Automated Placement of In-Network ACL Rules

Wafik Zahwa, Abdelkader Lahmadi, Michael Rusinowitch, Mondher Ayadi

▶ To cite this version:

Wafik Zahwa, Abdelkader Lahmadi, Michael Rusinowitch, Mondher Ayadi. Automated Placement of In-Network ACL Rules. 2023 IEEE 9th International Conference on Network Softwarization (NetSoft), Jun 2023, Madrid, Spain. pp.486-491, 10.1109/NetSoft57336.2023.10175436 . hal-04236850

HAL Id: hal-04236850

https://inria.hal.science/hal-04236850

Submitted on 11 Oct 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Automated Placement of In-Network ACL Rules

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 deploying distributed Access Control Lists (ACLs) in a software-defined network can ensure their internal services and hosts connectivity, security and reliability. ACLs are often deployed in a switch using Ternary Content-Addressable Memory (TCAM). Since TCAM memory is often too limited to store a large ACL, one has to split the lists and distribute the parts on several switches in such a way that every packet travelling from a source to a destination undergoes the required match-action rules. In this paper, we develop and compare three algorithms based on graph theory and Reinforcement Learning (RL) techniques to automatically distribute ACLs across networks switches, while minimizing their TCAM memory occupancy. We compare the three algorithms on several network topologies to evaluate their efficiency in terms of memory occupancy.**

***Index Terms*—ACL, automated rule placement, SDN, mincut, greedy algorithm, reinforcement learning.**

I. Introduction

Today's computer networks are so complex that the management of their core functions such as filtering, routing, load balancing, traffic engineering, and firewalling has become a critical problem for their operators. Increasingly, the evolution towards software-defined networking (SDN) and the virtualization of services and functions (NFV) offers simplified deployment and management possibilities. The controller, i.e., the centralized element of the SDN paradigm, is responsible of the control plane of the network, by managing, configuring, monitoring, and troubleshooting virtual network infrastructure. In particular SDN allows one to deploy easily Access Control Lists.

Access Control Lists (ACL) are usually created by network administrators to dictate the action taken for each packet entering or leaving a network switch. However, the growing number of network control requirements (unwanted traffic, QoS, network flow control) leads to continuous increase of the number of rules in these lists [1] (size of ACLs can reach 100K entries or even larger [11]). ACL rules are stored in tables implemented through the use of TCAM memories that provide fast match-action operation as they use a parallel search system. However, they require a high amount of energy [2] and their cost is high [5]. While the most widely-used TCAM chips currently have a capacity of either 10Mb (33k ACL entries of IPv6) or 20Mb, which is sufficient for most small to medium-sized networks, there are larger options available. For instance, as of 2021, the largest TCAM chip available has a size of 144Mb and can support up to 512k entries. Despite the increase in TCAM sizes, the need for more rules continues to grow. Deploying large TCAM chips introduces potential drawbacks regarding power consumption and generated heat, leading to higher energy costs [12]. Therefore to avoid relying on large and expensive memory capacities in network switches, an alternative is to rely on smaller capacity switches and to split ACLs and distribute their parts over the network in such a way that provides the same control functionality between end-points. Consider for instance a core network as shown in Fig. 1 with 4 switches (a, b, c, d) having 10Mb TCAM capacity which means each switch can afford 33000 ACL entries. If we assume that we have 60000 ACL rules to install, we should in this case split the ACL table into two sub-tables f_0 and f_1 then distribute these sub-tables on all paths between source and destination in order to preserve the same security policy imposed by the initial table. There are only two solutions to the in-network placement problem of these rules sets, one of which is shown in Fig. 1. We note that we cannot avoid redundancy in some cases since st-path $a \to b \to c \to d$ contains rules set f_0 (or f_1) twice. TABLE I shows an example of rules that can be ACL entries.

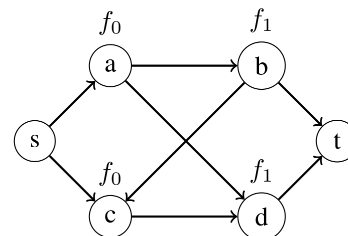


Fig. 1: Placement of a set of rules in a network.

