# In-Network ACL Rules Placement using Deep Reinforcement Learning

Wafik Zahwa, Abdelkader Lahmadi, Michael Rusinowitch, Mondher Ayadi

# In-Network ACL Rules Placement using Deep Reinforcement Learning

Wafik Zahwa[†*], Abdelkader Lahmadi[*], Michael Rusinowitch[*], Mondher Ayadi[†]
[*] Université de Lorraine, CNRS, Inria, Loria, F-54000 Nancy, France, {firstname.lastname}@inria.fr
[†] NUMERYX, France, w.zahwa@numeryx.fr

***Abstract*—Automatically distributing Access Control Lists (ACLs) within a software-defined network plays a critical role in ensuring seamless connectivity, security, and reliability for internal services and hosts. Typically implemented in switches using Ternary Content-Addressable Memory (TCAM), ACLs placement face challenges due to the limited capacity of TCAM memory. To address this, large ACLs must be divided and distributed across multiple switches, ensuring that each packet traveling from source to destination undergoes the necessary match-action rules. In this paper, we propose a novel approach that combines graph-embedding neural networks (GNN) with deep Q-learning (DQN) to automate the distribution of ACLs across network switches while minimizing TCAM memory usage. By allowing additional constraints and evaluating our trained models on both synthetic and real-world network topologies, we show that our approach has a placement success score up to 99% on unseen graphs.**

***Index Terms*—ACL, automated rule placement, SDN, graph embedding, reinforcement learning, deep Q-learning.**

## I. Introduction

In modern computer networks, the management of fundamental operations like filtering, routing, load balancing, traffic engineering, and firewall provisioning has evolved into a critical challenge for network operators due to the increasing complexity of these systems. However, the shift towards software-defined networking (SDN) has simplified deployment and management operations by introducing an abstract network view, simplifying the expression of policies for network programming and configuration. These policies, comprising high-level formal statements, dictate packet processing within the network and are commonly translated into specific rules or configurations, such as access control lists (ACLs).

ACL rules are stored in tables implemented through the use of ternary content addressable memories (TCAM). A standard TCAM in a switch typically accommodates a few thousand entries or rules [1]. Moreover, given that TCAM processing is power-intensive, even with current TCAM sizes, it could consume up to 60% of the total power utilized by a network switch [2]. Consequently, optimizing TCAM utilization is imperative not only due to its finite nature but also for mitigating power consumption and heat generation concerns [3]. Utilizing large TCAM chips may raise concerns regarding increased power consumption and heat generation, consequently elevating energy expenses [4]. Currently, the prevailing TCAM chips commonly come with capacities of either 10Mb (supporting 33k ACL entries of IPv6) or 20Mb, meeting the requirements of most small to medium-sized networks. However, larger options exist, exemplified by the largest TCAM chip as of 2021 with a size of 144Mb, capable of accommodating up to 512k entries.

Despite the expansion in TCAM sizes, the demand for additional rules persists for network control, including managing unwanted traffic, Quality of Service (QoS), and network flow, results in a continual growth of the number of ACL rules within these lists [5]. Thus, to mitigate reliance on large and costly memory capacities within network switches, an alternative approach involves leveraging smaller capacity switches and partitioning ACLs, distributing their segments throughout the network to maintain equivalent control functionality between endpoints.

Previous works [5], [6] have focused on distributing rules along single routing paths, which poses challenges when facing path failures or when congestion occurs, necessitating rule installation on backup paths to mitigate latency. To address this issue, Kanizo et al. [7] explore the utilization of multi-path routing by distributing rules across various paths, and proving that this problem is NP-hard. In addition, switches dynamically adjust their routing paths, making it challenging to predict the path taken by a packet, and existing approaches become inefficient in handling such situations for rules placement. In this work, we propose to distribute rule sets along *every path from source to destination*, named "st-paths", to mitigate the impact of dynamic routing and ensures comprehensive rule coverage across all potential paths. Reinforcement learning (RL) provides an efficient and scalable approach to optimize problems by approximating solutions and automating problem-solving processes. Therefore we use RL, specifically deep Q-learning (DQN) combined with graph neural networks (GNN), to automatically address the ACL rules placement problem. We then utilize these learned models to distribute rules on unseen network topologies, ensuring coverage of all potential routing paths between a given source and destination. Specifically, our contributions can be summarized as follows:

