of compression are constrained by the structural redundancy present in the rule set, eviction exploits traffic skewness, which is often substantially stronger than rule redundancy in real deployments. As a result, significant reductions in active table occupancy can be achieved even when opportunities for rule aggregation are limited. In this sense, eviction constitutes a pragmatic and operationally attractive response to the scalability pressures imposed by large firewall ACLs.

Nevertheless, eviction-based ACL management remains subject to several important limitations. Its effectiveness depends heavily on skewed traffic distributions; when traffic is more uniformly distributed across rules, the gains become marginal and worst-case scenarios may require most entries to remain resident in fast memory. In addition, many

eviction strategies are inherently reactive, relying on historical counters or recent observations. Sudden shifts in traffic patterns may therefore cause temporary degradation until the system adapts and those approaches can be expensive and not easy to implement, because of large signaling overhead [24]. Eviction also introduces multi-stage processing pipelines, since displaced rules must still be enforced through fallback mechanisms such as software processing or secondary lookup structures. This increases architectural complexity and may introduce additional latency for affected packets [76]. More importantly, eviction does not fundamentally reduce the total size of the ACL policy. The complete rule set must still be stored, maintained, and consistently managed elsewhere. As enterprise and data-center policies continue to grow in scale and heterogeneity, merely relocating rules does not resolve the underlying structural scalability problem. Finally, these approaches remain essentially local, as they optimize memory usage within individual devices without addressing how policies should be enforced cooperatively across multiple network elements. Consequently, while eviction techniques improve memory efficiency and runtime adaptability, they do not fully solve the scalability and correctness challenges of ACL management. These limitations have motivated the development of more structural solutions, notably rule splitting and distribution approaches, which are discussed in the following section.

2.3.3 Split and Distribution

Rule splitting and distribution approaches constitute a more structural response to the scalability limitations of large ACL tables. Instead of attempting to fit the complete rule set into a single device through minimization or dynamically managing a constrained local table through eviction, these methods exploit the distributed nature of modern networks by partitioning the policy into smaller components and enforcing them cooperatively across multiple nodes. The central idea is that a packet does not need to encounter the entire ACL at a single switch, provided that the sequence of switches traversed along its path collectively applies all required rules while preserving the semantics of the original policy [24]. In this paradigm, the network itself becomes a distributed enforcement substrate, where memory resources are aggregated across several devices rather than concentrated in a single table, as illustrated in Figure 2.7.

The literature generally distinguishes two complementary stages within this family of approaches: rule decomposition and rule distribution. Rule decomposition consists of transforming an initial ACL into smaller, semantically equivalent subsets or rule fragments that can be deployed independently while preserving the global behavior of the original policy. Several decomposition strategies have been proposed in the literature. One of the earliest and most influential concepts is the One Big Switch (OBS) abstraction, which models the network as a single logical switch and maps global policies into local forwarding and filtering rules distributed across physical devices while preserving centralized semantics [57]. Building on this principle, later works investigated more explicit classifier partitioning mechanisms. In particular, Palette introduced two decomposition methods for transforming an original rule table into multiple subtables [25]. The first, Pivot Bit Decomposition (PBD), recursively partitions rules by replacing wildcard bits with fixed values, whereas the second, Cut-Based Decomposition (CBD), relies on a directed dependency

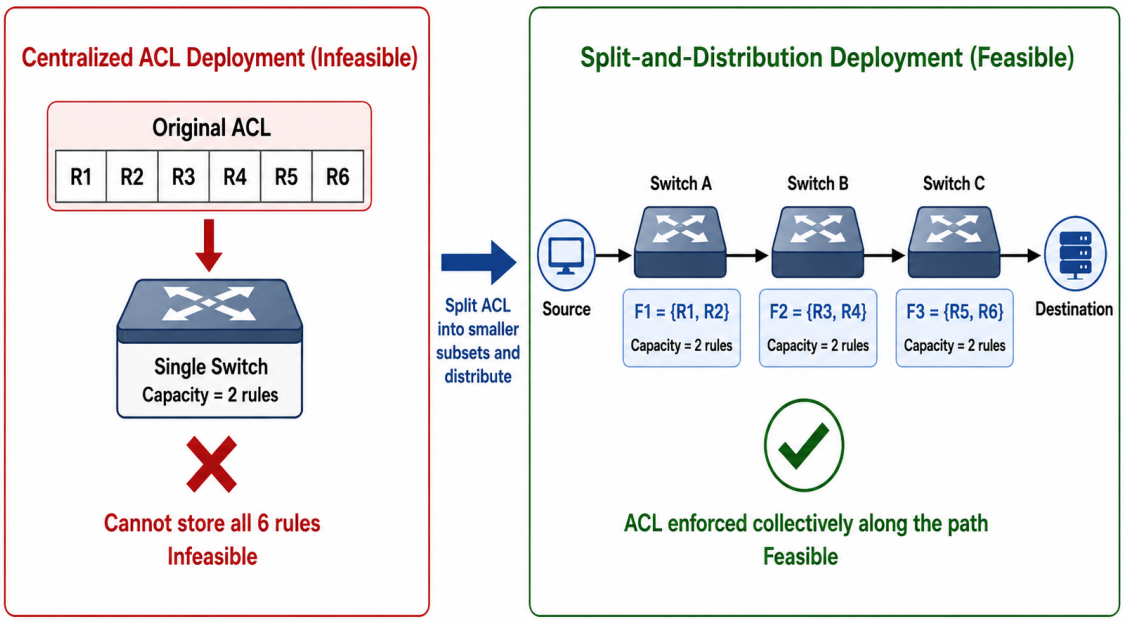


Figure 2.7 – Example of split-and-distribution based ACL placement

graph whose vertices represent rules and whose edges capture dependency relationships; the rule set is then divided by cutting selected dependencies. Additional techniques based on classifier transformation and topological reorganization further improved the ability to partition large rule sets efficiently [60]. More recently, Abboud et al. proposed the Double Mask representation and related decomposition mechanisms tailored to ACL structures under constrained memory budgets, showing that careful preprocessing can substantially reduce the size and complexity of the resulting rule subsets [61, 56]. Although decomposition is an important prerequisite, it remains only the first stage of the overall problem. Once the original rule set has been partitioned, the remaining challenge is to determine where each resulting component should be installed within the network so as to satisfy memory and policy constraints.

The second stage, namely rule distribution or placement, concerns the allocation of decomposed rules over network nodes under operational constraints. This problem has received increasing attention with the emergence of programmable and centrally managed networks. Nguyen et al. [24] identified rule placement as a fundamental optimization problem and highlighted its importance for routing, access control, traffic engineering, and middlebox deployment. Other works have explored policy steering and distributed enforcement mechanisms in which packet processing functions are executed across multiple devices rather than concentrated at a single point. For example, SIMPLE [77] showed how centralized control can coordinate the placement of network functions along traffic paths, while FlowTags [78] demonstrated consistent policy enforcement across distributed middleboxes.

Compared with compression and eviction, split-and-distribution approaches offer several decisive advantages. First, they directly address the root cause of the scalability prob-

lem, namely the mismatch between growing ACL sizes and the limited memory capacity of individual switches. Rather than attempting to fit a large policy into a constrained local table, they exploit the aggregate memory resources available across the network [24, 56]. Second, unlike compression methods, their effectiveness does not depend on structural redundancies or wildcard aggregation opportunities within the rule set. Even highly heterogeneous or weakly compressible ACLs can be efficiently deployed through distribution [25, 61]. Third, unlike eviction strategies, they do not rely on traffic skewness or reactive replacement mechanisms. Consequently, they avoid cache misses, temporary policy absence, and performance instability caused by changing traffic conditions [76, 79]. Fourth, these approaches preserve deterministic and complete policy enforcement, which is particularly important in security-sensitive ACL scenarios where every packet must be evaluated correctly along its path [78, 77]. For these reasons, rule splitting and distribution are widely regarded as one of the most promising directions for scalable ACL management in large and dynamic networks [24, 56].

The practical relevance of this approach has been considerably reinforced by recent advances in Software-Defined Networking (SDN) [80, 36]. Traditional distributed firewall deployment was historically difficult because each device had to be configured independently, often with limited visibility of the global topology and traffic behavior. By contrast, SDN introduces a logically centralized control plane with a global network view, enabling coordinated rule installation across multiple switches, rapid policy updates, topology-aware decisions, and automated adaptation to routing changes [81, 36]. This programmability transforms distributed ACL enforcement from a largely theoretical concept into an operationally feasible solution. In addition, SDN controllers provide the computational capabilities required to execute optimization algorithms, heuristics, and machine-learning-based methods for deciding how rules should be placed under evolving constraints [82].

Approach	Definition	Techniques	Main advantages	Main limitations
Rule compression	Reduces the number of ACL entries by merging, encoding, or minimizing semantically equivalent rules.	Range encoding [58], SRGE [59], Double Mask [61], TCAM Razor [64], Bit Weaving [13], MINNIE [66], and incremental wildcard aggregation approaches.	- Hardware-compatible; no topology changes required. - Preserves a single enforcement point. - Effective when structural redundancy is high.	- Effectiveness depends on rule redundancy. - Cannot resolve exhaustion when compressed rules still exceed TCAM capacity. - Degrades per-flow visibility and fine-grained monitoring.

Approach	Definition	Techniques	Main advantages	Main limitations
Rule eviction	Maintains only selected rules in fast memory and offloads less useful entries to auxiliary or slower processing structures.	LRU/LFU/ FIFO-based replacement [69, 70], state-based eviction [71], timeout-based mechanisms [72, 73, 74, 75], and hybrid hardware/software table approaches.	- Preserves rule ordering and conflict semantics without rewriting rules. - Exploits traffic skewness to reduce active table occupancy. - Low verification overhead compared to compression.	- Effectiveness depends heavily on skewed traffic; degrades under uniform distributions. - Reactive: vulnerable to sudden traffic shifts and cache misses. - Introduces multi-stage pipelines and additional latency.
Split and distribution	Partitions the ACL into smaller rule subsets distributed across multiple switches, so that packets encounter all relevant rules along their traversal path.	OBS [57], Palette [25], [83], Nguyen et al. [24], SIMPLE [77], and FlowTags [78].	- Directly addresses the root cause by aggregating memory across the network. - Independent of structural redundancy. - Does not rely on traffic skewness or reactive replacement. - Preserves deterministic and complete policy enforcement. - Naturally supported by SDN centralized control.	Requires solving a non-trivial placement problem to preserve policy correctness under memory, routing, and dependency constraints.

Table 2.2 – Comparative summary of the three main families of approaches to ACL memory optimisation in TCAM-based networks.

From an optimization perspective, once a rule set has been decomposed, the distribution problem can be formalized as deciding which rule subset should be installed on which node so that several constraints are simultaneously satisfied. These constraints typically include memory limitations at each switch, complete policy coverage along admissible paths, rule ordering dependencies, consistency constraints, and minimization of redundant placements. Depending on the operational context, additional requirements may also arise, such as resilience to routing changes, path selection constraints, bandwidth-

awareness, or bounded enforcement distance [24]. The objective is therefore not merely to place rules, but to compute an efficient global allocation that preserves policy correctness while minimizing resource consumption.

Table 2.2 summarizes the three principal families of approaches discussed in this section, highlighting their underlying principles, representative works, main advantages, and limitations. Among them, split-and-distribution emerges as the most structurally suitable direction for scalable ACL management, while introducing the non-trivial placement problem examined in the following section.

2.4 Distributed ACL Placement Approaches

Distributed ACL placement refers to the problem of taking an already decomposed security policy, namely a collection of rule subsets that collectively preserve the semantics of the original ACL, and determining on which network nodes these subsets should be installed so that all relevant traffic paths are correctly filtered while respecting switch memory capacities and inter-rule dependencies. This class of approaches emerged initially as a response to the limitations of centralized ACL deployment and later gained further importance as an effective alternative to the shortcomings of compression and eviction techniques. In general, the rule distribution problem is formulated as a constrained optimization model whose purpose is to decide which rules should be installed on which switches while satisfying policy correctness, memory limitations, and dependency constraints among rules. The optimization objectives may vary according to the targeted application. Typical goals include minimizing the total number of installed rules [57, 84, 85], reducing energy consumption [86], minimizing wasted bandwidth through early filtering [87], or maximizing traffic satisfaction and service performance [88]. Existing approaches to this problem can be broadly grouped into three families: heuristic and graph-based methods, exact optimization methods based on Integer Linear Programming (ILP), and more recent learning-based approaches.

2.4.1 Heuristic, Graph-Based, and Architectural Approaches

Heuristic and graph-based methods constitute the earliest and historically most influential family of approaches for distributed ACL rule placement. Their emergence is motivated by a practical observation: the placement of filtering rules over multiple switches under memory constraints rapidly becomes combinatorial (NP-hard), making exact optimization is difficult to apply at realistic scales. Consequently, many researchers proposed decomposition strategies, greedy algorithms, path-cover formulations, or topology-aware approximations capable of producing feasible solutions within short execution times. These methods established several foundational concepts still used today, including rule decomposition, path coverage, switch load balancing, and distributed utilization of network-wide TCAM resources. However, their computational efficiency is generally achieved by embedding restrictive assumptions regarding routing models, topology structure, path selection, or optimization objectives, which later become major limitations when moving toward dynamic and large-scale environments [24].

Among the earliest representative frameworks, OBS (One Big Switch) proposed by Kang et al. [57], introduced a highly influential abstraction in which the entire network is viewed as a logically centralized switch. Within this paradigm, policy specification, routing decisions, and rule compilation are separated into distinct stages, after which policies are distributed across physical switches while respecting rule-space limitations. A major contribution of OBS lies in showing that rule placement can be interpreted as a geometric covering and packing problem, enabling efficient heuristics for TCAM allocation. The framework also considers rule priorities and permits rule migration along paths to rebalance memory usage. Despite its conceptual importance, OBS presents several limitations. First, traffic flows are associated with predefined paths, which significantly reduces robustness under routing changes or multipath forwarding. Second, resource allocation is performed path by path, which may reserve memory for rarely used paths while neglecting globally efficient placements. Third, policies may be replicated across all selected paths, generating unnecessary rule duplication and increasing TCAM pressure. Finally, although effective for forwarding-rule compilation, the framework was not specifically designed for ACL filtering semantics under strict memory constraints, where path coverage guarantees and selective rule deployment become central concerns. A second milestone is Palette, introduced by Kanizo *et al.* [25], which extended rule distribution beyond restricted topological settings and modeled the problem through graph-theoretic formulations. Palette demonstrated that distributed switch memories can be exploited jointly by combining rule decomposition with path-coverage reasoning over arbitrary SDN topologies. Its main strength lies in its simplicity, relatively low computational overhead, and practical ability to leverage aggregate network memory without relying on expensive exact solvers. Nevertheless, Palette remains fundamentally greedy. Decisions are made according to immediate coverage gains, without anticipating future congestion of strategic nodes such as articulation points or major routing junctions. Under tight capacities, this often leads to premature saturation of critical switches and incomplete final placements. In addition, the framework was not conceived to integrate richer operational constraints such as rule ordering, traffic-aware placement, bandwidth preservation, or uncertainty over future routing paths. As a result, its effectiveness decreases when placement objectives move beyond simple path coverage.

A second family of works approached the problem from a system-design perspective, focusing primarily on switch-table scarcity and control-plane scalability rather than explicit ACL placement. DIFANE [23] delegates rule ownership to authority switches in order to reduce controller involvement. DevoFlow [89] minimizes controller interactions by handling micro-flows directly in the data plane. vCRIB [90] virtualizes rule management in cloud environments, while Infinite CacheFlow [91] explores remote rule caching to extend the effective capacity of flow tables. These systems are important because they established that TCAM scarcity is a structural bottleneck in SDN and that distributing rule ownership can substantially improve scalability. Their main strength is therefore architectural: they reduce pressure on individual switches and lower signaling overhead. However, they are not designed to solve the distributed ACL placement problem itself. In particular, they do not determine how pre-decomposed filtering rules should be allocated so that every relevant traffic path intersects a valid enforcement point, nor do they preserve the semantic dependencies between distributed ACL fragments.

The literature later diversified into specialized heuristics tailored to particular deployment contexts. MoRule [92] addresses mobility-aware placement for SDN-enabled access networks by minimizing rule-space occupation while maintaining a threshold of local processing for mobile users. Huang et al. [93] proposed a partition-and-allocation strategy for heterogeneous TCAM capacities, ensuring semantic preservation across switches with different memory sizes. Angelos et al. [94] developed a dynamic placement mechanism for hybrid hardware/software tables, increasing software-table utilization while controlling delay and packet loss. OPTree [95] introduced a structural data model that exploits serial and parallel relationships between neighboring devices when placing rules. The value of these studies lies in their practical realism: they explicitly address mobility, heterogeneous hardware, hybrid memories, or local structural dependencies that are often ignored in abstract formulations. However, their common weakness is specialization. Each method is strongly tuned to a particular operational setting and objective, making transferability difficult when multiple constraints must be handled simultaneously or when the topology changes significantly.

A related research direction concerns service-function and middlebox placement. Although these works do not directly target ACL distribution, they remain relevant because they also seek to position security functions within a network graph. Controller-placement studies and service-function-chain placement works have shown that topology structure, redundancy, and path diversity strongly influence placement quality [96, 97, 98, 99, 100]. Their main strength is to highlight the importance of topology-aware reasoning in network placement problems. However, they operate at a coarser granularity by placing complete functions such as firewalls or controllers rather than distributing the rules of a single ACL over multiple switches. Consequently, they do not address the finer challenge of simultaneously satisfying switch memory limits, path-wise filtering coverage, and rule-order semantics.

Other works explored hybrid rule-management architectures. Wang et al. [101] classify rules into path-based and node-based categories, pre-install path-based rules, and manage node-based chunks reactively through lazy updates. This contribution is valuable because it recognizes that distinct classes of rules may require distinct management strategies, thereby reducing controller-switch interactions. Nevertheless, such architectures are inherently tied to a predefined internal taxonomy of rules and a specific update mechanism. As a consequence, extensibility becomes difficult when new constraints or alternative placement objectives must be introduced. The approach improves rule management efficiency, but it does not provide a general optimization model for distributing ACL subsets over constrained path sets. Finally, several recent systems revisit the problem from the perspective of in-switch representation rather than network-wide placement. PortCatcher [16], for example, reduces TCAM usage by offloading parts of multi-field packet matching to SRAM. Its strengths include substantial memory savings and very low deployment latency, both of which are attractive for reactive security defenses. However, the solution addresses only the local storage dimension of the problem. It does not determine on which network nodes decomposed ACL fragments should be installed so that all routing and policy constraints remain satisfied. In this sense, such works reinforce the importance of network-wide placement optimization rather than replacing it. Table 2.3 summarizes the heuristic, graph-based, and architectural approaches discussed in this subsection by

comparing their main ideas, placement scope, strengths, and limitations.

Approach / Work	Main idea	Placement scope	Strengths	Limitations
OBS [57]	Logical one-switch abstraction for rule distribution.	Rule compilation over predefined paths.	Introduces path-based rule allocation.	Static paths, per-path allocation, and possible rule replication.
Palette [25]	Graph-based decomposition and greedy placement.	Path-covering over SDN topologies.	Simple and exploits aggregate TCAM.	Greedy; limited support for richer constraints.
DIFANE, DevoFlow, vCRIB, Infinite CacheFlow [23, 89, 90, 91]	Distributed ownership or rule caching.	Flow-table management and TCAM relief.	Reduces control-plane and TCAM pressure.	No guarantee for ACL path-wise coverage.
MoRule, Huang et al., Angelos et al., OPTree [92, 93, 94, 95]	Context-specific placement heuristics.	Mobility, heterogeneous TCAMs, or hybrid tables.	Captures practical deployment settings.	Limited transfer to new constraints and topologies.
Middlebox / SFC placement [96, 97, 98, 99, 100]	Places complete functions along paths.	Function-level placement.	Highlights topology and path diversity.	Too coarse for rule-level ACL placement.
Wang et al., PortCatcher [101, 16]	Hybrid or in-switch rule representation.	Local or hybrid storage optimization.	Reduces local TCAM usage.	Not a network-wide ACL placement model.

Table 2.3 – Comparative synthesis of heuristic, graph-based, and architectural approaches related to distributed ACL rule placement.

Despite the diversity of heuristic, graph-based, and system-oriented solutions, the state of the art still exhibits several important limitations. First, many existing approaches rely on static routing assumptions or a single predetermined path, whereas modern networks increasingly operate under dynamic routing and failure-driven path changes. Second, most heuristic methods optimize memory occupancy or signaling overhead but do not explicitly guarantee robust path-wise enforcement of distributed ACL policies. Third, several approaches are tightly specialized to particular topologies, deployment hypotheses, or hardware architectures, which limits their generalization to heterogeneous and large-scale networks. Fourth, current non-learning methods often require manual tuning and must be redesigned when constraints such as bandwidth restrictions or rule ordering are introduced. Finally, topology-awareness remains underexploited: most methods do not learn structural regularities from graphs nor adapt their decisions across variable-size networks.

2.4.2 Exact Optimization Approaches

Exact optimization methods in distributed ACL placement adopt a fundamentally different perspective from heuristic techniques. Rather than prioritizing computational speed, they aim to compute provably correct and globally efficient placements under explicitly modeled constraints. This family of methods is particularly attractive for distributed ACL deployment because the problem is inherently combinatorial: rule subsets must be assigned to switches, switch memory capacities must be respected, admissible traffic paths must remain correctly filtered, and policy semantics such as priority, ordering, or dependency relations must be preserved. For these reasons, a significant portion of the literature has relied on Integer Linear Programming (ILP), Mixed-Integer Linear Programming (MILP), and more recently satisfiability-based formulations such as MaxSMT.

Their principal strength is expressive rigor; their recurring limitation is that this rigor often becomes computationally expensive as the topology size, path diversity, or constraint set increases.

One of the earliest formal studies directly related to distributed ACL placement is the work of Zhang et al. [84], who modeled rule placement as an ILP optimization problem. Given a network topology, predefined traffic paths, and a set of firewall rules, their framework determines how rules should be distributed across switches while respecting local memory constraints. A key strength of this contribution lies in its precise mathematical formulation, which enables the optimization of multiple objectives, including minimizing the total number of installed rules or enforcing filtering decisions closer to traffic ingress points. Moreover, the model explicitly captures rule interactions more rigorously than purely greedy approaches. However, the approach remains limited by the computational complexity of ILP solvers, which may hinder scalability on large topologies or under frequent network updates. In addition, the formulation assumes that routing paths are known in advance and remain stable during optimization. This assumption is restrictive in dynamic environments where traffic may follow multiple feasible paths or where routing changes after failures and load-balancing decisions.

A second influential line of work is OFFICER [88], which formulates OpenFlow rule allocation and endpoint-policy enforcement as an integer optimization problem complemented by heuristic accelerations. Its major contribution is to connect rule allocation with end-to-end policy satisfaction and network resource availability, rather than treating placement as a purely local memory management problem. OFFICER also incorporates routing flexibility into the optimization process, thereby recognizing the strong interaction between path selection and rule occupancy. This makes it conceptually close to later path-aware placement models. Nevertheless, this flexibility comes at a cost. The framework may intentionally reroute flows along longer paths to improve rule-space efficiency [24], which can degrade latency or consume additional bandwidth. Furthermore, the method remains oriented toward general OpenFlow rule allocation rather than the semantics of ordered ACL-subset placement. It therefore broadens the optimization space but does not fully address the controlled distributed filtering problem.

Several subsequent works extended exact formulations toward joint routing and placement objectives. Giroire et al. [86] proposed an ILP for energy-aware routing under rule-space limitations, showing that routing quality and switch memory consumption are tightly coupled optimization dimensions. Similarly, Huang et al. [102] studied the joint optimization of rule allocation and traffic engineering in SDN. These studies are valuable because they demonstrate that rule placement cannot be treated independently from routing decisions. They also show the versatility of mathematical programming in integrating multiple operational objectives within a unified model. However, their focus remains on forwarding efficiency, energy reduction, or traffic balancing rather than distributed ACL enforcement. They do not consider rule decomposition, filtering semantics, or guaranteed coverage of all feasible traffic paths. Consequently, while methodologically relevant, they only partially overlap with the distributed ACL placement problem addressed in this thesis. A more recent example in the same direction is the work of Behera et al. [103], who investigated flow-rule placement for energy-aware SDN routing using a combination of ILP and heuristics. Their objective was to reduce power consumption by intelligently dis-

tributing rules across switches while maintaining routing efficiency. The main strength of this study lies in demonstrating that placement models can incorporate secondary objectives such as sustainability and operational cost, thereby extending the relevance of rule placement beyond memory management alone. However, the model remains centered on forwarding-rule optimization rather than security-policy distribution. It does not address ACL rule splitting, ordering semantics, or robust enforcement under routing uncertainty. As such, it should be viewed as a neighboring optimization problem rather than a direct solution to distributed ACL placement.

Among the works most closely related to path-diverse placement is Raptor [85]. Raptor formulates multi-path rule placement as an ILP whose objective is to maximize rule sharing across multiple paths, then applies graph partitioning techniques to restore rule ordering. This contribution is particularly important because it directly recognizes that a flow may traverse multiple alternative paths while still requiring consistent policy enforcement. Its strengths include better scalability than earlier exact formulations, support for fast failover, and explicit treatment of multipath consistency. However, two limitations remain significant. First, the optimization core still grows rapidly with topology size and path multiplicity. Second, rule ordering is not handled natively inside the mathematical formulation but delegated to a post-processing phase. This separation reduces extensibility when richer constraints must be integrated.

The dynamic dimension of exact optimization was addressed more explicitly by Chang et al. [87]. Their work recognizes an important operational issue: policy updates should not require recomputing the entire placement from scratch. They therefore proposed a resource-constrained splitting algorithm that preserves unaffected placements while recomputing only the modified portion of the ruleset. The principal strength of this contribution is its practical relevance for distributed firewall updates, where minimizing recomputation time is essential. However, the approach remains within the traditional exact-optimization paradigm. As additional constraints are introduced, repeated solving can still become expensive, and the method remains dependent on the tractability of the underlying linearized model. It improves incremental recomputation, but it does not fundamentally remove the structural scalability limitations of solver-based approaches.

More recently, Bringhenti *et al.* proposed a formal framework for the automated deployment of distributed firewalls in virtualized environments [104, 105, 106]. Their method models the network as a service graph and formulates placement and configuration as a partial weighted MaxSMT problem. Security requirements are translated into filtering rules, and the solver determines both the firewall locations and the rules installed at each instance. A major strength of this work is its correctness-by-construction philosophy: whenever a feasible solution exists, the generated deployment satisfies the specified constraints. In addition, placement and configuration are solved jointly rather than sequentially. However, the approach remains fundamentally centralized and solver-dependent. As the number of candidate locations, security requirements, and paths increases, the number of constraints grows rapidly, which may hinder scalability. Furthermore, the optimization assumes complete prior knowledge of the topology and is essentially offline, making repeated re-optimization necessary under routing changes or evolving policies.

Several later studies further emphasized the interaction between routing flexibility and switch memory constraints. Guo et al. [107] considered hybrid SDN networks in which

only part of the infrastructure is programmable. They formulated routing optimization under TCAM limitations as a MINLP with path-cardinality constraints, where limiting the number of paths used by each flow reduces forwarding-table occupancy. Similarly, Xu et al. [108] proposed a MILP for QoS-aware cloud-service routing in SDN-enabled IoT networks under TCAM capacity constraints, complemented by approximation algorithms for scalability. These works are significant because they clearly show that route diversity, although beneficial for traffic engineering, may dramatically increase rule occupancy. However, their objectives remain traffic engineering and load balancing rather than security-rule distribution. They do not model ACL decomposition, filtering dependencies, or the requirement that every feasible path intersects a valid enforcement point.

Recent studies, such as Ashraf et al. [109], confirm that exact formulations remain valuable tools for exposing the intrinsic structure of placement problems. Their multi-objective MILP jointly minimizes link utilization, number of installed rules, and switch-table occupancy under evolving traffic demands, and is complemented by greedy algorithms because the exact problem is NP-hard. This illustrates a recurring conclusion across the literature: exact formulations are indispensable for defining optimization problems rigorously and for generating strong baselines on moderate-size instances, yet they often become impractical when decisions must be recomputed repeatedly, under many constraints, or at a large scale. Table 2.4 summarizes the exact optimization approaches reviewed in this subsection, highlighting their modeling principles, placement scope, strengths, and main scalability limitations.

Approach / Work	**Main idea**	**Placement scope**	**Strengths**	**Limitations**
Zhang et al. [84]	ILP formulation for firewall rule placement.	Distributed placement over predefined paths.	Precise model with multiple objectives.	Limited scalability; assumes stable routing.
OFFICER [88]	Integer optimization for OpenFlow rule allocation.	Joint rule allocation and policy satisfaction.	Considers resources and routing flexibility.	May use longer paths; not ACL-subset specific.
Raptor [85]	ILP-based multipath rule placement.	Multipath placement and failover support.	Improves rule sharing across paths.	Cost grows with paths; ordering is post-processed.
Chang et al. [87]	Resource-constrained splitting with updates.	Dynamic firewall update management.	Avoids recomputing unchanged placements.	Still solver-dependent under many constraints.
Bringhenti et al. [104, 105]	MaxSMT-based distributed firewall deployment.	Joint firewall placement and configuration.	Correctness-by-construction.	Centralized and offline; constraints grow rapidly.
Routing-aware ILP/MILP works [107, 108, 109]	Joint routing and rule allocation.	Traffic engineering, energy, or QoS placement.	Shows interaction between routing and memory.	Focuses on forwarding, not ACL semantics.

Table 2.4 – Comparative synthesis of exact optimization approaches for distributed ACL rule placement and related rule-allocation problems.

In summary, ILP, MILP, and MaxSMT approaches provide the strongest formal baseline for distributed ACL placement. They are excellent at proving correctness, modeling constraints precisely, and identifying optimal trade-offs on moderate instances. However, the literature consistently shows three weaknesses: limited scalability, difficulty incorporating richer constraints without reformulation, and high recomputation cost in dynamic environments. These limitations motivate moving from exact formulations alone toward

approaches that preserve the problem's distributed ACL semantics while scaling better to larger path sets, richer constraints, and repeated placement decisions.

2.4.3 Machine Learning Approaches

Learning-based methods address distributed rule placement from the perspective of adaptability and scalability. Their underlying motivation is that classical optimization techniques, although rigorous, often become computationally expensive when the topology, path set, traffic demands, or deployment constraints evolve over time. In realistic networks, ACL placement is not a one-shot decision problem: routing changes, policy updates, failures, and traffic fluctuations may require repeated recomputation. In this context, recent research has explored whether a model can learn a placement policy that generalizes to unseen network instances instead of solving a full combinatorial optimization problem from scratch after every change. This direction is particularly attractive for distributed ACL placement, where memory constraints must be reconciled with rapid operational responsiveness.

Compared with heuristic and exact optimization approaches, the application of machine learning to distributed ACL rule placement remains relatively recent and significantly less explored. This observation is itself noteworthy. While machine learning has been widely adopted in SDN for routing, congestion control, anomaly detection, and resource orchestration, direct learning-based formulations for distributed ACL placement have long remained scarce. Consequently, the relevant literature for this subsection is drawn not only from security-policy placement studies, but also from the broader fields of networking intelligence and learning for combinatorial optimization.

A first relevant line of work concerns machine learning for routing and control optimization in SDN. Surveys such as Amin et al. [110] show that RL, deep learning (DL) [111], and neural approximators are increasingly used for routing optimization, congestion control, and adaptive resource management in programmable networks. These works demonstrate that learning-based approaches can handle large action spaces and sequential decision processes more efficiently than repeatedly solving exact optimization models. They also show strong adaptability in dynamic environments where traffic conditions evolve rapidly. However, their optimization objectives are generally network-centric metrics such as delay, throughput, load balancing, or resilience. They do not model the hard correctness constraints that arise when filtering policies must be distributed across multiple switches while ensuring that all relevant traffic paths remain protected. Therefore, although highly relevant methodologically, they do not directly solve the distributed ACL placement problem.

A second major direction is the use of GNNs for networking problems. Because communication networks are inherently graph-structured, GNNs have attracted considerable attention for their ability to perform relational reasoning and to generalize across varying topologies [112, 113]. Representative works such as RouteNet [114, 115] and subsequent studies [116, 117] have shown that graph-based models are particularly effective for capturing path interactions, congestion propagation, and structural bottlenecks in networks. Their principal strength is that they exploit topology directly rather than relying on handcrafted flat features. This is especially important when decisions depend on node

centrality, path overlap, articulation points, or graph connectivity. Nevertheless, these works primarily target performance prediction, routing evaluation, or generic network optimization. They do not directly address distributed ACL placement, nor do they model rule-order semantics, path-wise filtering guarantees, or switch-memory feasibility. Thus, while the representational power of GNNs is highly promising, it still requires adaptation to the specific placement problem.

At the SDN control level, broader surveys such as Faezi et al. [118] confirm that machine learning has become an important tool for automating network decisions in environments where handcrafted policies or repeated exact optimization are difficult to maintain. These studies highlight the relevance of data-driven control and the ability of learning methods to react to evolving network conditions. However, they remain broad in scope and do not specifically address distributed ACL placement under path-coverage and switch-capacity constraints. This leaves a clear gap between the general promise of machine learning in SDN and its concrete application to ACL rule distribution.

Reinforcement Learning (RL) has also been investigated in a number of adjacent networking problems, including admission control, traffic engineering, and service-function orchestration [119, 120, 121]. The strength of RL in these contexts lies in its capacity to learn sequential decisions under uncertainty and to improve through repeated interaction without explicitly solving a full combinatorial program at every step. This property is particularly relevant to distributed ACL placement, where assigning one subset of rules modifies the feasible placement choices of the remaining subsets. However, most existing RL-based networking approaches model constraints indirectly through reward shaping, which does not guarantee strict satisfaction of memory limits or path-wise policy coverage. Moreover, many methods rely on fixed-size feature encodings that generalize poorly across heterogeneous network topologies.

More specialized learning-based placement methods have recently emerged. Nguyen et al. proposed DeepPlace [122], a deep RL solution for adaptive flow-rule placement in software-defined IoT networks. DeepPlace jointly considers routing and rule placement, and demonstrates that DRL can preserve a high level of rule expressiveness while limiting QoS degradation under dynamic traffic conditions. This is an important contribution because it shows that learning can outperform static formulations in highly dynamic placement scenarios. However, the method remains oriented toward IoT flow-rule management rather than distributed ACL enforcement. Its objective focuses on maximizing match expressiveness and reducing QoS violations, rather than ensuring the consistent placement of pre-decomposed ACL subsets under memory, path-coverage, and dependency constraints. Consequently, it supports the case for learning-based placement, but does not fully address the security-policy distribution problem studied in this thesis.

A related contribution by Huang et al. [123] proposed a Q-learning-based predictive mobility-aware flow placement scheme for SDN-enabled IoT networks. The framework predicts future device mobility and proactively installs flow entries so as to reduce misses and controller interventions. Its main strength lies in coupling prediction with placement decisions, thereby recognizing that optimal placements depend not only on current states but also on expected traffic movement. This insight is also relevant to distributed ACL placement, where future routing behavior influences the best enforcement locations. Nevertheless, the work remains focused on mobility-aware forwarding management rather

than filtering-policy deployment. It does not address ordered ACL subsets, guaranteed filtering across all admissible paths, or the simultaneous satisfaction of memory and rule-dependency constraints.

Another neighboring but highly relevant direction concerns GNN-based ACL policy reasoning. Sun et al. [124, 116] demonstrated that graph neural networks can efficiently reason about distributed ACL policies under complex logical constraints, and in some cases outperform modern SAT/SMT solvers in synthesis time while maintaining correctness. The strength of these works is particularly notable because they explicitly target ACL semantics rather than generic forwarding-rule management. They therefore provide strong evidence that GNNs can successfully model security-policy constraints. However, their focus is policy synthesis and reachability reasoning rather than placement optimization. In other words, they determine which policy should hold, but not where pre-decomposed ACL subsets should be installed in the network so that routing, memory, and dependency constraints are jointly satisfied. Their relevance is therefore strong from a modeling perspective, but partial from a placement perspective.

The broader combinatorial optimization literature also provides important methodological foundations. Bello et al. [125] introduced neural combinatorial optimization with RL, showing that sequential graph optimization problems can be learned rather than solved through handcrafted heuristics. Kool et al. [126] later demonstrated that attention-based architectures can learn high-quality heuristics for routing-type problems, while Khalil et al. [127] and Barrett et al. [128] showed that graph embeddings combined with DRL can solve hard combinatorial problems such as minimum vertex cover, maximum cut, and traveling salesman variants. These studies are important because they establish the theoretical basis for replacing manually designed heuristics with learned policies capable of handling large structured search spaces. However, they are not ACL-specific and do not model filtering semantics, switch capacities, path-wise coverage, or rule ordering. More-

Approach / Work	**Main idea**	**Placement scope**	**Strengths**	**Limitations**
ML for SDN routing/control [110]	ML/RL for routing and adaptive control.	Network performance optimization.	Adapts to dynamic network conditions.	Does not model distributed ACL correctness.
GNNs for networking [114, 116, 117]	Learns topology-aware network representations.	Routing evaluation and performance prediction.	Captures paths and graph structure.	Does not solve ACL rule placement.
RL for networking problems [119, 120, 121]	Learns sequential network decisions.	Admission control, TE, or orchestration.	Suitable for placement-like decisions.	Constraints are often soft rewards.
DeepPlace [122]	DRL-based adaptive flow-rule placement.	Flow-rule placement and routing.	Shows adaptability in dynamic settings.	Targets QoS, not ACL semantics.
Mobility-aware Q-learning [123]	Predictive Q-learning for flow placement.	Mobility-aware forwarding management.	Anticipates mobility and reduces misses.	No ordered ACL or path-wise coverage.
GNN-based ACL reasoning [124, 116]	GNN reasoning over ACL policies.	Policy synthesis and reachability.	Directly relevant to ACL semantics.	Focuses on reasoning, not placement.
Neural combinatorial optimization [127, 128]	RL/GNNs for graph optimization.	Generic graph optimization.	Basis for learned placement policies.	Not ACL-specific; ignores rule semantics.

Table 2.5 – Comparative synthesis of machine-learning-based and graph-learning approaches related to distributed ACL rule placement.

over, many of these methods assume that each action corresponds directly to selecting a graph node or edge, whereas distributed ACL placement involves richer actions such as assigning rule subsets to switches under multiple interacting constraints. This distinction further motivates the specialized action design proposed in this thesis. Table 2.5 summarizes the learning-based and graph-learning approaches discussed in this subsection, emphasizing their relevance and remaining limitations with respect to distributed ACL rule placement.