Source Address	Source Port	Dest Address	Dest Port	Action
Any	Any	192.168.1.0	>1023	Allow
192.168.1.1	Any	Any	Any	Deny

TABLE I: Example of ACL entries.

Rules placement problem has been considered in previous works such as [6] where some rules are dispatched on a given set of paths, and where the set of paths can be arbitrary. The authors show that this problem is NP-hard. Their proposed heuristic solution to cope with the complexity is linear in the number of paths, but the number of paths can be exponential in the size of the network.

However routers may change their routing paths in a dynamic way and it is difficult to know which path is taken by

a packet. Therefore, due to this dynamic routing, it is difficult to anticipate which path is taken by a packet and therefore we propose in our work to distribute the rules sets on *every* path from source to destination. Such paths will be called an *st-paths*. Note that our problem is different from [6] since [6] consider the distribution of rules on *some* specified st-paths. We also note that the NP-hardness result in [6] is obtained with a set of paths that is a strict subset of the whole set of paths from source to destination and cannot be applied to our case.

Contributions of the paper: In this "all st-paths" setting we first show how the rules placement problem can be related to mincut computation in graphs and we derive a first algorithm based on this observation. Then we propose a polynomial-time heuristic based on installing an ACL primarily on switches located on a maximum number of st-paths that do not contain this ACL yet. Finally we design a reinforcement learning approach with a reward function that relies on the previous heuristic. The three proposed algorithms are compared on real-world topologies.

The remainder of this paper is organized as follows. Section II presents related work on the rule placement problem. In Section III, we define our network representation, formalize the rules placement problem and state the general principles and distribution constraints in this problem. We develop three algorithms to address rules placement problem, *s-t mincut*, *greedy*, and *reinforcement learning* rules placement respectively in Section IV, Section V and SectionVI. Simulation and results are detailed in Section VII. Finally, conclusion and future works are presented in Section VIII.

II. Related Work

The authors in [10] study OpenFlow rules placement problem and present a survey of the different proposals to solve it. They differentiate many applications that can benefit from OpenFlow rules placement solutions such as routing, security (access control), etc. Each of these applications requires a dedicated rules placement solution. Then they cite the challenges behind rules placement problems: resource limitations and signaling overhead and classify the exiting solutions for rules placement problems that address the switches memory constraints into three categories: (i) *eviction* by removing inactive rules to install recent rules; (ii) *compression* to reduce the number of required rules by using wildcard rules, and (iii) *split and distribution* of initial rules over all switches in the network.

In this paper, we rely on splitting and distributing ACLs to address resource limitations. In [1] the authors have proposed efficient algorithms to dispatch filter rules from a given set on all the network paths. However their approach is limited to networks that are series-parallel or close to this class. In *Palette* [6], the authors have proposed a heuristic solution to dispatch a set of rules on every path from a given a set of paths, and where the set of paths can be arbitrary. However the complexity of this heuristic is exponential with respect to the size of the network. In [8], the authors have proposed an ILP optimization solution to dispatch a set of rules on every path between source and destination. They consider two metrics to minimize: switches memory and wastage bandwidth, place rules as close as possible to the source. First, they group rules with same IP source to the same ingress SDN switch, then assign a capacity for each switch proportionally to the number of rules stored on it, apply the ILP optimization for each subset of firewall rules in parallel. Finally they develop an algorithm to distribute updates in case of rules changes. In [9], the authors have proposed an eviction solution based on deep reinforcement learning to address rules placement problem of SDN flow tables. The main idea of this work is to remove rules that are intended to be less used in order to make room for new rules that are prone to be used often.

As previous works we consider here the problem of distributing a rules set among network switches to meet end-point policies. However, in our approach, we distribute the rules over all paths between single source and single destination. Let us show the difference of our problem with the one handled by *Palette* [6] on a very simple example. Consider the network built from the set of paths $P = \{s \to a \to b \to t, s \to b \to c \to t\}$ such that there is space only for one rule on every switch. It is possible with *Palette* [6] to distribute consistently two different rules on every path above. However in our case this is not possible since any network that contains P should also contain the path $s \to b \to t$ on which there is no enough space for two rules.