- We design a novel algorithm coupling DQN and GNN techniques to automatically learn strategies for ACL rules placement problems.
- We integrate extensions to handle the placement problem with distance constraints.
- We conduct extensive evaluation of the algorithm on both synthetic and real-world network topologies.

The paper is structured as follows: Section II reviews related

works on the rule placement problem, graph embedding and deep reinforcement learning for combinatorial optimization, while Section III introduces our network representation, formalizes the rules placement problem, and outlines general principles and distribution constraints. Our approach is detailed in Section IV, followed by a description of our graph generation tool and training setup in Section V. Section VI summarizes our findings and discussions, and Section VII presents our conclusions and avenues for future research.

## II. Related works

The authors of [8] provide an overview of approaches to the OpenFlow rules placement problem, emphasizing its significance across various applications like routing and security, each necessitating tailored placement strategies. They discuss challenges such as resource constraints and signaling overhead, and categorize existing solutions into eviction, compression, and split and distribution approaches.

In our work, we focus on addressing resource limitations through the split and distribution of ACLs. However, in [5], the authors propose efficient algorithms for distributing filtering rules across network paths, albeit limited to networks that are series-parallel or closely resemble them. Another approach, presented in *Palette* [7], offers a heuristic solution for distributing a set of rules across every path from a given set, even when the set of paths is arbitrary which is inapplicable to our problem, where rules have to be distributed across all possible paths. However, the computational complexity of this heuristic grows exponentially with respect to the network size. The authors in [9] develop a solution for the rules placement problem based on MaxSMT but their solution is not suitable for dynamic routing.

In [10], the authors demonstrate the effectiveness of deep reinforcement learning in resolving rules placement problems, particularly within complex environments. Their proposed eviction strategy, based on deep reinforcement learning, targets the challenge of rule placement in SDN flow tables. The main idea involves removing less frequently used rules to make room for new rules that are expected to be accessed more frequently.

In recent studies [11], [12], the authors have integrated graph embedding and deep reinforcement learning to address combinatorial optimization problems on graphs, including minimum vertex cover, maximum cut, and traveling salesman. Graph embedding, known for its capability to compress intricate network structures into low-dimensional representations while preserving essential structural and relational details, is widely utilized across diverse fields and applications. These studies showcase the effectiveness of this fusion by training neural networks on a dataset and then utilizing the trained model to solve problems on larger graphs spanning various categories. However, in [11], [12], the authors consider the solution as a set of nodes. The challenge arises when the number of neurons in the final layer of the neural network must match the number of actions to be executed by the agent, which corresponds to the number of nodes [11], [12]. This challenge is easily solved when the number of actions is equal to the number of nodes because each vector node embedding results in a Q-value which is not suitable for our problem because the agent aims to choose which rule set to install on which node.

In this paper, we consider the problem of distributing rule sets among network switches to meet end-point policies. Unlike other works (e.g., [7]) we distribute the rules over *all paths* between source and destination by leveraging graph embedding techniques and deep reinforcement learning.

## III. Preliminaries

### A. Network representation and problem statement

We represent the network topology as an *st-dag*, which stands for a directed acyclic graph featuring a single source node, denoted as $s$, and a single destination node, referred to as $t$. Within an st-dag $G$, switches are represented by nodes, while network links are depicted by edges. Any route from the source $s$ to the destination $t$ is termed an st-path. Within a st-dag, every node is positioned along at least one st-path. We denote $n' < n$ if node $n'$ precedes node $n$ along some st-path.