Despite the growing success of machine learning in networking, the literature still leaves several important gaps when viewed from the perspective of distributed ACL placement. First, most existing works optimize routing efficiency, traffic engineering, or generic flow-table management rather than the enforcement of security policies distributed across multiple switches. Second, hard operational constraints such as switch-memory capacities, path-wise policy coverage, rule ordering, and semantic equivalence with the original ACL are rarely modeled explicitly, and are often approximated through soft rewards. Third, many RL approaches rely on fixed-size input/output representations that generalize poorly across heterogeneous topologies. Fourth, topology structure is frequently underexploited, despite the inherently graph-based nature of the placement problem. Finally, the literature seldom addresses the joint challenge of routing uncertainty, scalability, and practical deployment constraints within a unified framework. These limitations directly motivate the contributions of this thesis, which investigate distributed ACL placement under dynamic routing, graph-neural learning for topology-aware generalization, and routing-aware placement under bandwidth and rule-ordering constraints.

2.5 Summary

The rapid expansion of ACLs in modern networks has transformed ACL deployment into a major scalability challenge. While ACLs remain essential for enforcing security and traffic filtering policies, their implementation in TCAM-based switches is constrained by limited memory capacity, high hardware cost, and significant energy consumption. As enterprise networks, cloud infrastructures, and software-defined environments continue to grow in scale and complexity, efficient management of ACL rules has become a critical research issue. To address this problem, the literature has proposed three principal families of solutions: rule compression, rule eviction, and split-and-distribution approaches. Compression techniques aim at reducing the number of stored entries through rule minimization or aggregation, whereas eviction methods rely on dynamic replacement strategies that keep only the most relevant rules in memory. Although these approaches can provide temporary relief, they often suffer from policy distortion, computational overhead, limited adaptability, or degradation in classification performance.

In contrast, split-and-distribution techniques exploit the aggregate memory resources of multiple switches by decomposing ACL policies into smaller rule subsets and distributing them across the network. Nevertheless, the state of the art on distributed ACL placement still reveals several unresolved limitations and persistent research gaps. First, most existing approaches rely on static routing assumptions or a single predetermined path, whereas modern networks commonly operate under dynamic routing conditions

and failure-driven path changes. Second, heuristic methods are generally computationally efficient and often provide acceptable solutions in practice; however, they remain strongly problem-dependent, since each new operational constraint or modification of the objective function typically requires a complete redesign of the underlying model. Moreover, such methods often optimize memory occupancy or signaling overhead without explicitly guaranteeing robust path-wise enforcement of distributed ACL policies, and they rarely provide guarantees regarding optimality or robustness. Third, exact optimization approaches, particularly ILP formulations, can generate high-quality placements, yet their computational complexity increases rapidly with network size, policy scale, and interactions among constraints. In addition, every new requirement must be reformulated into linear constraints, which becomes increasingly difficult when modeling realistic networking behaviors and complex operational policies. As a result, the scalability and flexibility of current non-learning methods remain inherently limited when placement objectives evolve or additional constraints must be introduced. More recently, learning-based approaches have emerged as a promising alternative; however, their application to distributed ACL placement remains largely unexplored. To the best of our knowledge, this thesis constitutes the first work to investigate machine learning, and more specifically reinforcement learning combined with graph neural networks, for this problem. Furthermore, topology-awareness remains severely under-exploited in the literature, as most existing methods neither learn structural regularities from network graphs nor adapt their placement decisions across variable-size topologies. Moreover, when learning-based approaches are considered, critical operational requirements such as switch-memory capacities, path-wise policy coverage, rule ordering, and semantic equivalence with the original ACL are often incorporated only indirectly through reward shaping as soft constraints, rather than being enforced explicitly as hard constraints that must always be satisfied. More broadly, existing studies rarely formulate distributed ACL placement as a unified multi-objective optimization problem that simultaneously accounts for practical requirements such as routing uncertainty, bandwidth efficiency, placement locality, rule ordering dependencies, and scalability to large network topologies. Table 2.6 synthesizes these limitations by comparing the main families of distributed ACL placement approaches across the research dimensions addressed in this thesis, including routing assumptions, scalability, topology-aware generalization, constraint flexibility, and hard constraint enforcement. These observations highlight the need for scalable, adaptive, topology-aware, and constraint-aware placement mechanisms capable of optimizing multiple realistic operational requirements while maintaining high placement quality. This need directly motivates the contributions developed in this thesis.

These observations highlight the need for scalable, adaptive, topology-aware, and constraint-aware placement mechanisms capable of optimizing multiple realistic operational requirements while maintaining high placement quality. This need directly motivates the contributions developed in this thesis.

In this thesis, we focus on the distribution stage of the split-and-distribution paradigm. More precisely, we assume that the initial ACL table has already been decomposed into smaller rule subsets, and we address the problem of efficiently placing these subsets across the network topology. The main objective of this thesis is therefore to design an intelligent distributed filtering framework in which ACL rule subsets are strategically deployed over network devices while accounting for memory limitations and routing constraints.

The proposed framework seeks to optimize resource utilization while preserving policy correctness, guaranteeing the protection of relevant traffic paths, and satisfying practical deployment requirements. Furthermore, the proposed solution is architecture-agnostic and can be integrated into both SDN environments and conventional network infrastructures. Accordingly, Chapter 3 first investigates the distributed placement of ACL rules under routing uncertainty, where all feasible source-destination paths must remain protected despite possible route changes. Chapter 4 then introduces a graph neural reinforcement learning framework designed to learn scalable placement strategies while improving resource utilization and placement efficiency. Finally, Chapter 5 extends the problem to selected path sets and incorporates additional practical constraints, including bandwidth-aware filtering and precedence-preserving rule ordering. The next chapter presents the first contribution of this thesis and formulates the ACL placement problem under routing uncertainty.

Approach	Representative works	Routing model	Scalability to large topologies	Topology-aware generalization	Constraint flexibility	Hard constraint enforcement	Scope / problem addressed
Heuristic, graph-based, and architectural approaches	OBS [57], Palette [25], MoRule [92], OPTree [95], PortCatcher [16], DIFANE [23], SIMPLE [77].	Mostly static; generally based on predefined routes, selected path sets, or problem-specific deployment assumptions.	Partial	No	Limited; new constraints usually require redesigning the heuristic or the underlying architecture.	Partial	Flow-table management, TCAM scarcity reduction, or local rule optimization. These approaches are not designed to provide a general framework for distributed ACL semantics under routing uncertainty.
Exact optimization approaches (ILP / MILP / MaxSMT)	Zhang et al. [84], Raptor [85], Chang et al. [87], OFFICER [88], Bringhenti et al. [104], Giroire et al. [86].	Static or limited multi-path settings; routing paths are generally assumed to be known in advance.	Limited	No	Limited; incorporating new requirements typically requires reformulating the optimization model.	Yes	Strong formal guarantees on moderate-size instances. However, scalability decreases with network size, path diversity, and constraint complexity, while dynamic changes often require costly re-optimization.
Learning-based approaches(RL / DRL / GNN)	DeepPlace [122], mobility-aware Q-learning placement [123], RouteNet [114], GNN-based ACL reasoning, and neural combinatorial optimization studies [127, 128].	Mainly routing, traffic engineering, or flow-rule management; direct treatment of distributed ACL path coverage remains limited.	Promising	Partial	Partial; constraints are often integrated indirectly through reward design rather than through explicit placement guarantees.	Rarely	Routing, QoS, traffic engineering, and neighboring graph-optimization problems. These works do not directly address distributed ACL placement semantics under memory, path-coverage, and dependency constraints.
This thesis(C1, C2, C3)	Chapter 3: st-mincut, Greedy, RL. Chapter 4: GNN–DQN and distance-aware placement. Chapter 5: selected paths, bandwidth-aware placement, and rule ordering.	Addresses complementary routing settings: dynamic routing over all feasible paths, selected path sets, and routing uncertainty.	Yes	Yes	High; additional placement requirements are incorporated within a unified learning-based framework.	Yes	Distributed ACL placement under memory limitations, path-wise coverage, routing variability, bandwidth-aware filtering, and rule-ordering constraints.

Table 2.6 – Comparative gap analysis of distributed ACL placement approaches across the main research dimensions addressed in this thesis.

Chapter 3

Distributed ACL Placement Under Dynamic Routing Constraints

Contents

3.1 Introduction 45
3.2 Network Representation and Problem Statement 46
3.3 Algorithms for Distributed Placement of ACL Rules 49
3.3.1 st-mincut 49
3.3.2 Greedy Heuristic 52
3.3.3 Reinforcement Learning-Based Approach 55
3.4 Experiments and Results 57
3.4.1 Simulation Setup 57
3.4.2 Results and Discussion 59
3.5 Summary 64

3.1 Introduction

The distributed ACL placement problem, as introduced in Chapter 1, consists of assigning pre-decomposed rule subsets to a network of memory-constrained switches so that every packet encounters, along its traversal path, all filtering rules required to preserve the semantics of the original ACL. The split-and-distribution paradigm provides a structurally suitable response to TCAM scalability limitations by replacing centralized rule storage with cooperative enforcement across multiple devices. This paradigm, however, raises a fundamental question: how should rule subsets be placed so as to guarantee correct policy enforcement while minimizing the use of limited switch memory?

A major limitation of most existing distributed placement approaches is their reliance on static and fully known routing assumptions. Under such conditions, it is sufficient to ensure coverage only along a predefined set of paths. However, in practice, modern networks are inherently dynamic: routing decisions may evolve in response to link failures, load balancing, traffic-engineering policies, or topology reconfigurations. If traffic

is rerouted through paths not considered during placement, some packets may bypass the required filtering subsets, leading to silent violations of the intended security policy. This mismatch between static placement assumptions and dynamic routing behavior, therefore, creates a fundamental correctness risk. To address this issue, placement must be robust not merely to a fixed routing configuration, but to the entire set of feasible source-destination routes induced by the network topology. In other words, every possible source-destination path must collectively enforce the complete distributed ACL, regardless of the route ultimately selected at forwarding time. Ensuring this all-path coverage while minimizing total memory occupancy and respecting per-switch capacity constraints constitutes the central challenge studied in this chapter.

Accordingly, this chapter investigates distributed ACL placement under routing uncertainty. We represent the network as a source-destination directed acyclic graph and require that every feasible source-destination path encounters all rule subsets necessary to preserve the original ACL semantics. The objective is to compute a memory-efficient placement that guarantees complete policy enforcement independently of the actual route followed by traffic, while avoiding unnecessary rule duplication whenever possible. To solve this problem, we develop three complementary placement strategies. First, we introduce a graph-theoretic formulation based on the *st*-minimum cut, which yields an exact solution for the placement of a single rule subset and serves as the basis for an iterative extension to multiple subsets. Second, we propose a greedy heuristic that prioritizes nodes according to their contribution to path coverage. Third, we investigate RL-based approaches to assess whether sequential decision-making can produce effective placements under constrained memory conditions. Together, these methods provide complementary perspectives on the trade-off between optimality, computational efficiency, and adaptability in distributed ACL placement under dynamic routing.

The remainder of this chapter is organized as follows. Section 3.2 formally defines the placement problem, introduces the network model, and presents the main assumptions and constraints. Section 3.3 describes the proposed graph-theoretic, heuristic, and RL-based placement strategies. Section 3.4 evaluates these approaches on real-world network topologies and discusses their comparative performance. Finally, Section 3.5 summarizes the main findings and highlights the limitations that motivate the topology-aware learning framework introduced in the next chapter.

3.2 Network Representation and Problem Statement

We model the network topology as a *directed acyclic graph* (called st-DAG), denoted by $G(V, E)$, consisting of a single source node s and a single destination node t. In this representation, network switches are modeled as vertices V, while directed edges E correspond to communication links between switches. Any directed path from s to t in G is referred to as an *st-path*. By construction, every node in an st-DAG lies on at least one st-path. We denote by $n' < n$ that node n' precedes node n along at least one st-path in the DAG.

Each node is associated with a capacity constraint, defined as the maximum number of ACL rule sets that can be stored in the TCAM memory of the corresponding switch.

Due to the dynamic nature of routing, the exact path followed by packets at any given time is generally unknown and may vary over time. Consequently, the problem consists in distributing ACL rules across the network such that all possible st-paths from s to t are correctly covered.

The objective of this distribution is to efficiently utilize switch resources by minimizing the overall TCAM occupancy. In particular, redundant placement of the same rule along a single path should be avoided whenever possible, as it leads to unnecessary consumption of limited memory resources. However, depending on the network topology and capacity constraints, some degree of redundancy may be unavoidable. Therefore, the goal is to minimize rule duplication and overall memory usage while guaranteeing complete policy enforcement across all st-paths between the source and the destination.

Formal Problem Statement Given an st-DAG $G = (V, E)$, each node $v \in V$ is associated with a finite memory capacity $m(v)$, defined as the maximum number of ACL rules that can be installed on node n. Let $F = f_1, f_2, \ldots, f_k$ denote the set of ACL rules to be deployed in the network. The degree of a node $n \in V$ in the st-DAG G is defined as the number of edges incident to n and is denoted by $d_G(n)$. A weighted st-DAG is an st-DAG in which each edge $e \in E$ is assigned a real-valued weight. The weight of an edge e in graph G is denoted by $w_G(e) \in \mathbb{R}$. An st-DAG can equivalently be represented by a dictionary-based structure in which each node n is associated with two lists: the set of its immediate predecessors, denoted by $pred_G(n)$, and the set of its immediate successors, denoted by $succ_G(n)$. For any finite set S, the notation $|S|$ denotes its cardinality.

The problem addressed in this chapter concerns the placement of the rule sets $f_1, f_2, \ldots, f_k$ on the nodes of the graph G. The placement must satisfy two main conditions: (i) every st-path from s to t must include at least one occurrence of each rule in F, and (ii) the overall utilization of switch memory resources must be minimized.

We denote this problem by $PLACE(G, m, F)$. For any st-path p, we define its *placement capacity* (or *placement slots*), denoted by $path_capacity(p)$, as the sum of the memory capacities of the nodes composing the path, i.e., $path_capacity(p) = \sum_{v \in p} m(v)$, which represents the total number of rule slots available along p. It is important to note that $PLACE(G, m, F)$ admits no feasible solution if the placement capacity of a shortest st-path $p_{\min}$ is strictly smaller than k, i.e., $path_capacity(p_{\min}) < k$, since it would then be impossible to place all k rule sets along such a path. Conversely, when $path_capacity(p_{\min}) \geq k$, the coverage constraint in (i) can be satisfied. The notation used throughout the problem formulation is summarized in Table 3.1.

Proposition 1. *If for every st-path p, $\sum_{n \in p} m(n) \geq |F|$ then $PLACE(G, m, F)$ admits a solution.*

Proof. Let us assume that for every n, its memory is represented by an ordered list $l_n^1, l_n^2, \ldots, l_n^{m(n)}$ of memory cells. We define the distance of a cell l_n^i to s by the minimum on all st-path p containing n of $(\sum_{n' \prec n, n' \in p} m(n')) + i$. Then by hypothesis, for every st-path p and $1 \leq j \leq |F|$, there is a memory cell on p at distance j of s; we allocate this memory to f_j. This procedure defines a solution. □

Notation	Description
s	single source
t	single destination
F	set of ACL rules
$m(v)$	memory available on node v, i.e., maximum number of rules that can be installed on v
$d_G(n)$	number of edges incident to n
$w_G(e)$	weight of an edge e in G
$pred_G(n)$	list of immediate predecessors of node n in G
$succ_G(n)$	list of immediate successors of node n in G
$path_capacity(p)$	sum of the memory capacities of the nodes composing the path p
$p_{\min}$	shortest st-path within the graph G
$\|S\|$	cardinality of the finite set S

Table 3.1 – Notations for problem formulation

Illustrative Example: Redundancy Induced by Path Structure
This example illustrates the problem of distributing ACL rules so as to ensure coverage of all st-paths while minimizing the overall number of rules placed in the network under TCAM capacity constraints. Consider a core network represented by the st-DAG shown in Figure 3.1.

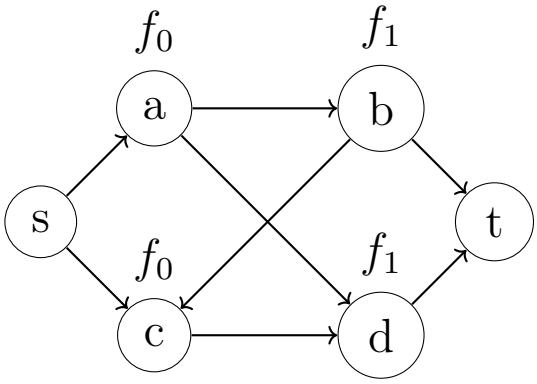


Figure 3.1 – Placement of a set of rules in a network

The network consists of four switches, denoted by (a, b, c, d), each equipped with a TCAM capacity of 2 Mb, corresponding to a maximum of 33,000 ACL entries per switch. The topology includes a single source node s and a single destination node t.

The set of all st-paths between s and t is given by:

$$s \to a \to b \to t, \quad s \to c \to d \to t, \quad s \to a \to d \to t, \quad s \to a \to b \to c \to d \to t.$$

Assume that the global ACL consists of 60,000 rules to be deployed. Since the TCAM capacity of a single switch is insufficient to store the entire rule set, the ACL table must be partitioned into two sub-tables, denoted by f_0 and f_1. These sub-tables must then be

distributed along the st-paths between the source and the destination so as to preserve the same end-to-end security policy enforced by the original ACL.

Under these constraints, the ACL placement problem admits only two feasible solutions. One such solution is illustrated in Figure 3.1. This example highlights that, due to the structure of the st-DAG, redundancy cannot always be avoided. In particular, along the st-path $a \to b \to c \to d$, the same rule set (f_0 or f_1) is necessarily encountered more than once. This redundancy is not the result of a suboptimal placement strategy but is instead inherent to the path structure and the imposed capacity constraints. This observation motivates the need for systematic placement strategies that ensure full st-path coverage while minimizing the overall number of rules installed in the network, which are presented in the next section .

3.3 Algorithms for Distributed Placement of ACL Rules

In this section, we present several strategies to solve the ACL placement problem. The proposed approaches differ in their underlying principles and computational complexity, ranging from graph-theoretic formulations to heuristic and reinforcement-learning-based methods. All strategies aim to ensure correct end-to-end ACL enforcement under TCAM capacity constraints while minimizing the overall number of rules installed in the network.

3.3.1 st-mincut

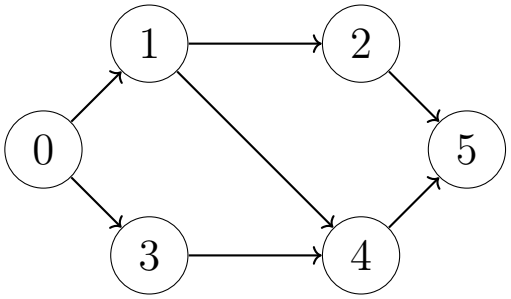
Applying st-mincut to One Rule Placement

We first consider the simplified setting in which a single rule set must be placed on an st-dag such that every path from the source s to the destination t contains at least one occurrence of this rule, while the total number of placements is minimized. For this case, we derive an *exact* polynomial-time solution by reducing the problem to a minimum st-cut problem.

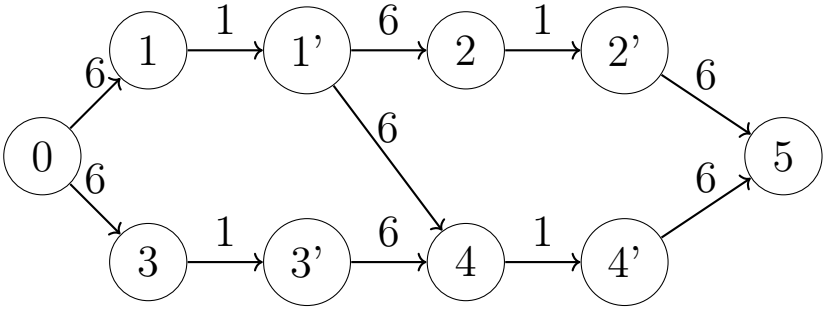
An st-cut (or cut, for brevity) in an st-DAG is defined as a set of edges C whose removal separates the graph into two disjoint subgraphs, one containing the source s and the other containing the destination t. In an unweighted graph, the value of a cut C is the number of edges it contains, whereas in a weighted graph, it is given by the sum of the weights of the edges in C. A minimum st-cut (mincut) is a cut with minimum value.

The minimum st-cut problem can be solved using the *max-flow min-cut theorem*, which states that in a flow network, the maximum achievable flow from s to t is equal to the minimum value of any st-cut [129]. Several polynomial-time algorithms exist to compute a minimum st-cut, including the Ford-Fulkerson algorithm [129] and the Edmonds-Karp algorithm [130], both of which iteratively increase the flow by exploiting augmenting paths until the maximum flow is reached. Depending on the chosen algorithm, the time complexity ranges from $O(|V| \times |E|^2)$ to $O(|V|^3)$ in the worst case.

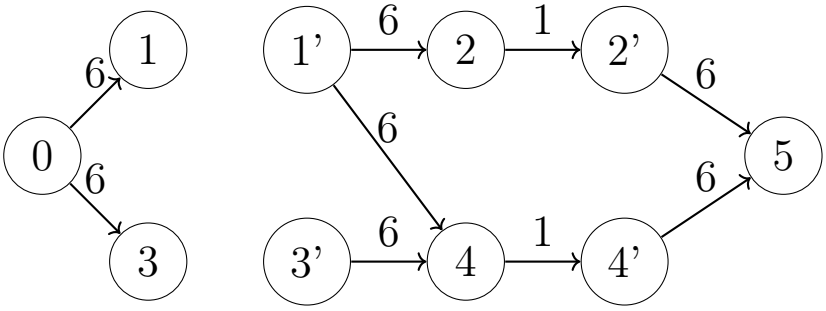
To adapt this algorithm to the rule placement problem, we transform the original st-DAG as follows. Each non-terminal node i is duplicated by introducing a new node i', and an edge (i, i') is added between them. The successors of node i in the original



(a) Graph G



(b) Augmented graph H



(c) H after applying the st-mincut

Figure 3.2 – Steps of the st-mincut-based Algorithm 3 for rule placement

graph become successors of i', while i' becomes the unique successor of i. Similarly, i is the unique predecessor of i'.

We then assign a weight of 1 to every edge of the form (i, i'), and a weight W_G to all other edges, where W_G is chosen as the number of nodes in the graph G. This construction ensures that any minimum st-cut contains only edges of the form (i, i'). Indeed, there always exists an st-cut composed exclusively of such edges, for instance, the set of edges (j, j') where j is a successor of s, whose total weight is strictly smaller than W_G. Conversely, any cut containing an edge not of the form (i, i') has a weight greater than or equal to W_G by construction.

As a result, once a minimum st-cut C is computed in the transformed graph, each edge in C corresponds to an edge (i, i'), and the rule set is placed on each original node i associated with such an edge. This guarantees that every st-path intersects at least one placement, while minimizing the total number of rule occurrences.

This procedure is illustrated in Figure 3.2 and formally described in Algorithm 3.

Algorithm 3: Rule placement with st-mincut

Input: G, st-dag; s, source ; t, destination
Output: O, set of nodes where to place the rule
Algorithm:

```
H ← G
for v ∈ nodes(H) do
    if v ∉ {s, t} then
        let v' be a new node:
        succ_H(v') ← succ_H(v)
        for j ∈ succ_H(v) do
            pred_H(j) ← remove(v, pred_H(j))
            pred_H(j) ← append(v', pred_H(j))
        end
        succ_H(v) ← ∅
        succ_H(v) ← append(v', succ_H(v))
        pred_H(v') ← append(v, pred_H(v'))
    end
end
for e ∈ edges(H) do
    if e == (v,v') then
        w_H(e) ← 1
    else
        w_H(e) ← W_H
    end
end
C ← st − mincut(H)
O ← {v|(v, v') ∈ C}
return O
```

Extending st-mincut to an Arbitrary Number of Rules

Given a set of k rule sets $F = \{f_1, f_2, \ldots, f_k\}$ to be placed on an st-dag G, the single-rule placement algorithm can be extended to handle multiple rule sets straightforwardly. The approach proceeds iteratively, placing rule sets one by one while preserving st-path coverage and respecting node capacity constraints. Specifically, for $j = 1, \ldots, k$, the following steps are performed as long as no failure is encountered in Step 4:

1. Associate each node $v \in V$ with a counter initialized to its memory capacity $m(v)$, representing the remaining number of rule sets that can be installed on v.
2. Assign a weight of 1 to each edge (v, v') if node v still has available memory to accommodate rule set f_j (i.e., its counter is strictly positive), and assign a weight W_G if node v has no remaining capacity (i.e., its counter equals 0).

3. Compute a minimum st-cut C_j in the resulting weighted graph.
4. If the value of the minimum cut is greater than or equal to W_G, terminate the procedure and declare failure; otherwise, place rule set f_j on every node v such that $(v, v') \in C_j$, and decrement the corresponding counter by one.

This algorithm runs in polynomial time, as it requires at most k minimum st-cut computations on graphs with a fixed vertex set and bounded edge weights. Each computation can be performed in $O(|V| \times |E|^2)$ to $O(|V|^3)$ time in the worst case, yielding an overall complexity of $O(k \times |V| \times |E|^2)$ to $O(k \times |V|^3)$. However, the approach does not always produce an optimal solution in terms of minimizing the total number of occupied switches. In particular, the greedy nature of the iterative placement may result in suboptimal utilization of switch memory resources.

3.3.2 Greedy Heuristic

In this section, we develop a greedy approach to address the ACL rule placement problem. The proposed algorithm has polynomial-time complexity; however, it does not guarantee the existence of a feasible solution in all cases. This greedy strategy is also employed to compute the reward function in the RL approach introduced in Section 3.3.3.

Applying Greedy Heuristic to One Rule Placement



(a) Graph G after topological sort



(b) Removing node 1 after installing a rule on it



(c) Cleaning the graph

Figure 3.3 – Steps of applying the greedy heuristic-based algorithm for rule placement

An st-DAG naturally induces a partial order on its nodes, defined by the transitive closure of the relation $a \prec b$ whenever there exists a directed edge from a to b. Under this ordering, the source node s is the minimal element, while the destination node t is the maximal one. This partial order can be extended to a total order by performing a topological sorting of the graph. After topological sorting, the nodes of an st-DAG G with $|V|$ vertices can be indexed by consecutive integers from 0 (corresponding to the source) to $|V| - 1$ (corresponding to the destination).

Once the graph is ordered, we perform a left-to-right traversal (from s to t) to compute, for each node v, the number of paths from the source s to v. This quantity, denoted by $leftcount_G(v)$, is computed as described in Algorithm 4. Similarly, a right-to-left traversal (from t to s) computes, for each node v, the number of paths from v to the destination t. This second traversal is a symmetric version of Algorithm 4, obtained by reversing the direction of the edges and exchanging the roles of the source and the destination, and yields the quantity $rightcount_G(v)$. The total number of st-paths that pass through a node v is then given by $pathcount_G(v) = leftcount_G(v) \times rightcount_G(v)$.

Algorithm 4: Left count

Input: H, topologically sorted st-dag with nodes $0, \ldots, N-1$; 0, source; $N-1$, destination
Output: $leftcount_H$, dictionary associating to each node the number of paths from 0 to this node
Algorithm:

```
for i = 1 to N − 1 do
 | leftcount_H[i] ← 0
end
leftcount_H[0] ← 1
for i = 1 to N − 1 do
 | for v ∈ pred_H(i) do
 |  | leftcount_H[i] ← leftcount_H[i] + leftcount_H[v]
 | end
end
return leftcount_H
```

After computing the path counts for all nodes, the algorithm selects a node u that maximizes $pathcount_G(u)$, i.e., the node that lies on the largest number of *st-paths*. A rule set is then installed on node u. The graph is subsequently updated by removing node u together with all its incident edges. In addition, the graph is iteratively cleaned by removing any non-destination node that no longer has successors and any non-source node that no longer has predecessors.

The same procedure is then repeated on the reduced graph until no node with a non-zero path count remains. At this point, all st-paths are covered by this rule set. The complete greedy algorithm for placing a single rule set is described in Algorithm 5, and Figure 3.3 illustrates the execution of the algorithm during the first iteration. Based on this construction, we can establish the following proposition:

Proposition 2. *Given a topologically sorted st-dag H, $pathcount_H$ can be computed in $O(N+M)$ where N is the number of nodes and M the number of edges of H.*

The computational complexity of the greedy placement algorithm is analyzed in the following proposition:

Proposition 3. *Let $G = (V, E)$ be an st-DAG. The greedy algorithm for the placement of a single rule described in Algorithm 5 runs in $O(|V| \times (|V| + |E|))$ time in the worst case.*

Algorithm 5: Greedy placement algorithm

Input: G, st-dag; s, source; t, destination.
Output: O, set of nodes where to place the rule.
Algorithm:

```
H[0..N − 1] ← topological_sort(G)
Compute pathcount_H
O ← ∅
while ∃v such that pathcount_H[v] ≠ 0 do
    node ← argmax_u(pathcount_H[u])
    O ← append(node, O)
    H ← remove(node, H)
    for v = node to 1 do
        if pred_H(v) = ∅ then
            H ← remove(v, H)
        end
    end
    for v = node to N − 1 do
        if succ_H(v) = ∅ then
            H ← remove(v, H)
        end
    end
    Compute pathcount_H
end
return O
```

Proof. The algorithm performs at most $|V|$ iterations. At each iteration, the computation of $pathcount_H$ requires $O(|V|+|E|)$ time, which dominates the cost of node selection and graph updates. Therefore, the overall complexity is $O(|V| \times (|V| + |E|))$. □

Applying Greedy Heuristic to an Arbitrary Number of Rules

Given a set of k rule sets $F = \{f_1, f_2, \ldots, f_k\}$ to be placed on an st-DAG G, the single-rule placement procedure can be extended straightforwardly to handle multiple rule sets. The method proceeds iteratively, placing rule sets one at a time while preserving st-path coverage and respecting node capacity constraints. Specifically, for $j = 1, \ldots, k$, the following steps are performed as long as no failure is encountered in Step 3:

1. Associate each node $v \in V$ with a counter initialized to its memory capacity $m(v)$, representing the remaining number of rule sets that can be installed on node v.
2. Apply the greedy placement algorithm for a single rule set to identify the nodes on which rule set f_j should be installed.
3. Install rule set f_j on the selected nodes and decrement the corresponding counters. If the resulting graph still contains uncovered st-paths (i.e., the set of nodes with non-zero $pathcount_H$ is non-empty) but no node along these paths has sufficient

remaining memory to accommodate f_j, the procedure terminates and no feasible solution is found.

The computational complexity of the proposed extension follows directly from the single-rule placement procedure. Indeed, for each rule set, the greedy algorithm is executed once on the graph. Given that the placement of a single rule set has a worst-case complexity of $O(|V| \times (|V| + |E|))$, the total complexity for placing k rule sets is $O(k \times |V| \times (|V| + |E|))$.

3.3.3 Reinforcement Learning-Based Approach

This section presents a reinforcement learning formulation of the ACL rule placement problem, leveraging both Q-learning and deep Q-learning to learn efficient placement strategies over network topologies.

RL Pipeline for Rule Placement

As illustrated in Figures 3.4 and 3.5, the RL environment corresponds to the network graph $G(V, E)$. At each time step t, the agent observes a state s_t describing the current network topology and the distribution of rule sets across the switches. Based on this state, the agent selects an action a_t corresponding to the placement of a rule set on a specific node.

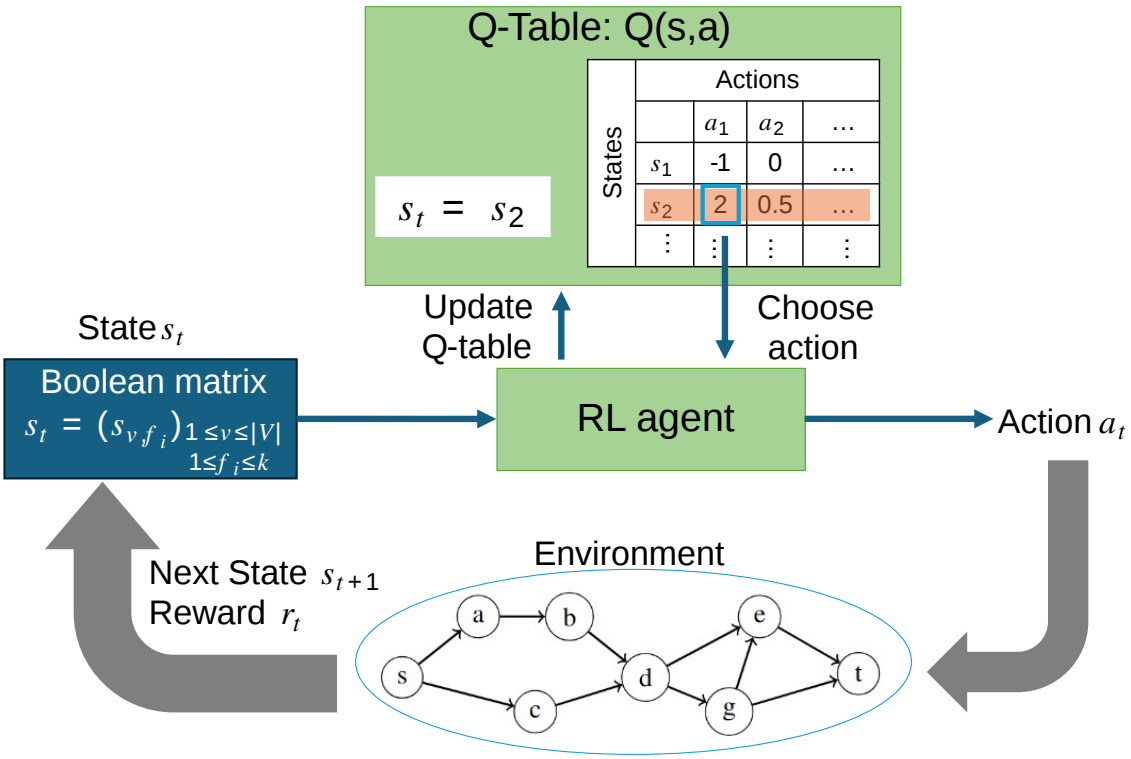


Figure 3.4 – Q-learning pipeline for ACL rule placement.

In both QL and DQN, the selected action aims to maximize the state–action value function $Q(s_t, a_t)$. The two approaches differ only in the way the Q-values are computed: tabular Q-learning relies on an explicit Q-table, whereas DQN approximates the Q-function using a NN, as described in Chapter 2, Section 2.2.2. After executing action a_t, the environment transitions to a new state s_{t+1} and returns a reward r_{t+1} that reflects the effectiveness of the placement decision. Through repeated interactions, the agent learns a placement policy adapted to the network topology and the ACL deployment objectives. Both QL and DQN share the same definitions of agent, action space, and reward function, which are detailed next.

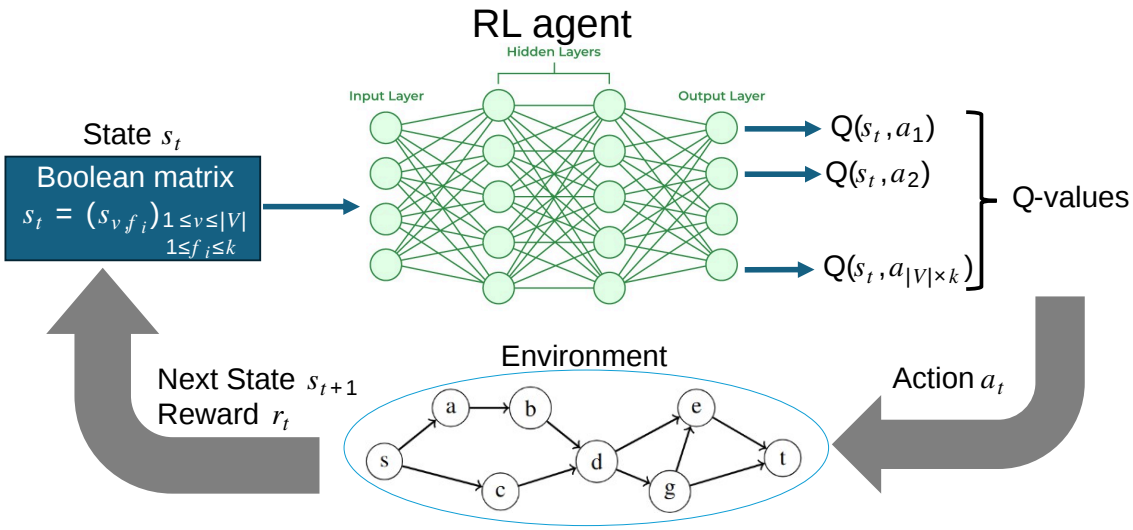


Figure 3.5 – Deep Q-learning pipeline for ACL rule placement.

RL Agent Components

We now define the RL components used for the ACL rule placement problem.

State: A state encodes both the current rule placement and the remaining path-coverage information over the network graph $G(V, E)$. It is represented by a matrix $S \in \mathbb{R}^{|V| \times 2k}$, where $|V|$ denotes the number of nodes in G and k is the number of rule sets, composed of two blocks: $S = [S^{place} \mid S^{cov}]$. The first block, $S^{place} \in \{0, 1\}^{|V| \times k}$, is a Boolean placement matrix. The entry S^{place}_{v,f_i} equals 1 if rule set f_i is installed on node v, and 0 otherwise. This block also allows the remaining memory of node v to be inferred through $m(v) - \sum_{i=1}^{k} S^{place}_{v,f_i}$. The second block, $S^{cov} \in \mathbb{R}^{|V| \times k}$, encodes the remaining coverage potential. The entry S^{cov}_{v,f_i} corresponds to the number of currently uncovered paths for rule set f_i that traverse node v. Therefore, a positive value of S^{cov}_{v,f_i} indicates that placing f_i on node v would contribute to covering additional paths. Consequently, the state provides the agent with both the already performed placements and the remaining coverage opportunities, allowing it to select actions that improve path coverage while respecting capacity constraints.

Action: An action $a(v, f_i)$ consists in installing rule set f_i on node v. Actions are classified into three categories:

— *Invalid actions*, which violate placement constraints (e.g., insufficient memory on v, v being the source or destination, f_i already installed on v, or v not belonging to any st-path);

— *Redundant actions*, which place f_i on v but do not contribute to covering any previously uncovered st-paths;

— *Valid actions*, which respect all constraints and contribute to improving path coverage.