III. Preliminaries

A. Network representation

We model the network as an *st-dag*, i.e., a directed acyclic graph with a unique source node s and a unique sink or destination node t. Given an st-dag G, its nodes represent the switches and its edges represent the network links. A path from s to t will be called an st-path. In an st-dag, every node lies on some st-path. We write $n' \prec n$ if node n' occurs before node n in some st-path. The degree of a node n in an st-dag G is the number of edges incident to n and will be denoted by $d_G(n)$. A weighted st-dag is an st-dag whose edges are labelled with a weight in $\mathbb{R}$. The weight of an edge e in st-dag H will be denoted by $w_H(e)$. An st-dag can be equivalently represented by a dictionary where the keys are the nodes and the value associated to a key n is a pair of lists: the list of immediate predecessors ($pred$) of n and the list of immediate successors ($succ$) of n. We will denote by $|S|$ the cardinality of set S.

B. Problem statement

The capacity of a node is the maximal size of a rule set that can be stored in the TCAM memory of the corresponding switch. Due to dynamic routing and the difficulty to anticipate which path is taken by a packet at some instant, the challenge behind our work is to distribute rules on all paths between source and destination in such a way the switches resources are optimized, e.g., the total rule space occupancy is minimized. For instance, we should avoid packets to encounter the same rules set more than once on a path. However this

is unavoidable in some cases, as shown by Fig. 1 where the same rule has to occur several times on the same path. Thus, to limit switches resources consumption we aim to minimize the space occupied by these rules, while covering all paths from source to destination.

Problem formulation: We are given an st-dag $G = (V, E)$, and a set of rules sets $F = \{f_1, f_2, ..., f_k\}$ of the same size (i.e., occupying the same memory space). We assume that every switch $v \in V$ has enough memory for only one rule set. The problem we consider is placing rules sets $f_1, f_2, ..., f_k$ on nodes in G such that 1) every path contains one occurrence of each of the rules sets and 2) the number of occupied switches is minimized. Note that the problem has no solution if the shortest path between s and t contains less than k switches. On the other hand, if the shortest path between s and t contains less than k switches then condition 1) is easily ensured.

In the rest of the paper, to simplify the presentation the terms *rule* and *rules set* are used interchangeably.

IV. Rules placement based on s-t mincut

A. Applying s-t mincuts to one rule placement

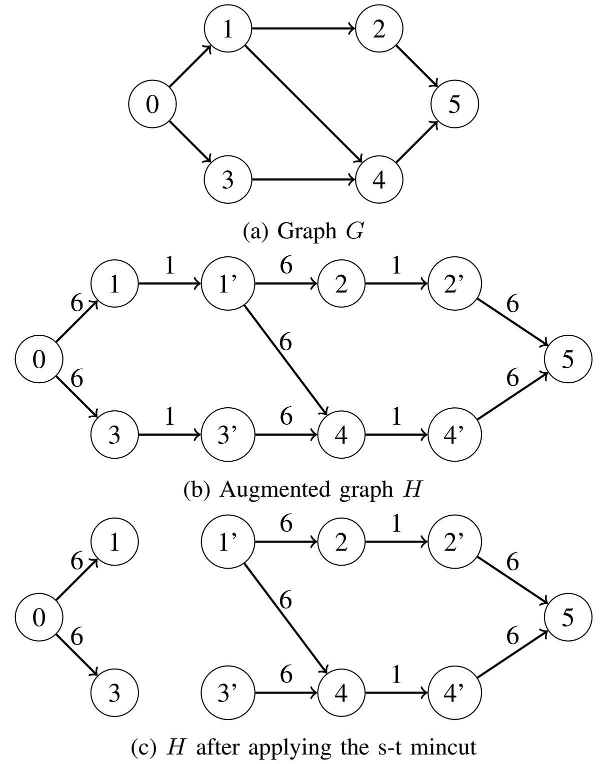
We will first consider the simple case of placing one rules set on an st-dag in such a way that every path from s to t contains at least one occurrence of the rules set and the number of rules set occurrence is minimized. For this case we can derive an *exact* polynomial time algorithm by an easy reduction to s-t mincut problem.

An s-t cut (or cut in short) of an st-dag is a set of edges C such that once they are removed, we get two disjoint subgraphs containing respectively the source and the destination.The value of a cut C is the number of edges in C for unweighted graphs and the sum of the weights of edges in C for weighted graphs. A minimum cut (or mincut in short) of an st-dag, is a cut with minimal value.