The node's capacity refers to the maximum number of rule sets that its corresponding switch's TCAM memory can accommodate. Given the dynamic nature of routing and the uncertainty regarding the path taken by packets at any given moment, our task entails efficiently distributing rules across all paths from the source to the destination. This optimization aims to ensure that switch resources are utilized effectively, minimizing the overall occupancy of rule space. For instance, it's crucial to prevent packets from encountering the same rule set multiple times along a path. Nevertheless, in certain scenarios, this redundancy cannot be entirely avoided. Therefore, our goal is to minimize the utilization of switch resources by reducing the space occupied by these rules, all while ensuring coverage of all paths from the source to the destination.

*Formal problem statement:* Given an st-dag $G = (V, E)$, such that every node $n$ has finite memory space $m(n)$ (which we define to be the maximum number of rules that can be installed on $n$) and a set of rules $F = \{f_1, f_2, ..., f_k\}$. The problem we are addressing involves the placement of rule sets $f_1, f_2, ..., f_k$ on nodes within graph $G$. This placement needs to satisfy two conditions: firstly, ensuring that each path includes one occurrence of every rule set, and secondly, minimizing the number of switches occupied. We denote this problem as $PLACE(G, m, F)$. It's important to note that $PLACE(G, m, F)$ lacks a solution if the shortest path between $s$ and $t$ involves fewer than $k$ switches. Conversely, if the shortest path between $s$ and $t$ involves fewer than $k$ switches, ensuring condition 1) becomes straightforward.

**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. □

### B. Rule Placement problem with distance constraints

To prove the flexibility of our DQN approach for addressing constrained rules placement problem, we introduce the following constraints: *minimizing the distance to the source* or *minimizing the distance to the destination*.

Filtering traffic closer to the source offers several benefits. Firstly, it reduces unnecessary network traffic, particularly useful during high-traffic periods like denial of service attacks (DOS). Secondly, enforcing security policies at the source immediately blocks potentially harmful traffic, reducing the risk of security breaches. Additionally, filtering at or near the source reduces the workload on intermediary network devices, enhancing network performance and scalability.

Alternatively, placing ACL rules closer to the destination enables more targeted security measures, ensuring that only authorized traffic reaches the destination. When ACLs are placed closer to the destination, administrators can create rules that specifically address the protocols and behaviors of applications being used. This allows for more targeted control over how applications interact within the network.

## IV. Deep Q-learning approach for rule placement

We address this optimization problem within the framework of a Markov decision process (MDP). In this setup, the agent encounters a state $s$ and selects an action $a$. Upon executing the chosen action, the agent receives a reward $r$, which summarizes the efficacy of action $a$ within the current state $s$. The Q-value of a given state-action pair, $(s, a)$, is determined by the discounted sum of immediate and future rewards.

Traditional Q-learning typically employs a lookup table to store Q-values for each state-action pair. However, this becomes impractical with expansive or continuous state spaces due to the table's immense size. Deep Q-Networks (DQN) [13] address this issue by approximating the Q-function using a neural network. A DQN furnishes a function $Q(s, a; \theta)$, where $\theta$ represents the network parameters (weights and biases), trained to approximate $Q^*(s, a)$, which denotes the maximum Q-values for each state-action pair under the optimal policy. Once trained, actions are selected greedily based on predicted Q-values to derive an estimate of the optimal policy. In our study, we employ a message-passing neural network (MPNN) as the deep Q-network to solve our placement problem.

### A. Action Space

In our placement problem, actions $a(v, f_i)$ represent the installation of rule set $f_i$ on node $v$. Actions fall into three categories: valid, invalid, and redundant. A valid action is one where the node has enough memory for the rule installation. An action is redundant if attempting to install rule set $f_i$ on node $v$ while all covered paths already have this rule set. An action is invalid if node $v$ is the source or destination, if node $v$ does not exist in the graph ($v \notin V$), if there is insufficient memory on node $v$, or if the rule set $f_i$ is already installed on node $v$.