Reward: The reward function is designed to guide the agent toward full st-path coverage while minimizing overall memory usage. A positive reward is assigned when an action covers new st-paths, whereas penalties are applied for invalid or redundant actions. The reward formulation is calculated using the greedy heuristic introduced in Section 3.3.2 and is defined as:

$$reward(a(v, f_i)) = \begin{cases} -\text{penalty}, & \text{if } a(v, f_i) \text{ is invalid or redundant,} \\ paths_{v,f_i} - occ, & \text{otherwise,} \end{cases}$$

where $paths_{v,f_i}$ denotes the number of previously uncovered st-paths that become covered by placing f_i on node v, and occ represents the total number of rule sets currently installed in the network, i.e., the overall memory occupation.

3.4 Experiments and Results

3.4.1 Simulation Setup

We evaluated the proposed algorithms on five real-world network topologies obtained from the Internet Topology Zoo [131]. Since these topologies are not directed acyclic graphs (DAGs), a preprocessing step was applied to convert each topology into an st-DAG. Specifically, a node with no predecessors was selected as the source s, while the node with the maximum distance from s was chosen as the destination t. During this process, some intermediate nodes may lie between s and t without successors. To ensure connectivity, for each such node n, we identified neighboring st-paths, selected a nearby node n' at random, and added a directed edge from n to n', thereby preserving reachability between s and t.

All algorithms were implemented in Python and executed on a laptop equipped with an AMD Ryzen 7 PRO 5850U CPU (1.90 GHz), 32 GB of RAM, running Windows 11. The algorithms were compared using two main performance metrics: *occupancy*, defined as the percentage of the total switch memory capacity occupied by rule sets (lower values indicate better performance), and the computation time required to obtain a solution for a given topology.

For the RL approaches, the penalty associated with invalid or redundant actions in the reward function was set to -100, encouraging the agent to avoid inefficient placements. Both QL and DQN were trained for 1000 episodes per graph. Each episode was limited to

a maximum of 500 steps, corresponding to 500 potential placement actions. An episode terminated under one of the following conditions: (i) all rule sets were successfully placed on all st-paths, (ii) the maximum number of steps was reached without achieving full coverage, or (iii) all nodes were saturated while uncovered paths still remained.

Both QL and DQN employed an ϵ-greedy exploration strategy with an initial exploration rate $\epsilon_0 = 1$, a minimum exploration rate $\epsilon_{\min} = 0.09$, and a decay factor defined as $\lambda = \exp\left(\frac{2\ln(0.5)}{M}\right)$, where M denotes the total number of episodes. This schedule ensures that the exploration rate decreases to 0.5 halfway through training, favoring exploration during the first half of the learning process and exploitation thereafter. The theoretical foundations and algorithmic details of QL and DQN are presented in Chapter 2, Section 2.2.2, Algorithms 1 and 2, respectively.

For the QL algorithm, the discount factor was set to $\gamma = 0.8$ to emphasize long-term rewards and promote strategic placement decisions, while the learning rate was set to $\alpha = 0.6$ to enable faster convergence through larger Q-value updates. For the DQN approach, the discount factor was also fixed at $\gamma = 0.8$. The learning rate was set to $\alpha = 10^{-5}$ to balance training stability and convergence speed. Training was performed using minibatches of size 32, and a target network update frequency $C = 1000$ steps was adopted, corresponding to the number of steps between successive synchronizations of the online and target networks.

Proposition 4. *Let the exploration rate be updated recursively according to $\epsilon_t = \lambda\,\epsilon_{t-1}$, with initial value $\epsilon_0 = 1$ and decay factor $\lambda \in (0,1)$. If λ is chosen as*

$$\lambda = \exp\left(\frac{2\ln(0.5)}{M}\right),$$

then the exploration rate satisfies $\epsilon_{M/2} = 0.5$. Moreover, the sequence $(\epsilon_t)_{t=0}^{M}$ is monotonically decreasing, and therefore

$$\epsilon_t \geq 0.5 \quad \textit{for } t \leq \frac{M}{2}, \qquad \epsilon_t \leq 0.5 \quad \textit{for } t \geq \frac{M}{2}.$$

Proof. Starting from the recursive definition

$$\epsilon_t = \lambda\,\epsilon_{t-1},$$

one obtains by successive substitution

$$\epsilon_t = \lambda\epsilon_{t-1} = \lambda^2\epsilon_{t-2} = \cdots = \lambda^t\epsilon_0.$$

Since $\epsilon_0 = 1$, it follows that

$$\epsilon_t = \lambda^t.$$

Evaluating at $t = \frac{M}{2}$ gives

$$\epsilon_{M/2} = \lambda^{M/2}.$$

Using the proposed value of λ, we obtain

$$\epsilon_{M/2} = \left(\exp\left(\frac{2\ln(0.5)}{M}\right)\right)^{M/2} = \exp\left(\frac{M}{2}\cdot\frac{2\ln(0.5)}{M}\right) = \exp(\ln(0.5)) = 0.5.$$

Since $\lambda \in (0,1)$, the sequence $(\epsilon_t)_{t=0}^{M}$ is strictly decreasing. Therefore,

$$\epsilon_t \geq 0.5 \quad \text{for } t \leq \frac{M}{2}, \qquad \epsilon_t \leq 0.5 \quad \text{for } t \geq \frac{M}{2}.$$

This concludes the proof. □

This result shows that the adopted decay schedule maintains a relatively high exploration rate during the first half of training, while progressively shifting the learning process toward exploitation during the second half.

Action-Masked DQN

To improve exploration efficiency and reduce the number of infeasible decisions during training, the DQN agent was further extended with an action-masking mechanism. At each decision step, a Boolean mask is dynamically computed from the current state in order to identify valid actions only. Actions violating placement constraints, such as selecting the source or destination node, exceeding the memory capacity of a switch, reinstalling an already deployed rule set, or selecting a node that does not contribute to additional path coverage, are masked and excluded from the action-selection process.

Formally, $\mathcal{A}$ denote the complete action space and $\mathcal{A}_{valid}(s_t) \subseteq \mathcal{A}$ the subset of feasible actions in state s_t. During both exploration and exploitation, the agent selects actions exclusively from $\mathcal{A}_{valid}(s_t)$. In the exploitation phase, Q-values associated with invalid actions are artificially suppressed before the arg max operation:

$$Q(s_t, a) = \begin{cases} Q(s_t, a), & a \in \mathcal{A}_{valid}(s_t), \\ -\infty, & \text{otherwise.} \end{cases}$$

This masking strategy significantly reduces the exploration of infeasible regions of the action space and accelerates convergence by guiding the agent toward constraint-compliant placements.

3.4.2 Results and Discussion

In our experiments, all switches are assumed to have identical capacity, allowing the installation of a single rule set per switch. The number of rules is set equal to the placement capacity of the shortest st-path, i.e., $k = path_capacity(p_{\min})$, corresponding to the number of switches along that path.

Table 3.2 reports the performance of the proposed algorithms on real-world network topologies from the Internet Topology Zoo. The results show that both the exact st-mincut formulation and the masked DQN consistently achieve the best placement quality in terms of memory occupancy across all evaluated topologies. The st-mincut approach computes optimal placements by exploiting the structural properties of the graph through minimum-cut optimization, while the masked DQN is able to learn equivalent placement strategies by restricting exploration to feasible actions only. In contrast, the standard DQN and QL approaches generally produce higher occupancy values on large and highly

		st-mincut		Greedy		QL		DQN		masked_DQN	
Graph	♯ of rules	Occupancy (%)	Time (s)	Occupancy (%)	Time (s)	Occupancy (%)	Time (s)	Occupancy (%)	Time (s)	Occupancy (%)	Time (s)
Sprint/11	3	55.56	0.0021	55.56	0.0087	55.56	621.32	55.56	13	55.56	6
Abvt/23	6	61.9	0.0062	71.43	0.022	71.43	6182.95	71.43	106	61.9	85
Agis/25	5	56.52	0.0061	-	-	78.26	2365.8	78.26	1352.03	56.52	338.26
Bics/34	8	46.88	0.036	-	-	96.88	9156.37	95.6	2257.58	46.88	824.843
NetworkUSA/35	7	45.45	0.03	48.48	0.036	51.52	8504.16	51.52	1102.4	45.45	239.6

Table 3.2 – Memory occupancy and processing time of our rules placement algorithms on real-world network topologies.

connected graphs due to the approximate nature of the learned policy and the difficulty of efficiently exploring the large action space.

The greedy heuristic performs well on small or moderately sized graphs such as Sprint and Abvt, but fails on larger and more densely connected topologies, including Agis and Bics. In contrast, NetworkUSA exhibits a relatively sparse structure with fewer overlapping paths, which simplifies the placement process and enables the greedy algorithm to find a feasible solution. The failure of the greedy approach on larger graphs can be attributed to its locally optimal decision-making strategy: at each iteration, it selects the node covering the maximum number of paths. While this maximizes short-term gain, it often saturates highly connected junction nodes early, thereby preventing the placement of remaining rule sets on some paths and ultimately leading to infeasible solutions.

RL-based approaches demonstrate a different behavior. Both QL and DQN are able to solve instances where the greedy heuristic fails, notably on the Agis and Bics topologies. This highlights the ability of RL to optimize long-term rewards rather than relying exclusively on immediate local gains. Unlike greedy placement, RL agents progressively learn deployment policies that preserve future placement flexibility across the remaining uncovered paths. On small graphs, DQN and QL solutions exhibit occupancy levels comparable to those obtained by exact and heuristic methods. However, as the graph size and connectivity increase, the occupancy obtained by standard QL and DQN becomes higher than that of the st-mincut solution. This behavior is mainly explained by the rapid growth of the action space, whose size is proportional to the product of the number of nodes and the number of rule sets. Moreover, only a small subset of this action space corresponds to valid placement decisions, since many actions violate capacity constraints, e.g., reinstall already deployed rule sets, target source or destination nodes, or do not contribute to additional path coverage. While exhaustive exploration of state-action pairs remains feasible for small topologies, it becomes computationally prohibitive for larger graphs. Consequently, the RL agent converges toward approximate solutions that remain feasible but are not always occupancy-optimal. This observation motivates the introduction of the masked DQN approach, which restricts the exploration process to feasible actions only in order to improve both convergence efficiency and placement quality.

The results also reveal a clear difference between tabular QL and neural-network-based DQN. Although both approaches produce similar occupancy values, DQN considerably reduces training time compared to QL, especially on large topologies. This improvement stems from the generalization capability of neural networks, which allows DQN to approximate the Q-function over similar states instead of explicitly storing and updating a large Q-table for every state-action pair. In contrast, QL suffers from the curse of dimensionality, making convergence increasingly slow as the state space expands. By leveraging

NNs as function approximators, DQN achieves substantially faster convergence, reducing the training time from approximately 30 minutes to about 13 seconds.

The introduction of action masking significantly improves both convergence efficiency and placement quality. By dynamically restricting the action-selection process to valid actions only, the masked DQN avoids exploring infeasible decisions. As shown in Table 3.2, this mechanism enables the masked DQN to recover occupancy values identical to those of the st-mincut approach on all evaluated topologies, while simultaneously reducing computation time compared to the standard DQN formulation. The benefits of action masking become particularly significant on large and highly constrained topologies such as Agis, Bics, and NetworkUSA, where the proportion of invalid actions grows substantially. In these scenarios, standard DQN spends a significant portion of training exploring infeasible regions of the action space, whereas masked DQN focuses the exploration process exclusively on constraint-compliant actions. This considerably accelerates convergence and improves policy quality. These results demonstrate that integrating structural constraints directly into the RL decision process can substantially enhance the scalability and effectiveness of DRL for distributed ACL placement problems.

It is important to note that the comparison of computation times between RL-based and non-RL-based approaches is not entirely fair. Exact and heuristic algorithms follow predefined decision rules and terminate once a solution is found, whereas RL algorithms are model-free and require an extensive training phase starting from random exploration. The reported execution time for RL corresponds to the full training process and the final policy extraction.

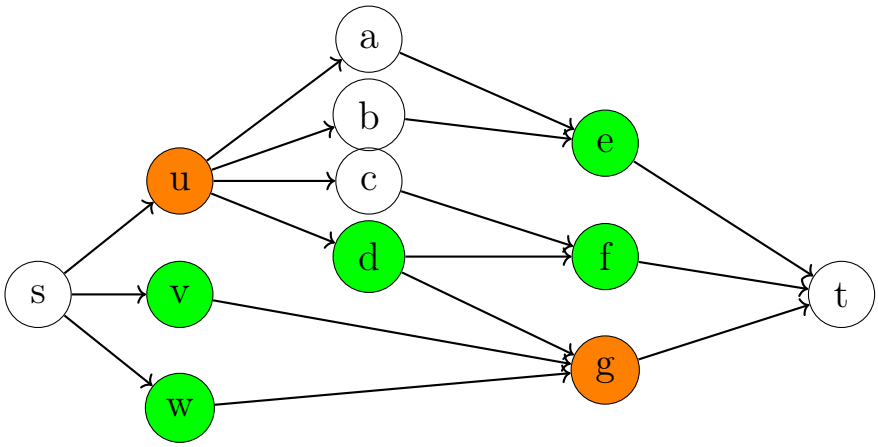


Figure 3.6 – Comparison between RL and non-RL based placements, illustrating the long-term optimality of RL decisions. The example considers two rule sets, f_1 and f_2, with unit switch capacities. Orange nodes denote the greedy placement of f_1, while green nodes denote the subsequent placement of f_2 needed to cover all paths. This illustrates how an initially greedy choice may increase final memory occupancy.

Local Coverage Maximization vs Long-Term Placement Efficiency.

To further illustrate the advantage of RL in terms of long-term decision-making, we consider the network shown in Figure 3.6, where each node has a capacity $m = 1$ and $F = \{f_1, f_2\}$. Greedy approach and st-mincut prioritize immediate coverage, placing rule f_1 on nodes u and g and rule f_2 on nodes e, f, d, v, and w, resulting in a memory occupancy of 70%. In contrast, the RL agent adopts a more strategic placement, installing f_1 on nodes u, v, and w and f_2 on nodes e, f, and g, thereby achieving a lower occupancy of 60%. This example illustrates how RL may temporarily sacrifice short-term optimality to achieve better long-term outcomes.

Failure of Greedy and st-mincut under Capacity Constraints

The previous results show that local sequential placement strategies may fail on some graph instances, even when a feasible placement exists. In particular, the greedy heuristic may fail because it selects, at each iteration, the node that covers the largest number of currently uncovered paths, without accounting for the impact of this choice on the placement of subsequent rule sets. Similarly, while the st-mincut formulation is exact for the placement of a single rule set, its iterative extension to multiple rule sets may also lead to suboptimal or infeasible placements, because each rule set is placed independently of future capacity requirements. The purpose of the following example is to illustrate this limitation and to highlight the advantage of RL-based placement, which can learn decisions that preserve long-term feasibility. Consider the st-DAG shown in Figure 3.7.

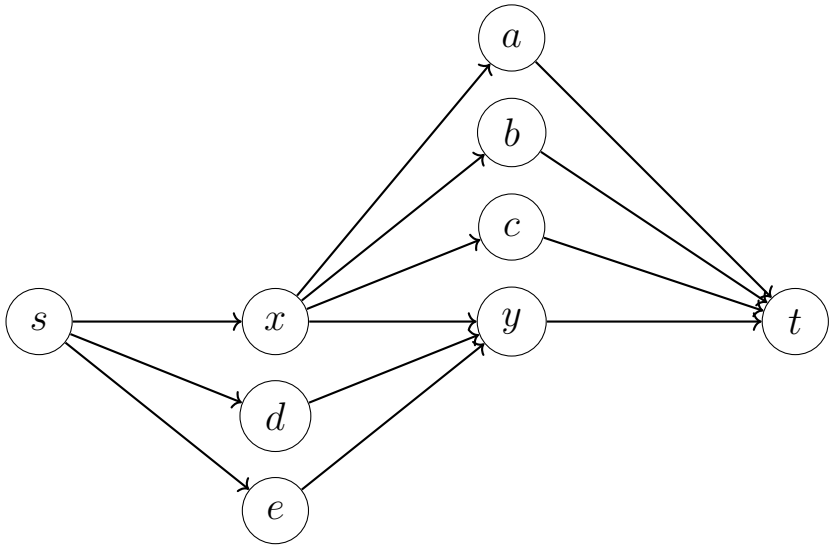


Figure 3.7 – Counterexample graph illustrating the failure of local sequential placement decisions.

The graph contains two rule sets, $F = \{f_1, f_2\}$, and all intermediate nodes have unit capacity, i.e., $m(v) = 1$. The set of admissible paths is:

$$
\begin{aligned}
p_0 &= s \to x \to y \to t, \\
p_1 &= s \to x \to a \to t, \\
p_2 &= s \to x \to b \to t, \\
p_3 &= s \to x \to c \to t, \\
p_4 &= s \to d \to y \to t, \\
p_5 &= s \to e \to y \to t.
\end{aligned}
$$

For the first rule set, the greedy heuristic selects node x, since it covers four paths, namely p_0, p_1, p_2, and p_3. After these paths are removed from the uncovered set, the remaining paths are p_4 and p_5, both of which pass through node y. The greedy heuristic, therefore, selects y for the same rule set. As a consequence, both nodes of the short path $p_0 = s \to x \to y \to t$ are saturated by f_1, leaving no available capacity on this path for f_2. The greedy procedure therefore fails, although the instance is feasible.

A similar failure may occur with the st-mincut approach. For a single rule set, the minimum cut is $\{x, y\}$, since this pair intersects all paths with only two placements. Hence, when the first rule set is placed, the algorithm installs it on x and y. However, since both nodes have unit capacity, the short path p_0 becomes fully saturated, and the second rule set cannot be installed on any node of this path. This does not contradict the optimality of st-mincut for the single-rule case; rather, it shows the limitation of applying it sequentially to multiple rule sets without anticipating future capacity requirements.

In contrast, the RL agent adopts a more strategic placement, installing f_1 on nodes $\{x, d, e\}$ and f_2 on nodes $\{y, a, b, c\}$. In this placement, every st-path contains both rule sets: path p_0 contains f_1 on x and f_2 on y; paths p_1, p_2, and p_3 contain f_1 on x and f_2 on a, b, and c, respectively; and paths p_4 and p_5 contain f_1 on d and e, respectively, and f_2 on y.

This example highlights a stronger limitation than the increased occupancy observed in the previous example. In this case, the locally sequential strategies do not merely produce a less efficient placement; they fail to find any feasible placement, although one exists. The failure is caused by the premature saturation of the critical short path $p_0 = s \to x \to y \to t$, where both available rule slots are consumed by the same rule set. By contrast, the RL-based approach can identify an alternative allocation that distributes the two rule sets across complementary nodes and preserves coverage on every admissible path. This illustrates the ability of RL to overcome specific feasibility failures of greedy and st-mincut strategies in constrained placement instances.

Finally, we note that the proposed algorithms are not limited to ACL rule placement and can be readily applied to other device or function placement problems in networking, as they operate independently of the specific semantics of the rule sets being distributed.

This section demonstrates that RL is capable of solving graph instances that cannot be handled by the other considered approaches. Furthermore, an important limitation remains: the learned policy is tightly coupled to the graph instance used during training. As a consequence, each new topology—or any modification to an existing one—requires a dedicated training phase. Addressing this generalization issue constitutes the main focus of the next chapter.

3.5 Summary

In this chapter, we addressed the problem of distributed ACL rule placement under dynamic routing constraints, with the objective of ensuring complete st-path coverage while minimizing switch memory occupancy. We first proposed a polynomial-time solution based on an st-mincut formulation, which provides an optimal placement when a single rule set is to be installed. We then introduced a greedy heuristic that iteratively selects nodes maximizing path coverage. Finally, we investigated RL-based approaches, including tabular QL, DQN, and masked DQN. The proposed methods were evaluated on five real-world network topologies and compared in terms of memory occupancy and computation time. The results show that the st-mincut approach serves as the reference optimal method for the single-rule case and achieves the lowest execution time. The greedy heuristic performs well on small and sparse graphs but fails on larger or highly connected topologies due to its locally optimal decision-making strategy. Tabular QL demonstrates that RL can discover feasible placement strategies in cases where greedy heuristics fail, highlighting the benefit of long-term sequential decision-making, but its convergence time becomes prohibitively high as the state-action space grows. DQN mitigates this limitation through neural-network-based function approximation, improving convergence speed compared to QL; however, standard DQN still suffers from inefficient exploration in large constrained action spaces and may produce higher occupancy than the st-mincut solution. In contrast, action-masked DQN explicitly integrates placement constraints into the decision process by restricting exploration to feasible actions only, enabling it to recover occupancy values identical to those of the st-mincut approach across all evaluated topologies while reducing convergence time. Overall, this chapter reveals a trade-off between optimality, computational efficiency, and adaptability. While st-mincut and heuristic methods provide fast and effective solutions under favorable conditions, RL offers greater flexibility in complex scenarios at the cost of training effort. Nevertheless, the learned policies remain topology-dependent, and any graph modification requires retraining, which motivates the graph-based RL extensions introduced in the next chapter.

Chapter 4

Topology-Aware Distributed Placement of ACLs Using GNN-DQN

Contents

4.1 Introduction 65
4.2 Learning over Variable-Size Graphs 66
4.2.1 Dimensional Normalization via Padding 67
4.2.2 Topology-Aware Representation via GNNs 68
4.3 ACL Rules Placement Using GNN-DQN 69
4.3.1 Architecture and Learning Framework 69
4.3.2 Agent State, Action, and Reward Design 72
4.4 Experimental Setup and Methodology 74
4.4.1 Experimental Configuration 74
4.4.2 Synthetic Network Topology Generation 75
4.5 Evaluation and Performance Analysis 77
4.5.1 Training Behavior and Model Performance 77
4.5.2 Generalization Across Graph Sizes and Rule Configurations . . 79
4.6 Distance-Constrained ACL Rule Placement 80
4.6.1 Motivation and Definition of Distance Constraints 80
4.6.2 Adaptation of the GNN-DQN Framework 83
4.6.3 Experimental Results 84
4.7 Summary 86

4.1 Introduction

The previous chapter investigated the distributed ACL placement problem under dynamic routing constraints and proposed exact, heuristic, and RL-based solutions to ensure *full st-path coverage* while minimizing switch memory occupancy under TCAM capacity constraints. The experimental results demonstrated that RL methods, in particular, are

capable of balancing short-term and long-term objectives, and can identify feasible solutions in complex graph instances where heuristic strategies may fail.

Despite these promising results, an important limitation of the RL framework presented in Chapter 3 lies in its strong dependency on the specific network topology used during training. The learned policy is intrinsically tied to the structure, size, and connectivity pattern of the training graph. Consequently, even minor structural modifications - such as the addition or removal of links, switch failures, planned maintenance operations, or traffic engineering adjustments - modify the state-action space and render the previously learned Q-function suboptimal or even invalid.

In operational networks, such structural changes are not exceptional but rather inherent to their dynamic nature. Link failures, node outages, routing updates, incremental deployment of new devices, and topology reconfigurations occur regularly. Under these conditions, retraining an RL agent from scratch for every topology modification becomes computationally expensive and operationally impractical, particularly in large-scale or highly dynamic environments. This lack of structural generalization significantly limits the scalability and real-world applicability of classical QL and DQN approaches for distributed rule placement. Addressing this issue requires shifting from topology-specific value learning toward topology-aware representation learning capable of capturing structural information in a transferable manner.

To overcome this limitation, this chapter introduces a learning framework that integrates Graph Neural Networks (GNNs) [47, 48] with DQN. By leveraging graph representation learning, the proposed GNN–DQN architecture encodes the structural properties of the network directly into latent node embeddings. These embeddings enable the agent to operate on variable-size graphs, preserve relational dependencies between nodes, and exploit permutation-invariant representations of the topology. As a result, the learning process becomes more scalable and structurally adaptive across different network instances, while preserving the original objective of ensuring complete st-path coverage with minimal memory occupancy.

Section 4.2 motivates the need for graph-aware learning and presents the techniques adopted to train the RL agent on graphs of varying sizes, highlighting the advantages of GNN-based representations. Section 4.3 details the proposed GNN–DQN algorithm for distributing ACL rules across all st-paths. Section 4.4 introduces the synthetic graph generation tool and the experimental setup, while Section 4.5 reports and analyzes the obtained results. Section 4.6 presents an early filtering constraint designed to improve training efficiency, and Section 4.7 concludes the chapter.

4.2 Learning over Variable-Size Graphs

In standard DQN formulations, the dimension of the NN input corresponds to the dimension of the state representation, while the dimension of the output layer equals the size of the action space. This assumption implicitly requires both the state and action spaces to have fixed dimensions. However, in graph-based problems, the state typically encodes the graph structure together with its current status (e.g., node attributes or rule placements matrix), and is therefore inherently proportional to the number of nodes in the

graph. Similarly, the action space is often defined over graph elements-such as selecting a node or a node-rule pair-and thus its size also scales with the number of nodes.

Consequently, when applying DQN directly to graph-based problems, both the input layer and the output layer become dependent on the graph size. This dependency introduces several critical challenges. First, NNs designed with fixed input and output dimensions cannot naturally accommodate graphs of varying sizes. If the graph size changes, the dimensionality of both the state representation and the action space changes accordingly, requiring a redefinition of the NN architecture. Second, such architectural modifications imply reinitializing parameters, retraining the network, and reconstructing the Q-function approximation from scratch, thereby increasing computational overhead and reducing scalability.

Moreover, dynamically changing the network architecture during training may destabilize the learning process. If the graph becomes larger, additional neurons and connections must be introduced and initialized with random weights. These newly introduced parameters may generate unreliable Q-value estimates and perturb previously learned representations. Conversely, if the graph becomes smaller, the output layer may still contain actions corresponding to nodes that no longer exist in the graph, leading to invalid action selections and inefficient exploration. Overall, the direct application of classical DQN to variable-size graphs lacks structural generalization and suffers from poor adaptability to topology changes.

To address these limitations, it is necessary to decouple the NN architecture from the graph size by enforcing fixed input and output dimensions. Two complementary solutions are investigated in this chapter: padding-based dimensional normalization and GNN-based topology-aware representation learning. These approaches are detailed in the following subsections.

4.2.1 Dimensional Normalization via Padding

A straightforward approach to handle variable-size graphs within a classical DQN framework consists in artificially enforcing fixed input and output dimensions through a padding mechanism [132, 133, 134]. This approach consists of padding smaller graphs to match the size of the largest graph in the dataset. By introducing dummy nodes and masked actions, padding ensures uniform input and output dimensions across all instances.

Let $N_{\max}$ denote the maximum number of nodes considered across all training graphs. For a graph $G = (V, E)$ with $|V| < N_{\max}$, the state representation matrix from the previous chapter is

$$S \in \{0,1\}^{|V|\times 2\times |F|} \tag{4.1}$$

where $|F|$ is the number of rules to be installed, is extended to a fixed-size matrix

$$\tilde{S} \in \{0,1\}^{N_{\max}\times 2\times |F_{\max}|} \tag{4.2}$$

by introducing $N_{\max} - |V|$ dummy nodes and padding the rule dimension up to $|F_{\max}|$ when necessary. These additional rows are filled with zeros and do not correspond to actual switches in the network. Similarly, the action space is expanded to a fixed size of $N_{\max} \times$

$|F_{\max}|$, and actions associated with dummy nodes are masked to prevent their execution. Similarly, the action space is expanded to a fixed size of $N_{\max} \times |F_{\max}|$. Actions associated with dummy nodes or padded rule positions are masked to prevent their selection during training and inference. This procedure ensures dimensional consistency, allowing the use of a standard fully connected DQN architecture without modifying the network structure when processing graphs of different sizes. From an implementation perspective, padding is simple and compatible with classical DRL pipelines.

However, padding addresses only the dimensionality issue and does not incorporate any structural information about the graph topology. The neural network still receives a flattened representation of the state, in which nodes are implicitly ordered and treated independently. As a result, the model does not exploit relational dependencies between nodes, nor does it benefit from permutation-invariant representations. More importantly, padding assumes that the positional index of a node in the state matrix carries consistent semantic meaning across different graphs. In practice, this assumption does not hold. A node that plays a critical structural role in one topology - e.g., a highly connected junction or a bottleneck node - may correspond to a completely different structural position in another graph. Since padding preserves only dimensional alignment and not relational context, the same index in the input matrix may represent nodes with entirely different structural importance across training instances.

Consequently, the agent may associate learned Q-values with positional indices rather than with structural roles. This mismatch can destabilize the learning process and impair convergence when generalizing across graphs. Preliminary experiments conducted during this work revealed that DQN with padding struggled to converge reliably when trained on graphs of varying sizes and connectivity patterns. The learned policy exhibited high variance and failed to generalize across topologies.

These observations suggest that resolving dimensional inconsistency alone is insufficient. A representation mechanism capable of encoding relational and structural information is required to achieve meaningful generalization across graphs. This motivates the adoption of GNNs, which are discussed in the next subsection.

4.2.2 Topology-Aware Representation via GNNs

As discussed in Chapter 2 (Section 2.2.3), GNNs are specifically designed to operate on graph-structured data by incorporating structural inductive biases such as permutation invariance and locality. In contrast to padding-based approaches, which rely on fixed positional indexing, GNNs construct representations that are intrinsically dependent on the topology of the graph.

Building upon the state representation introduced in expression 4.1, which encodes the distribution of rule sets over the nodes of the graph, we replace the flattened matrix-based encoding with a topology-aware representation learned through a GNN. Rather than treating nodes as independent entries in a fixed-dimensional tensor, the GNN computes node embeddings that jointly capture node-level attributes and structural information derived from the connectivity of the graph.

Through the message-passing mechanism described in Chapter 2, node representations are iteratively refined by aggregating information from neighboring nodes. After L layers,

each node embedding $h_v^{(L)}$ incorporates multi-hop structural context, enabling the agent to reason not only about the local state of a node but also about its role within the global st-DAG structure. This representation provides several fundamental advantages over padding-based encoding:

— **Size invariance:** The same GNN architecture can process graphs of varying sizes without altering the number of learnable parameters.

— **Permutation invariance:** Node embeddings are independent of arbitrary indexing, eliminating positional bias introduced by fixed matrix representations.

— **Structural awareness:** The importance of a node is inferred from its relational context within the graph rather than from its position in the state tensor.

— **Parameter sharing:** Identical transformation functions are applied across all nodes, promoting inductive generalization across different topologies.

In the distributed ACL placement problem, the relevance of a node is determined by its structural contribution to st-path coverage. Nodes that constitute bottlenecks or lie on a large number of st-paths naturally acquire embeddings that reflect their strategic importance, whereas peripheral nodes obtain representations consistent with their limited influence. Unlike padding, which preserves only dimensional consistency, GNN-based representations dynamically adapt to the underlying topology.

Consequently, the Q-function is no longer learned over a flattened state vector tied to a specific graph instance. Instead, it operates on topology-aware node embeddings that encode relational dependencies and multi-hop structural information. This shift from positional encoding to structural representation enables the RL agent to generalize across graphs with different sizes and connectivity patterns, while preserving the objective of ensuring complete st-path coverage under TCAM capacity constraints. The integration of these topology-aware embeddings within the DQN framework is presented in the next section.

4.3 ACL Rules Placement Using GNN-DQN

4.3.1 Architecture and Learning Framework

GNN-DRL agent design.

This section addresses the same distributed rule placement problem introduced in Chapter 3, namely the placement of all rule sets over all st-paths of a graph $G(V, E)$ under node capacity constraints. The objective remains to ensure that every st-path contains exactly one occurrence of each rule while minimizing memory occupation.

Figure 4.1 illustrates the proposed GNN–DQN architecture. The environment corresponds to the network graph $G(V, E)$ together with the current distribution of installed rule sets. At each discrete time step t, the agent observes the current placement configuration over the network graph, represented by state s_t, which encodes both the topology of $G(V, E)$ and the set of rule sets currently installed on the switches. Based on this configuration, the agent selects an action a_t corresponding to the installation of a specific rule on a candidate node. The environment then updates the rule distribution and returns a

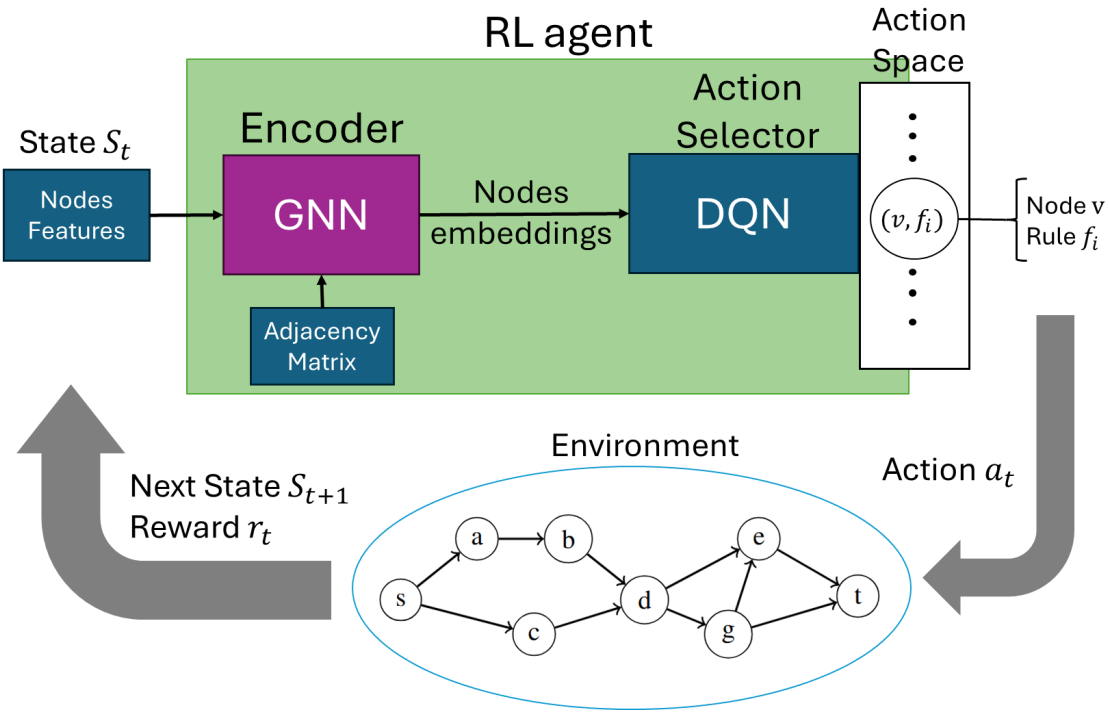


Figure 4.1 – GNN-DQN architecture dedicated to rule placement.

reward r_t that reflects the contribution of this placement to st-path coverage and memory efficiency.

To obtain a topology-aware representation of the state, we employ MPNN [55] as a graph encoder. The MPNN processes the graph together with node features encoding rule occupancy and structural attributes, and produces node embeddings $h_v^{(L)}$ after L message-passing iterations. These embeddings capture both local and multi-hop structural information, as detailed in Chapter 2.

Although several GNN variants exist, GCN, GAT, and GraphSAGE, we adopt the MPNN framework as the graph encoder due to its suitability for the structural properties of the distributed ACL placement problem. Our environment is an st-DAG in which the placement decision depends primarily on multi-hop structural dependencies and path coverage relationships rather than on local homophily or neighbor similarity. The MPNN framework provides a general and expressive formulation capable of propagating information across multiple hops through iterative message passing. This property is essential for capturing the global impact of installing a rule on a given node, since such a decision influences multiple st-paths simultaneously. Unlike GCN, which relies on normalized neighborhood averaging and is particularly effective under homophily assumptions, the ACL placement problem does not exhibit similarity-based neighborhood structure. Moreover, while attention-based mechanisms such as GAT can learn adaptive importance weights over neighbors, the underlying st-DAG topology is structured and deterministic rather than noisy, reducing the need for attention-driven filtering. GraphSAGE, which is

designed for large-scale inductive settings with neighborhood sampling, is not required in our context due to the moderate size of the network graphs. The MPNN formulation offers a principled balance between expressive power and architectural simplicity. It preserves permutation invariance, naturally incorporates edge directionality, and enables multi-hop structural reasoning without introducing unnecessary complexity. Therefore, it constitutes an appropriate and well-justified choice for encoding topology-aware representations in our proposed GNN–DQN framework. These embeddings are subsequently passed to the DQN, which serves as the decision-making component. The Q-function is approximated as

$$\hat{Q}(s_t, a_t; \Theta) = F\left(\left[I\left(\sum_{u \in V} h_u^{(T)}\right),\ J\left(h_v^{(T)}\right)\right]\right), \tag{4.3}$$

where $h_v^{(L)}$ denotes the embedding of the node targeted by action a_t, and $\sum_{u \in V} h_u^{(L)}$ provides a global graph representation obtained through permutation-invariant aggregation. The functions $I(\cdot)$, $J(\cdot)$, and $F(\cdot)$ are learnable nonlinear transformations implemented as neural network layers. This formulation enables the Q-network to jointly consider (i) the global structural state of the graph and (ii) the local structural role of the candidate node. Unlike the classical DQN formulation in Chapter 3, which relied on flattened state representations tied to a specific topology, the proposed architecture operates on topology-aware embeddings and is therefore adaptable to graphs of varying sizes. After executing action a_t, the environment transitions to a new state s_{t+1} and returns reward r_t. The interaction continues iteratively, allowing the agent to learn a placement policy that maximizes the cumulative expected reward while satisfying coverage and capacity constraints.

Encoding the joint rule–node action space Most existing GNN-based RL approaches define the action space directly over the nodes of the graph. In such settings, each action corresponds to selecting a single node, and the Q-network outputs one scalar value per node embedding. This modeling paradigm is adopted in several combinatorial optimization works combining GNNs with RL, such as the framework of Khalil et al. [127], where actions correspond to node selection, and the approach of Barrett et al. [128], where node addition or removal decisions are encoded through the state rather than by expanding the output layer. Similar node-level action modeling is also employed in neural combinatorial optimization methods for routing and graph search problems [125, 126].

In contrast, the distributed ACL placement problem requires selecting a pair (v, f), where $v \in V$ is a candidate node and $f \in F$ is a rule set. The resulting action space has cardinality $|V| \times |F|$, which exceeds the standard node-level output dimensionality commonly adopted in graph-based RL architectures. This structured action space necessitates an explicit modeling strategy to generate $|V| \times |F|$ action values while preserving permutation invariance and structural inductive bias.