The minimum s-t cut problem can be solved using the *max-flow min-cut theorem*, which states that in a flow network, the maximum flow is equal to the minimum value of any minimum s-t cut [3]. There are several polynomial time algorithms that can be used to find the minimum s-t cut in a graph, such as the Ford-Fulkerson [3] and the Edmonds-Karp [4] algorithms. Both of them use the idea of augmenting paths to increase the flow in the network until it reaches the maximum flow.

So, in order to adapt the algorithm to our placement problem, we first duplicate each non terminal node. The duplication of node i is done by inserting another node i' after i. So, the successors of i are now successors for i' and the only successor for i is i' Further, the only predecessor of i' is i.

Then every edge of type (i, i') is assigned the weight 1 and the other edges are assigned a weight W_G, equal to the number of nodes in G. This is to ensure that a minimum st-cut will only contain edges of type (i, i'): we can always find an st-cut with all edges of type (i, i') (for instance by taking the set of (j, j') where j is a successor of s) and such cut has weight $< W_G$. Conversely, if an st-cut contains an edge not of type (i, i') then its weight is by construction $\geq W_G$. Hence once a minimum st-cut C has been computed, we know each element



(a) Graph G

(b) Augmented graph H

(c) H after applying the s-t mincut

Fig. 2: Steps of st-mincut based Alg. 1 for rules placement.

of C is an edge of type (i, i') and the rules are to be placed on every initial node i of such edges.

This procedure is illustrated in Fig. 2 and detailed in Algo. 1.

B. Extension to an arbitrary number of rules sets

Given k rules sets $f_1, f_2, \ldots, f_k$ to place on st-dag G one can easily generalize the algorithm for one rules set. We iterate the following operations while there is no failure at step 3 below,

1) assign weight W_G to every edge (v, v') such that v is occupied by some rules set f_l with $l \leq j - 1$;
2) find a minimum st-cut C_j of the resulting graph;
3) if the mincut has value $\geq W_G$ then exit with fail else add rules set f_j to every n such that $(n, n') \in C_j$.

The algorithm is polynomial time since it is essentially k times computation of st-mincut on graphs with the same set of nodes and bounded weights. However there are some cases where it does not provide an optimal solution, i.e., the number of occupied switches is not the minimum possible. Note that we have not required that the rules have to appear in the same given order on every st-path.

V. Greedy rules placement

We are going to present a greedy approach to solve the rules placement problem. The proposed algorithm is polynomial but its success is not guaranteed. This algorithm is also applied to compute the reward in the reinforcement learning approach presented in Section VI.

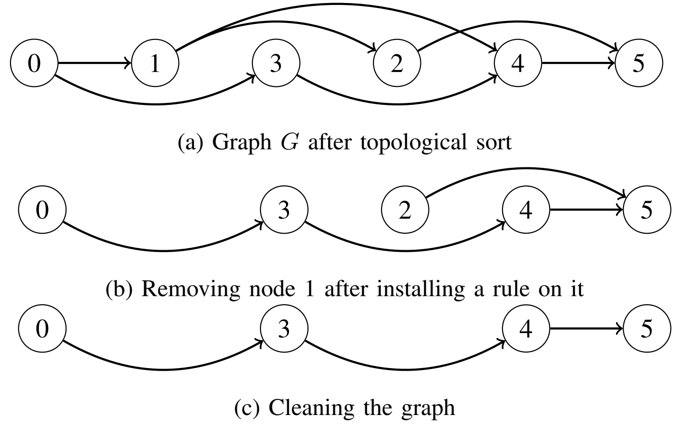
Algorithm 1: Rule placement with s-t mincut

Input: G, st-dag; s, source ; t, destination
Output: O, set of nodes where to place the rule
Algorithm:
$H \leftarrow G$
for $v \in nodes(H)$ **do**
  if $v \notin \{s, t\}$ **then**
    let v' be a new node:
    $H[v'][succ] \leftarrow H[v][succ]$
    for $j \in H[v][successor]$ **do**
      $H[j][pred] \leftarrow remove(v, H[j][pred])$
      $H[j][pred] \leftarrow append(v', H[j][pred])$
    end
    $H[v][succ] \leftarrow \emptyset$
    $H[v][succ] \leftarrow append(v', H[v][succ])$
    $H[v'][pred] \leftarrow append(v, H[v'][pred])$
  end
end
for $e \in edges(H)$ **do**
  if *e == (v,v')* **then**
    $w_H(e) \leftarrow 1$
  else
    $w_H(e) \leftarrow W_H$
  end
end
$C \leftarrow s - t_mincut(H)$
$O \leftarrow \{v|(v, v') \in C\}$
return O