### B. State Space

A state characterizes the current occupation of nodes by the rule set at some stage of solution construction. A state includes the graph $G(V, W)$ with node features. Each node $v$ is associated with $k$ features vectors $x_v^1, \ldots, x_v^k$ where $k$ is the number of rule sets. Given a specific rule set $f_i$ and a specific node $v$, $x_v^i$ has 5 components:

- $x_v^i[1]$ Node degree (i.e., number of edges incident to $v$);
- $x_v^i[2]$ Boolean, true if the rule set $f_i$ 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 $f_i$;
- $x_v^i[5]$ 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 set to install on which node) within a given state.

*Extending the state for handling distance constraints:* To adapt the model for solving the placement problem with the distance constraints (placing rules either close to the source or destination), we need to modify the feature vectors by adding two components to the vector features $x_v$, considering a specific rule set $f_i$ and a specific node $v$:

- $x_v^i[6]$ indicates whether $f_i$ should be placed near the source (1 for yes, 0 for no);
- $x_v^i[7]$ indicates whether $f_i$ should be placed near the destination (1 for yes, 0 for no);

These features help differentiate between rule sets based on their placement preferences.

### C. Reward

Our objective is to efficiently distribute rule sets along all paths between the source and destination nodes, optimizing switch resources. The reward obtained by the agent increases when placing a rule set on a node covering previously uncovered paths, while penalties are incurred for actions resulting in redundancy among rule sets. The reward of the action $a(v, f_i)$ of installing rule set $f_i$ on node $v$, is:

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

where $paths_{v,f_i} \in [0, 1]$ is the ratio of new paths that get covered due to this action, i.e. the paths from $s$ to $t$ that contain node $v$ and did not contain the rules set $f_i$ before this action over the total number of paths. The term $occ_v \in [0, 1]$ is the memory occupancy ratio of $v$, i.e., the number of rule sets installed on node $v$ over its total capacity. Additionally,

$paths_{v,f_i}$ and $x_v^i[3]$ and $x_v^i[4]$ are determined using a greedy polynomial algorithm (as in [14]).

*Extending the reward to handle additional distance constraints:* In order to effectively tackle the placement problem while considering the distance constraints, such as placing rules near the source, or near the destination, the reward of the action $a(v, f_i)$ of installing rule set $f_i$ on node $v$, is modified:

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

where $dist_{s,f_i}$ is the normalized distance between source-labeled rule sets and the source $s$. The normalized distance measures the number of nodes between the source and the labeled rule sets divided by the longest distance from the source in the graph, and similarly $dist_{f_i,d}$ is the normalized distance between destination-labeled rule sets and the destination $d$, calculated in the same manner.

## V. Experiments

### A. Synthetic graphs generation

We first evaluate the DQN algorithm on randomly generated st-dags. An st-dag can be represented using a dictionary, where each node $v \in V$ serves as a key, and the value associated with each key $v$ comprises two lists: one listing the immediate predecessors ($pred$) of node $v$, and the other listing the immediate successors ($succ$) of node $v$. The graph generator inputs are: the targeted networks size measured by its number of nodes $|V|$, the probability $p$ of an edge creation between a couple of nodes, the number $k$ of graphs to be generated, and the maximum connection depth $h$ of the generated st-dags, defined as an upper bound of $\{i; v \in V, v + i \in succ(v)\}$.

The tool initially generates $k$ random dags of size $|V|$, with the node set $\{0, 1, \ldots, |V|-1\}$. For each pair of nodes $(v, u)$ with $v < u$, a Bernoulli trial with success probability $p$ is conducted. If successful, an oriented edge $v \rightarrow u$ is created. Subsequently, the graph is cleansed if necessary, by removing nodes that cannot be connected to the source or destination by a directed path, thereby ensuring that only topologically ordered st-dags with source 0 and destination $|V| - 1$ are produced.