To accommodate this structured action space without altering the underlying message-passing formulation, we construct an augmented representation composed of $|F|$ copies of the original graph. Each copy corresponds to a specific rule set and models its placement

independently, while node capacity constraints are enforced globally by sharing capacity-related features across all copies. This layered construction enables the Q-network to estimate action values over the joint rule-node space while preserving permutation invariance and structural inductive bias.

Learning algorithm. Algorithm 6 presents the proposed GNN–DQN training procedure for the distributed rule placement problem. Training is conducted over M episodes. At the beginning of each episode, a graph $G(V, E)$ is sampled from the dataset $\mathcal{S}$ and the environment is reset. The node features, denoted by $features$, are then initialized using the structural attributes of the graph and an initial empty placement state, where no rule set has yet been installed. The definitions of node features, state encoding, action space, and reward function are detailed in Section 4.3.2. At each time step t, node embeddings are computed through the GNN message-passing procedure, producing topology-aware representations of the current placement configuration. Based on these embeddings, an action a_t is selected using an ϵ-greedy strategy, balancing exploration and exploitation. The selected action corresponds to installing a rule on a candidate node, after which the environment updates the placement configuration and returns the corresponding reward r_t. Transitions $(s_t, a_t, r_t, s_{t+1}, done_t)$ are stored in the replay buffer, and the DQN parameters are updated by minimizing the temporal-difference loss using minibatch stochastic gradient descent, as described in Chapter 2. The target network is periodically synchronized to stabilize learning. An episode terminates when either all st-paths are successfully covered under capacity constraints or a predefined step limit $MaxTry$ is reached. To evaluate generalization across topologies, the trained agent is periodically assessed on a validation set composed of graphs not seen during training. Validation is performed every val_freq episodes to monitor performance on unseen network instances.

4.3.2 Agent State, Action, and Reward Design

The RL setting remains consistent with the placement objective introduced in Chapter 3, namely achieving complete st-path coverage while respecting switch capacity constraints. In this section, we describe the state, action, and reward formulation adapted to the GNN-DQN framework.

State Space A state represents the current rule installation configuration over the network graph $G(V, E)$. It comprises the graph structure (i.e., the adjacency matrix) together with node feature representations that encode the placement status of all rule sets. For each rule f_i in the set of rules F, we associate a distinct copy of node features over the same underlying graph G. Consequently, each node $v \in V$ is associated with $|F|$ feature vectors $x_v^1, \ldots, x_v^{|F|}$, where each $x_v^i \in \mathbb{R}^5$ corresponds to rule set f_i. This layered representation enables independent modeling of rule-specific coverage, while preserving a common structural backbone. For a given node v and rule set f_i, the feature vector x_v^i consists of: x

$x_v^i[1]$ Node degree (i.e., number of edges incident to v);

$x_v^i[2]$ Boolean, true if the rule f_i is installed on node v and false otherwise;

$x_v^i[3]$ Number of paths covered by v and that contain f_i;

$x_v^i[4]$ Number of paths covered by v and that does not contain f_i;

Algorithm 6: GNN-DQN training procedure for distributed rule placement

Initialize replay buffer D;
Initialize online parameters θ and target parameters $\theta^- \leftarrow \theta$;
for *episode* $= 1$ **to** M **do**
  if *episode modulo* $k = 0$ **then**
    Sample a graph $G(V, E)$ from dataset $\mathbb{S}$ and reset the environment;
  end
  Initialize node features $features \leftarrow$ InitFeatures(G);
  for $t = 1$ **to** $MaxTry$ **do**
    Observe placement configuration (state)
      $s_t \leftarrow concatenate(features, adjacency_matrix_G)$ `// Topology-aware encoding (GNN)`
    $\mu_t \leftarrow \text{MPNN}(s_t)$; `// node embeddings after message passing`
    `// Action selection using DQN on embeddings`
    Select a_t using ϵ-greedy with Q-values $\hat{Q}(\mu_t, a; \theta)$;
    Execute a_t, observe reward r_t, next state s_{t+1}, and termination flag $done_t$;
    Update node features $features \leftarrow$ InitFeatures(G);
    Calculate the next state
     $s_{t+1} \leftarrow concatenate(features, adjacency_matrix_G)$
    Store $(s_t, a_t, r_t, s_{t+1}, done_t)$ in D;
    `// Standard DQN update (Chapter 2)`
    Update θ from minibatches sampled from D and periodically synchronize
     $\theta^- \leftarrow \theta$;
    if $done_t$ **then**
      break
    end
    $s_t \leftarrow s_{t+1}$;
  end
  if *episode* mod $val_freq = 0$ **then**
    Evaluate the learned policy on graphs from a validation set $\mathbb{S}_{val}$;
  end
end

$x_v^i[5]$ Remaining capacity of v (i.e., currently available memory of this switch);

$x_v^i[2]$ and $x_v^i[5]$ are specific to each node, varying locally, while $x_v^i[1], x_v^i[3]$ and $x_v^i[4]$ are global, representing the overall state of the graph and the episode context. The general purposes of each of the components are: $x_v^i[2]$ and $x_v^i[5]$ to offer valuable insights for assessing the legitimacy of an action while $x_v^i[1], x_v^i[3]$ and $x_v^i[4]$ to provide valuable guidance for selecting the most suitable action (i.e., deciding which rule to install on which node) within a given state.

Action Space The action space remains unchanged with respect to Chapter 3 (Section 3.3.3.0). An action $a(v, f_i)$ corresponds to installing rule set f_i on node v. Actions are categorized as valid, invalid, or redundant according to memory constraints, struc-

tural feasibility, and coverage contribution. The key difference does not lie in the action definition itself, but in how action values are computed: instead of relying on a flattened state representation, the Q-function now operates on node embeddings generated by the GNN encoder.

Reward The reward formulation follows the same structural definition introduced in Chapter 3 (Section 3.3.3.0); however, in the present chapter, each component is normalized. In Chapter 3, the agent was trained on a single fixed topology. Consequently, the structural role of a node and the number of st-paths passing through it remained constant throughout training, and the reward magnitude associated with a given action was directly comparable across episodes.

In contrast, the current framework considers graphs of varying sizes and densities. As a result, the same action $a(v, f_i)$ may produce substantially different absolute coverage gains across different topologies, since the number of st-paths and the structural importance of node v depend on the underlying graph. Without normalization, reward magnitudes would scale with graph size and density, potentially biasing the learning process and hindering generalization. Normalization, therefore, ensures that reward values remain comparable across heterogeneous graph instances.

The objective remains to distribute rule sets efficiently along all st-paths while minimizing switch resource utilization. Positive rewards are assigned when installing a rule on a node results in newly covered paths, whereas penalties are incurred for invalid or redundant actions. The reward associated with action $a(v, f_i)$ is defined as:

$$Reward(a(v, f_i)) = \begin{cases} -penalty & \text{if } a(v, f_i) \text{ is invalid or redundant,} \\ paths_{v,f_i} - occ_v & \text{otherwise.} \end{cases}$$

Here, $paths_{v,f_i} \in [0, 1]$ denotes the normalized ratio of newly covered paths resulting from installing rule f_i on node v, computed as the number of previously uncovered paths containing v divided by the total number of paths in the graph. The term $occ_v \in [0, 1]$ represents the memory occupancy ratio in the graph G, defined as the number of rules currently installed on G divided by the total capacity. The quantities $paths_{v,f_i}$ as well as the feature components $x_v^i[3]$ and $x_v^i[4]$ are computed using the greedy polynomial-time procedure described in Chapter 3 (Section 3.3.2).

4.4 Experimental Setup and Methodology

4.4.1 Experimental Configuration

Experiments were conducted using graphs generated by the tool described in subsection 4.4.2. The dataset consists of 1000 graphs for training, 200 for validation, and 500 for testing. The agent is trained and evaluated across graphs of varying sizes and structural characteristics in order to assess its generalization capability.

The algorithm was implemented in Python and executed on a remote GPU server equipped with an Intel(R) Xeon(R) Gold 6258R CPU @ 2.70GHz, 503 GB of RAM, run-

ning Ubuntu 22.04.3 LTS, and an NVIDIA RTX A6000 GPU (Driver Version 530.30.02, CUDA 12.1).

Evaluation Metrics

The performance of the proposed GNN-DQN framework is assessed using two primary metrics. The first metric, referred to as *occupancy*, measures the percentage of total switch capacity utilized by the installed rules; lower occupancy values indicate more efficient use of network resources. The second metric is the *success rate*, defined as the percentage of valid solutions obtained on unseen test graphs. This metric evaluates the agent's ability to generalize across topologies of varying size and structural characteristics.

Training Configuration

The penalty associated with invalid or redundant actions in the reward function was set to -10, encouraging the agent to avoid infeasible or inefficient placements. The GNN-DQN agent was trained for $M = 12000$ episodes over the training dataset. To promote exposure to diverse topologies, the sampled graph was changed every $k = 2$ episodes. Each episode was limited to a maximum of 500 steps, corresponding to potential placement actions. An episode terminated under one of the following conditions:

1. all rules were successfully placed on all paths,
2. the maximum number of steps was reached without achieving full coverage,
3. all nodes became saturated while uncovered paths remained.

An ϵ-greedy exploration strategy was adopted with initial exploration rate $\epsilon_0 = 1$ and minimum exploration rate $\epsilon_{\min} = 0.09$. The decay factor was defined as $\lambda = \exp\left(\frac{2\ln(0.5)}{M}\right)$, ensuring that ϵ decreases to 0.5 halfway through training. This schedule promotes extensive exploration during the early stages and gradually shifts toward exploitation.

The discount factor was set to $\gamma = 0.8$, and the learning rate to $\alpha = 10^{-5}$ to ensure stable convergence. Training was performed using minibatches of size 32. The target network was synchronized with the online network every $C = 1000$ steps. Validation was conducted every $val_freq = 50$ episodes using the validation set to monitor generalization performance. The complete GNN-DQN training procedure is summarized in Algorithm 6.

4.4.2 Synthetic Network Topology Generation

Due to the absence of publicly available benchmark datasets for st-DAGs, we developed a dedicated graph generation tool to construct synthetic instances with controlled structural properties. The generator produces k directed acyclic graphs (DAGs) based on the following input parameters: the number of nodes n, an edge creation probability p, a maximum forward connection depth h, and the desired number of graphs k. The overall generation procedure is summarized in Algorithm 7.

For each generated graph, the node set is defined as $V = \{0, 1, \ldots, n-1\}$. To ensure acyclicity and topological ordering, directed edges are only considered from node v to node u such that $v < u \leq v + h$. For each admissible pair (v, u), a Bernoulli trial with success probability p determines whether the directed edge $v \to u$ is added. This forward-only

construction guarantees the absence of cycles. To enforce the st-DAG structure, node 0 is designated as the source and node $n-1$ as the destination. After edge generation, nodes that are not reachable from the source or that do not lead to the destination are removed. The resulting graphs are therefore connected, topologically ordered st-DAGs.

To align the synthetic graphs with realistic network connectivity patterns, we considered structural characteristics observed in real-world communication networks, such as those reported in the Internet Topology Zoo [131]. In practical network infrastructures, switch connectivity is typically sparse and structured to support efficient forwarding and redundancy while avoiding cycles. Nodes generally connect to a limited number of neighbors, often between two and four, leading to tree-like or layered topologies with low average degree. To replicate these properties while preserving sufficient path diversity, we set the edge creation probability to $p = 0.2$ and selected maximum connection depths $h \in \{3, 5, 7\}$ depending on the target graph size. This configuration produces moderately sparse st-DAGs that balance connectivity, structural diversity, and computational tractability.

Algorithm 7: Synthetic st-DAG generation

Input: Number of nodes n, edge probability p, maximum connection range h
Output: Synthetic st-DAG $G = (V, E)$
$V \leftarrow \{0, 1, \ldots, n-1\}$;
$E \leftarrow \emptyset$;
$s \leftarrow 0$;
$t \leftarrow n-1$;
foreach $v \in V \setminus \{t\}$ **do**
  Define the candidate successor set:

$$\mathcal{S}_v = \{u \in V \mid v < u \leq \min(v+h, n-1)\}$$

  if $\mathcal{S}_v \neq \emptyset$ **then**
    foreach $u \in \mathcal{S}_v$ **do**
      Add edge (v, u) to E with probability p;
    end
  end
end
Construct $G = (V, E)$;
repeat
  Remove all non-destination nodes with no successors;
  Remove all non-source nodes with no predecessors;
  Update V and E accordingly;
until *no additional node is removed*;
return G;

4.5 Evaluation and Performance Analysis

4.5.1 Training Behavior and Model Performance

We conducted a comprehensive experimental study using datasets generated by the proposed synthetic graph generation tool. For each graph size, the dataset comprises 1000 graphs for training, 200 for validation, and 500 for testing. Four graph sizes were considered, namely $n \in 10, 15, 20, 25$. The number of rule sets to be installed was fixed for each configuration and defined as $\frac{1}{2} \times path_capacity(p_{min})$, corresponding to half of the available placement slots along the shortest st-paths.

The validation set consists of graphs that are entirely distinct from those used during training. It is employed to evaluate the generalization capability of the learned policy at regular intervals ($val_freq = 50$ *episodes*). For each graph size, a separate model was trained while keeping the number of rule sets fixed.

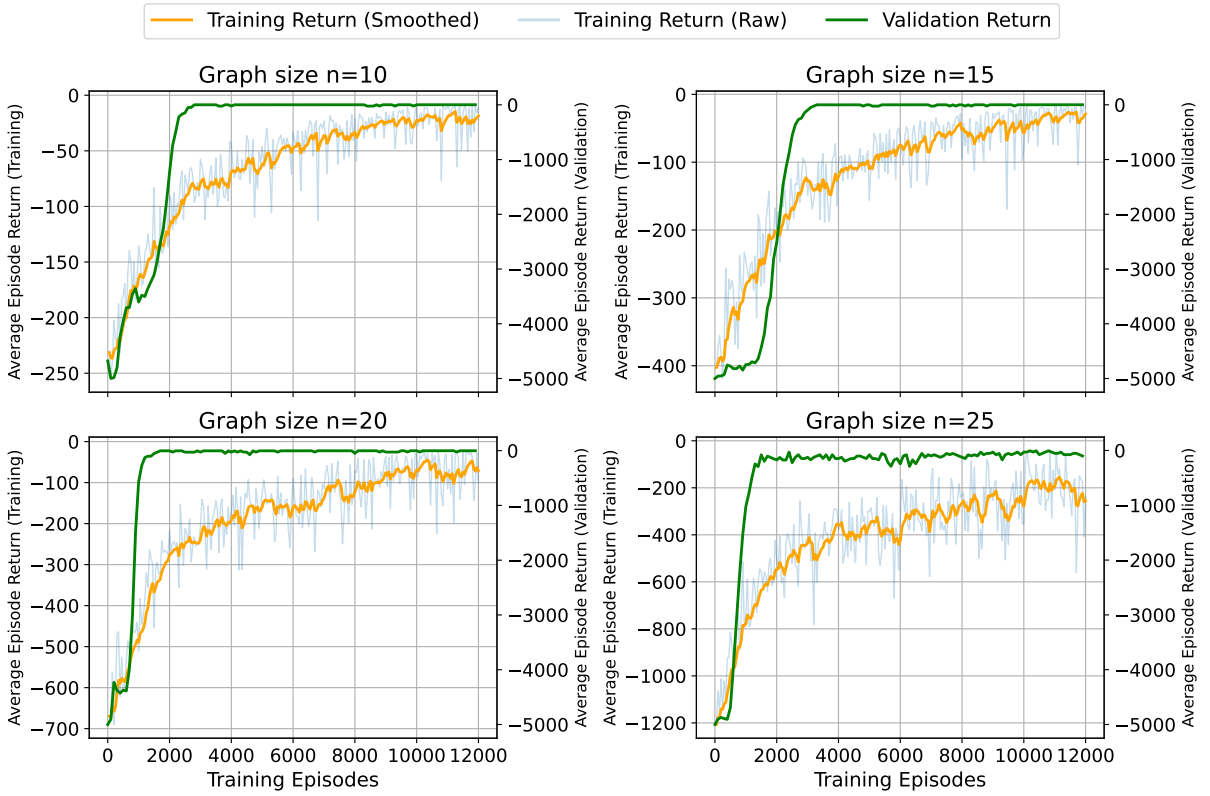


Figure 4.2 – Learning curves of the GNN-DQN agent showing the evolution of the average episode return during training and validation for different graph sizes ($n = 10, 15, 20, 25$).

Figure 4.2 illustrates the evolution of the average episode return during training for the four graph sizes. In RL, the episode return corresponds to the cumulative reward obtained over a complete episode. For each subplot, the solid orange curve represents the smoothed training return, the lighter curve corresponds to the raw training return, and the solid green curve denotes the validation return evaluated periodically during

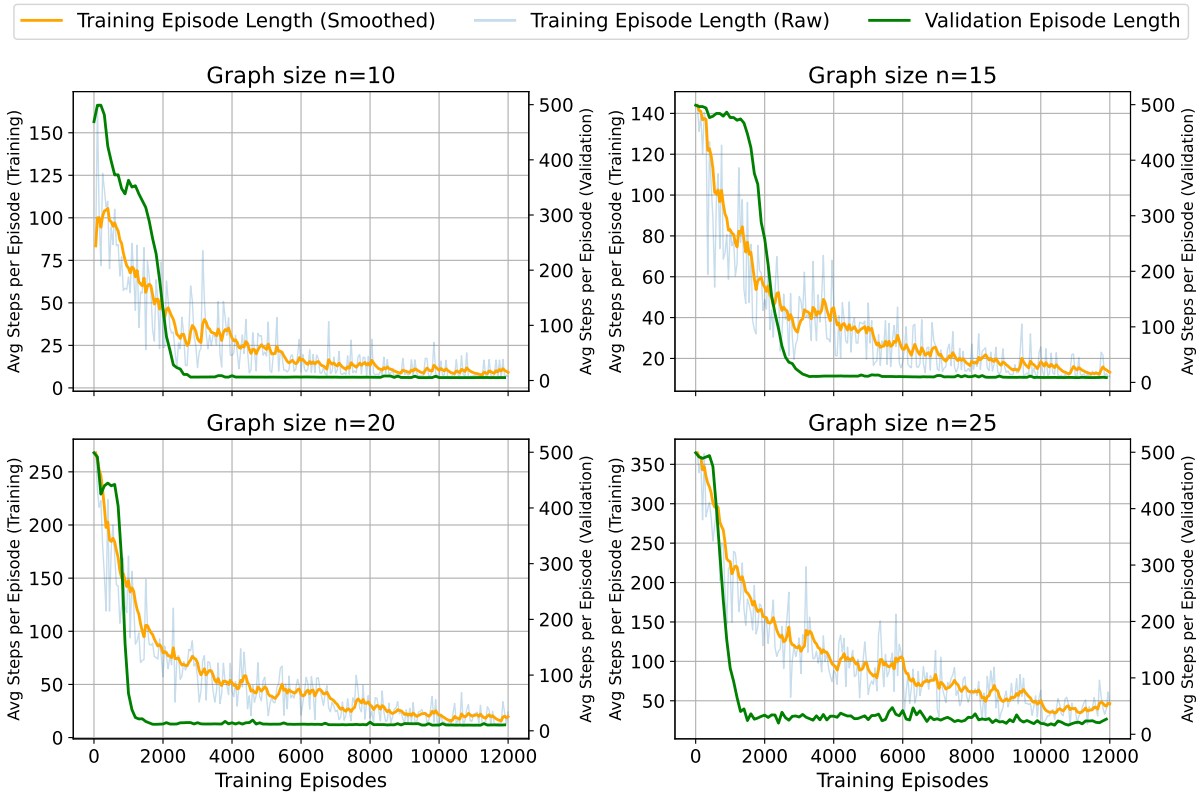


Figure 4.3 – Learning curves of the GNN-DQN agent showing the evolution of the average episode length during training and validation for different graph sizes ($n = 10, 15, 20, 25$).

training. Due to the difference in scale between training and validation returns, dual y-axes are employed: the left axis corresponds to training performance, while the right axis corresponds to validation performance. Furthermore, Figure 4.3 presents the evolution of the mean episode length during training and validation. The episode length is defined as the total number of actions executed by the agent within a single episode. The same visualization strategy is adopted as in the reward plots.

From these figures, two complementary behaviors clearly emerge: the average episode return increases progressively, while the episode length decreases over time. This reflects both improved policy quality and increased decision efficiency. At the early stages of training, the agent operates without prior knowledge and relies heavily on exploratory actions. Given the stringent feasibility constraints imposed on the action space, valid actions are relatively rare compared to invalid or redundant ones. As defined in Section 4.3.2, valid actions yield rewards in the interval $[0, 1]$, whereas invalid or redundant actions incur a penalty of -10. Consequently, during initial exploration, the agent accumulates large negative returns due to frequent infeasible decisions. This also explains the large episode lengths observed at the beginning of training. A similar phenomenon is observed in the early validation phase. Since validation graphs are unseen during training, the initial policy generalizes poorly, resulting in low returns and high episode lengths. As training progresses, however, the agent gradually learns to associate states with high-value actions.

The frequency of invalid and redundant actions decreases substantially, leading to an increase in cumulative return toward zero. Simultaneously, the number of actions required to reach a feasible solution decreases, indicating more efficient rule placement strategies. The convergence trends observed in both training and validation curves demonstrate that the learned policy not only improves on the training distribution but also generalizes effectively to unseen graphs.

To further assess generalization performance, we evaluated the trained models on a separate testing dataset composed of 500 entirely unseen graphs per graph size. The results are summarized in Table 4.1. The proposed GNN–DQN framework successfully identifies valid rule placement solutions in 99% of the test graphs, highlighting the robustness and scalability of the learned policy across varying graph structures.

Graph size	# rules	Avg. Occ. (%)	Avg. Time (s)	Success (%)
10	3	43.2	0.029	99.9
15	3	39.69	0.0322	99.6
20	4	41.16	0.0489	99.8
25	4	45.46	0.09436	99.8

Table 4.1 – Memory occupancy and processing time of DQN on 500 test graphs for each graph size.

4.5.2 Generalization Across Graph Sizes and Rule Configurations

To further assess the generalization capability of the proposed GNN–DQN framework, we conducted a series of additional experiments designed to evaluate robustness across varying rule configurations, increased graph sizes, and real-world network topologies.

First, we evaluated the model trained on graphs of size $n = 25$ with a fixed training configuration of four rule sets on a new set of 500 graphs of the same size, while *varying the number of rule sets at test time*. This setting intentionally departs from the training distribution: changing the number of rule sets modifies both the action space and the state representation, and thus exposes the agent to decision scenarios that were not encountered during training. As reported in Table 4.2, the trained policy still achieves a success rate of 99.9%. Similar behavior is observed for the models trained on graph sizes $n \in \{10, 15, 20\}$.

			# rules			
Model trained on	Test size	Metrics	3	4	5	6
Graph size 25	25	Avg. Occ. (%)	39.19	48.98	54.13	53.47
		Avg. Time (s)	0.11177	0.16	0.1932	0.2275

Table 4.2 – Testing a trained model with fixed graph size 25 and four rule sets on graphs with the same size but a variable number of rule sets.

Secondly, we evaluated the scalability of the proposed framework by testing the models trained on graph sizes 25 and 20 on a new dataset of 600 graphs of size 30, while

simultaneously varying the number of rule sets. Increasing the graph size introduces new structural patterns and connectivity configurations that were not observed during training. At the same time, modifying the number of rule sets alters the action space and the combinatorial complexity of the placement problem. This experimental setting, therefore, allows us to assess both structural scalability and adaptability to different rule configurations. Results are reported in Table 4.3, which provides the average memory occupancy and processing time for each configuration. In addition to these metrics, we observed that the model trained on graphs of size 25 successfully identifies valid placements in approximately 95% of the test instances, whereas the model trained on size 20 achieves a success rate of about 80%. These results indicate that the agent trained on smaller graphs is capable of transferring its learned structural representations to larger and more complex topologies. Although performance slightly degrades as the structural gap between training and testing graphs increases, the agent maintains a high level of feasibility and efficiency, demonstrating robust generalization beyond the original training scale.

			# rules			
Model trained on	Test size	Metrics	5	6	7	8
Graph size 25	30	Avg. Occ. (%)	53.56	56.62	56.18	50
		Avg. Time (s)	0.324	0.355	0.375	0.19
Graph size 20	30	Avg. Occ. (%)	56.98	59	65.06	50
		Avg. Time (s)	0.326	0.357	0.412	0.17

Table 4.3 – Testing trained models with fixed graph size and number of rule sets on larger graphs with a variable number of rule sets.

Finally, we evaluated the transfer capability of the proposed GNN-DQN framework on real-world network topologies obtained from the Internet Topology Zoo [131]. These instances provide realistic connectivity patterns, and testing on Topology Zoo graphs therefore constitutes a relevant assessment of whether a policy learned on synthetic st-DAGs remains effective when applied to real network structures. In this experiment, we deployed the model trained on synthetic graphs of size $n = 20$ with a fixed training configuration of four rule sets on Topology Zoo graphs of varying sizes and with varying numbers of rule sets. The objective is to assess both scalability and robustness under changes in topology and rule configuration, and to verify the agent's behavior beyond the synthetic distribution. The results reported in Table 4.4 show that the trained policy is able to compute feasible rule placements on these real-world instances while maintaining reasonable memory occupancy and processing time, which supports the practical relevance of the proposed approach.

4.6 Distance-Constrained ACL Rule Placement

4.6.1 Motivation and Definition of Distance Constraints

Beyond the objective of ensuring complete st-path coverage while respecting switch memory limitations, practical ACL deployment may also be subject to additional place-

Graph / size	# rules	Occupancy (%)	Time (s)
Sprint/11	3	38.89	0.06
Abvt/23	6	50	0.22
Agis/25	5	50	0.25
Bics/34	8	31.81	0.39
NetworkUSA/35	7	42.1875	0.55

Table 4.4 – Memory occupancy and processing time of DQN on real-world network topologies using a model trained on graphs of size 20 with four rule sets.

ment preferences related to the relative position of the filtering point in the network. In operational environments, security administrators do not always seek only a feasible placement; they may also require that some rule sets be installed *closer to the traffic source* or, conversely, *closer to the destination*. These preferences arise from concrete architectural and security considerations and therefore motivate the introduction of additional distance-aware constraints in the placement problem.

Placing ACL rules close to the source corresponds to an *early filtering* strategy. The main rationale is to stop unwanted traffic as soon as possible after it enters the network. This strategy is particularly relevant for traffic identified as suspicious, unnecessary, or clearly unauthorized. By blocking such traffic near its origin, the network avoids carrying useless packets through intermediate switches, preserves forwarding resources, and limits congestion in the core of the infrastructure. This behavior is especially desirable in situations involving high traffic volumes, such as scanning campaigns, denial-of-service attempts, or massive flows generated by compromised hosts. Figure 4.4 illustrates this source-oriented placement preference, where filtering is performed close to the ingress in order to stop malicious traffic before it propagates through the network core. Early filtering also reduces the attack surface by preventing malicious traffic from reaching deeper portions of the network, thereby limiting the risk of lateral propagation and minimizing the exposure of internal services. A simple real-world example is that of a university or enterprise campus network in which several access switches connect student laboratories, office machines, servers, and data center resources. Suppose that a large number of anomalous flows originate from a specific access segment due to misconfigured devices or infected hosts. In such a case, placing the corresponding filtering rules near the source access switch is preferable because the unwanted traffic is discarded before consuming aggregation and backbone resources. From a security standpoint, this strategy implements a form of containment: suspicious traffic is isolated at the edge of the network before it can affect shared infrastructure or sensitive internal zones.

In contrast, placing ACL rules close to the destination corresponds to a *destination-oriented filtering* strategy. This configuration is useful when security policies are tightly coupled to the protected service, application, or subnet rather than to the source of the traffic. In many real deployments, critical resources such as databases, authentication servers, industrial controllers, or internal applications are protected by security controls deployed in their immediate vicinity. This design allows policies to be expressed in a more service-aware and context-specific manner, since the filtering logic is applied near

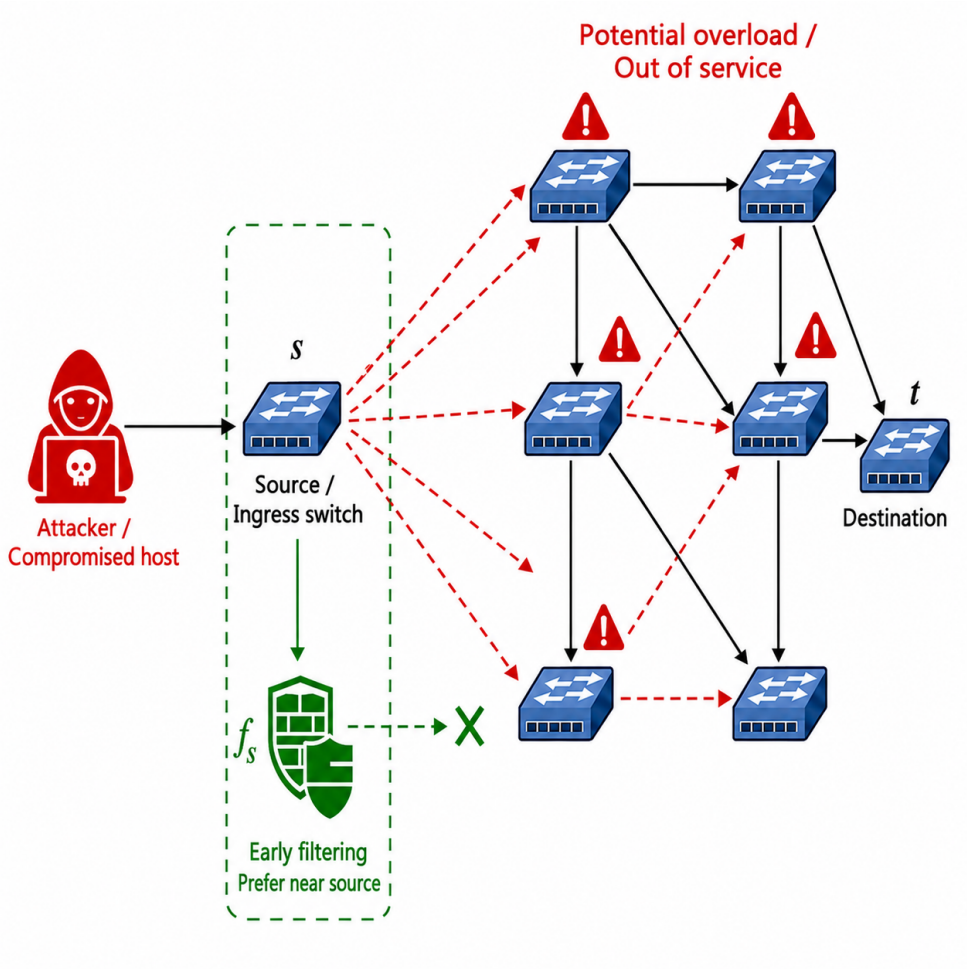


Figure 4.4 – Illustration of a source-oriented distance constraint for early filtering.

the resource being protected. Such an approach is particularly appropriate when different destinations require different levels of protection, or when access control depends on application semantics that are meaningful only near the final service location.

As an illustrative example, consider the network shown in Figure 4.5, where traffic traverses an internal interconnected switch network toward a protected data-center domain. This domain represents a logical destination composed of several service zones with distinct security requirements, such as a public web zone, an application zone, and a protected database zone. Each egress switch provides access to one internal zone and acts as a destination-side enforcement point for the ACL rules associated with that zone. For instance, the web-zone egress may allow public HTTPS traffic, whereas the database-zone egress may only allow traffic from authorized application servers and deny other flows. This setting is realistic in data centers and enterprise networks, where service zones, trust boundaries, and micro-segmentation mechanisms are commonly used to enforce service-specific protection. Therefore, placing ACL rules near the corresponding egress switches aligns the filtering decision with the security perimeter of the protected service. More generally, destination-side placement is consistent with architectures in which security

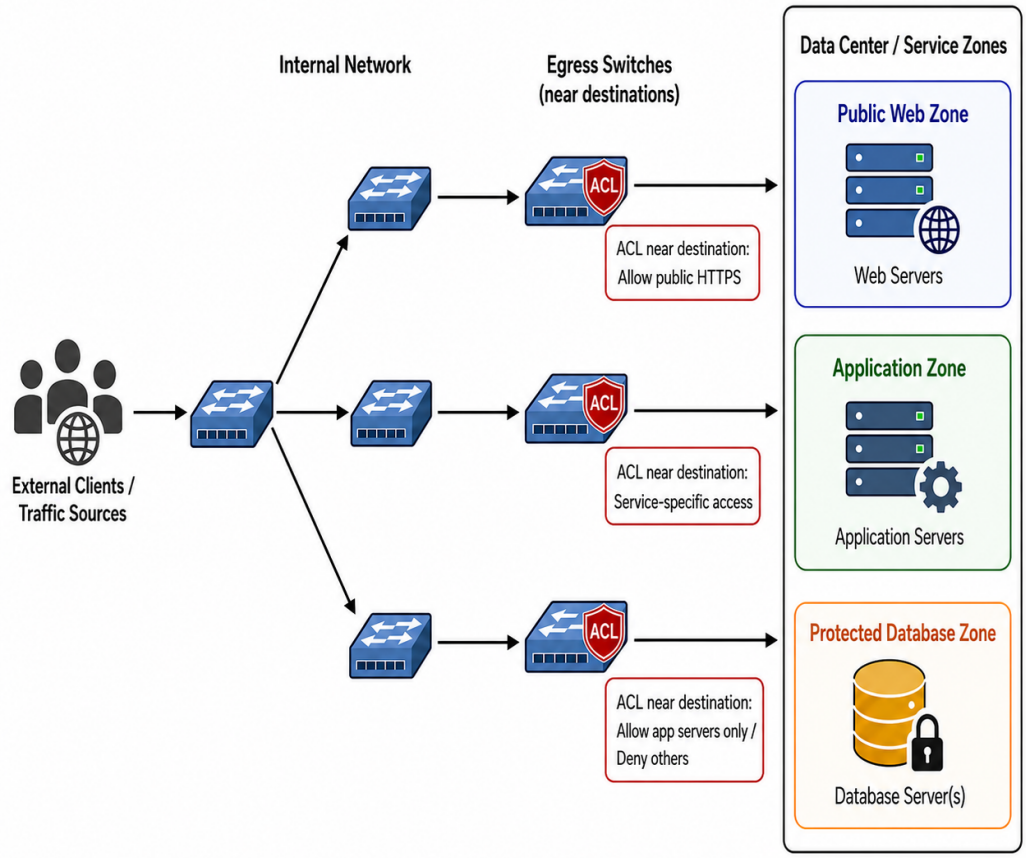


Figure 4.5 – Illustration of a destination-oriented distance constraint for service-specific protection.

enforcement is organized around service zones, trust boundaries, or micro-segmentation principles.

These two placement preferences, source-oriented and destination-oriented, do not replace the original objective of the distributed ACL placement problem, but rather refine it by introducing an additional notion of *where* a rule should preferably be installed among all feasible candidate nodes. Accordingly, the placement problem is extended with distance constraints that bias the agent toward nodes located either near the source or near the destination, depending on the semantic label associated with the rule set. Source-labeled rule sets are encouraged to be installed on nodes that are topologically close to the source, whereas destination-labeled rule sets are encouraged to be installed near the destination.

4.6.2 Adaptation of the GNN-DQN Framework

To extend the proposed GNN-DQN framework to the placement problem with additional distance constraints, only two components of the RL formulation require adaptation: the node feature representation and the reward function. The action space remains unchanged, since an action is still defined as $a(v, f_i)$, corresponding to the installation of rule set f_i on node v.

First, the node feature vector associated with each pair (v, f_i) is enriched with two additional binary components in order to encode the placement preference of the considered rule set. More precisely, $x_v^i[6]$ indicates whether rule set f_i should be placed close to the source node s (1 if yes, 0 otherwise), while $x_v^i[7]$ indicates whether f_i should be placed close to the destination node d (1 if yes, 0 otherwise). These additional features allow the GNN encoder to incorporate, directly into the state representation, the semantic placement preference associated with each rule set.

Second, the reward function is extended with a distance-dependent penalty term in order to bias the agent toward source-side or destination-side placements according to the label of the considered rule set. The modified reward is defined as:

$$Reward(a(v, f_i)) = \begin{cases} -penalty, & \text{if } a(v, f_i) \text{ is invalid or redundant,} \\ paths_{v,f_i} - occ_v - dist_{s,f_i}, & \text{if } f_i \text{ is source-labeled,} \\ paths_{v,f_i} - occ_v - dist_{f_i,d}, & \text{if } f_i \text{ is destination-labeled,} \\ paths_{v,f_i} - occ_v, & \text{otherwise,} \end{cases}$$

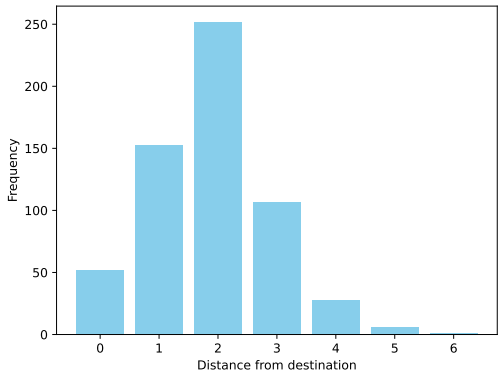
where $dist_{s,f_i}$ denotes the normalized distance between the source s and the selected node v for source-oriented rule sets, and $dist_{f_i,d}$ denotes the normalized distance between the selected node v and the destination d for destination-oriented rule sets. In this way, the agent is encouraged to favor placements that satisfy not only the original path-coverage and memory objectives introduced previously, but also the desired topological proximity constraint. The baseline reward structure remains the same as in Section 4.3.2, and only an additional distance-aware term is incorporated.

4.6.3 Experimental Results

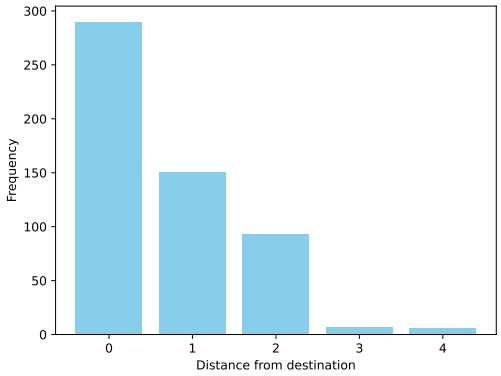
To evaluate the proposed extension under additional distance constraints, the GNN-DQN agent was trained on the same datasets previously generated for the baseline placement problem, namely 1000, 200, and 500 graphs for training, validation, and testing, respectively, for each graph size in the set $\{10, 15, 20, 25\}$. The same environment configuration was preserved, including the underlying graph generation process. For each topology, the number of rule sets to be installed was fixed to $\frac{1}{2} \times path_capacity(p_{\min})$, which corresponds to half of the available placement slots along the shortest st-paths.