An st-dag defines a partial order on nodes by taking the transitive closure of the relation $a \prec b$ if there is an edge from a to b. The source is the smallest element and the destination is the largest one. We can extend it to a linear order by applying a topological sorting. Once an st-dag G with N nodes has been sorted we can consider that the nodes are identified by non negative integers from 0, the source, to $N-1$ the destination.

Then, we traverse the graph once from left to right to compute for every node i the number of paths from source to i as shown in Algo. 2. A right to left traversal computes also the number of paths from i to destination. We do not detail this latter algorithm as it is a symmetric version of Algo. 2 derived by reversing network links, and replacing the source by the destination. Algo. 2 (resp., its symmetric) outputs $leftcount_G$ (resp., $rightcount_G$), a dictionary associating to each node the number of paths from this node to the source (resp., the destination). The number $pathcount_G(i)$ of paths from source to destination passing by node i is easily obtained as the product $leftcount_G(i) \times rightcount_G(i)$.

Finally after computing the number of paths for each node from s to t, we choose a node u that covers a maximal number of paths, i.e., with the maximum value in $pathcount_G$, and we install a rules set on it. Then we proceed to clean the graph by removing u with all its edges and also (iteratively) the non-destination nodes that remain without successor and the non-source nodes that remain without predecessor. After



(a) Graph G after topological sort

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

(c) Cleaning the graph

Fig. 3: Steps of applying our greedy heuristic based algorithm for rules placement.

cleaning the graph, we repeat the same procedure until no node can be selected with $pathcount \neq 0$. This means that *all paths are covered by the rule*. This algorithm for one rule (or one rules set) is shown in Algo. 3. Fig. 3 illustrates the algorithm steps in the first iteration. We can easily prove the following proposition:

Proposition 1. *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.*

When we have k rules to distribute on the network, one has to iterate k times the same procedure but choosing nodes u that covers a maximal number of paths *and are not yet occupied by another rule*.

Algorithm 2: 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**
  $leftcount_H[i] \leftarrow 0$
end
$leftcount_H[0] \leftarrow 1$
for $i = 1$ *to* $N-1$ **do**
  for $v \in H[i][pred]$ **do**
    $leftcount_H[i] \leftarrow$
      $leftcount_H[i] + leftcount_H[v]$
  end
end
return $leftcount_H$

VI. Reinforcement learning for rule placement

Reinforcement Learning (RL) can be characterised by a method of trial and error in which an agent observes the current state s of the environment in order to take an action a