### B. Simulation setup

In our study, we utilized the Q-learning Algorithm 1, to address the placement problem. We conducted experiments where we trained and evaluated the Q-learning agent on graphs generated using our tool described in Section V-A.

The algorithm is coded in Python and the experiments are executed on a remote GPU server featuring an Intel(R) Xeon(R) Gold 6258R CPU @ 2.70GHz, 503GB of RAM, running Ubuntu 22.04.3 LTS (Jammy Jellyfish) OS, and equipped with an NVIDIA RTX A6000 GPU. The server's GPU is powered by Driver Version 530.30.02 and CUDA Version 12.1.

In evaluating our algorithm, we defined two key metrics: the *Occupancy* metric, indicating the percentage of the switches' total capacity occupied by rule sets, where lower occupancy is preferable; and the *success rate* metric, denoting the percentage of valid solutions returned for unseen graphs during training.

During training, we set the discount factor $\gamma$ to 0.8 to prioritize long-term rewards, while the learning rate $lr$ was set to $10^{-5}$ for stable optimization. Additionally, a penalty of $-10$ was applied in the reward function to discourage invalid actions. We utilized mini-batch sizes of 32 and conducted $k = 2$ episodes per graph, with $k$ representing the number of episodes for each graph. Furthermore, we employed a target update frequency $C$ of 1000, transferring weights from the online network to the target network every 1000 steps, and a validation frequency, *val_freq*, of 50 episodes to evaluate the neural network's behavior on unseen graphs.

**Algorithm 1:** Deep Q-learning for placement problem

```
Initialize replay memory D to capacity N
Initialize online network with random weights θ
Initialize target network with weights θ' = θ
for episode = 1 to M do
    if episode mod k = 0 then
        Sample a graph G(V, W) from dataset 𝕊
    end
    for t=1 to T do
        a = { random action a' ∈ 𝒜        with prob. ε,
            { argmax_{a'∈𝒜} Q(s, a'; θ)   otherwise.
        Execute action a_t and observe reward r_t and
          next state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}, isterminal_t)
          in D
        Set s_{t+1} = s_t
        Sample random minibatch of transitions
          (s_i, a_i, r_i, s_{i+1}, isterminal_i) from D
        Calculate loss
        Update θ by SGD
        Every C steps reset θ' = θ
    end
    if episode mod val_freq = 0 then
        online_net.eval(validtion set)
    end
end
```

## VI. Results and discussion

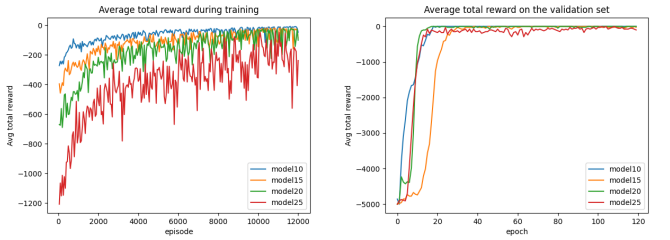
### A. Training results and performance

We conducted a series of experiments by generating 1000, 200, and 500 graphs for training, validation, and testing respectively with different graph sizes for each of them: 10, 15, 20, and 25. The validation set, consisting of graphs not present in the training data, was utilized to evaluate the model's ability to generalize to unseen data during training, with assessments conducted at regular intervals (val_freq = 50 episodes). For each trained model, we fixed the number of rule sets.

Figure 1a illustrates the progression of the average reward throughout the training process in each experiment. Observing Figure 1a, it becomes apparent that the average reward increases along the training process. Initially, the agent operates with minimal knowledge by employing entirely random actions to explore the environment. This is why a lower reward is observed initially. As training progresses, however, the agent gains familiarity with optimal actions to take in various states, resulting in fewer actions required to reach a solution. As the reward encourages the agent to minimize the occupancy by placing rule sets on junction nodes (maximize $paths_{v,f_i}$), the observed increase in reward during training indicates that the agent is learning to take the appropriate actions in each state.