To assess the effect of the newly introduced constraints, we define an additional performance metric *distance*. For a source-labeled rule, this metric measures the distance from the source to the first node on which the rule is installed. Similarly, for a destination-labeled rule set, it measures the distance from the destination to the first node on which the rule is installed. This metric is intended to quantify how close the placement effectively is to the desired side of the network.

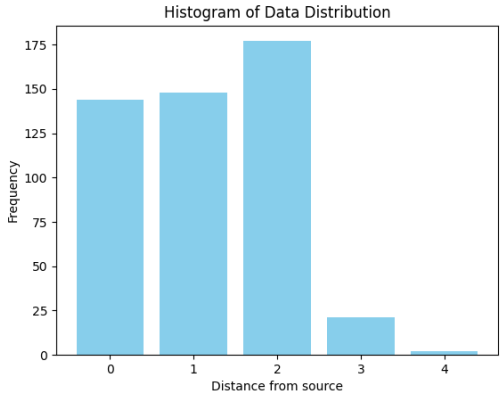
Figure 4.6 reports the distribution of this distance metric on the 500 graphs of the testing set, before and after incorporating the additional distance constraints. More precisely, Figures 4.6a and 4.6b illustrate the behavior for destination-labeled rule sets, while Figures 4.6c and 4.6d present the corresponding results for source-labeled rule sets. For conciseness, only the results obtained on graphs of size 25 are shown, since the same trend was consistently observed for the other graph sizes.



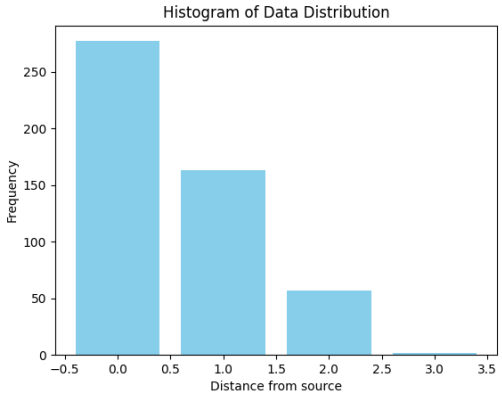
(a) Distance between destination-labeled rules and the destination before adding the constraint.



(b) Distance between destination-labeled rules and the destination after adding the constraint.



(c) Distance between source-labeled rules and the source before adding the constraint.



(d) Distance between source-labeled rules and the source after adding the constraint.

Figure 4.6 – Distribution of placement distances for destination-labeled and source-labeled rules before and after introducing the distance constraint. The first row corresponds to destination-oriented rules, while the second row corresponds to source-oriented rules.

The obtained results clearly show that the proposed agent successfully adapts to the modified objective. Although the distance constraints are introduced as soft constraints through the reward function, the learned policy exhibits a strong tendency to place labeled rule sets as close as possible to the desired endpoint. In particular, after incorporating the constraint, the proportion of rule sets placed at distance 0 from the destination or the source increases significantly, while the number of placements at larger distances decreases accordingly. This confirms that the reward shaping strategy effectively biases the placement policy toward source-side or destination-side enforcement, without compromising the original objective of the problem.

An additional advantage of the proposed GNN-DQN framework lies in its flexibility with respect to objective modifications. In contrast, the heuristic st-mincut introduced in Chapter 3.3.1, as well as heuristic approaches in general, cannot naturally accommodate such additional constraints. Indeed, a heuristic is typically designed for a specific objective function and relies heavily on the designer's prior understanding of the problem structure. As a result, any change in the optimization objective generally requires a substantial redesign of the heuristic itself. This lack of adaptability constitutes another limitation of the st-mincut approach, in addition to the shortcomings already discussed in the previous chapter, namely its local decision-making nature and its tendency to favor short-term gains.

Overall, these results further highlight the effectiveness of the proposed GNN-DQN approach. Beyond its ability to produce feasible placement solutions, it also provides a flexible framework capable of incorporating new constraints through appropriate reward design, thereby enabling a better trade-off between multiple placement objectives.

4.7 Summary

In this chapter, we have addressed the distributed rule placement problem by formulating it as a sequential decision-making task and solving it using a RL approach. Building upon the limitations of heuristic and exact methods, we proposed a learning-based framework that leverages a GNN combined with DQN to capture both the structural properties of the network and the combinatorial nature of the placement decisions. The proposed approach enables the agent to learn effective placement strategies directly from interaction with the environment, while generalizing across different graph instances and sizes. Extensive experiments demonstrated that the GNN-DQN model achieves strong performance in terms of solution quality and scalability, while maintaining the ability to adapt to varying network topologies and constraints. In particular, the use of graph-based representations allows the model to exploit relational information between nodes, which is critical for capturing path dependencies and placement feasibility. Moreover, the RL framework offers flexibility for incorporating additional constraints and optimization objectives without requiring an explicit redesign of the solution method. Despite these advantages, the formulation considered in this chapter assumes that rule placement must be ensured across all possible paths between source and destination nodes. While this assumption provides a comprehensive coverage guarantee, it may lead to overly conservative solutions and increased resource consumption in practical settings. In real-world networks, traffic is often

routed according to predefined strategies, such as shortest-path routing or protocol-driven forwarding decisions, which restrict the set of feasible paths that are actually used. Motivated by this observation, the next chapter extends the proposed framework to a more general and realistic setting, where rule placement is performed over a subset of selected paths rather than the entire path space. This extension enables a more efficient use of network resources while remaining consistent with operational routing policies.

Chapter 5

Constrained ACL Rule Placement over Selected Paths

Contents

5.1 Introduction 89
5.2 Problem Formulation 90
5.2.1 Motivating Example 91
5.2.2 Network Representation and Problem Statement 93
5.2.3 Rule Placement under Constraints 95
5.3 ACL Rule Placement Framework 96
5.3.1 Agent State, Action, and Reward Design 97
5.3.2 Reward Shaping under Bandwidth Constraint 98
5.3.3 Handling Rule Ordering Constraint 98
5.4 Evaluation Methodology and Baseline Comparison 101
5.4.1 Baseline Algorithms for ACL Rule Placement 101
5.4.2 Experimental Setup 102
5.5 Experimental Results and Performance Analysis 103
5.5.1 Performance With Underutilized Capacity 103
5.5.2 Performance under Near Maximum Capacity 106
5.5.3 Effect of the Bandwidth Constraint 108
5.5.4 Generalization to Real-World Network Topologies 109
5.6 Discussion and Practical Deployment Considerations 112
5.6.1 Mininet Deployment 112
5.6.2 Discussion 113
5.7 Summary 115

5.1 Introduction

In the previous chapters, the ACL rule placement problem was addressed under the assumption of routing uncertainty. Specifically, we considered that traffic between a source

and a destination may traverse any feasible path in the network. As a result, rule placement was designed to ensure coverage over all possible source-destination paths, guaranteeing correctness independently of the underlying routing behavior.

While this assumption provides robustness, it leads to an overly conservative formulation of the problem. In operational networks, routing is not arbitrary but governed by predefined protocols and policies, such as shortest-path routing or ECMP. Consequently, traffic is typically concentrated over a limited subset of paths, while many alternative paths are rarely, if ever, utilized. Enforcing rule placement over all possible paths may therefore introduce unnecessary redundancy, increase TCAM consumption, and reduce placement efficiency without tangible benefits in practice.

Motivated by this observation, this chapter introduces a more general and realistic formulation of the placement problem, where rules are deployed over a selected subset of source-destination paths. This subset reflects the effective routing behavior of the network and can be derived from routing configurations or traffic engineering strategies. By focusing on relevant paths only, the placement process becomes both more efficient and better aligned with real-world network operations. This formulation not only generalizes the previous problem setting but also bridges the gap between theoretical placement models and practical routing-aware deployments.

To address this generalized problem, we extend the learning-based framework developed in Chapter 4. The proposed approach leverages a GNN-DQN architecture to learn placement strategies that adapt to both the network topology and the selected path set. The model is designed to optimize rule distribution while ensuring coverage of the chosen paths and minimizing resource usage.

In addition to the core placement objective, we incorporate practical constraints that arise in real deployments. These include memory limitations of network devices, rule ordering requirements to preserve policy semantics, and bandwidth-aware placement strategies that favor early filtering of traffic. These constraints are integrated directly into the learning process, allowing the agent to jointly optimize multiple objectives while preserving policy correctness and operational efficiency.

The remainder of this chapter is organized as follows. Section 5.2 introduces a motivating example of the placement problem under different constraints and provides a formal definition of the problem. Section 5.2.3 presents the additional constraints considered in this work, namely bandwidth optimization and rule ordering. Section 5.3 details the proposed GNN-DQN architecture for the placement problem, along with its extensions to handle these constraints. Section 5.4 describes the experimental setup and the baseline methods used for comparison. Section 5.5 reports and analyzes the simulation results. Finally, Section 5.7 concludes the chapter.

5.2 Problem Formulation

This section first describes a motivating example of in-network ACL rules placement and then presents the formulation of the placement problem and its constraints. The notation used for the problem formulation is summarized in Table 5.1.

Notation	Description
s	single source
t	single destination
P	set of paths
F	set of ACL rules
$m(v)$	memory available on node v, i.e., maximum number of rules that can be installed on v
$\|\|p\|\|$	length of path p, i.e., number of links contained in p (not necessarily st-path)
$paths_v$	subset of paths in P that contain node v
$\|P\|, \|paths_v\|$	number of paths in P, in $\|paths_v\|$
$min(P)$	shortest length of a path in P
$path_capacity(p)$	sum of the memory capacities of the nodes composing the path p
P_G	set of all paths between s and t within a graph G
cv	coverage percentage, i.e., ratio of paths included in P (and therefore that should be covered)

Table 5.1 – Notations for problem formulation

5.2.1 Motivating Example

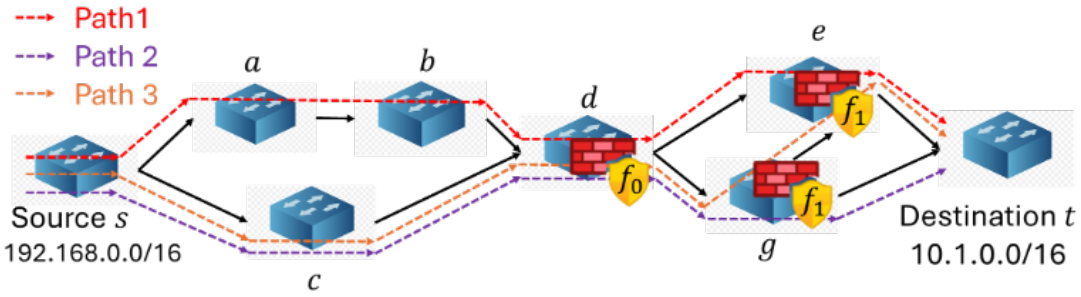


Figure 5.1 – A network topology for a placement scenario with 2 rules f_0 and f_1 to be placed on 3 paths: Path 1, Path 2, and Path 3.

Consider the network topology in Fig. 5.1, where s represents the source, t is the destination, and $a, b, c, d, e,$ and g are intermediate switches; each switch can accommodate one rule set (capacity = 1). The ACL table shown in Fig. 5.2 (Table 5.2a) is split into two rule sets, f_0 and f_1 (Tables 5.2b and 5.2c), based on the action taken by each rule. The goal is to place these rule sets across the network while satisfying a set of predefined constraints. The default rule in f_0 and f_1 is not fixed but varies depending on the relative position of each rule set in the processing sequence. Specifically, the rule set that appears earlier in the pipeline adopts a default action that permits unmatched traffic, allowing it to proceed to subsequent rule sets for further evaluation. In contrast, the rule set positioned at the end of the sequence enforces a default action that denies all unmatched traffic. This is because, at this stage, the traffic has already traversed the entire original

Source IP	Source Port	Dest. IP	Dest. Port	Protocol	Action
192.168.1.10	Any	10.1.20.16	80	TCP	Allow
192.168.0.0/16	Any	10.1.2.0/24	Any	Any	Allow
192.168.3.30	Any	10.1.2.10	443	TCP	Deny
192.168.1.10	Any	10.1.10.50	Any	ICMP	Allow
192.168.1.10	53	10.1.1.30	Any	UDP	Allow
192.168.3.30	Any	10.1.3.40	Any	ICMP Type $\neq 8$	Deny
Any	Any	Any	Any	Any	Deny

(a) Initial ACL table

Source IP	Source Port	Dest. IP	Dest. Port	Protocol	Action
192.168.1.10	Any	10.1.20.16	80	TCP	Allow
192.168.0.0/16	Any	10.1.2.0/24	Any	Any	Allow
192.168.1.10	Any	10.1.10.50	Any	ICMP	Allow
192.168.1.10	53	10.1.1.30	Any	UDP	Allow
Default rule					

(b) First subtable: f_0

Source IP	Source Port	Dest. IP	Dest. Port	Protocol	Action
192.168.3.30	Any	10.1.2.10	443	TCP	Deny
192.168.3.30	Any	10.1.3.40	Any	ICMP Type $\neq 8$	Deny
Default rule					

(c) Second subtable: f_1

Figure 5.2 – Example of a simplified ACL table divided into two subtables, f_0 and f_1, according to the action associated with each rule.

ACL table without satisfying any matching condition and therefore reaches the default deny rule of the initial policy.

First, when considering only the constraint of covering all paths in $P = \{[s, a, b, d, e, t], [s, c, d, g, t], [s, c, d, g, e, t]\}$ with the minimal number of rules, f_0 is placed at node d, and f_1 is placed at nodes e and g. The second rule in Table 5.2b permits all traffic to the subnet 10.1.2.0/24, while the first rule in Table 5.2c drops traffic to a specific IP address, 10.1.2.10, within this subnet. This makes f_1 more important than f_0. Therefore, when incorporating the ordering constraint where f_1 has a higher priority than f_0, the placement is adjusted by swapping their positions. In this case, f_1 is placed at node d, and f_0 is placed at nodes e and g. Moreover, the second rule in Table 5.2c denies ICMP traffic with any type other than Echo Request (Type 8), specified as $ICMP\ Type! = 8\ Deny$. Based on standard domain knowledge about ICMP types, this rule is interpreted as allowing basic connectivity verification (e.g., ping) while blocking other ICMP functionalities such as error reporting or redirection, which could be exploited in attacks or introduce unnecessary network traffic. By restricting ICMP to Echo Requests only, such rules are commonly used to mitigate certain ICMP-based DDoS attacks. Therefore, to optimize bandwidth usage

across network links, this rule should be placed as close to the source as possible to prevent flooding network links before the packet is filtered at nodes e and g in the initial placement. Thus, when extending the problem to include a bandwidth constraint in addition to the coverage constraint, the placement changes further: f_1 is placed at nodes a and c, while f_0 remains at node d. This example highlights how the placement of ACL rules evolves in response to different constraints, demonstrating the flexibility required to meet practical network objectives.

5.2.2 Network Representation and Problem Statement

We consider the same network representation introduced in the previous chapter, where the topology is modeled as an *st-dag* $G = (V, E)$, i.e., a directed acyclic graph with a unique source node $s \in V$ and a unique destination node $t \in V$. Nodes represent network switches, while edges correspond to communication links. A path from s to t is referred to as an *st-path*, and we write $n' < n$ when node n' precedes node n along such a path. Each node has a finite capacity corresponding to the TCAM size of the associated switch.

In Chapter 4, the placement problem assumed that traffic may traverse any feasible st-path, requiring coverage over all such paths. In this chapter, we adopt a more general setting where traffic is restricted to a predefined subset of paths P, reflecting the network routing behavior. Consequently, coverage is enforced only over the paths in P.

Formally, given a set of rule sets, the objective is to determine their placement across the nodes of G such that each rule set is encountered at least once along every path in P, while respecting node capacity constraints and minimizing the overall TCAM usage. As in the previous formulation, redundant installations of the same rule set along a path should be minimized, although they may remain unavoidable due to capacity constraints and interactions between paths in P.

Problem statement Given an st-dag $G(V, E)$, such that every node v has finite memory space $m(v)$ (which we define to be the maximum number of rules that can be installed on v), a set of k rules $F = \{f_1, f_2, ..., f_k\}$, and a set of paths $P = \{p_1, p_2, ..., p_f\}$. We will denote by P_G the set of all paths between the source and the destination within G, and by $paths_v$ the set of all paths from P that pass through node v. If $p \in paths_v$ we will say that p is covered by node v. We also say that a path $p \in P$ is covered by a rule f if p is covered by a node v holding f. We will denote by $|S|$ the cardinality of set S, and by $min(P)$ the shortest length of a path in a set of paths P, where the length of a path is given by the number of its nodes. For any path $p \in P$, we define its *placement capacity* (or *placement slots*), denoted by $path_capacity(p)$, as the sum of the memory capacities of the nodes composing the path, i.e., $path_capacity(p) = \sum_{v \in p} m(v)$, which represents the total number of rule slots available along p. The problem addressed in this chapter consists of determining the placement of the rule sets $f_1, f_2, \ldots, f_k$ on the nodes of graph G traversed by the selected paths in P. We denote this problem as $PLACE(G, m, F, P)$.

$PLACE(G, m, F, P)$ can be formulated as a combinatorial optimization problem, as defined by Equations 5.1–5.3, with the objective of minimizing the total number of rule placements across the network. Let $X = (x_{v,f})$ be a binary allocation matrix, where $x_{v,f} = 1$ indicates that rule $f \in F$ is placed on switch $v \in G$, and $x_{v,f} = 0$ otherwise.

Additionally, let δ be a binary indicator such that $\delta_{p,v} = 1$ if path $p \in P$ is covered by switch v, i.e., $p \in paths_v$, and $\delta_{p,v} = 0$ otherwise. The formulation must satisfy three key constraints:

1. **Resource minimization:** the overall number of rule placements must be minimized (Equation 5.1);
2. **Capacity constraints:** the number of rules assigned to any switch must not exceed its predefined memory capacity (Equation 5.2);
3. **Path coverage:** each path $p \in P$ must include at least one instance of every rule $f \in F$ (Equation 5.3).

$$\min \quad \sum_{v \in V} \sum_{f \in F} x_{v,f} \tag{5.1}$$

Subject to,

$$\forall v \in V \quad \sum_{f \in F} x_{v,f} \leq m(v), \tag{5.2}$$

$$\forall f \in F, \forall p \in P \quad \sum_{v \in V} \delta_{p,v} \cdot x_{v,f} > 0 \tag{5.3}$$

It is important to note that $PLACE(G, m, F, P)$ has no feasible solution if the placement capacity of a shortest path $p_{\min} \in P$ is strictly smaller than k, i.e., $path_capacity(p_{\min}) < k$, since it would then be impossible to place all k rule sets along such a path. Conversely, if the shortest path in P comprises more than k placement slots, ensuring the first condition is straightforward. We can show that $PLACE(G, m, F, P)$ is an *NP-hard* problem even when m is uniformly equal to 1 and the set of rules is a singleton (i.e., $k = 1$).

In other words, we show in Theorem 1 that given an st-dag, a set of st-paths P, and $l \in \mathbb{N}$, distributing a rule on l switches such that every st-path in P is covered by one of these switches, is NP-hard. Therefore, minimizing the number of switches where to install a rule to cover all paths in P is also NP-hard.

Theorem 1. *The following problem is NP-hard: Given an st-dag G, a subset P of st-paths from G and a positive integer l, decide whether we can place the rule on l switches in such a way that every path from P contains at least one of these l switches.*

Proof. We reduce to our problem the *vertex cover problem* (known to be NP-hard [135]): given a graph H and positive integer l, does H have a vertex cover of size at most l ? Given a graph H we define an st-dag G by first choosing an arbitrary strict ordering $<$ on the vertices V of H and by orienting every edge (u, v) of H in such a way that $u < v$. Then we add two new vertices s and t and oriented edges (s, u), (u, t) for every $u \in V$. We consider the set of st-paths $P = \{(s, u, v, t) | u, v \in V \text{ and } (u, v) \text{ is an edge of } G\}$. Note that the size of P is polynomial in the size of H. We can also check that H has a vertex cover of size l if and only if there exist l switches ($\neq s, t$) where to place the rule so that every $p \in P$ contains at least one of these switches. □

We remark that the previous problem is also NP-complete since, after guessing l switches of G where to install the rule, we can check in polynomial time that every st-path in P contains at least one occurrence of the rule. We also note that, by a slight extension of the proof of Theorem 1, we can generalize this result to any fixed set of m rules:

Theorem 2. *Let $\{f_1, \ldots, f_m\}$ be a fixed set of rules, with $m \geq 1$. The following problem is NP-complete: Given an st-dag G, a subset P of st-paths from G, and integer $l \geq 1$, decide whether we can place the rules on l switches in such a way that every path in P contains at least one occurrence of each f_i ($i = 1, \ldots, m$).*

Finally let us mention that the related problem of placing m rules $\{f_1, \ldots, f_m\}$ on l switches of an st-dag G such that for every rule f_i, *every* st-path in G is covered by f_i can be solved in polynomial time [136].

5.2.3 Rule Placement under Constraints

Beyond the core placement objective introduced in the previous section, practical network deployments impose additional constraints that must be explicitly incorporated into the placement strategy to ensure both efficiency and correctness. In this chapter, we consider two fundamental constraints, namely bandwidth-related considerations and rule priority, which play a critical role in real-world network operations.

Rule Placement under Bandwidth Constraint

Efficient utilization of network bandwidth is a key requirement in communication networks, as traffic consumes resources along the links interconnecting switches. Consequently, limiting unnecessary traffic propagation across the network is essential to maintain high performance and avoid congestion. A common and effective strategy is to place filtering rules with a *deny* action as close as possible to the source node along the forwarding path. By filtering unwanted or irrelevant traffic at an early stage, packets that do not satisfy the policy are prevented from traversing multiple links, thereby reducing bandwidth consumption across the network. This early filtering mechanism limits congestion, improves resource utilization, and ensures that only valid traffic is forwarded through the network infrastructure [87].

It is important to note that this constraint differs from the distance-based constraint introduced in Chapter 4. While the latter focuses on minimizing the placement distance along the shortest paths, reflecting typical routing behavior where traffic is forwarded along shortest routes, the bandwidth constraint adopts a more general perspective. Specifically, it aims at minimizing the propagation of unwanted traffic across all paths originating from the source node, rather than restricting the optimization to shortest paths only. This distinction is particularly relevant in dynamic or congested network conditions. For example, in the presence of high traffic loads or denial-of-service (DoS) attacks, shortest paths may become saturated, leading to traffic being redirected over alternative paths. In such scenarios, limiting bandwidth consumption requires considering all potential forwarding

paths, ensuring that undesired traffic is filtered as early as possible, regardless of the path it may eventually take.

Rule Placement under Ordering Constraint

Handling the ordering or priority of rules is a critical constraint in the placement of ACL rules. This constraint ensures that rules with greater specificity or importance are applied before the other more general ones, which is essential for accurate traffic management and security. For example, when rules are written using prefixes and wildcard bits, a rule with a more specific match should take precedence over a broader one [83]. This prevents broader rules from inadvertently overriding more targeted, critical policies. Imagine two rules: one that permits all traffic to a subnet 192.168.1.0/24 and another that blocks access to a specific IP within that subnet 192.168.1.5. The second rule, with the larger prefix /32, has higher priority and must be placed before the general rule /24. Failing to respect this order could allow traffic that should be blocked, creating a security breach.

Additionally, in modern SDN environments, ACLs are no longer limited to binary actions such as *allow* or *drop*. The *action* field has been extended to support more expressive functionalities, including routing directives such as *forward_to_port*. ACLs that specify such forwarding behavior are commonly referred to as *routing policies*, whereas traditional security-focused ACLs are categorized as *endpoint policies*. Endpoint policies—responsible for security, access control, and Quality of Service (QoS)—must be prioritized over routing policies, which focus on determining packet paths [85, 57]. Endpoint policies directly control packet filtering, meaning they should always be applied first to ensure proper enforcement of access and security measures before routing decisions are made. For instance, if an endpoint policy blocks traffic to a specific port, it should override a routing policy that would otherwise allow that traffic to flow. Properly managing this rule priority prevents security vulnerabilities and ensures optimal network performance by placing critical rules first and avoiding policy conflicts.

5.3 ACL Rule Placement Framework

The proposed approach builds upon the GNN-DQN framework introduced in Chapter 4 (see Section 4.3). The overall RL pipeline is preserved, while its components are adapted to the generalized placement problem defined over a subset of paths P. At each decision step, the agent observes the current placement state of the network and interacts with the environment by selecting a rule-node installation action, after which the environment returns the next state and the corresponding reward.

Within this framework, the GNN acts as an encoder of the graph-structured state. It receives the current network representation, including the topology and node features, and transforms it into topology-aware node embeddings that capture both local and global structural information. These embeddings are then passed to the DQN, which acts as the decision-making component by estimating the action values associated with candidate placements and selecting the most appropriate action according to the learning policy.

As illustrated in Figure 4.1 and detailed in Algorithm 6 of Chapter 4, the training procedure remains unchanged and relies on the same learning mechanisms, including experience replay, target network synchronization, and ϵ-greedy exploration. In this chapter, the main adaptations concern the problem-dependent components of the framework, namely the state representation and the interpretation of path-related features with respect to the selected path set P.

5.3.1 Agent State, Action, and Reward Design

State As in Chapter 4 (Section 4.3.2), a state encodes the current distribution of rule sets over the network graph $G = (V, E)$ together with node-level features. Each node $v \in V$ is associated with k feature vectors $x_v^1, \ldots, x_v^k$, where k denotes the number of rule sets. For a given rule f_i and a specific node v, the feature vector x_v^i consists of six components.

$x_v^i[1]$ Node degree (i.e., number of edges incident to v);

$x_v^i[2]$ Boolean indicator of whether rule f_i is installed on node v;

$x_v^i[3]$ Number of paths in P passing through v that already contain rule f_i;

$x_v^i[4]$ Number of paths in P passing through v that do not yet contain rule f_i;

$x_v^i[5]$ Remaining capacity of node v (available TCAM space);

$x_v^i[6]$ Boolean indicator of whether node v belongs to at least one path in P.

The components $x_v^i[2], x_v^i[5]$ and $x_v^i[6]$ are node-specific and vary locally, while $x_v^i[1], x_v^i[3]$, and $x_v^i[4]$ are global, reflecting the overall state of the graph and the episode's context. The roles of these components are as follows: $x_v^i[2], x_v^i[5]$ and $x_v^i[6]$ provide key insights to evaluate the validity of an action, while $x_v^i[1], x_v^i[3]$, and $x_v^i[4]$ offer essential guidance for determining the most appropriate action, such as selecting which rule to install on which node in a given state.

Compared to Chapter 4, the key difference lies in the computation of path-related features, which are now restricted to the subset P instead of the full set of feasible st-paths. This modification enables the agent to focus on the effective routing structure of the network.

Action The action space retains the same structure as in Chapters 3 and 4: an action $a(v, f_i)$ consists of installing rule set f_i on node v. However, the validity criterion is adapted to the selected-path setting considered in this chapter. In addition to the previously defined structural and capacity requirements, an action is considered invalid if node v does not belong to any path in the predefined paths set P, since installing a rule on such a node cannot contribute to the targeted policy enforcement. Among the remaining feasible actions, an action is classified as redundant if installing f_i on v does not cover any previously uncovered path in P for that rule set. Otherwise, the action is valid, as it respects all placement constraints and contributes to improving policy coverage over the selected paths.

Reward The reward function follows the same formulation as in Chapter 4, adapted to the subset of paths P. The objective remains to maximize coverage while minimizing resource usage and avoiding redundant placements. The reward for taking action $a(v, f_i)$, which installs rule f_i on node v, is determined accordingly:

$$reward\big(a(v,f_i)\big) = \begin{cases} -penalty, & \text{if } a(v,f_i) \text{ is invalid or redundant,} \\ paths_{v,f_i} - occ, & \text{otherwise.} \end{cases}$$

where $paths_{v,f_i} \in [0,1]$ represents the ratio of new paths that are covered by this action. Specifically, it is the ratio of paths in P that pass through node v and did not previously contain the rule f_i, to the total number of paths in P. The term $occ \in [0,1]$ denotes the memory occupancy ratio of the graph G, calculated as the total number of rules installed across all nodes in G relative to its total capacity. The total capacity is determined by the number of nodes in G that participate in paths within P, multiplied by each node's capacity m.

5.3.2 Reward Shaping under Bandwidth Constraint

To effectively address the placement problem while considering the bandwidth constraint, specifically by placing deny rules as close to the source s as possible, the state and the actions remain the same but the reward for the action $a(v, f_i)$ of installing rule f_i on node v is adjusted accordingly:

$$reward\big(a(v,f_i)\big) = \begin{cases} -penalty, & \text{if } a(v,f_i) \text{ is invalid or redundant,} \\ paths_{v,f_i} - occ - dist_{s,v,f_i}, & \text{otherwise.} \end{cases}$$

where

$$dist_{s,v,f_i} = \frac{1}{|paths_v|} \sum_{p \in paths_v} \frac{||p_i||}{||p||}$$

and where p_i is the prefix path of p that goes from s to the first node that contains rule f_i. Since matched packets are processed at the first switch where rule f_i is installed, only the first occurrence of the rule placement along a path should be considered when computing the distance. The value $||p_i||$ is the length of p_i. This length is normalized by the length of p. Finally, $dist_{s,v,f_i}$ is obtained by averaging the result across the number of paths in P that contain v. In summary, $dist_{s,v,f_i}$ represents the average normalized distance, measured in hop count, from the source node s to the first occurrence of rule f_i across all paths in P that pass through node v.

5.3.3 Handling Rule Ordering Constraint

To incorporate the ordering constraint within the placement problem, we follow the same modeling approach as in the previous sections. The state, action space, and reward function remain unchanged, while the definition of valid actions is extended to enforce rule priority relationships. Let f_i and f_j be two rules such that f_i has higher priority than

f_j. The action $a(v, f_i)$, corresponding to the installation of f_i on node v, is considered valid only if no lower-priority rule f_j is installed on any node that precedes v along the paths in P that pass through v ($paths_v$). Conversely, the action $a(v, f_j)$ is valid only if no higher-priority rule f_i is installed on any node that follows v along these paths. This ensures that rule evaluation order is preserved along all paths in P.

The incorporation of the ordering constraint introduces additional dependencies between placement decisions, which fundamentally alters the nature of the decision process. Specifically, the validity of an action no longer depends solely on the current placement configuration, but also on information that is not explicitly captured in the state representation, such as the priority relationships between rules. As a result, two identical states in terms of rule placement may correspond to different feasibility conditions depending on the underlying ordering requirements. This situation violates the Markov property, which requires that the current state fully encapsulates all information necessary to evaluate future actions. In the present formulation, the state encodes only the placement configuration and does not include rule priority relationships, which are essential for determining ordering feasibility. Consequently, the validity of an action cannot be inferred solely from the current state and may implicitly depend on unobserved aspects of the decision process. As a result, the problem becomes non-Markovian from the agent's perspective.

This contrasts with the bandwidth constraint, which can be naturally incorporated through reward shaping without affecting the Markovian structure of the problem. The ordering constraint directly impacts the feasibility of actions, making it incompatible with the standard RL formulation unless additional information is incorporated into the state. As standard RL algorithms such as Q-learning rely on the Markov assumption, this lack of state completeness may lead to suboptimal decisions and inefficient training.

To address this limitation, we propose two complementary approaches that enable the enforcement of ordering constraints while preserving the effectiveness of the learning framework. The first approach consists in reformulating the problem so as to recover a Markovian structure by augmenting the state representation. The second approach enforces the ordering constraint through a post-processing phase applied after solving the basic placement problem.

Markovian Reformulation for Joint Placement and Ordering

To enable the integration of the ordering constraint within the GNN-DQN framework, the state representation is augmented with two additional features. First, each rule f_i is associated with a priority level that encodes its relative importance, allowing the agent to distinguish between higher- and lower-priority rules. Second, for each candidate action $a(v, f_i)$, a Boolean indicator is introduced to specify whether the action is valid with respect to the ordering constraint. This validity indicator is computed within the environment based on the current placement configuration, the rule priority hierarchy, and the relative ordering of nodes along the paths in P. By explicitly encoding both rule priorities and ordering feasibility, the augmented state provides the agent with all the necessary information to evaluate candidate actions without relying on the history of previous decisions. Consequently, the ordering-constrained placement problem can be reformulated as a Markov Decision Process (MDP), allowing the direct application of the GNN-DQN

framework.

Incorporating rule ordering requirements directly into ILP-based formulations, such as Raptor [85], can substantially increase the size of the optimization model. Unlike coverage or capacity constraints, ordering constraints must represent precedence relationships between pairs of rules and relate these dependencies to their possible placement locations along the admissible paths. In Raptor, these dependencies are captured through a rule-dependency graph: a directed edge from rule f_i to rule f_j is introduced whenever both rules may match the same packet and f_i has a higher priority than f_j. The placement model must then ensure that, on every relevant path, f_i is encountered either at the same switch as f_j or at a switch preceding the one on which f_j is installed.

The explicit enforcement of this requirement is computationally demanding because it must be considered for each dependent pair of rules and for each possible pair of candidate placement nodes. In the worst case, if F denotes the set of applicable rules, V the set of candidate switches, and P the set of relevant paths, the number of ordering related combinations may grow as $O\left(|P|\,|F|^2\,|V|^2\right)$. Here, the factor $|F|^2$ reflects the number of rule pairs whose relative priorities may need to be represented, while $|V|^2$ accounts for the possible pairs of switches on which two dependent rules may be installed. These placement relationships must additionally be verified along each path in P. Consequently, the direct integration of ordering constraints can rapidly make the ILP formulation difficult to solve as the number of rules, candidate placement locations, or admissible paths increases. For this reason, ILP-based approaches relax priority ordering requirements during its initial allocation phase and subsequently adjusts the placement through a dedicated correction phase.

Post-Processing for Ordering Enforcement

As an alternative to modifying the learning process, the ordering constraint can be enforced through a post-processing phase applied after solving the placement problem without ordering constraints. Given a graph $G(V, E)$, a set of rules F, and a set of paths P, a solution S of $PLACE(G, m, F, P)$ assigns each rule in F to specific nodes in G to ensure all paths in P are covered. We select a *pivot* which is a minimal set of nodes $\mathcal{S} \in S$ that contains all occurrences of some rule. We compute the distances (shortest path length) $d(s, \mathcal{S})$ from s to $\mathcal{S}$ and $d(\mathcal{S}, t)$ from $\mathcal{S}$ to t. Depending on these distances, we modify the graph as follows. If $d(\mathcal{S}, t) \times m \geq |F| - 1$, a virtual source s' is created that is connected to every node of the pivot set, with higher-priority rules placed on the pivot, after which the GNN-DQN agent is called to place the remaining rules after the pivot in the modified graph. If $d(s, \mathcal{S}) \times m \geq |F| - 1$, the node in the pivot set are connected to a virtual destination t', lower-priority rules are placed on the pivot, and the GNN-DQN agent is called in order to place the remaining rules before the pivot. If neither path length exceeds $|F| - 1$, we redefine the pivot as the next smallest set in S possibly larger than the current pivot. This approach guarantees priority preservation if the ordering algorithm is completed successfully.

5.4 Evaluation Methodology and Baseline Comparison

5.4.1 Baseline Algorithms for ACL Rule Placement

In this chapter, we have compared the placement generated by our GNN-DQN algorithm with four state-of-the-art algorithms, including Palette [25], Raptor [85], Raptor++ [87], and an updated version of st-mincut [28].

st-mincut: In our previous work [28], we propose a polynomial heuristic based on the computation of a minimum cut in a graph, to address the rule placement problem across *all paths* in a network. In this approach, an st-cut in an st-dag is defined as a set of nodes C whose removal results in two disjoint subgraphs, one containing the source and the other the destination. The value of a cut C is given by the number of nodes it contains, and a minimum cut (or mincut) is the cut with the fewest nodes. To apply the st-mincut algorithm in [28] to the placement problem over a specified subset of paths P, we simply run this algorithm on the subgraph $G|_P$ of G whose nodes and edges are in P. To illustrate this, consider the graph presented in Fig. 5.1, where P contains the first and second paths, *Path 1* and *Path 2*. Based on these paths, the subgraph $G|_P$ is derived by removing the edge (g, e).

Raptor: In [85], the authors reformulate the placement problem as an ILP-based approach. They represent the objective function using a boolean allocation matrix of dimensions *(number of rules)* $\times$ *(number of nodes)*, aiming to minimize the total number of placed rules. Each element in this matrix, $x_{i,j}$, equals 1 if rule i is placed in switch j, and 0 otherwise. The objective is to minimize the sum of all matrix elements while satisfying two key constraints: (1) the total number of rules assigned to any switch must not exceed its capacity, and (2) each rule must appear at least once on every path within the specified set of paths, P.

Palette: In [25], the authors introduce a heuristic approach to the placement problem, representing rules as colors and iteratively selecting uncolored nodes until each path includes at least one selected node. They present two algorithm variants: *1-GREEDY*, which selects a node that maximizes the number of uncovered paths at each step, and *q-GREEDY*, which generalizes this by selecting a set of up to q nodes to maximize path coverage. The primary goal of their approach is to maximize the number of rules by counting the sets returned by the algorithm, creating a distinction between the two variants. However, in our scenario with a fixed number of rules, both variants yield the same outcome.

Raptor++: After extending our placement algorithm to minimize wasted bandwidth, we further compared our results to those in [87]. The authors in this work [87] tackled the placement problem with a bandwidth constraint by reformulating it as an ILP. Minimizing wasted bandwidth involves reducing the distance between the ingress switch and the first node where rules are placed. Reformulating and solving this constraint in an ILP-based

approach is challenging, as it is inherently quadratic and requires transformation into a linear form through the introduction of additional decision variables and constraints. The authors' approach introduces a dual-objective function: the first entity minimizes wasted bandwidth, while the second entity minimizes the number of installed rules. This second entity is identical to the Raptor formulation [85], sharing the same constraints, decision variables, and objective function. Building on this, we integrated the bandwidth minimization entity into Raptor's codebase, solving the extended ILP using Gurobi instead of PuLP as used in [87]. This extended version of Raptor is referred to as *Raptor++*.