Algorithm 3: 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 H[v][pred] = ∅ then
            H ← remove(v, H)
        end
    end
    for v = node to N-1 do
        if H[v][succ] = ∅ then
            H ← remove(v, H)
        end
    end
    Compute pathcount_H
end
return O
```

and transition into a new state. The end of each interaction is marked by the agent receiving a reward r, which represents a numerical value that the agent aims to maximise by optimising its decision making in the long run [7]. Recent works have shown the interest of RL for the implementation and control of network functions, also to make these networks more autonomous [13]. Here, we rely on the *Q-learning* algorithm to distribute ACL rules in a network. Q-learning is a model-free reinforcement-learning algorithm that seeks to learn the optimal action-value function (Q-function) that estimates the expected cumulative reward for taking a particular action in a given state and following the optimal policy thereafter. The Q-values are updated at each step over a given number of iterative episodes. An episode is defined as a sequence of states and actions ending in a terminal state. In our work, an episode starts from an initial graph without any rules set installed on the nodes and ends when all paths are covered or when the maximum number of steps allowed in an episode has been reached.

Therefore we define the RL components as follow:

- **State:** the state represents the actual situation of the environment. The environment in our problem is the st-dag G with ACLs currently installed on it. The state is represented as a boolean matrix S with N rows and C columns, where N is the number of nodes in G and C is the number of rules sets. If rules set j is installed on node i then the element $S_{i,j}$ is equal to 1 and otherwise 0;
- **Action:** actions are the agent's methods to change states. The actions here are denoted by $A(node_i, rule_j)$, meaning that rules set j gets installed on node i. An action is valid if there is enough memory on the node to install the rule.
- **Reward:** a reward is an immediate feedback sent by the environment to evaluate the last action of the agent. Our goal is to distribute rules sets on all paths between source and destination in such a way the switches resources are optimized. So the reward should guide the agent where to place rules sets in order to reach our final goal. We increase the reward when placing a rules set on some node that covers one or several paths that were not covered before. Conversely we decrease the reward (get a penalty) when this action creates some redundancy among rules sets. We calculate the reward based on the greedy algorithm defined in Section V. So, the reward of the action $A(i,j)$ is:

$$reward = paths_{i,j} - occ_i \quad (1)$$

where $paths_{i,j}$ is the number of new paths that get covered due to this action, i.e., the paths from s to t that contain node i and did not contain the rules set j before this action. The term occ_i is the number of rules sets installed on node i, i.e., the memory occupancy of i. First, we calculate the path count as done in the greedy algorithm then the agent chooses an action either by exploitation or exploration. If this action is valid, the rule will be installed then the graph is cleaned and we update the path count. The agent goes on choosing actions until all paths are covered.

The agent uses an exploration-exploitation (ϵ-*Decay*) strategy to balance between trying out new actions (exploration phase with probability p_ϵ) and exploiting its knowledge (exploitation phase with probability $1 - p_\epsilon$) to maximize its reward and updates Q-values accordingly (equation 2). The ϵ-*Decay* strategy improves the performance of RL algorithms as it allows to explore the environment in the early stages of learning and then become more exploitative as it becomes more experienced (i.e., start with a large value for ϵ and decays after each episode).

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max(Q(s',a')) - Q(s,a)] \quad (2)$$

where s stands for the current state (current assignment of rules sets to switches) and s' is the next state of the agent after taking action a and r stands for the reward. Parameter α is the learning rate that determines how much the new estimate of the Q-value should overwrite the old estimate and γ is the discount factor that determines how much the future rewards should be discounted when estimating the expected cumulative reward.

Finally, the optimal action for each state is the one that maximizes the Q-value function for this state. Indeed, we consider the goal of maximizing the reward.

VII. EXPERIMENTS

A. Simulation setup

In our experiments, we have tested our three algorithms on five real-world network topologies available from Internet Topology Zoo [14]. The algorithms are implemented in Python and executed on a laptop computer with AMD Ryzen 7 PRO 5850U with Radeon Graphics 1.90 GHz CPU, 32 GB of RAM and last version of Windows 11 OS.

We compare the algorithms according to the *Occupancy* metric that denotes the percentage of the switches total capacity that is occupied by the rules sets: less occupancy is better.

For RL algorithm, we set γ, the discount factor, to 0.8 in order to place a greater emphasis on long-term rewards, which can lead to more far-sighted and strategic behavior and α, the learning rate, to 0.6 in order to make large updates to the Q-values, which can lead to faster convergence.