Figure 1b also depicts the average reward within the validation set, comprising graphs unseen by the neural network during training. The upward trend in reward within the validation set indicates the agent's capability to generalize across diverse graphs. Specifically, this suggests that the agent effectively adapts to unseen scenarios, optimizing its performance.

In this direction, Table I provides a summary of the testing outcomes for each trained model across 500 graphs. Remarkably, our agent successfully identifies a solution for rule set placement in 99% of the provided graphs, underscoring its robust performance and generalizability.




(a) Average total reward during training for different graph sizes

(b) Average total reward on the validation set for different graph sizes

Fig. 1: Average reward during training and validation of different models on different graph sizes.

| 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 I: Memory occupancy and processing time of DQN with 500 graphs testing sets for each graph size.

### *B. Evaluation of trained models generalisation*

To demonstrate the effectiveness of our methodology and the capacity of graph embedding combined with DQN in capturing graph features for our combinatorial optimization problems, we have conducted different testing experiments. Firstly, we tested our trained model on graphs of size 25 and fixed 4 rules on a different set of 500 graphs with the same size, but we varied the number of rule sets in each test (the number of rule sets in each graph is taken to be the number of nodes in the shortest paths between the source and the destination over 2). Table II shows the results of this experiment proving that the agent trained on a fixed number of rule sets can find solutions for other graphs of the same size but variable number of rule sets with a success rate of 99.9%. This behavior is also observed in all 3 models 10, 15, and 20. Secondly, we have tested our agent trained on graphs of size 25, and 20 on 600 graphs of size 30 with a variable number of rule sets. Table III shows the results of this experiment proving that the agent can capture the necessary features from lower-size graphs with a fixed number of rule sets and find solutions for graphs of larger size and variable number of rule sets with a success rate of 95% for model trained on size 25 or 80% for model trained on size 20. Finally, we have tested our trained model on graphs of size 20 on graphs built from real-world network provided by Internet topology zoo [15]. Table IV shows the results of this experiment proving that the agent trained on fixed-size graphs 20 and fixed number of rule set 4 can find solutions for variable graph sizes with variable number of rule sets.

| | | | # 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 II: Testing a trained model with fixed graphs size (25) and number of rules (4) on graphs with same size but variable number of rule sets.

| | | | # 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 III: Testing trained models with fixed graphs size and number of rules on larger graph size and variable number of rule sets.

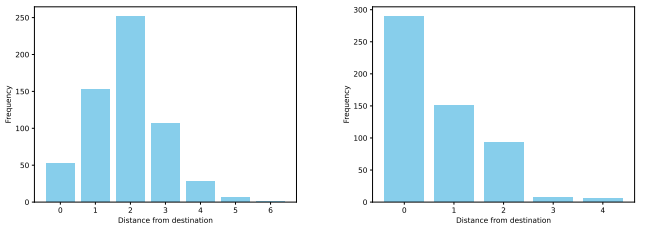
| Graph /size | ♯ of rules sets | Occupancy (%) | Time (s) |
|---|---|---|---|
| Sprint/11 | 3 | 38.89 | 0.06 |
| Abvt/23 | 6 | 50 | 0.22 |
| Agis/25 | 5 | 50 | 0.25 |
| Bics/34 | 8 | 31.81 | 0.39 |
| NetworkUSA/35 | 7 | 42.1875 | 0.55 |

TABLE IV: Memory occupancy and processing time of DQN on real-world network topologies by using a trained model of graphs with size 20 and 4 rules sets.