5.4.2 Experimental Setup

The experimental setup follows the same configuration as described in Chapter 4 (Section 4.4), ensuring consistency in the evaluation and enabling a direct comparison between the two problem formulations. The GNN-DQN algorithm (Algorithm 6) is employed to solve the generalized placement problem over a subset of paths P, using the same graph generation process detailed in Section 4.4.2. The implementation was carried out in Python and executed on the same computational platform as in Chapter 4, namely a remote GPU server equipped with an Intel(R) Xeon(R) Gold 6258R CPU @ 2.70 GHz, 503 GB of RAM, running Ubuntu 22.04.3 LTS, and an NVIDIA RTX A6000 GPU (Driver Version 530.30.02, CUDA 12.1).

Evaluation Metrics

We retain the two primary metrics introduced in Chapter 4, namely *Occupancy* and *Success Rate*. Occupancy measures the percentage of total switch capacity utilized by the installed rules, with lower values indicating more efficient resource usage. Success Rate corresponds to the percentage of valid placement solutions obtained on previously unseen graphs, reflecting the generalization capability of the learned policy.

To assess the impact of the bandwidth constraint, we introduce an additional metric, denoted by *dist*, which captures the relative position of rule placements with respect to the source node. For a given rule f_i, this metric measures the average distance from the source s to the first node where the rule is installed, computed over all paths in P. For a set of k rules, *dist* is defined as the average of these distances across all rules. This metric provides insight into the effectiveness of early traffic filtering and the extent to which the placement strategy reduces bandwidth consumption.

Training Configuration

The training protocol follows the same configuration as in Chapter 4 to ensure consistency. The GNN-DQN agent is trained for $M = 12000$ episodes over the training dataset, with a new graph sampled every $k = 2$ episodes to promote exposure to diverse topologies. Each episode is limited to a maximum of 500 steps, corresponding to potential placement actions, and terminates when either (i) all rules are successfully placed on all paths in P, (ii) the maximum number of steps is reached, or (iii) all nodes become saturated while uncovered paths remain. An ϵ-greedy exploration strategy is adopted with initial exploration rate $\epsilon_0 = 1$ and minimum exploration rate $\epsilon_{\min} = 0.09$. The decay factor is

defined as $\lambda = \exp\left(\frac{2\ln(0.5)}{M}\right)$, ensuring that ϵ decreases to 0.5 halfway through training, thereby balancing exploration and exploitation. The discount factor is set to $\gamma = 0.8$, and the learning rate to $\alpha = 10^{-5}$ to ensure stable convergence. A penalty of -10 is applied to invalid or redundant actions in the reward function. Training is performed using mini-batches of size 32, and the target network is synchronized with the online network every $C = 1000$ steps. Validation is conducted every $val_freq = 50$ episodes using the validation set to monitor generalization performance.

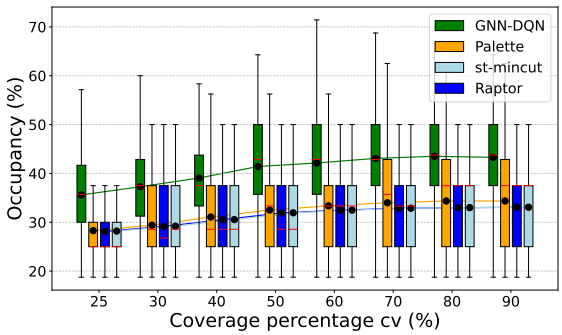
5.5 Experimental Results and Performance Analysis

We conducted a series of experiments by generating 1,000 graphs for training, 200 for validation, and 500 for testing, with each dataset including graphs of sizes 10, 15, 20, and 25 nodes. For each trained model, we set a fixed number of rules. The *coverage percentage*, denoted as cv, is a real number within the range $[0, 1]$ that specifies the number of paths included in P. The number of paths in P is obtained by multiplying the total number of paths in the graph G by the coverage percentage cv, resulting in $|P| = |P_G| \times cv$. Once the size of P is computed, $|P|$ paths are randomly selected from P_G. The agent was trained once using a coverage percentage of $cv = 50\%$. In subsequent experiments, this trained model was directly applied to entirely new graphs, unseen during the training phase, with a variable number of paths adjusted by varying the coverage percentage cv.

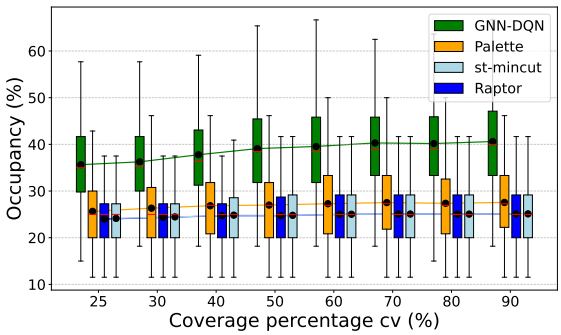
5.5.1 Performance With Underutilized Capacity

To demonstrate the effectiveness of our approach and the ability of graph embeddings combined with DQN to capture graph features in solving combinatorial optimization problems, we conducted a series of testing experiments. First, we evaluated our trained model on new graphs of the same sizes as those used during training, varying the coverage percentage from 25% to 90%. Increasing the coverage percentage for the same graph results in the generation of a new graph, as the number of paths in P increases. The additional paths may introduce new edges and nodes, altering the graph structure. Since the upper bound on rules is determined by the number of nodes in the shortest path within P, we fixed the number of rules to half the placement slots in that path, $|F| = \frac{1}{2} \times path_capacity(min(P))$. This number of rules remained constant across both training and testing, set to 3 for graphs with sizes 10 and 15, and to 4 for sizes 20 and 25. The agent's results were then compared to those of Palette, Raptor, and st-mincut.

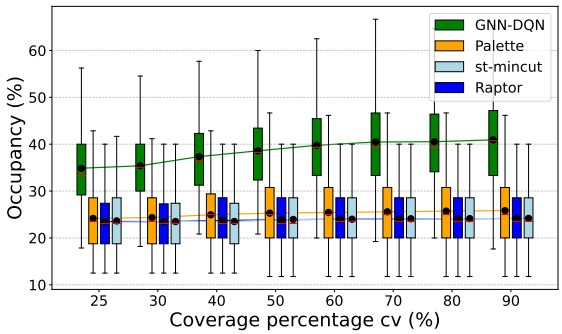
Figures 5.3a, 5.3b, 5.3c, and 5.3d depict the occupancy statistics of 500 graphs of each size for the four algorithms, with occupancy plotted against an increasing number of paths to cover within P, representing a higher coverage percentage. Each box illustrates the distribution of the occupancy values for 500 graphs of a given size at a specific coverage percentage, while the corresponding point (black point) indicates the average occupancy across these graphs. The results indicate that Raptor and st-mincut consistently provide the best solutions, followed by Palette and then GNN-DQN. We noticed too that for small graph sizes, like 10, the solution provided by the GNN-DQN approach is close to the solution provided by other algorithms.



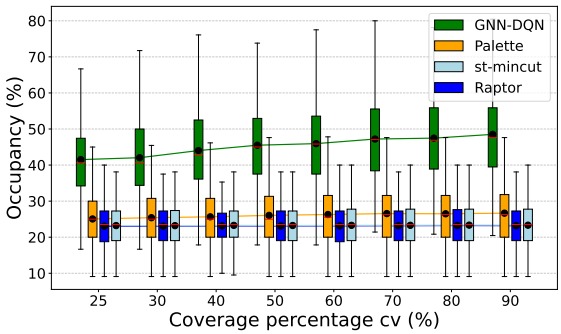
(a) Occupancy distribution with graphs of size 10



(b) Occupancy distribution with graphs of size 15



(c) Occupancy distribution with graphs of size 20



(d) Occupancy distribution with graphs of size 25

Figure 5.3 – Occupancy distribution of the four approaches across four graph sizes with varying *cv*.

This result can be attributed to two primary factors: (1) the simplicity of the test setup and (2) the complexity of hyperparameter tuning. The simplicity of the test setup offers substantial flexibility, as the number of available slots for rule placement is significantly larger (double) than the number of rules to be placed. This excess in placement slots provides enough space for multiple trials and variations, ensuring a minimal likelihood of failing to assign all rules to suitable slots. On the other hand, approaches like Palette, st-mincut, and Raptor utilize pre-defined mechanisms to address the placement problem systematically. These methods follow fixed procedures, such as identifying nodes that cover the maximum number of paths, installing rules on those nodes, and repeating this process until all rules are installed. Their efficiency in this setup stems from the surplus of placement slots relative to the number of rules. In contrast, the GNN-DQN agent does not rely on such predefined strategies. Instead, it begins with no prior knowledge and depends on exploration to learn effective solutions. This approach necessitates careful tuning of several hyper-parameters, including the number of message-passing steps, trials per episode, episode counts, and action space dimensions. Each of these factors plays a crucial role in shaping the agent's learning and performance, adding complexity to the process.

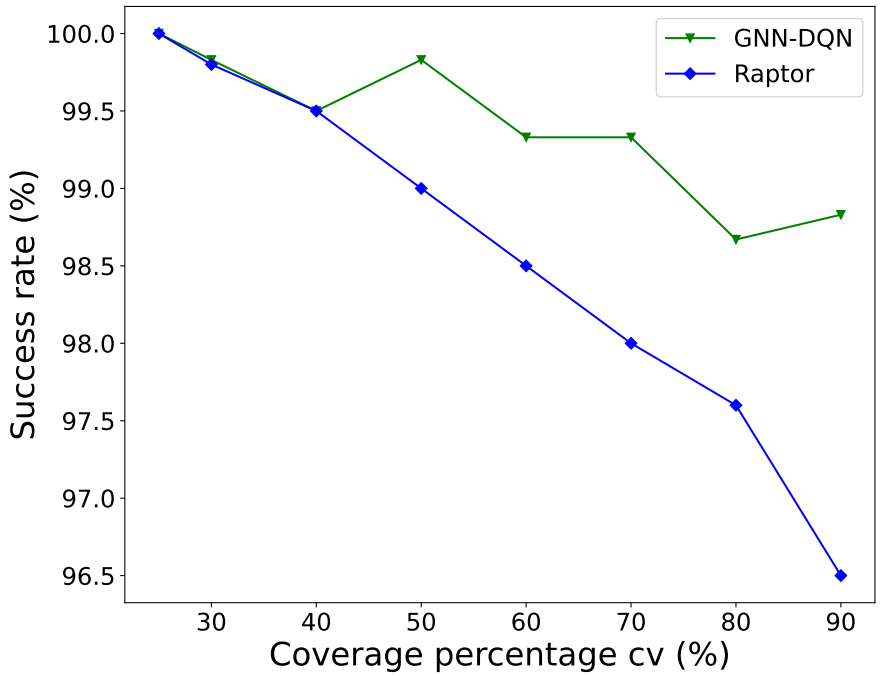


Figure 5.4 – Success rates for Raptor and GNN-DQN algorithms over 500 graphs of size 25 while increasing the coverage percentage cv.

The GNN-DQN agent operates by associating each state with a set of candidate actions and selecting the action that maximizes the estimated Q-value. In the considered formulation, the size of the action space grows as the product of the number of nodes and the number of rules, leading to a combinatorial increase in possible decisions as the graph size increases. Moreover, the proportion of valid actions remains relatively small due to the multiple constraints imposed on the placement process. In particular, actions must satisfy structural and resource constraints, including the exclusion of source and destination nodes, adherence to node capacity limits, avoidance of redundant placements, and participation in at least one path within the considered set P. As a result, a large fraction of the action space corresponds to invalid or redundant (non-contributing) actions. Given that the number of interaction steps per episode is fixed to 500 regardless of the graph size, the agent is able to explore a significant portion of the action space in smaller graphs, where the number of possible actions is limited. This facilitates convergence toward high-quality solutions comparable to those obtained by alternative methods. In contrast, for larger graphs characterized by a substantially expanded action space and a sparse distribution of valid actions, the fixed interaction budget becomes insufficient to adequately explore the decision space. Consequently, the agent faces increased difficulty in effectively exploring and exploiting candidate actions, which may prevent it from identifying solutions of comparable quality to those produced by other approaches. This observation is consistent with the findings of Chapter 3, where the introduction of action masking significantly

improved the behavior of the RL agent. By explicitly removing invalid and redundant actions from the decision process, the agent was able to focus its exploration on feasible and contributing placements. As a result, the masked RL formulation recovered solutions with the same placement quality as st-mincut, demonstrating that the main difficulty does not lie in the learning model itself, but in the inefficient exploration of a large and highly constrained action space.

Despite those complexities, our GNN-DQN agent, trained only once at a 50% coverage level, demonstrated remarkable generalization power, achieving over 95% success on new graph sets across all tested sizes and coverage levels. Additionally, we observed that Raptor's success rate decreases as the coverage percentage increases. Fig. 5.4 illustrates the success rate of both Raptor and GNN-DQN for size-25 graphs, showing that our model, despite being trained on other graphs, achieves a higher success rate on larger graphs than the ILP-based Raptor.

To further demonstrate the limitations of ILP-based approaches on larger instances, we generated 30 graphs each for sizes 30 and 40, comparing the results across all four algorithms. Table 5.2 presents the average occupancy for these 30 graphs per size, highlighting that Raptor fails to effectively handle larger instances, while the GNN-DQN agent can adapt and provide feasible solutions. Additionally, these results emphasize that our st-mincut approach consistently delivers the best solution in terms of occupancy, even for larger graph instances

Graph size-rules	Avg. paths	GNN-DQN	Palette	st-mincut	Raptor
30-5	408	74	32	26	–
40-7	930	80.64	20.97	20.97	–

Table 5.2 – Average occupancy of the four algorithms on 30 graphs of each graph size with $cv = 25\%$.

5.5.2 Performance under Near Maximum Capacity

To further assess the performance of our GNN-DQN approach against existing methods with more challenging conditions, we conducted experiments on the same graph sets as before but increased the number of rules from half to nearly the total number of placement slots in the shortest paths, i.e., $|F| = path_capacity(min(P)) - 1$. This increase greatly reduces placement flexibility, as the number of rules approaches the available slots, making the problem significantly more constrained.

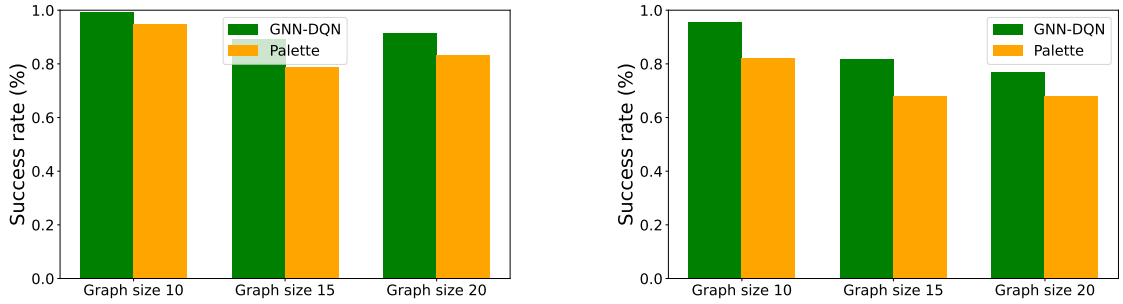
Graph size-rules	Avg. paths	GNN-DQN	Palette	st-mincut	Raptor
10–5	4.502	65.45 ± 8.60	53.23 ± 11.12	52.97 ± 10.91	51.27 ± 11.01
15–5	27.69	67.17 ± 8.63	52.05 ± 10.04	49.52 ± 8.56	43.70 ± 8.71
20–7	13.47	69.58 ± 8.90	54.63 ± 9.92	52.80 ± 9.07	46.74 ± 9.40

Table 5.3 – Average occupancy of the four algorithms with $|F| = path_capacity(\min(P)) - 1$ and $cv = 25\%$.

Graph size-rules	Avg. paths	GNN-DQN	Palette	st-mincut	Raptor
10–5	9.552	71.79 ± 8.60	61.31 ± 11.15	59.59 ± 10.21	57.25 ± 10.96
15–5	55.892	72.14 ± 8.85	53.82 ± 9.95	50.44 ± 8.26	44.85 ± 8.64
20–7	27.45	76.02 ± 7.26	56.72 ± 9.48	53.70 ± 8.48	47.35 ± 9.06

Table 5.4 – Average occupancy of the four algorithms with $|F| = path_capacity(\min(P)) - 1$ and $cv = 50\%$.

Tables 5.3 and 5.4 present the average occupancy achieved by each algorithm across 500 graphs of each size for coverage percentages of 25% and 50%, respectively. The results demonstrate that Raptor outperforms st-mincut in terms of average occupancy with this setup, which contrasts with earlier experiments where both algorithms exhibited similar performance. Additionally, Palette ranks third in solution quality (average occupancy), while the GNN-DQN agent comes last.



(a) Success rate for $cv = 25\%$

(b) Success rate for $cv = 50\%$

Figure 5.5 – Success rate comparison of GNN-DQN and Palette with $|F| = path_capacity(min(P)) - 1$ and different cv.

Figures 5.5a and 5.5b illustrate a comparison of the success rates between the GNN-DQN agent and Palette for coverage percentages of 25% and 50%. These results show that the GNN-DQN agent exhibits a notable ability to generalize and adapt to stricter constraints, maintaining a higher success rate than Palette in highly constrained condition (near maximum capacity).

As previously explained, these results can be attributed to the challenges in optimizing the GNN-DQN hyperparameters and the relative simplicity of the placement problem under one constraint which covers all given paths with the minimum number of rules. Even after increasing the number of rules, the complexity of the problem itself did not change significantly. Instead, the problem shifted to relying on a global or long-term perspective of the placement process. Specifically, increasing the number of rules from half the number of nodes in the shortest path to one less than this number results in the number of rules being nearly equal to the available placement slots on the nodes, compelling the algorithm to adopt a more strategic approach. Any algorithm must account for the remaining rules and available nodes to exploit resources optimally.

Palette, for example, installs rules incrementally across all given paths by prioritizing junction nodes, nodes that cover the maximum number of paths. However, due to the structured nature of the graph, and the eventual saturation of junction nodes, Palette often fails to find enough free spaces to complete all paths, leading to a reduced success rate. This highlights the advantage of using GNN-DQN, as the agent's design inherently focuses on maximizing long-term gains and has a global view of the environment. By taking a comprehensive approach to rule placement, the GNN-DQN agent is better equipped to handle the constrained nature of the problem, even under conditions where strategic planning is critical.

5.5.3 Effect of the Bandwidth Constraint

To further evaluate the performance and capability of the GNN-DQN approach in handling the placement problem under a multi-constrained setting, we extended our analysis to include an additional constraint: minimizing wasted bandwidth. Using the same graphs from the previous section, we trained four models across various graph sizes (10, 15, 20, and 25) to address the ACL rules placement problem under this new constraint. During training, the number of rules was fixed to $|F| = 0.5 \times path_capacity(min(P))$ to evaluate the models' performance under a more relaxed configuration. In real-world networks, rules are constructed based on network administrators' specific objectives, such as priority, action, and IP ranges. Since the placement of rules with a "deny" action was not predetermined, this experiment aims to minimize the distance of all rules from the source node. Notably, Palette and st-mincut could not adapt to solve the placement problem with the bandwidth constraint. This limitation arises because these heuristics are specifically designed for a unique objective function, leveraging deep knowledge and familiarity with the problem. Any modification to the objective function would require developing entirely new heuristical approaches.

	GNN-DQN			Raptor++		
Graph size-rules	Avg. occ(%)	Avg. time(s)	Avg. dist(link)	Avg. Occ(%)	Avg. Time(s)	Avg. dist(link)
10-3	34.21	0.0126	2.41	38.52	0.0143	1.367
15-3	33.18	0.0329	2.53	31.97	0.0743	1.33
20-4	32.31	0.041	3.11	31.91	0.0891	1.51
25-4	33.43	0.086	2.98	27.5	0.165	1.51

Table 5.5 – Performance of GNN-DQN and Raptor++ for the placement problem with bandwidth constraint and $cv = 25\%$.

Tables 5.5 and 5.6 summarize the performance of each algorithm in terms of average occupancy, average execution time, and average distance, evaluated on 500 unseen graphs for coverage percentages of 25% and 50%, respectively. The results highlight that GNN-DQN, despite involving computationally intensive operations like message passing and matrix multiplication within its neural network layers, is faster than Raptor++. In terms of occupancy, GNN-DQN outperforms Raptor++ on smaller graphs (size 10) where the small action space allows the agent to explore a greater variety of possible actions. For larger graphs, where the action space grows proportionally, GNN-DQN delivers solutions

	GNN-DQN			Raptor++		
Graph size-rules	Avg. occ(%)	Avg. time(s)	Avg. dist(link)	Avg. Occ(%)	Avg. Time(s)	Avg. dist(link)
10-3	40.65	0.0147	2.24	43.49	0.0241	1.352
15-3	35.76	0.0247	2.51	31.51	0.0898	1.33
20-4	34.27	0.0566	2.95	31.37	0.1549	1.51
25-4	36.24	0.0701	2.96	25.55	0.1231	1.508

Table 5.6 – Performance of GNN-DQN and Raptor++ for the placement problem with bandwidth constraint and $cv = 50\%$.

that are close to those of Raptor++. Regarding the average distance metric, Raptor++ demonstrates better performance. This is due to its sequential strategy of prioritizing nodes closer to the source, gradually expanding outward (e.g., saturating nodes at distance 1 before moving to distance 2, and so on).

In contrast, GNN-DQN employs a reward-driven approach, as detailed in Section 5.3.2. Its reward function consists of three components: the first encourages placing rules on junction nodes, the second minimizes the total number of rules, and the third reduces the distance between the source and the nodes where rules are installed. Unlike Raptor++, GNN-DQN balances these objectives by selecting the closest junction nodes to the source that cover all paths in P, thereby optimizing a trade-off among the competing goals. To further analyze the trade-offs between the components of the reward function, we modified it into a weighted sum for valid actions to assess the influence of its competing objectives. Specifically, the reward was reformulated as: $reward(a(v, f_i)) = \alpha \times (paths_{v,f_i} - occ) - (1 - \alpha) \times dist_{s,f_i,v}$. Here, the weight parameter α determines the relative importance of optimizing occupancy versus minimizing distance.

Fig. 5.6 illustrates the evolution of average occupancy and average distance as α increases for graph size 15 (similar results observed for other graph sizes). When α is small, the reward prioritizes minimizing the distance, leading to a lower average distance at the expense of higher occupancy. Conversely, as α approaches 1, the reward shifts focus to reducing occupancy, resulting in lower occupancy but higher average distance. This demonstrates how the balance between the two objectives can be adjusted by tuning α, offering a flexible means to adapt the solution to different optimization priorities.

Figures 5.7a and 5.7b compare the success rates of the GNN-DQN agent and Raptor++ for coverage percentages of 25% and 50%. These results demonstrate that GNN-DQN, applied to unseen graphs, achieves a higher success rate than Raptor++. This highlights the scalability of GNN-DQN in addressing large graphs and its effectiveness in handling multi-constrained problems, outperforming ILP-based approaches like Raptor and Raptor++.

5.5.4 Generalization to Real-World Network Topologies

In all previous experiments, we utilized synthetic graphs generated with our tool, as described in Chapter 4 (Section 4.4.2). These synthetic graphs were designed to closely mimic real-world networks by incorporating characteristics observed in the Internet Topology Zoo topologies. To evaluate the performance of our GNN-DQN agent on real-world networks, we repeated the previous experiments using actual topologies from the Internet

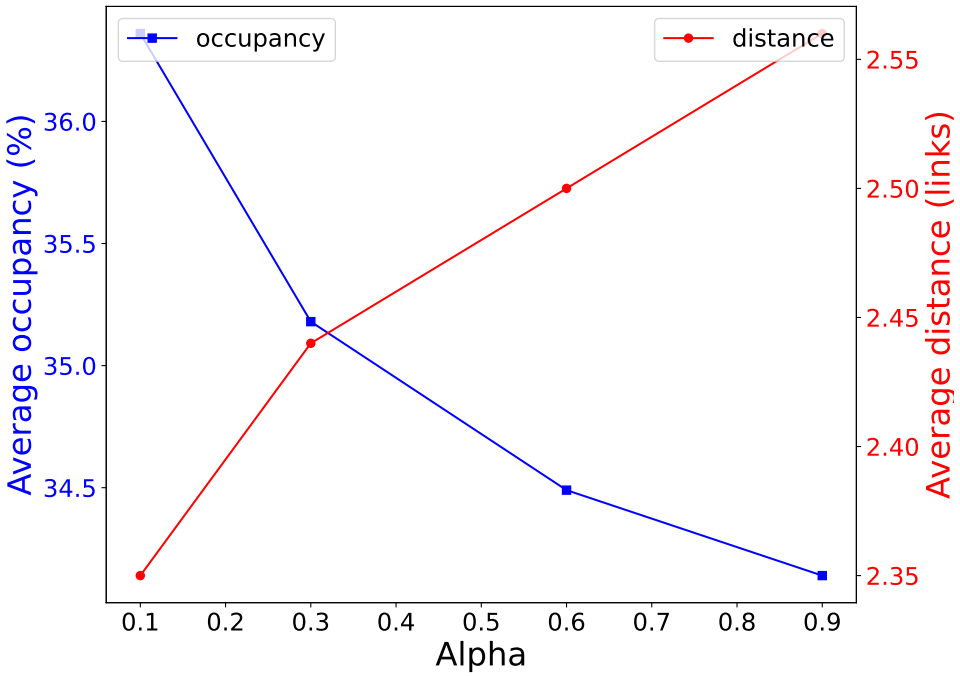
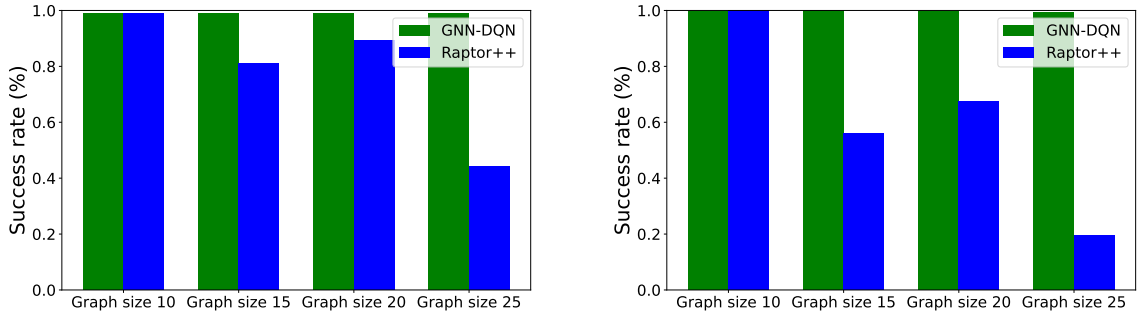


Figure 5.6 – Variation of average occupancy and average distance with α, illustrating the trade-off between occupancy (left y-axis) and distance (right y-axis) for graph size 15.



(a) Success rate for $cv = 25\%$

(b) Success rate for $cv = 50\%$

Figure 5.7 – Success rate comparison of GNN-DQN and Raptor++ for the placement problem with the bandwidth constraint for different cv.

Topology Zoo [131]. For this evaluation, we employed the previously trained GNN-DQN agent to compare its behavior on real-world graphs against other algorithms.

Tables 5.7 present the results of the four approaches applied to five graphs from the Internet Topology Zoo under the first experimental setup described in Section 5.5.1. This setup involves covering all paths in P with minimum occupancy, where $|F| = 0.5 \times path_capacity(min(P))$, the number of rules to install is half the available placement slots

Graph size–rules	Paths	GNN–DQN	Palette	st-mincut	Raptor
Sprint 11–3	7	31.25	25.00	25.00	25.00
Abvt 23–6	27	31.57	26.32	26.32	26.32
Agis 25–5	118	50.00	26.08	21.74	21.74
Bics 34–8	117	39.06	15.62	15.62	15.62
NetworkUSA 35–7	3	39.47	18.42	18.42	18.42

(a) Occupancy on real-world topologies with $cv = 50\%$.

Graph size-rules	Paths	GNN-DQN	Palette	st-mincut	Raptor
Sprint 11–3	14	27.77	22.22	22.22	22.22
Abvt 23–6	54	52.38	23.81	23.81	23.81
Agis 25–5	236	56.52	26.08	21.74	21.74
Bics 34–8	234	37.50	15.62	15.62	15.62
NetworkUSA 35–7	7	30.30	21.21	21.21	21.21

(b) Occupancy on real-world topologies with $cv = 100\%$.

Table 5.7 – Occupancy of the four approaches on real-world topologies with $|F| = 0.5 \times path_capacity(\min(P))$ for different values of cv.

Graph size-rules	Paths	GNN-DQN	Palette	st-mincut	Raptor
Sprint 11–5	7	68.75	50.00	50.00	50.00
Abvt 23–11	27	68.42	57.89	52.63	52.63
Agis 25–9	118	78.30	–	47.83	47.83
Bics 34–13	117	71.87	–	32.81	32.81
NetworkUSA 35–13	3	65.78	36.84	36.84	36.84

(a) Occupancy on real-world topologies with $cv = 50\%$.

Graph size-rules	Paths	GNN-DQN	Palette	st-mincut	Raptor
Sprint 11–5	14	61.11	44.44	44.44	44.44
Abvt 23–11	54	85.71	64.28	54.76	54.76
Agis 25–9	236	76.26	–	47.83	–
Bics 34–13	234	70.31	–	32.81	–
NetworkUSA 35–13	7	63.63	48.48	40.91	40.91

(b) Occupancy on real-world topologies with $cv = 100\%$.

Table 5.8 – Occupancy of the four approaches on real-world topologies with $|F| = path_capacity(\min(P)) - 1$ for different values of cv.

in the shortest paths in P, and cv takes two values: 50% and 100%. Similarly, Tables 5.8 show results under the same experimental setup but with $|F| = path_capacity(min(P)) - 1$, maintaining cv at 50% and 100%. Finally, Tables 5.9 highlight the results of the placement problem with a bandwidth constraint for the GNN-DQN and Raptor++ approaches,

		GNN-DQN		Raptor++	
Graph size-rules	paths	Occupancy(%)	Avg. dist(links)	Occupancy(%)	Avg. dist(links)
Sprint 11-3	7	31.25	2.33	25	1.33
Abvt 23-6	27	36.84	5.33	-	-
Agis 25-5	118	60.86	2.6	-	-
Bics 34-8	117	50	5	-	-
NetworkUSA 35-7	3	21.05	6	18.42	2.28

(a) Performance of GNN-DQN and Raptor++ for the placement problem with bandwidth constraint and $cv = 50\%$.

		GNN-DQN		Raptor++	
Graph size-rules	paths	Occupancy(%)	Avg. dist(links)	Occupancy(%)	Avg. dist(links)
Sprint 11-3	14	27.78	1.67	22.22	1.33
Abvt 23-6	54	42.85	5.33	-	-
Agis 25-5	236	52.17	3.2	-	-
Bics 34-8	234	45.31	4.625	-	-
NetworkUSA 35-7	7	22.72	4.28	21.21	2.14

(b) Performance of GNN-DQN and Raptor++ for the placement problem with bandwidth constraint and $cv = 100\%$.

Table 5.9 – Performance of GNN-DQN and Raptor++ for the placement problem on real-world topologies with bandwidth constraint and different cv.

using the same Internet Topology Zoo graphs.

These results reinforce the findings from previous experiments. ILP-based approaches like Raptor and Raptor++ exhibit efficiency for small-scale decision variables, such as smaller graphs, fewer rules, or fewer paths in P. However, they struggle with larger-scale scenarios. Heuristic methods like Palette lack a global perspective and optimize gain at each step, which leads to failures in constrained scenarios requiring a holistic approach. Additionally, these heuristics are tailored to solve single-objective problems, and introducing additional constraints necessitates a comprehensive reformulation of the heuristic, which is both complex and time-consuming. In contrast, GNN-DQN demonstrates robustness and adaptability. It generalizes effectively across unseen graphs of varying sizes and settings, where the number of paths and rules change dynamically. Moreover, GNN-DQN accommodates new constraints seamlessly, proving its versatility and scalability in diverse problem scenarios.

5.6 Discussion and Practical Deployment Considerations

5.6.1 Mininet Deployment

To validate the effectiveness of the proposed approach in a realistic setting, we emulated the network topology depicted in Fig. 5.1 using Mininet, with all switches connected to a Ryu controller responsible for installing the rules prior to traffic execution.

Each switch is constrained by a TCAM capacity of $m = 2$. The set of forwarding paths is defined as $P = \{\text{Path 2}, \text{Path 3}, [s, a, b, d, g, t], [s, c, d, e, t]\}$.

The initial ACL, derived from a real enterprise firewall configuration, is decomposed into five rule sets: f_1 Ingress Filtering, f_2 Transport and Port Filtering, f_3 Context-Aware Filtering, f_4 Reverse Path Protection, and f_5 Last Switch Enforcement. Synthetic traffic flows are generated to match the header fields associated with each rule, enabling a controlled evaluation of the placement strategies.

Performance is assessed using two metrics. The first one, *path coverage*, measures the percentage of rule sets that are successfully deployed along all paths in P. The second one, *correctness*, evaluates whether the network behavior complies with the intended policy, i.e., whether permitted packets reach their destination and denied packets are effectively dropped. Since unmatched packets are dropped by default, correctness is determined by the absence of false negatives, corresponding to allowed packets that are incorrectly discarded.

The Palette algorithm proceeds by iteratively selecting junction nodes, i.e., nodes that maximize the number of covered paths, and incrementally placing rules. However, due to the structural constraints of the topology and the limited TCAM capacity, these junction nodes quickly become saturated, preventing the algorithm from completing the placement across all paths. As a result, Palette fails to install all rule sets, achieving only partial deployment with four rules (f_1: [d], f_2: [d], f_3: [c, g], f_4: [c, g]), corresponding to a path coverage of 80%. This incomplete placement leads to degraded correctness, with performance varying depending on the missing rule set: f_1 (66%), f_2 (56%), f_3 (92.67%), f_4 (88%), and f_5 (97.33%).

In contrast, the proposed GNN-DQN approach, leveraging its global representation of the network state and its ability to optimize long-term rewards, successfully installs all rule sets (f_1: [d], f_2: [c, e], f_3: [c, g], f_4: [g, e], f_5: [d]). This results in full path coverage (100%) and ensures correct policy enforcement across all evaluated traffic scenarios, achieving 100% correctness.

5.6.2 Discussion

This work does not aim to outperform ILP-based or heuristic approaches in terms of optimality metrics such as rule occupancy, as GNN-DQN is a neural approximator meant to emulate their behavior. Rather, the objective is to demonstrate the distinctive advantages of the GNN-RL framework in terms of scalability, generalization, and adaptability across diverse deployment scenarios.

Scalability

The first experiment (Section 5.5.1) highlights the scalability of the proposed approach. While ILP-based methods such as Raptor and Raptor++ achieve optimal or near-optimal solutions on small instances, their performance degrades significantly as the network size increases due to the exponential growth in decision variables. In contrast, once trained, the GNN-DQN agent can be applied to larger and previously unseen topologies without retraining. Nevertheless, the computational overhead associated with training and inference

makes GNN-DQN less suitable for small-scale networks, where simpler methods remain more efficient.

Generalization and Robustness

The trained agent demonstrates strong generalization capabilities, successfully handling unseen graphs with varying sizes, densities, path configurations, and rule sets. It achieves a success rate of 95% with an average inference time of 0.14 s, indicating its robustness to variations in network structure and problem instances.

Adaptability and Flexibility

The adaptability of the proposed approach is illustrated through multiple experiments. In the example presented in Chapter 3 (Fig. 3.6), heuristic methods such as Palette and st-mincut converge to the same suboptimal solution due to their greedy, stepwise nature and lack of global reasoning. In contrast, the GNN-DQN agent identifies a placement that optimizes long-term objectives. This behavior is further confirmed in the second experiment 5.5.2 and the Mininet-based evaluation, where the limitations of heuristic approaches become evident under constrained conditions.

Furthermore, the third experiment (Section 5.5.3) demonstrates the flexibility of the GNN-DQN framework in handling multiple constraints simultaneously. Heuristic approaches require substantial reformulation for each new constraint, while ILP-based methods quickly become computationally intractable, achieving only 17% success in this setting. In contrast, GNN-DQN incorporates additional constraints seamlessly within its learning framework, maintaining a high success rate of 99%. This flexibility is particularly evident when considering the ordering constraint. Heuristic methods such as st-mincut and Palette are not readily adaptable to handle ordering dependencies and would require a complete reformulation of their underlying logic. Similarly, ILP-based approaches face significant scalability challenges, as jointly modeling placement and ordering leads to a quadratic increase in the number of constraints, rendering the problem intractable for realistic network sizes. In contrast, the GNN-DQN framework enables a straightforward integration of ordering constraints through state augmentation and action feasibility modeling, allowing for a joint optimization of placement and ordering within a unified learning process.

Placement with	Underutilized capacity $\lvert F\rvert = 0.5 \times min(P) \times m$			Near maximum capacity $\lvert F\rvert = min(P) \times m - 1$			Bandwidth constraint			Ordering constraint	
Graph parameters	size ≤ 15	density ≥ 0.5	size ≥ 20	size ≤ 15	density ≥ 0.5	size ≥ 20	size ≤ 15	density ≥ 0.5	size ≥ 20	Joint	Post-processing
Palette	+	+	+	+	-	-	-	-	-	-	+
st-mincut	+	+	+	+	-	+	-	-	-	-	+
Raptor	+	+	-	+	-	-	+	-	-	-	+
GNN-DQN	+	+	+	+	+	+	+	+	+	+	+

Table 5.10 – Summary table highlighting the strengths and weaknesses of each algorithm across various placement setups and constraints.

Table 5.10 summarizes the overall results. In conclusion, while ILP-based methods suffer from scalability limitations and heuristic approaches lack flexibility, the GNN-DQN

framework provides a scalable, generalizable, and robust solution for ACL rule placement in complex and dynamic network environments.

5.7 Summary

This chapter addressed the ACL rule placement problem under more realistic network conditions by restricting the placement to a predefined subset of routing paths rather than considering all possible source-to-destination paths. This formulation reflects practical networking scenarios where routing policies, such as shortest-path or traffic engineering strategies, inherently constrain the set of feasible paths. To capture additional real-world requirements, two critical constraints were introduced: the bandwidth wastage constraint, which enforces the early placement of filtering rules to avoid unnecessary resource consumption, and the ordering constraint, which ensures consistency in rule evaluation when dependencies exist between rules. To solve this extended problem, a GNN combined with DQN (GNN-DQN) framework was proposed, leveraging its ability to encode graph topology and learn adaptive placement policies. The state representation was enriched to incorporate both structural and constraint-related features, while the action space and reward function were carefully designed to enforce feasibility and guide the learning process.