B. Results and discussion

In our evaluation, all switches have the same capacity (one rules set per switch) and *the number of rules sets is equal to the number of switches in the shortest paths*. TABLE II shows that rules placement solution based on s-t mincut overcomes the other two algorithms in term of occupancy percentage (memory occupancy), better utilization of limited resources, and time to get the solution. We have noticed that the greedy based solution has the same effect on resource utilization as the RL based solution (occupancy percentages are almost equal) but the greedy based solution is faster than RL. The large time observed in RL with respect to the other two approaches is due to the fact that RL is a model free algorithm starting with no knowledge (by choosing random action and observing the effect of this action on the environment) till it reaches the solution. Instead, we can observe that the greedy approach may fail to provide a solution (e.g., Agis and Bics topologies) when at some step there is no space to install additional rules sets.

		s-t mincut		Greedy		RL	
Graph	♯ of rules sets	Occupancy (%)	Time (s)	Occupancy (%)	Time (s)	Occupancy (%)	Time (s)
Sprint/11	3	55.56	0.0021	55.56	0.0087	55.56	621.32
Abvt/23	6	61.9	0.0062	71.43	0.022	71.43	6182.95
Agis/25	5	56.52	0.0061	-	-	78.26	2365.8
Bics/34	8	46.88	0.036	-	-	96.88	9156.37
NetworkUSA/35	7	45.45	0.03	48.48	0.036	51.52	8504.16

TABLE II: Memory occupancy and processing time of our rules placement algorithms on real-world network topologies.

We can also apply our algorithms to multiple sources and multiple destinations graphs by merging all sources to one virtual source and all destinations to one destination. Furthermore, our algorithms can be adapted to other device or function placement problems. It is worth to note that our distribution scheme responds to any switch failure due to the fact that the SDN controller can re-route traffic through alternatives paths to avoid the failed switch since all paths are already covered by the rules.

VIII. CONCLUSION

In this paper, we developed three algorithms for in-network rules placement and we compared their performance on real-world network topologies with respect to switches space occupancy. Our results show the efficiency of the s-t mincut approach in resources utilization, overcoming greedy and RL based algorithms. Although the greedy algorithm does not always guarantee a solution, it is generally faster than RL. As future work we plan to investigate the exact complexity of our placement problem. We will also consider other resource constraints than space such as bandwidth, energy consumption and rules priority. We believe that the RL approach will perform better than the other algorithms in multi-constraint situations and in some specific network topologies (e.g., fat tree topology). We aim to further link the learning process in RL algorithms with the graphs properties in order to learn an efficient policy.

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] A. Abboud, R. Garcia, A. Lahmadi, M. Rusinowitch, and A. Bouhoula. Efficient Distribution of Security Policy Filtering Rules in Software Defined Networks. In IEEE 19th International Symposium on Network Computing and Applications (NCA), pp. 1-10, 2020.

[2] C. R. Meiners, A. X. Liu, and E. Torng, Bit weaving: A nonprefix approach to compressing packet classifiers in TCAMs. IEEE/ACM Transactions on Networking, vol. 20, no. 2, pp. 488–500, April 2012.

[3] L. R. Ford Jr. et D. R. Fulkerson. Maximal flow through a network . Canadian Journal of Mathematics, vol. 8, 1956, p. 399–404.

[4] J. Edmonds, R. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM. 19 (2), pp. 248–264, 1972.

[5] P. C. Lekkas, Network Processors: Architectures, Protocols, and Platforms. New York, NY, USA: McGraw-Hill, 2003

[6] Y. Kanizo, D. Hay, and I. Keslassy. Palette: Distributing tables in software-defined networks. in 2013 Proceedings IEEE INFOCOM, pp. 545–549.

[7] N. N. Khumalo, O. O. Oyerinde and L. Mfupe. Reinforcement Learning-Based Resource Management Model for Fog Radio Access Network Architectures in 5G. in IEEE Access, vol. 9, pp. 12706-12716, 2021.

[8] Y. -W. Chang and T. -N. Lin. An Efficient Dynamic Rule Placement for Distributed Firewall in SDN. GLOBECOM 2020, IEEE Global Communications Conference, Taipei, Taiwan, 2020, pp. 1-6

[9] M. Jiménez-Lázaro, J. Berrocal and J. Galán-Jiménez. Deep Reinforcement Learning Based Method for the Rule Placement Problem in Software-Defined Networks. NOMS 2022, IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, 2022, pp. 1-4.

[10] X. -N. Nguyen, D. Saucez, C. Barakat and T. Turletti. Rules Placement Problem in OpenFlow Networks: A Survey,in IEEE Communications Surveys & Tutorials, vol. 18, no. 2, pp. 1273-1286, Secondquarter 2016.

[11] Cisco. [n.d.]. Software-Defined Networking and Network Programmability: Use Cases for Defense and Intelligence Communities. https://bit.ly/33SBtnK

[12] F. Yu, T.V. Lakshman, M.A. Motoyama, and R.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 (pp. 105-113).

[13] Nick Feamster and Jennifer Rexford. Why (and How) Networks Should Run Themselves. In Proceedings of the Applied Networking Research Workshop (ANRW '18), ACM, 2018.

[14] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," Selected Areas in Communications, IEEE Journal on, vol. 29, pp. 1765 –1775, october 2011.