### *C. Results with distance constraints*

We trained four models on various graph sizes (10, 15, 20, and 25) to address the ACL rules placement problem under two specified constraints: close to the source or to the destination. Specifically, we assessed the placement solutions generated by the agent both before and after incorporating these constraints, focusing on graphs of size 25 (with similar results observed for other graph sizes). Figure 2a illustrates the distribution of distances of placement solution obtained

before integrating the near-destination constraint, while Figure 2b depicts the agent's output after incorporating this constraint. Notably, the agent demonstrates a tendency to minimize the distance between rule sets labeled as "close to the destination" and the destination itself as the number of rule sets at distance 0 increases, resulting in a gradual decrease in distance from the destination. These results are observed symmetrically to the source.




(a) Placement distances to the destination before adding the constraint

(b) Placement distances to the destination after adding the constraint

Fig. 2: Distribution of rules placement distances between destination-labeled rule sets and destination before (a) and after (b) adding the distance constraint.

### D. DQN vs st-mincut

It is worth noting that the st-mincut approach in our previous work [14] has globally better performance than our novel DQN approach. However, DQN offers significant advantages in its ability to adapt to various constraints like closeness to the source or the destination, unlike st-mincut, which is unable to handle them. DQN enables the agent to make decisions that may seem suboptimal in the short term but lead to greater rewards in the long run, showcasing its advantage over st-mincut in finding optimal solutions as illustrated below. Let us consider that we have to install two rule sets $f_1, f_2$ on the graph in Figure 3. Each node can afford one rule set. The solution returned by st-mincut is $u$ and $g$ in the first cut and $e, f, d, v$ and $w$ in the second cut resulting in 70% as occupancy. Instead, the DQN agent chooses to lose in the first installation by placing the rule set $f_1$ on nodes $u, v$, and $w$ and then installing the rule set $f_2$ on the nodes $e, f$, and $g$ resulting in 60% as occupancy.

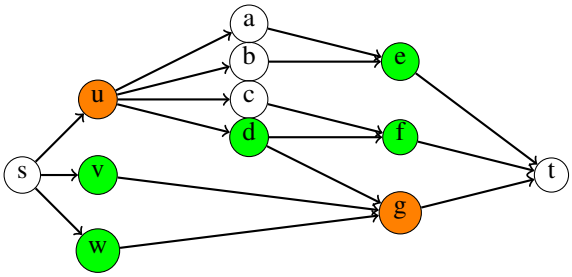



Fig. 3: Counterexample to st-mincut heuristics.

## VII. Conclusion

In this paper, we have coupled graph embedding and deep Q-learning techniques for solving the ACL rules placement problem. This combination enables the trained agent to generalize its learned models to solve similar problems on unseen graphs from various categories. These categories include graphs of the same size but with varying numbers of rule sets, larger-sized graphs with variable numbers of rule sets, and graphs with variable sizes and numbers of rule sets. Moreover, we have shown the flexibility of our approach compared to other heuristics. As future research, we plan to conduct a comparative analysis between our DQN approach and ILP-based rules placement. Furthermore, we aim to incorporate real-world industrial constraints to evaluate the adaptability and effectiveness of DQN in constrained placement problems.

## Acknowledgements

This work is supported by a CIFRE convention between the ANRT (National Association of Research and Technology) and the company NUMERYX Technologies.

## References

[1] Curtis Yu, Cristian Lumezanu, Harsha V Madhyastha, and Guofei Jiang. Characterizing rule compression mechanisms in software-defined networks. In *Passive and Active Measurement (PAM): 17th International Conference,Heraklion, Greece, March 31-April 1*. Springer, 2016.

[2] 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.

[3] 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.

[4] 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.

[5] Ahmad Abboud, Rémi Garcia, Abdelkader Lahmadi, Michaël Rusinowitch, and Adel Bouhoula. Efficient distribution of security policy filtering rules in software defined networks. In *19th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2020.

[6] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. Scalable rule management for data centers. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

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

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

[9] 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.

[10] 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.

[11] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 2017.

[12] 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.

[13] 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.

[14] 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.

[15] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 2011.