The experimental results confirm both the effectiveness and robustness of the proposed approach. Through extensive evaluation, the GNN-DQN agent was compared against state-of-the-art methods, including Palette [25], Raptor [85], and Raptor++ [87], across a wide range of network configurations. The results indicate that, while heuristic and optimization-based approaches can achieve competitive performance in specific scenarios, particularly on small-scale or sparse graphs, they suffer from inherent limitations in terms of scalability and flexibility. In particular, these methods struggle to scale to large or dense topologies due to the rapid growth in computational complexity. Furthermore, they are typically designed for a fixed problem formulation, and any modification of the placement objective or the introduction of additional constraints requires substantial redesign. In contrast, the proposed GNN-DQN framework consistently achieves high success rates while maintaining efficient resource utilization, even in large-scale and complex network settings. Moreover, it demonstrates strong adaptability in handling multiple interacting constraints within a unified learning framework. This capability highlights the advantage of learning-based approaches in capturing complex dependencies that are difficult to model explicitly using traditional methods. Overall, these findings validate the potential of GNN-DQN as a scalable, flexible, and robust solution for ACL rule placement in practical networking environments.

Chapter 6

General Conclusion

Modern networks must enforce increasingly large and complex security policies under strict hardware constraints. Access Control Lists (ACLs) remain a central mechanism for traffic filtering and access control, but their deployment is limited by the finite capacity, cost, and energy consumption of TCAM-based switches. As discussed throughout this thesis, these limitations make centralized ACL deployment increasingly difficult and motivate more scalable enforcement strategies. While compression and eviction techniques can reduce memory pressure in specific settings, they do not fully eliminate the structural mismatch between growing rule sets and limited per-device memory. Therefore, this thesis focused on the split-and-distribution paradigm, in which an ACL is decomposed into smaller rule subsets and distributed across multiple network devices so that the original policy semantics are preserved along traffic paths.

However, distributing ACL rule subsets across a network introduces a challenging placement problem. Determining where each subset should be installed so that the distributed system preserves the semantics of the original centralized ACL is far from trivial. Traffic may traverse multiple possible paths, particularly in dynamic or load-balanced environments, requiring policy enforcement to remain effective regardless of the selected route. At the same time, switch capacities are limited and often heterogeneous, making placement decisions strongly interdependent. Furthermore, rule dependencies and ordering constraints must be preserved to guarantee policy correctness. Existing solutions do not address these challenges in a unified manner, and there remains a need for approaches that are simultaneously scalable, flexible, and capable of accommodating the diverse operational constraints encountered in real-world deployments.

6.1 Achievements

This thesis proposes three complementary contributions to the distributed ACL placement problem, each addressing one of the research gaps identified in Chapter 2. The contributions are presented in order of increasing generality and operational realism.

The state of the art surveyed in Chapter 2 revealed three main limitations in distributed ACL placement. First, most existing approaches assume static routing or a single predetermined path, leaving enforcement under routing uncertainty insufficiently ad-

dressed. Second, heuristic methods are highly problem-specific and require expert redesign when objectives or constraints change, while exact optimization formulations rapidly lose scalability as network size and constraint complexity increase. Third, learning-based solutions remain scarcely explored for this problem; to the best of our knowledge, this thesis is the first to combine RL and GNN for distributed ACL placement. Moreover, no existing work jointly addresses routing uncertainty, scalable placement, bandwidth efficiency, and rule ordering within a unified framework. These gaps motivate the three contributions developed in this thesis.

C1: Distributed ACL placement under routing uncertainty (Chapter 3).
Chapter 3 addresses the problem of distributed ACL placement under dynamic routing, where all feasible source-destination paths must remain protected despite possible route changes. The problem is formalized over source-destination directed acyclic graphs (st-DAGs), and three complementary placement strategies are introduced. First, an exact polynomial-time algorithm based on an st-mincut formulation provides an optimal solution for the single-rule case. Second, a greedy heuristic selects nodes according to their contribution to newly covered st-paths, offering a lightweight solution for the multi-rule case, particularly on sparse graphs. Third, reinforcement-learning approaches, including tabular QL, DQN, and action-masked DQN, are investigated to learn adaptive placement policies under memory constraints. Experiments on five real-world topologies from the Internet Topology Zoo show that the st-mincut approach consistently achieves the best placement quality in terms of memory occupancy among the evaluated methods. The results also show that action-masked DQN achieves occupancy values identical to those of the st-mincut approach on all evaluated instances, while reducing convergence time. The results also show that RL can find feasible placements in cases where the greedy heuristic and st-mincut fail, thereby highlighting the relevance of sequential decision-making for distributed ACL placement and the trade-off between optimality, efficiency, and adaptability.

C2: Topology-aware scalable placement via GNN-DQN (Chapter 4).
A key limitation identified at the end of Chapter 3 is that RL-based placement policies remain tied to the topology used during training: any structural change, such as a link failure, routing update, or switch addition, may invalidate the learned Q-function and require full retraining. Since such changes are inherent to operational networks, Chapter 4 addresses this limitation by integrating GNNs with DQN, allowing the agent to encode network structure into transferable latent node embeddings. The resulting GNN-DQN architecture learns placement policies that generalize across graph instances of different sizes and structures by exploiting permutation-invariant representations and relational information between nodes. A central architectural contribution is the explicit modeling of the joint rule-node action space. Unlike standard GNN-based RL approaches, where actions often correspond to selecting a single node, distributed ACL placement requires selecting a pair (v, f), where v is a candidate node and f is a rule set. Since this structured action space has cardinality $|V| \times |F|$, an augmented representation composed of $|F|$ copies of the original graph is constructed, enabling the Q-network to estimate action

values over the full joint space while preserving the message-passing structure. In addition, due to the absence of public benchmark datasets for st-DAGs, Chapter 4 introduces a synthetic graph generation tool with controlled structural properties calibrated from Internet Topology Zoo networks. This tool provides the experimental substrate for training and evaluation in Chapters 4 and 5. Experiments show that GNN-DQN maintains strong placement quality and scalability across varying network configurations, while capturing path dependencies and feasibility constraints that static methods cannot exploit. Finally, the framework is extended to distance-constrained placement through reward shaping alone, demonstrating its flexibility in incorporating new operational constraints without modifying the underlying architecture.

C3: Constraint-aware ACL placement over selected paths (Chapter 5). Chapters 3 and 4 considered the conservative setting in which placement must guarantee coverage over all possible source-destination paths. While robust to routing uncertainty, this requirement may induce unnecessary rule replication when routing follows predefined strategies, such as shortest-path or traffic-engineering policies. Therefore, Chapter 5 extends the placement framework to a more realistic setting where rule placement is restricted to a predefined subset of routing paths reflecting the effective routing behavior of the network. In this context, two additional operational constraints are incorporated: a bandwidth-wastage constraint, which promotes early filtering to reduce unnecessary resource consumption along traffic paths, and a rule-ordering constraint, which preserves semantic consistency when dependencies exist between rule subsets. The GNN-DQN framework is adapted to handle these interacting constraints jointly through enriched state representations, a modified action space, and reward shaping. Extensive experiments against Palette, Raptor, and Raptor++ on synthetic and real-world topologies show that the proposed approach achieves high placement success rates and efficient TCAM utilization. The results further confirm that heuristic and optimization-based methods remain competitive on small or sparse instances but degrade as graph size and constraint complexity increase, while requiring substantial redesign when objectives or constraints change. In contrast, GNN-DQN maintains adaptability and robust performance across the evaluated settings within a unified model.

6.2 Limitations

Although the contributions developed in this thesis provide scalable and constraint-aware solutions for distributed ACL placement, several limitations remain. These limitations mainly concern the network model, the learning formulation, and the separation between ACL decomposition and rule placement.

Single source-to-destination network model.
The network model adopted throughout this thesis represents the topology as a single-source, single-destination DAG. This formulation, while appropriate for studying the fundamental placement problem in a tractable setting, does not directly capture the multi-commodity nature of real networks, where multiple source-destination pairs coexist and

may share intermediate switches. When all pairs are governed by a common ACL policy, the model can be extended straightforwardly by introducing a virtual source and a virtual destination, connecting all sources and destinations to them respectively, and applying the proposed algorithms without modification. In the more general case, where each source-destination pair enforces its own distinct policy, a natural extension consists of solving the placement problem independently for each pair using the corresponding subset of routing paths. However, this iterative pair-wise approach does not account for memory contention on shared intermediate nodes: solving each pair's placement independently may saturate switches that lie on the paths of multiple pairs, leading to globally suboptimal or even infeasible solutions. The root cause of this limitation is that each placement decision is made locally, without global awareness of the aggregate resource demand imposed by the remaining pairs.

Restriction to off-policy RL and the action masking trade-off.

The RL framework developed in this thesis relies exclusively on DQN, an off-policy algorithm. This design choice is directly motivated by the structure of the action space: since the invalidity and redundancy of actions vary across graph instances, invalid and redundant actions are modeled as soft constraints through strict reward penalties rather than as hard exclusions. The objective of this formulation is to train the agent to learn, from the structural features of each graph instance, which actions are valid and which among them are optimal. This approach leads to a large action space in which valid actions represent only a small fraction of all possible decisions, resulting in higher computational cost during training and a slight deviation in solution quality compared to exact and heuristic methods, as observed in Chapter 5.

An alternative formulation, investigated in Chapter 3, consists of applying action masking: at each decision step, a dynamically computed validity function identifies all feasible actions and forces the agent to select exclusively among them. This mechanism avoids wasted exploration on infeasible regions of the action space and, as demonstrated in Chapter 3, recovers placement quality equal to that of the best algorithm while reducing computation time relative to the soft-constraint DQN formulation. However, action masking requires a handcrafted validity function that must be recomputed at every time step and tailored to the specific constraint structure of the problem, limiting its generalization across different problem settings. Beyond the masking question, the soft-constraint formulation has a more fundamental consequence: it restricts the framework to off-policy algorithms. DQN handles sparse and constrained action spaces efficiently by reusing past experience through its replay buffer, allowing the agent to learn from invalid action outcomes without repeatedly re-encountering them. In contrast, on-policy algorithms, such as PPO, sample actions from the current policy and discard collected trajectories after each update. In sparse action spaces, this leads to repeated exploration of invalid decisions, slow convergence, and potential policy collapse when action masking is not applied. Therefore, the current formulation prohibits the direct adoption of on-policy methods, which offer advantages in terms of policy stability and theoretical convergence guarantees.

ACL decomposition assumed to be given.

All contributions in this thesis presuppose that the original ACL has already been partitioned into a set of rule subsets before the placement stage begins. The decomposition problem, i.e., determining how to partition the ACL into subsets of appropriate size while preserving semantic correctness, minimizing inter-subset dependencies, and ensuring that the resulting subsets are compatible with the capacity constraints of the target network, is not addressed. In practice, the quality of the decomposition directly conditions the difficulty of the placement problem and the efficiency of the resulting distributed policy. A decomposition that produces overly large or strongly dependent subsets may render the placement problem infeasible or significantly degrade the occupancy achieved by otherwise effective placement strategies.

6.3 Future Work

The contributions and limitations of this thesis open several directions for future research, both for extending the proposed placement models and for addressing broader challenges in distributed ACL management.

A first important direction concerns the extension of the proposed framework to the multi-commodity placement problem. Such an extension would require coordinating placements over shared switches and limited TCAM resources, since decisions made for one pair may affect the feasibility and efficiency of others. Multi-agent RL constitutes a promising direction for this extension. Each agent could be associated with one source-destination pair, while a cooperative learning mechanism would encourage agents to coordinate their decisions and preserve global feasibility across shared network resources.

A second direction is to improve the way placement constraints are integrated into the learning process. The results obtained with action-masked DQN showed that explicitly restricting the agent to feasible actions can significantly improve exploration efficiency and solution quality. Extending this mechanism to the GNN-DQN framework would make it possible to study how action masking affects both convergence and generalization across different graph instances. A more systematic integration of action masking could also facilitate the use of alternative RL algorithms, such as actor-critic or policy-gradient methods, by preventing repeated exploration of invalid decisions and improving sample efficiency.

A third direction is to jointly address ACL decomposition and rule placement. The work presented in this thesis assumes that the initial ACL has already been decomposed into smaller rule subsets and focuses on deciding where these subsets should be installed. Future work could therefore investigate a joint optimization framework in which the original ACL is decomposed while taking into account the network topology, switch capacities, routing paths, rule dependencies, and placement objectives. Such an integrated approach could reduce unnecessary dependencies between rule subsets, improve placement feasibility, and further decrease TCAM occupancy. This would move the proposed framework toward a complete split-and-distribution pipeline, from the original ACL table to the final deployment over the network.

Finally, a broader research direction is to move toward autonomous ACL management

in self-driving networks. In many operational environments, the security policy implemented across the network is not always explicitly documented or expressed in a formal high-level language. Instead, it is often embedded in vendor-specific ACL configurations that have evolved over time and may have been inherited from previous operators. This makes policy understanding, verification, and update difficult. Future work could investigate the use of Large Language Models (LLMs) to extract the underlying security intents implemented by existing ACL tables. These inferred intents could then be used to reason about the current policy, detect inconsistencies, guide policy updates, and generate new ACL configurations. In a longer-term perspective, an autonomous ACL management system could observe network traffic, infer the security intents encoded in the current ACL, update the policy according to new requirements or threats, decompose the resulting ACL into rule subsets, and distribute them efficiently across the network. Such a framework would combine intent extraction, policy update, ACL decomposition, and distributed placement into a unified closed-loop system for scalable and adaptive network security management.

Bibliography

[1] Palo Alto Networks. 2025 report exposes widespread device security risks. [Online]. Available: https://www.paloaltonetworks.com/blog/network-security/2025-report-exposes-widespread-device-security-risks/. Accessed: 2026-04-28.

[2] Statista. Internet of Things (IoT) and non-IoT active device connections worldwide from 2010 to 2025. [Online]. Available: https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/. Accessed: 2026-05-06.

[3] IoT Analytics. Number of connected iot devices growing 14% to 21.1 billion. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/. Accessed: 2026-05-06.

[4] Flexera Cloud Report. Flexera 2023 state of the cloud report. [Online]. Available: https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/. Accessed: 2026-05-06.

[5] Packetlabs. Cybersecurity statistics to know for 2024. [Online]. Available: https://www.packetlabs.net/posts/the-top-cybersecurity-statistics-for-2024/. Accessed: 2026-05-06.

[6] Rob Sobers. 139 cybersecurity statistics and trends [updated 2025]. [Online]. Available: https://www.varonis.com/blog/cybersecurity-statistics/. Accessed: 2026-05-06.

[7] Check Point Research (2024). 300 cyber security statistics, facts & figures. [Online]. Available: https://www.keepnetlabs.com/blog/cyber-security-statistics-and-trends/. Accessed: 2026-05-06.

[8] Dustin Volz and Robert McMillan. Chinese and iranian hackers are using U.S. AI products to bolster cyberattacks. *The Wall Street Journal*, 2025. [Online]. Available: https://www.wsj.com/tech/ai/chinese-iranian-hackers-ai-cyberattacks. Accessed: 2026-05-06.

[9] Amirreza Niakanlahiji, Mir Mehedi Pritom, Bei-Tseng Chu, and Ehab Al-Shaer. Predicting zero-day malicious ip addresses. In *Proceedings of the 2017 Workshop on Automated Decision Making for Active Cyber Defense*, pages 1–6, 2017.

[10] MIT Sloan / Fortinet (2024). Cybersecurity statistics 2025: Rising threats and industry impact. [Online]. Available: https://www.fortinet.com/resources/cyberglossary/cybersecurity-statistics/. Accessed: 2026-05-06.

[11] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2002.

[12] Kostas Pagiamtzis and Ali Sheikholeslami. Content-addressable memory (cam) circuits and architectures: A tutorial and survey. *IEEE journal of solid-state circuits*, 41(3):712–727, 2006.

[13] Chad R Meiners, Alex X Liu, and Eric Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. *IEEE/ACM Transactions on Networking*, 2011.

[14] Banit Agrawal and Timothy Sherwood. Modeling TCAM power for next generation network devices. In *IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2006.

[15] Fang Yu, TV Lakshman, Martin Austin Motoyama, and Randy H Katz. Ssa: A power and memory efficient scheme to multi-match packet classification. In *Proceedings of the 2005 ACM Symposium on Architecture for Networking and Communications Systems*, 2005.

[16] Changhun Jung, Sian Kim, Rhongho Jang, David Mohaisen, and DaeHun Nyang. A scalable and dynamic acl system for in-network defense. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1679–1693, 2022.

[17] Cisco Systems. Validate Security ACLs on Catalyst 9000 Switches, 2022. [Online]. Available: `https://www.cisco.com/c/en/us/support/docs/switches/catalyst-9500-series-switches/217266-validate-security-acls-on-catalyst-9000.html`. Accessed: 2026-05-08.

[18] Cisco Systems. Migration to Silicon One Q200-Based Platforms: C9500X/C9600X, 2024. [Online]. Available: `https://www.ciscolive.com/c/dam/r/ciscolive/global-event/docs/2024/pdf/BRKARC-2096.pdf`. Accessed: 2026-05-08.

[19] NVIDIA. NVIDIA Spectrum SN4000 Series Switches Datasheet, 2022. [Online]. Available: `https://www.nvidia.com/content/dam/en-zz/Solutions/networking/br-sn4000-series.pdf`. Accessed: 2026-05-08.

[20] Hamed Khanmirza. Wildminnie: compression of software-defined networking (sdn) rules with wildcard patterns. *PeerJ Computer Science*, 8:e809, 2022.

[21] Abdulhadi Alahmadi and Tae Sun Chung. An effective selection of memory technologies for tcam to improve the search operations: Demonstration of memory efficiency in sdn recovery. *Electronics*, 13(4):707, 2024.

[22] Ankur Mudgal, Abhishek Verma, Munesh Singh, Kshira Sagar Sahoo, Erik Elmroth, and Monowar Bhuyan. Flora: Flow table low-rate overflow reconnaissance and detection in sdn. *IEEE Transactions on Network and Service Management*, 2024.

[23] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. Scalable flow-based networking with difane. *ACM SIGCOMM Computer Communication Review*, 40(4):351–362, 2010.

[24] Xuan-Nam Nguyen, Damien Saucez, Chadi Barakat, and Thierry Turletti. Rules placement problem in openflow networks: A survey. *IEEE Communications Surveys & Tutorials*, 2015.

[25] Yossi Kanizo, David Hay, and Isaac Keslassy. Palette: Distributing tables in software-defined networks. In *Proceedings IEEE INFOCOM*, 2013.

[26] Wafik Zahwa, Abdelkader Lahmadi, Michael Rusinowitch, and Mondher Ayadi. Deep reinforcement learning for in-network placement of acl rules under constraints. In *2025 21st International Conference on Network and Service Management (CNSM)*, pages 1–7. IEEE, 2025.

[27] Wafik Zahwa, Abdelkader Lahmadi, Michael Rusinowitch, and Mondher Ayadi. In-network ACL rules placement using deep reinforcement learning. In *2024 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*. IEEE, 2024.

[28] Wafik Zahwa, Abdelkader Lahmadi, Michael Rusinowitch, and Mondher Ayadi. Automated placement of in-network ACL rules. In *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, 2023.

[29] D Brent Chapman and Elizabeth D Zwicky. *Building internet firewalls*. O'Reilly & Associates, Inc., 1995.

[30] Ehab Al-Shaer. *Automated Firewall Analytics: Design, Configuration and Optimization*. Springer, 2014.

[31] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, 2014.

[32] Hazem Hamed and Ehab Al-Shaer. Taxonomy of conflicts in network security policies. *IEEE Communications Magazine*, 44(3):134–141, 2006.

[33] Ehab S Al-Shaer and Hazem H Hamed. Modeling and management of firewall policies. *IEEE Transactions on network and service management*, 1(1):2–10, 2008.

[34] Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and Prasant Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 15–pp. IEEE, 2006.

[35] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A {NICE} way to test {OpenFlow} applications. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 127–140, 2012.

[36] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2014.

[37] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[38] DatabaseTown. Basics of Reinforcement Learning (Algorithms, Applications & Advantages), 2023. [Online]. Available: `https://databasetown.com/basics-of-reinforcement-learning/`. Accessed: 2026-05-14.

[39] Richard S Sutton, Andrew G Barto, et al. Reinforcement learning. *Journal of Cognitive Neuroscience*, 11(1):126–134, 1999.

[40] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[41] James R Norris. *Markov chains*. Cambridge university press, 1998.

[42] Thomas M. Moerland, Joost Broekens, Aske Plaat, and Catholijn M. Jonker. Model-based reinforcement learning: A survey. *Foundations and Trends in Machine Learning*, 16(1):1–118, 2023.

[43] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[44] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

[45] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[46] Sun-Chong Wang. Artificial neural network. In *Interdisciplinary computing in java programming*. Springer, 2003.

[47] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[48] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[49] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI open*, 1:57–81, 2020.

[50] Lilapati Waikhom and Ripon Patgiri. Graph neural networks: Methods, applications, and opportunities. *arXiv preprint arXiv:2108.10733*, 2021.

[51] William L Hamilton. *Graph representation learning*. Morgan & Claypool Publishers, 2020.

[52] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[53] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[54] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[55] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*. PMLR, 2017.

[56] Ahmad Abboud. *Efficient Rules Management Algorithms in Software Defined Networking*. PhD thesis, Université de Lorraine, 2021.

[57] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the" one big switch" abstraction in software-defined networks. In *Proceedings of the ninth ACM Conference on Emerging Networking Experiments and Technologies*, 2013.

[58] Huan Liu. Efficient mapping of range classifier into ternary-cam. In *Proceedings 10th Symposium on High Performance Interconnects*. IEEE, 2002.

[59] Anat Bremler-Barr and Danny Hendler. Space-efficient tcam-based classification using gray coding. *IEEE Transactions on Computers*, 61(1):18–30, 2010.

[60] Chad R Meiners, Alex X Liu, and Eric Torng. Topological transformation approaches to optimizing tcam-based packet classification systems. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):73–84, 2009.

[61] Ahmad Abboud, Abdelkader Lahmadi, Michael Rusinowitch, Miguel Couceiro, Adel Bouhoulal, and Mondher Avadi. Double mask: An efficient rule encoding for software defined networking. In *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 186–193. IEEE, 2020.

[62] Qunfeng Dong, Suman Banerjee, Jia Wang, Dheeraj Agrawal, and Ashutosh Shukla. Packet classifiers in ternary cams can be smaller. In *Proceedings of the joint international conference on measurement and modeling of computer systems*, pages 311–322, 2006.

[63] Alex X Liu and Mohamed G Gouda. Complete redundancy removal for packet classifiers in tcams. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):424–437, 2008.

[64] Alex X Liu, Chad R Meiners, and Eric Torng. Tcam razor: A systematic approach towards minimizing packet classifiers in tcams. *IEEE/ACM Transactions on Networking*, 18(2):490–500, 2009.

[65] David A Applegate, Gruia Calinescu, David S Johnson, Howard Karloff, Katrina Ligett, and Jia Wang. Compressing rectilinear pictures and minimizing access control lists. In *SODA*, volume 7, pages 1066–1075, 2007.

[66] Myriana Rifai, Nicolas Huin, Christelle Caillouet, Frédéric Giroire, D Lopez-Pacheco, Joanna Moulierac, and Guillaume Urvoy-Keller. Too many sdn rules? compress them with minnie. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2015.

[67] Shouxi Luo, Hongfang Yu, et al. Fast incremental flow table aggregation in sdn. In *2014 23rd international conference on computer communication and networks (ICCCN)*, pages 1–8. IEEE, 2014.

[68] Wolfgang Braun and Michael Menth. Wildcard compression of inter-domain routing tables for openflow-based software-defined networking. In *2014 Third european workshop on software defined networks*, pages 25–30. IEEE, 2014.

[69] Anilkumar Vishnoi, Rishabh Poddar, Vijay Mann, and Suparna Bhattacharya. Effective switch memory management in openflow networks. In *Proceedings of the 8th ACM international conference on distributed event-based systems*, pages 177–188, 2014.

[70] Eun-Do Kim, Seung-Ik Lee, Yunchul Choi, Myung-Ki Shin, and Hyoung-Jun Kim. A flow entry management scheme for reducing controller overhead. In *16th International Conference on Advanced Communication Technology*, pages 754–757. IEEE, 2014.

[71] Kalapriya Kannan and Subhasis Banerjee. Flowmaster: Early eviction of dead flow on sdn switches. In *International Conference on Distributed Computing and Networking*, pages 484–498. Springer, 2014.

[72] Liang Xie, Zhifeng Zhao, Yifan Zhou, Gang Wang, Qianlan Ying, and Honggang Zhang. An adaptive scheme for data forwarding in software defined network. In *2014 Sixth International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–5. IEEE, 2014.

[73] Huikang Zhu, Hongbo Fan, Xuan Luo, and Yaohui Jin. Intelligent timeout master: Dynamic timeout for sdn-based data centers. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 734–737. IEEE, 2015.

[74] Xin Wang, Cheng Wang, Changjun Jiang, Lei Yang, Zhong Li, and Xiaobo Zhou. Rule optimization for real-time query service in software-defined internet of vehicles. *arXiv preprint arXiv:1503.05646*, 2015.

[75] Manuel Jiménez-Lázaro, Javier Berrocal, and Jaime Galán-Jiménez. Deep reinforcement learning based method for the rule placement problem in software-defined networks. In *NOMS, IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2022.

[76] Babangida Isyaku, Mohd Soperi Mohd Zahid, Maznah Bte Kamat, Kamalrulnizam Abu Bakar, and Fuad A Ghaleb. Software defined networking flow table management of openflow switches performance and security challenges: A survey. *Future Internet*, 12(9):147, 2020.

[77] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. Simple-fying middlebox policy enforcement using sdn. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 27–38, 2013.

[78] Seyed Kaveh Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 19–24, 2013.

[79] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[80] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. *ACM SIGCOMM computer communication review*, 37(4):1–12, 2007.

[81] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

[82] Md Faizul Bari, Arup Raton Roy, Shihabur Rahman Chowdhury, Qi Zhang, Mohamed Faten Zhani, Reaz Ahmed, and Raouf Boutaba. Dynamic controller provisioning in software defined networks. In *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*, pages 18–25. IEEE, 2013.

[83] Ahmad Abboud, Rémi Garcia, Abdelkader Lahmadi, Michaël Rusinowitch, and Adel Bouhoula. Efficient distribution of security policy filtering rules in software defined networks. In *Proceedings IEEE NCA*, 2020.

[84] Shuyuan Zhang, Franjo Ivancic, Cristian Lumezanu, Yifei Yuan, Aarti Gupta, and Sharad Malik. An adaptable rule placement for software-defined networks. In *2014 44th annual IEEE/IFIP international conference on dependable systems and networks*. IEEE, 2014.

[85] Pravein Govindan Kannan, Mun Choon Chan, Richard TB Ma, and Ee-Chien Chang. Raptor: Scalable rule placement over multiple path in software defined networks. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, 2017.

[86] Frédéric Giroire, Joanna Moulierac, and Truong Khoa Phan. Optimizing rule placement in software-defined networks for energy-aware routing. In *2014 IEEE Global Communications Conference*, pages 2523–2529. IEEE, 2014.

[87] Yu-Wei Chang and Tsung-Nan Lin. An efficient dynamic rule placement for distributed firewall in SDN. In *IEEE Global Communications Conference*. IEEE, 2020.

[88] Xuan-Nam Nguyen, Damien Saucez, Chadi Barakat, and Thierry Turletti. Officer: A general optimization framework for openflow rule allocation and endpoint policy enforcement. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 478–486. IEEE, 2015.

[89] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 254–265, 2011.

[90] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. {vCRIB}: Virtualized rule management in the cloud. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, 2012.

[91] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Infinite cacheflow in software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 175–180, 2014.

[92] He Li, Peng Li, and Song Guo. Morule: Optimized rule placement for mobile users in sdn-enabled access networks. In *2014 IEEE global communications conference*, pages 4953–4958. IEEE, 2014.

[93] Jen-Feng Huang, Guey-Yun Chang, Chun-Feng Wang, and Chih-Hao Lin. Heterogeneous flow table distribution in software-defined networks. *IEEE Transactions on Emerging Topics in Computing*, 4(2):252–261, 2015.

[94] Angelos Mimidis-Kentis, Artur Pilimon, Jose Soler, Michael Berger, and Sarah Ruepp. A novel algorithm for flow-rule placement in sdn switches. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 1–9. IEEE, 2018.

[95] Wenjie Li, Zheng Qin, Keqin Li, Hui Yin, and Lu Ou. A novel approach to rule placement in software-defined networks based on optree. *IEEE Access*, 7:8689–8700, 2019.

[96] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. *ACM SIGCOMM Computer Communication Review*, 42(4):473–478, 2012.

[97] Petra Vizarreta, Carmen Mas Machuca, and Wolfgang Kellerer. Controller placement strategies for a resilient sdn control plane. In *2016 8th international workshop on resilient networks design and modeling (RNDM)*, pages 253–259. IEEE, 2016.

[98] Stuart Clayman, Lefteris Mamatas, and Alex Galis. Experimenting with control operations in software-defined infrastructures. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 390–396. IEEE, 2016.

[99] Faizul Bari, Shihabur Rahman Chowdhury, Reaz Ahmed, Raouf Boutaba, and Otto Carlos Muniz Bandeira Duarte. Orchestrating virtualized network functions. *IEEE Transactions on Network and Service Management*, 13(4):725–739, 2016.

[100] Hongyi Huang and Wenfei Wu. Hypersfp: Fault-tolerant service function chain provision on programmable switches in data centers. In *NOMS*, volume 2022, pages 1–9, 2022.

[101] Lei Wang, Qing Li, Richard Sinnott, Yong Jiang, and Jianping Wu. An intelligent rule management scheme for software defined networking. *Computer Networks*, 144:77–88, 2018.

[102] Huawei Huang, Peng Li, Song Guo, and Baoliu Ye. The joint optimization of rules allocation and traffic engineering in software defined network. In *2014 IEEE 22nd International Symposium of Quality of Service (IWQoS)*, pages 141–146. IEEE, 2014.

[103] Sibananda Behera, Namita Panda, Sudhansu Shekhar Patra, Tabrej Khan, and Lalbihari Barik. Flow-rule placement in software-defined networks for energy-aware routing. In *2025 International Conference on Sustainable Communication Networks and Application (ICSCN)*, pages 367–373. IEEE, 2025.

[104] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Fulvio Valenza, and Jalolliddin Yusupov. Automated firewall configuration in virtual networks. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[105] Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. Automating the configuration of firewalls and channel protection systems in virtual networks. In *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, pages 474–479. IEEE, 2023.

[106] Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. A demonstration of verefoo: an automated framework for virtual firewall configuration. In *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, pages 293–295. IEEE, 2023.

[107] Yingya Guo, Huan Luo, Zhiliang Wang, Xia Yin, and Jianping Wu. Routing optimization with path cardinality constraints in a hybrid sdn. *Computer Communications*, 165:112–121, 2021.

[108] Shizhong Xu, Xiong Wang, Guangxu Yang, Jing Ren, and Sheng Wang. Routing optimization for cloud services in sdn-based internet of things with tcam capacity constraint. *Journal of Communications and Networks*, 22(2):145–158, 2020.

[109] Usman Ashraf, Adnan Ahmed, Stefano Avallone, and Pasquale Imputato. Traffic evolution in software defined networks. *Computer Networks*, 255:110852, 2024.

[110] Rashid Amin, Elisa Rojas, Aqsa Aqdus, Sadia Ramzan, David Casillas-Perez, and Jose M Arco. A survey on machine learning techniques for routing optimization in sdn. *IEEE Access*, 9:104582–104611, 2021.

[111] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 2015.

[112] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[113] Natalia Vesselinova, Rebecca Steinert, Daniel F Perez-Ramirez, and Magnus Boman. Learning combinatorial optimization on graphs: A survey with applications to networking. *IEEE Access*, 2020.

[114] Krzysztof Rusek, José Suárez-Varela, Paul Almasan, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Routenet: Leveraging graph neural networks for network modeling and optimization in sdn. *IEEE Journal on Selected Areas in Communications*, 2020.

[115] Krzysztof Rusek, José Suárez-Varela, Albert Mestres, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Unveiling the potential of graph neural networks for network modeling and optimization in sdn. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 140–151, 2019.

[116] Haifeng Sun, Xingjian Liao, Jingyu Wang, Qi Qi, Zirui Zhuang, Jianxin Liao, and Dapeng Oliver Wu. Fast and scalable acl policy solving under complex constraints with graph neural networks. *IEEE/ACM Transactions on Networking*, 32(5):4175–4190, 2024.

[117] Weiwei Jiang, Haoyu Han, Yang Zhang, Ji'an Wang, Miao He, Weixi Gu, Jianbin Mu, and Xirong Cheng. Graph neural networks for routing optimization: Challenges and opportunities. *Sustainability*, 16(21):9239, 2024.

[118] Sahar Faezi and Alireza Shirmarz. A comprehensive survey on machine learning using in software defined networks (sdn). *Human-Centric Intelligent Systems*, 3(3):312–343, 2023.

[119] Albert Mestres, Alberto Rodriguez-Natal, Josep Carner, Pere Barlet-Ros, Eduard Alarcón, Marc Solé, Victor Muntés-Mulero, David Meyer, Sharon Barkai, Mike J Hibbett, et al. Knowledge-defined networking. *ACM SIGCOMM Computer Communication Review*, 47(3):2–10, 2017.

[120] Zhiyuan Xu, Jian Tang, Jingsong Meng, Weiyi Zhang, Yanzhi Wang, Chi Harold Liu, and Dejun Yang. Experience-driven networking: A deep reinforcement learning based approach. In *IEEE INFOCOM 2018-IEEE conference on computer communications*, pages 1871–1879. IEEE, 2018.

[121] Zili Meng, Minhu Wang, Jiasong Bai, Mingwei Xu, Hongzi Mao, and Hongxin Hu. Interpreting deep learning-based networking systems. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 154–171, 2020.

[122] Tri Gia Nguyen, Trung V Phan, Dinh Thai Hoang, Hai Hoang Nguyen, and Duc Tran Le. Deepplace: Deep reinforcement learning for adaptive flow rule placement in software-defined iot networks. *Computer communications*, 181:156–163, 2022.

[123] Gan Huang, Ihsan Ullah, Hanyao Huang, and Kyung Tae Kim. Predictive mobility and cost-aware flow placement in sdn-based iot networks: a q-learning approach. *Journal of Cloud Computing*, 13(1):26, 2024.

[124] Xingjian Liao, Haifeng Sun, Jingyu Wang, Qi Qi, Zirui Zhuang, Jianxin Liao, and Guang Yang. Solving distributed acl policies under complex constraints with graph neural networks. In *2023 IEEE 31st International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2023.

[125] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[126] Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

[127] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in Neural Information Processing Systems*, 2017.

[128] Thomas Barrett, William Clements, Jakob Foerster, and Alex Lvovsky. Exploratory combinatorial optimization with reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020.

[129] Lester R Ford Jr and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8:399–404, 1956.

[130] Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

[131] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 2011.

[132] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *International conference on machine learning*, pages 2014–2023. PMLR, 2016.

[133] Jiawei Zhang. Segmented graph-bert for graph instance modeling. *arXiv preprint arXiv:2002.03283*, 2020.

[134] Kiarash Zahirnia, Oliver Schulte, Parmis Naddaf, and Ke Li. Micro and macro level graph modeling for graph variational auto-encoders. *Advances in Neural Information Processing Systems*, 35:30347–30361, 2022.

[135] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[136] Donald K. Wagner. Disjoint $(s,\ t)$-cuts in a network. *Networks*, 1990.

List of Figures

1.1 A network topology for a placement scenario with 2 rules f_0 and f_1 to be placed. . . . 4
1.2 Overview of the thesis organization, illustrating the relationships between research questions, proposed contributions, and corresponding chapters. . . 7

2.1 Illustration of the TCAM matching process for ACL rule lookup. . . . 12
2.2 Illustrative example of reinforcement learning, in which an agent improves its behavior through interaction with the environment and evaluative reward feedback [38]. . . . 13
2.3 Overview of the main RL elements: agent, environment, states, actions, and rewards. . . . 14
2.4 Message-passing mechanism in an MPNN for updating a target node representation. . . . 22
2.5 Example of ACL rule compression through prefix aggregation. . . . 24
2.6 Example of eviction-based ACL management. . . . 26
2.7 Example of split-and-distribution based ACL placement 28

3.1 Placement of a set of rules in a network 48
3.2 Steps of the st-mincut-based Algorithm 3 for rule placement 50
3.3 Steps of applying the greedy heuristic-based algorithm for rule placement . 52
3.4 Q-learning pipeline for ACL rule placement. . . . 55
3.5 Deep Q-learning pipeline for ACL rule placement. . . . 56
3.6 Comparison between RL and non-RL based placements, illustrating the long-term optimality of RL decisions. The example considers two rule sets, f_1 and f_2, with unit switch capacities. Orange nodes denote the greedy placement of f_1, while green nodes denote the subsequent placement of f_2 needed to cover all paths. This illustrates how an initially greedy choice may increase final memory occupancy. . . . 61
3.7 Counterexample graph illustrating the failure of local sequential placement decisions. . . . 62

4.1 GNN-DQN architecture dedicated to rule placement. . . . 70
4.2 Learning curves of the GNN-DQN agent showing the evolution of the average episode return during training and validation for different graph sizes ($n = 10, 15, 20, 25$). . . . 77

4.3 Learning curves of the GNN-DQN agent showing the evolution of the average episode length during training and validation for different graph sizes ($n = 10, 15, 20, 25$). 78
4.4 Illustration of a source-oriented distance constraint for early filtering. . . . 82
4.5 Illustration of a destination-oriented distance constraint for service-specific protection. 83
4.6 Distribution of placement distances for destination-labeled and source-labeled rules before and after introducing the distance constraint. The first row corresponds to destination-oriented rules, while the second row corresponds to source-oriented rules. 85

5.1 A network topology for a placement scenario with 2 rules f_0 and f_1 to be placed on 3 paths: Path 1, Path 2, and Path 3. 91
5.2 Example of a simplified ACL table divided into two subtables, f_0 and f_1, according to the action associated with each rule. 92
5.3 Occupancy distribution of the four approaches across four graph sizes with varying cv. 104
5.4 Success rates for Raptor and GNN-DQN algorithms over 500 graphs of size 25 while increasing the coverage percentage cv. 105
5.5 Success rate comparison of GNN-DQN and Palette with $|F| = path_capacity(min(P))-1$ and different cv. 107
5.6 Variation of average occupancy and average distance with α, illustrating the trade-off between occupancy (left y-axis) and distance (right y-axis) for graph size 15. 110
5.7 Success rate comparison of GNN-DQN and Raptor++ for the placement problem with the bandwidth constraint for different cv. 110

A.1 Topologie réseau illustrant un scénario de placement avec deux règles f_0 et f_1 à placer. 145

List of Tables

2.1 Example of an ACL rule table. 11
2.2 Comparative summary of the three main families of approaches to ACL memory optimisation in TCAM-based networks. 30
2.3 Comparative synthesis of heuristic, graph-based, and architectural approaches related to distributed ACL rule placement. 34
2.4 Comparative synthesis of exact optimization approaches for distributed ACL rule placement and related rule-allocation problems. 37
2.5 Comparative synthesis of machine-learning-based and graph-learning approaches related to distributed ACL rule placement. 40
2.6 Comparative gap analysis of distributed ACL placement approaches across the main research dimensions addressed in this thesis. 44

3.1 Notations for problem formulation 48
3.2 Memory occupancy and processing time of our rules placement algorithms on real-world network topologies. 60

4.1 Memory occupancy and processing time of DQN on 500 test graphs for each graph size. 79
4.2 Testing a trained model with fixed graph size 25 and four rule sets on graphs with the same size but a variable number of rule sets. 79
4.3 Testing trained models with fixed graph size and number of rule sets on larger graphs with a variable number of rule sets. 80
4.4 Memory occupancy and processing time of DQN on real-world network topologies using a model trained on graphs of size 20 with four rule sets. . 81

5.1 Notations for problem formulation 91
5.2 Average occupancy of the four algorithms on 30 graphs of each graph size with $cv = 25\%$. 106
5.3 Average occupancy of the four algorithms with $|F| = path_capacity(\min(P)) - 1$ and $cv = 25\%$. 106
5.4 Average occupancy of the four algorithms with $|F| = path_capacity(\min(P)) - 1$ and $cv = 50\%$. 107
5.5 Performance of GNN-DQN and Raptor++ for the placement problem with bandwidth constraint and $cv = 25\%$. 108

5.6 Performance of GNN-DQN and Raptor++ for the placement problem with bandwidth constraint and $cv = 50\%$. 109
5.7 Occupancy of the four approaches on real-world topologies with $|F| = 0.5 \times path_capacity(\min(P))$ for different values of cv. 111
5.8 Occupancy of the four approaches on real-world topologies with $|F| = path_capacity(\min(P)) - 1$ for different values of cv. 111
5.9 Performance of GNN-DQN and Raptor++ for the placement problem on real-world topologies with bandwidth constraint and different cv. 112
5.10 Summary table highlighting the strengths and weaknesses of each algorithm across various placement setups and constraints. 114

Acronyms

ACL	Access Control List
CAM	Content Addressable Memory
CBD	Cut-Based Decomposition
CIDR	Classless Inter-Domain Routing
CNN	Convolutional Neural Network
DDoS	Distributed Denial of Service
DL	Deep Learning
DoS	Denial of Service
DQN	Deep Q-Network
DRAM	Dynamic Random Access Memory
DRL	Deep Reinforcement Learning
ECMP	Equal-Cost Multi-Path
GAT	Graph Attention Network
GCN	Graph Convolutional Network
GIN	Graph Isomorphism Network
GNN	Graph Neural Network
HTTPS	HyperText Transfer Protocol Secure
ICMP	Internet Control Message Protocol
ILP	Integer Linear Programming
IoT	Internet of Things
IP	Internet Protocol
LLM	Large Language Model
MDP	Markov Decision Process
MILP	Mixed-Integer Linear Programming
MINLP	Mixed-Integer Nonlinear Programming
ML	Machine Learning
MPNN	Message Passing Neural Network
NN	Neural Network
PBD	Pivot Bit Decomposition
PPO	Proximal Policy Optimization
QL	Q-learning
QoS	Quality of Service
RAM	Random Access Memory
RL	Reinforcement Learning

RNN	Recurrent Neural Network
SAT	Boolean Satisfiability
SDN	Software-Defined Networking
SFC	Service Function Chain
SMT	Satisfiability Modulo Theories
SRAM	Static Random Access Memory
SRGE	Short Range Gray Encoding
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VLAN	Virtual Local Area Network

Appendix A

Résumé de la thèse en français

A.1 Contexte

Les réseaux actuels transportent une quantité considérable, et en constante augmentation, de trafics de nature très diverse. Les hôpitaux échangent par exemple des données médicales sensibles avec des médecins à distance, les usines connectent des milliers de capteurs à des systèmes de contrôle centralisés, les banques traitent chaque seconde des millions de transactions, et les plateformes vidéo diffusent des contenus haute définition à des millions d'utilisateurs dans le monde. Cette diversité n'est pas fortuite: elle reflète l'évolution profonde des réseaux au cours des deux dernières décennies, aussi bien dans leur architecture que dans leurs usages. Cette complexité se manifeste également au sein des réseaux d'entreprise. Un réseau d'entreprise typique peut aujourd'hui connecter environ 35 000 équipements appartenant à près de 80 catégories différentes, dont près d'un tiers fonctionne en dehors du contrôle direct des équipes informatiques [1]. Ces équipements non administrés incluent notamment des objets connectés, tels que des téléviseurs intelligents et des capteurs, ainsi que des ordinateurs portables ou smartphones personnels utilisés par les employés. Les réseaux deviennent ainsi de plus en plus hétérogènes, ce qui rend leur contrôle et leur sécurisation plus difficiles. Chacun de ces services impose des exigences de communication spécifiques, qui doivent être appliquées au sein du réseau. Les données sensibles ne doivent être accessibles qu'aux utilisateurs autorisés, les systèmes industriels doivent rester isolés du trafic externe, les opérations financières nécessitent une protection stricte contre les accès non autorisés, et les applications multimédias doivent bénéficier d'un traitement efficace du trafic sans compromettre la sécurité. Ces exigences, appelées *politiques de sécurité*, définissent la manière dont le trafic réseau doit être contrôlé et protégé. À mesure que les réseaux continuent de croître, ces politiques deviennent plus volumineuses et plus complexes, ce qui rend leur gestion et leur application de plus en plus difficiles. Comprendre cette complexité croissante et ses conséquences constitue le point de départ de cette thèse.

En pratique, les politiques de sécurité sont exprimées sous la forme d'exigences de haut niveau décrivant la manière dont le trafic réseau doit être contrôlé. Pour être appliquées par les équipements réseau, ces politiques doivent être traduites en règles ou configurations concrètes, le plus souvent sous la forme des listes de contrôle d'accès (ACLs). Une

ACL peut être comprise comme une liste d'instructions placée au point d'entrée d'un équipement réseau, tel qu'un routeur ou un commutateur, afin de déterminer la manière dont les paquets entrants doivent être traités. Chaque instruction, appelée règle, définit une condition, par exemple le fait qu'un paquet provienne d'une adresse IP donnée et cible un service ou un port spécifique, ainsi qu'une action associée, telle que l'autorisation du paquet, son blocage ou l'application d'un traitement particulier.

La taille des ACLs a fortement augmenté sous l'effet de deux facteurs principaux. Le premier est l'expansion rapide et la diversification des équipements connectés, qui nécessitent des politiques de sécurité de plus en plus fines. En 2021, environ 13,8 milliards d'équipements étaient connectés à Internet dans le monde [2], un nombre qui devrait atteindre 39 milliards d'ici 2030 [3]. Ces équipements incluent des smartphones, des capteurs industriels, des dispositifs médicaux, des véhicules et des appareils intelligents, chacun présentant des schémas de communication et des besoins de sécurité spécifiques. Chaque catégorie d'équipements nécessite donc des règles dédiées indiquant quels systèmes peuvent être contactés et dans quelles conditions, ce qui augmente directement la taille et la complexité des ACLs. Cette tendance est encore renforcée par le cloud computing et les environnements multi-locataires, dans lesquels plusieurs clients, ou tenants, partagent une même infrastructure physique tout en exécutant des applications et services indépendants. Afin de garantir la sécurité et la confidentialité, chaque tenant doit être strictement isolé des autres. Son trafic doit donc être contrôlé et séparé au moyen de règles ACL spécifiques. Ainsi, lorsque le nombre de tenants augmente, le nombre de règles nécessaires croît également de manière significative [4].

Le second facteur est l'évolution permanente de l'environnement des cybermenaces, qui entraîne une expansion continue des politiques ACL. Les règles de sécurité doivent être régulièrement mises à jour afin de répondre à de nouvelles menaces, vulnérabilités et activités malveillantes. Les attaques actuelles sont variées et de plus en plus sophistiquées. Par exemple, les attaques par rançongiciel (ransomware), dans lesquelles les attaquants bloquent l'accès aux données et exigent une rançon pour le rétablir, ont touché plus de 72% des organisations dans le monde en 2023 [5]. Les attaques par déni de service distribué, ou DDoS, qui consistent à submerger un système par un volume massif de trafic afin de le rendre indisponible, ont augmenté de 46% au cours du premier semestre 2024 par rapport à 2023 [6]. Plus largement, le nombre de cyberattaques a augmenté de 30% en 2024, atteignant en moyenne 1,636 attaques hebdomadaires par organisation [7]. Les attaquants exploitent également des vulnérabilités logicielles, parfois à l'aide d'outils d'intelligence artificielle [8], et recourent à des techniques telles que l'envoi de courriels trompeurs contenant du contenu malveillant, ou phishing, la compromission de partenaires externes de confiance, ou attaques de la chaîne d'approvisionnement, ainsi que le vol d'identifiants légitimes. Une difficulté majeure réside dans le fait que de nombreuses attaques proviennent d'adresses IP qui ne sont identifiées comme malveillantes qu'après leur observation. De nouvelles règles de filtrage doivent donc être ajoutées en continu. Plus de 5 000 nouvelles adresses IP malveillantes sont découvertes chaque jour [9], et certains ensembles de règles contiennent déjà plus de 250 000 entrées dédiées au blocage de menaces connues. En parallèle, les exigences réglementaires contribuent également à cette croissance. Plus de 170 lois relatives à la protection des données ont été introduites dans le monde en 2023-2024 [10], chacune imposant des contraintes sur la manière dont les données sensi-

bles doivent être traitées. La traduction de ces exigences en règles ACL applicables ajoute ainsi une couche supplémentaire de complexité aux politiques de sécurité réseau.

À ce stade, une question naturelle se pose : si les réseaux nécessitent davantage de règles, pourquoi ne pas simplement les stocker toutes sur chaque équipement réseau ? La réponse tient au matériel utilisé pour appliquer les ACLs. Les équipements réseau doivent inspecter chaque paquet et déterminer, en une fraction de seconde, quelle règle doit s'appliquer. Aux débits actuels, où des millions de paquets sont traités chaque seconde, les règles ne peuvent pas être vérifiées séquentiellement. Les équipements réseau s'appuient donc sur la Ternary Content Addressable Memory (TCAM) [11, 12], une mémoire spécialisée capable de comparer un paquet à l'ensemble des règles simultanément, en parallèle et en une seule opération. La TCAM prend en charge des valeurs ternaires, à savoir 0, 1 et '*' pour représenter une valeur indifférente, ce qui en fait une technologie standard, et quasi incontournable, pour l'application des ACLs dans les routeurs et commutateurs modernes.

Cependant, ces performances ont un coût. La TCAM est beaucoup plus coûteuse à fabriquer que les mémoires traditionnelles, consomme une quantité importante d'énergie, pouvant atteindre jusqu'à 60% de la consommation totale d'un commutateur, et occupe une surface importante sur la puce [13, 14, 15]. Par conséquent, sa capacité reste intrinsèquement limitée. En pratique, alors que la taille des ACLs pouvait déjà atteindre 100 000 règles dès 2022 [16], ce nombre devrait continuer à croître rapidement dans les prochaines années en raison de l'expansion continue des équipements connectés, des services cloud et des exigences de sécurité discutées précédemment. À l'inverse, les plateformes de commutation modernes montrent clairement que l'application des ACLs reste contrainte par des ressources TCAM limitées. Par exemple, les commutateurs de la série Cisco Catalyst 9000 prennent en charge entre environ 1 000 et 18 000 entrées ACL IPv4, selon le modèle considéré [17]. La capacité effective est encore plus faible pour IPv6, car chaque règle peut consommer plusieurs entrées TCAM, généralement deux, en raison de son expansion au niveau matériel. Des contraintes similaires sont observées dans des architectures plus récentes : les commutateurs basés sur Cisco Silicon One Q200 offrent environ 8K entrées ACL IPv4 ou 4K entrées ACL IPv6 [18]. Même si certaines plateformes modernes à hautes performances, telles que NVIDIA Spectrum SN4000 [19], annoncent des capacités TCAM partagées ou externes beaucoup plus importantes, ces ressources restent finies et sont souvent partagées entre les ACLs, les routes, les entrées MAC, les tunnels et d'autres fonctions de commutation.

Ainsi, malgré les progrès matériels, la capacité limitée de la TCAM reste une contrainte majeure pour le déploiement des ACLs. La croissance continue des ensembles de règles ACL, combinée à des ressources matérielles limitées et partagées, crée un défi persistant de passage à l'échelle pour les opérateurs réseau. Cette limitation est régulièrement soulignée dans des travaux récents [20, 21, 22] et motive directement la recherche d'approches plus efficaces pour la gestion des ACLs. Cette thèse s'inscrit précisément dans cette problématique.

A.2 Problématique

La complexité croissante des politiques ACL, combinée à la capacité limitée et partagée des TCAMs, soulève un défi fondamental de passage à l'échelle. En pratique, il devient souvent impossible de stocker l'ensemble d'une table ACL sur un seul commutateur tout en garantissant un traitement des paquets à la fois correct et efficace. Par conséquent, assurer l'application correcte des politiques de sécurité à l'échelle du réseau devient de plus en plus difficile, en particulier lorsque les infrastructures s'étendent et que les contraintes opérationnelles se multiplient. Parmi les différentes approches proposées dans la littérature, le paradigme de découpage et de distribution [23, 24, 25] constitue l'une des directions les plus pertinentes sur le plan architectural. Son principe consiste à décomposer l'ACL initiale en sous-ensembles de règles plus petits, puis à les installer sur plusieurs commutateurs, de manière à ce que tout paquet traversant le réseau rencontre, le long de son chemin, l'ensemble des règles qui lui sont applicables. Cette approche permet de contourner les limitations de capacité des TCAMs en répartissant la charge de filtrage à travers le réseau, sans recourir à une approximation de la politique ni à un remplacement réactif des règles.

Cependant, l'adoption de ce paradigme introduit un nouveau problème, à la fois central et non trivial : déterminer où placer chaque sous-ensemble de règles afin que le système distribué applique collectivement la même politique que l'ACL centralisée initiale. Ce problème est particulièrement difficile pour plusieurs raisons. Premièrement, le trafic entre une source et une destination peut suivre plusieurs chemins possibles, notamment dans les réseaux dynamiques ou utilisant des mécanismes d'équilibrage de charge, ce qui impose de garantir l'existence d'au moins un point de filtrage valide sur chaque chemin. Deuxièmement, les capacités mémoire des commutateurs sont limitées et hétérogènes, ce qui rend les décisions de placement interdépendantes. Troisièmement, les dépendances entre règles ainsi que leurs contraintes d'ordre doivent être préservées afin de garantir la correction sémantique de la politique. Les solutions existantes ne traitent pas conjointement l'ensemble de ces défis : les approches d'optimisation exacte deviennent coûteuses sur le plan computationnel lorsque la taille du réseau et le nombre de contraintes augmentent, tandis que les approches heuristiques sont souvent spécifiques à un problème donné et s'adaptent difficilement à l'évolution des contraintes.

Répondre à ces défis nécessite donc de concevoir des stratégies de placement à la fois scalables et adaptatives, capables d'aller au-delà des formulations exactes classiques. L'objectif de cette thèse est ainsi d'aborder le problème de gestion des ACLs en s'appuyant sur des capacités TCAM réduites, en découpant la table ACL initiale en sous-ensembles de règles plus petits, puis en distribuant ces sous-ensembles dans le réseau de manière à préserver la même fonctionnalité de contrôle entre les extrémités communicantes. Formellement, le problème étudié dans cette thèse peut être formulé comme suit : étant donné une topologie réseau, un ensemble de chemins de routage et une ACL décomposée, comment assigner les sous-ensembles de règles aux commutateurs de sorte que : (i) la sémantique de l'ACL soit préservée, c'est-à-dire que tous les paquets soient correctement filtrés conformément à la politique initiale ; (ii) les contraintes mémoire des commutateurs soient respectées ; et (iii) des contraintes opérationnelles supplémentaires, telles que la variabilité du routage, les limitations de bande passante ou les dépendances entre règles, soient

satisfaites ?

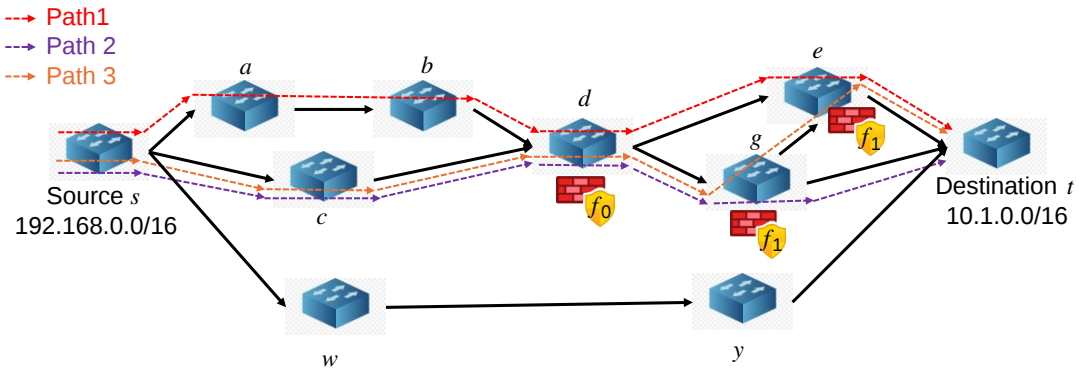


Figure A.1 – Topologie réseau illustrant un scénario de placement avec deux règles f_0 et f_1 à placer.

Considérons le réseau illustré dans la Figure A.1, composé d'un noeud source s, d'un noeud destination t, de commutateurs intermédiaires $a - g$, ainsi que des noeuds w et y, chacun disposant d'une capacité d'une règle. Supposons que l'ACL initiale soit décomposée en deux sous-ensembles de règles, notés f_0 et f_1. Dans un premier cas, si l'objectif consiste simplement à couvrir l'ensemble des chemins de routage de l'ensemble $\mathcal{P}$ = Path 1, Path 2, Path 3 tout en minimisant le nombre de règles installées, un placement faisable consiste à affecter f_0 au noeud d et f_1 aux noeuds e et g. Cependant, si f_1 possède une priorité plus élevée que f_0, c'est-à-dire si elle doit être évaluée en premier afin de préserver la sémantique de l'ACL initiale, le placement doit être adapté en conséquence. Dans ce cas, f_1 est affectée au noeud d, tandis que f_0 est placée aux noeuds e et g. Supposons maintenant que f_1 soit principalement composée de règles dont l'action associée est drop. Afin de réduire la consommation inutile de bande passante, il devient souhaitable d'appliquer ces règles le plus tôt possible le long du chemin suivi par le trafic. Sous cette contrainte, f_1 est déplacée plus près de l'entrée du réseau, à savoir vers les noeuds a et c, tandis que f_0 reste placée au noeud d. Enfin, dans un environnement de routage dynamique où le chemin effectivement suivi par le trafic ne peut pas être prédit à l'avance, tous les chemins possibles entre s et t doivent être couverts. Cette exigence conduit à étendre le placement, avec f_0 affectée au noeud w et f_1 au noeud y. Cet exemple montre que le placement des règles ACL doit s'adapter à des contraintes multiples et évolutives, incluant la couverture des chemins, l'ordre d'évaluation des règles et les considérations liées à la bande passante. Il met ainsi en évidence la nécessité de stratégies de placement flexibles et conscientes des contraintes opérationnelles.

À partir des défis identifiés, cette thèse vise à répondre aux questions de recherche suivantes, liées au déploiement scalable et efficace de politiques ACL distribuées :

RQ1: *Comment placer de manière optimale des sous-ensembles de règles ACL dans un réseau de commutateurs à capacité mémoire limitée, afin de garantir l'application complète de la politique sur* tous *les chemins possibles entre une source et une destination, indépendamment des changements de routage ?* La plupart des approches existantes supposent un routage statique ou un chemin unique prédéterminé. Or, dans les réseaux modernes,

le trafic entre une source et une destination peut suivre plusieurs chemins en raison de la dynamique du routage, de l'équilibrage de charge ou de défaillances. Cette question porte donc sur la manière de garantir que chaque chemin possible soit correctement filtré, tout en évitant une réplication inutile des règles et en respectant les contraintes mémoire des commutateurs.

RQ2: *Comment rendre le placement distribué des ACLs scalable et adaptable à des topologies réseau de grande taille et dynamiques ?* Les approches d'optimisation exacte deviennent coûteuses lorsque la taille du réseau, le nombre de règles et le nombre de contraintes augmentent. Cette question examine dans quelle mesure des méthodes fondées sur l'apprentissage et tenant compte de la structure du graphe peuvent fournir des stratégies de placement efficaces et généralisables à différentes configurations réseau.

RQ3: *Comment intégrer des contraintes opérationnelles pratiques dans le problème de placement des ACLs ?* Dans les déploiements réels, les décisions de placement doivent prendre en compte des contraintes supplémentaires, telles que la bande passante disponible, les dépendances entre règles et les politiques de routage prédéfinies. Cette question vise à étudier comment intégrer ces contraintes dans le processus de placement tout en préservant la correction de la politique et le passage à l'échelle.

A.3 Notre contribution

Cette thèse propose trois contributions complémentaires au problème du placement distribué des ACLs, chacune répondant à l'une des limitations de recherche identifiées au Chapitre 2. Ces contributions sont présentées selon un ordre croissant de généralité et de réalisme opérationnel.

L'état de l'art présenté au Chapitre 2 a mis en évidence trois limitations principales dans les travaux existants sur le placement distribué des ACLs. Premièrement, la plupart des approches supposent un routage statique ou un chemin unique prédéterminé, ce qui laisse encore insuffisamment traitée la question de l'application correcte des politiques en présence d'incertitude de routage. Deuxièmement, les méthodes heuristiques restent fortement dépendantes du problème considéré et nécessitent une redéfinition nécessitant une connaissance approfondie et une compréhension experte pour modéliser correctement les changements lorsque les objectifs ou les contraintes évoluent, tandis que les formulations d'optimisation exacte perdent rapidement en scalabilité lorsque la taille du réseau et la complexité des contraintes augmentent. Troisièmement, les solutions fondées sur l'apprentissage restent encore peu explorées pour ce problème ; à notre connaissance, cette thèse constitue le premier travail combinant RL et GNN pour le placement distribué des ACLs. De plus, aucun travail existant ne traite conjointement, dans un cadre unifié, l'incertitude de routage, la scalabilité du placement, l'efficacité en bande passante et les contraintes d'ordre entre règles. Ces limitations identifiées motivent les trois contributions développées dans cette thèse.

A.3.1 Placement distribué des ACLs sous incertitude de routage (Chapitre 3).

Le Chapitre 3 traite le problème du placement distribué des ACLs dans un contexte de routage dynamique, où l'ensemble des chemins réalisables entre une source et une destination doit rester protégé malgré d'éventuels changements de route. Le problème est formalisé sur des graphes orientés acycliques source-destination, ou st-DAGs, et trois stratégies de placement complémentaires sont proposées. Premièrement, un algorithme exact en temps polynomial, fondé sur une formulation par st-mincut, permet d'obtenir une solution optimale dans le cas d'une seule règle. Deuxièmement, une heuristique gloutonne sélectionne les noeuds en fonction de leur contribution à la couverture de nouveaux chemins, offrant ainsi une solution légère pour le cas multi-règles, en particulier sur des graphes creux. Troisièmement, plusieurs approches par apprentissage par renforcement, incluant QL tabulaire, DQN et DQN avec masquage des actions, sont étudiées afin d'apprendre des politiques de placement adaptatives sous contraintes de mémoire. Les expériences menées sur cinq topologies réelles issues de l'Internet Topology Zoo montrent que l'approche fondée sur le st-mincut obtient systématiquement la meilleure qualité de placement, en termes d'occupation mémoire, parmi les méthodes évaluées. Les résultats montrent également que DQN avec masquage des actions atteint, sur toutes les instances considérées, des valeurs d'occupation identiques à celles obtenues par l'approche st-mincut, tout en réduisant le temps de convergence. Les résultats montrent également que le RL peut trouver des placements faisables dans des cas où l'heuristique gloutonne et st-mincut échouent, mettant ainsi en évidence l'intérêt de la prise de décision séquentielle pour le placement distribué des ACLs, ainsi que le compromis entre optimalité, efficacité et adaptabilité.

A.3.2 Placement scalable sensible à la topologie via GNN-DQN (Chapitre 4).

Une limitation importante identifiée à la fin du Chapitre 3 est que les politiques de placement fondées sur le RL restent liées à la topologie utilisée durant l'entraînement : toute modification structurelle, telle qu'une panne de lien, une mise à jour du routage ou l'ajout d'un commutateur, peut invalider la fonction Q apprise et nécessiter un réentraînement complet. Or, ces changements font partie du fonctionnement normal des réseaux opérationnels. Le Chapitre 4 répond donc à cette limitation en intégrant les GNNs avec DQN, ce qui permet à l'agent d'encoder la structure du réseau sous la forme de représentations latentes généralisables associées aux noeuds. L'architecture GNN-DQN obtenue apprend ainsi des politiques de placement capables de généraliser à des instances de graphes de tailles et de structures différentes, en exploitant des représentations invariantes aux permutations ainsi que les relations entre les noeuds. Une contribution architecturale centrale réside dans la modélisation explicite de l'espace d'actions conjoint règle-noeud. Contrairement aux approches RL fondées sur les GNNs, où une action correspond souvent à la sélection d'un seul noeud, le placement distribué des ACLs nécessite de sélectionner une paire (v, f), où v est un noeud candidat et f un ensemble de règles. Comme cet espace d'actions structuré possède une cardinalité $|V| \times |F|$, une représentation augmentée composée de $|F|$ copies du graphe initial est construite. Cette représentation permet au

réseau de neurones d'estimer les valeurs des actions sur l'ensemble de l'espace conjoint, tout en préservant la structure de passage de messages. En outre, en raison de l'absence de bases de données publiques de référence pour les st-DAGs, le Chapitre 4 introduit un outil de génération de graphes synthétiques présentant des propriétés structurelles contrôlées et calibré à partir de topologies issues de l'Internet Topology Zoo. Cet outil fournit le support expérimental utilisé pour l'entraînement et l'évaluation dans les Chapitres 4 et 5. Les expériences montrent que GNN-DQN maintient une bonne qualité de placement et une bonne scalabilité pour différentes configurations réseau, tout en capturant les dépendances de chemins et les contraintes de faisabilité que les méthodes statiques ne peuvent pas exploiter. Enfin, le cadre proposé est étendu au placement sous contrainte de distance, dont la mise en œuvre ne nécessite qu'une intégration au niveau de la récompense, ce qui démontre sa flexibilité pour intégrer de nouvelles contraintes opérationnelles sans modifier l'architecture globale.

A.3.3 Placement des ACLs sensible aux contraintes sur des chemins sélectionnés (Chapitre 5).

Les Chapitres 3 et 4 ont considéré une configuration conservatrice dans laquelle le placement doit garantir la couverture de tous les chemins possibles entre une source et une destination. Bien que cette formulation soit robuste face à l'incertitude de routage, elle peut entraîner une réplication inutile des règles lorsque le routage suit des stratégies prédéfinies, telles que le plus court chemin ou des politiques de traffic engineering. Le Chapitre 5 étend donc le cadre de placement à un contexte plus réaliste, dans lequel le placement des règles est limité à un sous-ensemble prédéfini de chemins de routage reflétant le comportement effectif du réseau. Dans ce contexte, deux contraintes opérationnelles supplémentaires sont intégrées : une contrainte de gaspillage de bande passante, qui favorise un filtrage près de la source afin de réduire la consommation inutile de ressources le long des chemins de trafic, et une contrainte d'ordre entre règles, qui préserve la cohérence sémantique lorsque des dépendances existent entre sous-ensembles de règles. Le cadre GNN-DQN est adapté afin de traiter conjointement ces contraintes interdépendantes, au moyen de représentations d'état enrichies, d'un espace d'actions modifié et d'une formulation de récompense adaptée. Des expériences approfondies, menées face à Palette, Raptor et Raptor++ sur des topologies synthétiques et réelles, montrent que l'approche proposée atteint des taux élevés de succès de placement ainsi qu'une utilisation efficace de la TCAM. Les résultats confirment également que les méthodes heuristiques et fondées sur l'optimisation restent compétitives sur des instances de petite taille ou peu denses, mais que leurs performances se dégradent lorsque la taille du graphe et la complexité des contraintes augmentent, tout en nécessitant une révision importante lorsque les objectifs ou les contraintes changent. À l'inverse, GNN-DQN conserve une bonne capacité d'adaptation et des performances robustes dans les différents scénarios évalués, au sein d'un modèle unifié.

A.4 Limitations

Bien que les contributions développées dans cette thèse proposent des solutions scalables et adaptées aux contraintes pour le placement distribué des ACLs, plusieurs limitations persistent . Ces limitations concernent principalement le modèle réseau considéré, la formulation par apprentissage, ainsi que la séparation entre la décomposition des ACLs et le placement des règles.

Modèle réseau à source et destination uniques.
Le modèle réseau adopté tout au long de cette thèse représente la topologie sous la forme d'un DAG à source unique et destination unique. Cette formulation permet de capturer les caractéristiques essentielles du problème de placement tout en conservant une formulation analytique et algorithmique tractable, mais elle ne reflète pas directement la nature multi-commodity des réseaux réels, où plusieurs paires source-destination coexistent et peuvent partager des commutateurs intermédiaires. Lorsque toutes les paires sont soumises à une même politique ACL, le modèle peut être étendu de manière directe en introduisant une source virtuelle et une destination virtuelle, reliées respectivement à l'ensemble des sources et destinations, puis en appliquant les algorithmes proposés sans modification. Dans le cas plus général où chaque paire source-destination applique sa propre politique, une extension naturelle consiste à résoudre le problème de placement séparément pour chaque paire, en utilisant le sous-ensemble de chemins de routage correspondant. Cependant, cette approche itérative paire par paire ne tient pas compte de la concurrence pour les ressources mémoire au niveau des commutateurs intermédiaires partagés. En effet, résoudre indépendamment le placement de chaque paire peut saturer des commutateurs situés sur les chemins de plusieurs paires, ce qui peut conduire à des solutions globalement sous-optimales, voire infaisables. Cette limitation provient du fait que chaque décision de placement est prise localement, sans vision globale de la demande agrégée en ressources imposée par les autres paires.

Restriction aux méthodes RL off-policy et compromis lié au masquage des actions.
Le cadre RL développé dans cette thèse repose exclusivement sur DQN, un algorithme off-policy. Ce choix est directement lié à la structure de l'espace d'actions : comme la validité et la redondance des actions varient d'une instance de graphe à l'autre, les actions invalides et redondantes sont modélisées comme des contraintes souples à travers des pénalités de récompense strictes, plutôt que comme des exclusions explicites. L'objectif de cette formulation est d'entraîner l'agent à apprendre, à partir des caractéristiques structurelles de chaque graphe, quelles actions sont valides et lesquelles sont optimales parmi elles. Cette approche conduit toutefois à un espace d'actions de grande taille, dans lequel les actions valides ne représentent qu'une faible proportion des décisions possibles. Il en résulte un coût computationnel plus élevé durant l'entraînement, ainsi qu'un léger écart de qualité des solutions par rapport aux méthodes exactes et heuristiques, comme observé au Chapitre 5.

Une formulation alternative, étudiée au Chapitre 3, consiste à appliquer un masquage des actions. À chaque étape de décision, une fonction de validité calculée dynamiquement

identifie l'ensemble des actions réalisables et contraint l'agent à choisir uniquement parmi celles-ci. Ce mécanisme évite une exploration inutile des régions infaisables de l'espace d'actions et, comme montré au Chapitre 3, permet d'atteindre une qualité de placement équivalente à celle du meilleur algorithme, tout en réduisant le temps de calcul par rapport à la formulation DQN fondée sur des contraintes souples. Toutefois, le masquage des actions nécessite une fonction de validité définie manuellement, recalculée à chaque étape et adaptée à la structure spécifique des contraintes du problème, ce qui limite sa généralisation à d'autres contextes de placement. Au-delà de la question du masquage, la formulation par contraintes souples entraîne une conséquence plus fondamentale : elle limite le cadre proposé aux algorithmes off-policy. DQN gère efficacement les espaces d'actions de grande taille comportant une faible proportion d'actions valides et contraints en réutilisant les expériences passées grâce à son mécanisme de replay d'expérience, ce qui permet à l'agent d'apprendre à partir des actions invalides sans devoir les réexplorer continuellement. À l'inverse, les algorithmes on-policy, tels que PPO, échantillonnent les actions à partir de la politique courante et écartent les trajectoires collectées après chaque mise à jour. Dans des espaces d'actions où les actions valides sont rares par rapport à l'ensemble des actions possibles, cela entraîne une exploration répétée de décisions invalides, une convergence lente et une forte dégradation des performances d'apprentissage lorsque le masquage des actions n'est pas utilisé. Par conséquent, la formulation actuelle ne permet pas d'adopter directement des méthodes on-policy, bien que celles-ci présentent des avantages en matière de stabilité de la politique et de garanties théoriques de convergence.

Décomposition des ACLs supposée donnée.
Toutes les contributions présentées dans cette thèse supposent que l'ACL initiale a déjà été décomposée en un ensemble de sous-ensembles de règles avant le début de l'étape de placement. Le problème de décomposition, c'est-à-dire la manière de partitionner l'ACL en sous-ensembles de taille appropriée tout en préservant la correction sémantique, en minimisant les dépendances entre sous-ensembles et en garantissant leur compatibilité avec les contraintes de capacité du réseau cible, n'est pas traité. En pratique, la qualité de cette décomposition conditionne directement la difficulté du problème de placement ainsi que l'efficacité de la politique distribuée obtenue. Une décomposition produisant des sous-ensembles trop volumineux ou fortement dépendants peut rendre le problème de placement infaisable, ou dégrader de manière significative l'occupation obtenue, même lorsque les stratégies de placement utilisées sont efficaces.

A.5 Travaux futurs

Les contributions et les limitations de cette thèse ouvrent plusieurs perspectives de recherche, à la fois pour étendre les modèles de placement proposés et pour traiter des problématiques plus larges liées à la gestion distribuée des ACLs.

Une première direction importante concerne l'extension du cadre proposé au problème de placement multi-commodity. Une telle extension nécessiterait de coordonner les décisions de placement sur des commutateurs partagés et sous contraintes de capacité TCAM, dans la mesure où les décisions prises pour une paire source-destination peuvent affecter la

faisabilité et l'efficacité du placement pour les autres paires. Le RL multi-agent constitue une piste prometteuse dans ce contexte. Chaque agent pourrait être associé à une paire source-destination, tandis qu'un mécanisme d'apprentissage coopératif encouragerait les agents à coordonner leurs décisions afin de préserver la faisabilité globale sur les ressources réseau partagées.

Une deuxième direction consiste à améliorer la manière dont les contraintes de placement sont intégrées dans le processus d'apprentissage. Les résultats obtenus avec DQN fondé sur le masquage des actions ont montré que restreindre explicitement l'agent aux actions faisables peut améliorer de manière significative l'efficacité de l'exploration ainsi que la qualité des solutions obtenues. Étendre ce mécanisme au cadre GNN-DQN permettrait d'étudier l'impact du masquage des actions à la fois sur la convergence et sur la capacité de généralisation à différentes instances de graphes. Une intégration plus systématique du masquage des actions pourrait également faciliter l'utilisation d'autres algorithmes RL, tels que les méthodes actor-critic ou policy-gradient, en évitant l'exploration répétée de décisions invalides et en améliorant l'efficacité en nombre d'échantillons.

Une troisième direction consiste à traiter conjointement la décomposition des ACLs et le placement des règles. Les travaux présentés dans cette thèse supposent que l'ACL initiale a déjà été décomposée en sous-ensembles de règles plus petits et se concentrent sur la décision de placement de ces sous-ensembles dans le réseau. Des travaux futurs pourraient donc étudier un cadre d'optimisation conjoint, dans lequel l'ACL initiale serait décomposée en tenant compte simultanément de la topologie réseau, des capacités des commutateurs, des chemins de routage, des dépendances entre règles et des objectifs de placement. Une telle approche intégrée pourrait réduire les dépendances inutiles entre sous-ensembles de règles, améliorer la faisabilité du placement et diminuer davantage l'occupation des TCAMs. Elle permettrait ainsi de faire évoluer le cadre proposé vers une chaîne complète de découpage et de distribution, depuis la table ACL initiale jusqu'au déploiement final dans le réseau.

Enfin, une perspective plus large consiste à évoluer vers une gestion autonome des ACLs dans les réseaux self-driving. Dans de nombreux environnements opérationnels, la politique de sécurité appliquée dans le réseau n'est pas toujours explicitement documentée ni exprimée dans un langage formel de haut niveau. Elle est souvent intégrée dans des configurations ACL spécifiques aux fournisseurs, qui ont évolué au fil du temps et peuvent avoir été héritées d'administrateurs précédents. Cette situation rend la compréhension, la vérification et la mise à jour des politiques particulièrement difficiles. Des travaux futurs pourraient explorer l'utilisation des Large Language Models (LLMs) afin d'extraire les intentions de sécurité implémentées par les tables ACL existantes. Ces intentions inférées pourraient ensuite être utilisées pour raisonner sur la politique actuelle, détecter des incohérences, guider les mises à jour de politique et générer de nouvelles configurations ACL. À plus long terme, un système autonome de gestion des ACLs pourrait observer le trafic réseau, inférer les intentions de sécurité encodées dans l'ACL courante, mettre à jour la politique en fonction de nouvelles exigences ou menaces, décomposer l'ACL résultante en sous-ensembles de règles, puis les distribuer efficacement dans le réseau. Un tel cadre combinerait l'extraction d'intentions, la mise à jour des politiques, la décomposition des ACLs et le placement distribué au sein d'un système en boucle fermée unifié, destiné à une gestion scalable et adaptative de la sécurité réseau.