2018 Doctoral Thesis

The Packet Classification Problem and its Solutions

Supervisor Prof. Ken TANAKA

Field of Information Sciences, Graduate School of Science, Kanagawa University

Student ID Number:201670146

Takashi HARADA

Contents

1 Introduction **1**

1.1 Research Background 1
1.2 Purpose of This Study 2
1.3 Studies on Classification 3
1.3.1 Policy and Configuration Checking 4
1.3.2 Rule List Optimization 6
1.3.3 Specialized Data Structures 7
1.4 Organization of This Thesis 10

2 Optimal Rule Ordering **11**

2.1 Relaxed ORO 11
2.2 RRO is **NP**–hard 16
2.3 Zero–Suppressed Binary Decision Diagrams 27
2.4 Computing The Number of Evaluated Packets 28
2.4.1 Complexity of Computing the Number of Evaluated Packets 28
2.5 Manipulating $M(r_i)$ and $E(\mathcal{R}_\sigma, i)$ via ZDDs 29
2.6 Rule Pairing Algorithm 32
2.6.1 Complexity Analysis 34
2.6.2 Experiments 35
2.7 Improving Reordering Methods based on [75] 41
2.7.1 Interchange Adjacent Rules 41
2.7.2 Interchange of Single Rule and Consecutive Rules 42
2.7.3 Adaptive Reordering Algorithm 44
2.7.4 Experiments 45

3 Optimizing Rule List **48**

3.1 Single Machine Job Sequencing Problem 48
3.2 Rule List Reconstruction Method via Inclusive Rules 49
3.2.1 Experiments 50
3.2.2 Difference between Single–Machine Job Sequencing Problem and RORO . 53
3.3 Rewriting Rules to Oriented Trees 54
3.3.1 Rewriting Rules 54

3.3.2 Merging Rules 54
3.3.3 Experiments 55

4 Determining Equivalence of the Rule List Policies 60
4.1 Determining Equivalence of Rule List Policies via ZDD 60
4.2 Determining Equivalence of Rule Lists with Multiple Actions 65
4.3 Experiments 66

5 Run-Based Trie 69
5.1 Simple Search 69
5.2 Decision Tree constructed from RBT 70
5.3 Decision Tree based on RBT with Pointers 72
5.3.1 RBT with Pointers 72
5.3.2 Decision Tree based on RBTwP 72
5.4 Cascaded Circular–RBT 75
5.4.1 Circular–RBT 75
5.4.2 Consecutive Ones Property and Circular Ones Property 78
5.4.3 Cascaded Circular–RBT 79
5.4.4 Experiments 81

6 Conclusions 85

A Simulated Annealing for RORO 97
A.1 Multi–Start Local Optimization (MSL) 97
A.2 Simulated Annealing 98
A.3 Experiments 98

B Research Achievement 101
B.1 Journals 101
B.2 Conferences 101
B.3 Technical Reports 101
B.4 Programs 102

Chapter 1

Introduction

1.1 Research Background

The number of malicious programs on the Internet has been growing at an alarming rate, and the attacks on various important agencies are increasingly complicated. These malevolent programs can result in unauthorized access to governments or companies, the spread of ransomware, and information leakage. With the rapid growth in the Internet of Things, the number of devices connected to the Internet has increased significantly, and unsecured equipment now poses critical problem as a gateway for distributed denial of service attacks. Packet classification is an effective countermeasure against such cyber–attacks. It is also the key technology for network management tasks such as quality of service (QoS), load balancing, and network function virtualization (NFV). Packet classifiers determines the behavior of incoming packets through a comparison with the operational classification policy. A classification policy is generally a list of classification rule.

The linear search classification algorithm assigns prior actions to each packet according to the classification policy. These actions are determined by comparing the packet header with classification rules until a match is found. Because the processing latency of packet classification is proportional to the number of rules, a large number of rules can result in serious communication delay. To solve this problem, several techniques for reconstructing the rule list have been developed, and specialized data structures and hardware solutions have been proposed.

The rapid growth of NFV [55] and software–defined networking (SDN) [54] has led to the need for a technique that efficiently classifies packets without specialized hardware such as ternary content addressable memory (TCAM) or field–programmable gate arrays [16, 39, 47, 69, 70]. In general, packet classifiers use five fields to classify packets: source address, destination address, source port number, destination port number, and protocol of the packet. These fields are represented as prefix/range patterns such as `133.72.*.*` and `0-65535`, and most existing algorithms only handle these types of patterns. In addition to these patterns, researchers have introduced *arbitrary bitmask patterns* like `*.72.*.141` to represent more complex fields [50, 57]. Clearly, a prefix pattern is a special case of an arbitrary bitmask pattern. In general, translation

from prefix/range patterns to arbitrary bitmask patterns is less efficient than that from arbitrary bitmask pattern to range patterns. As the virtual switch in the NFV environment specifies more fields, it is necessary to develop a packet classification technique based on arbitrary bitmask patterns. Therefore, developing a method with arbitrary bitmask rules is a worthy subject of study.

In addition to specialized data structure solutions, rule–list–based algorithms decrease the classification latency by reordering the rules or reconstructing the rule list [19, 21, 22, 29, 30, 32, 36, 56, 60, 61, 65, 66, 75–77]. In the former task, the rules are reordered according to weights representing the frequency of matching against packets, thus preserving the classification policy. The latter task involves constructing a rule list so that heavy rules are placed above light rules. As the rule list generated by such algorithms must retain the original classification policy, some method for checking whether the rule list satisfies the policy is required. Although there are various complex schemes for checking the network configuration, there is no simple algorithm for determining the equivalence of rule list policies. As most research on the problem of reordering rules or reconstructing rule lists does not confirm whether the resulting rule list retains the original policy, some proposed methods have no awareness of policy violations.

1.2 Purpose of This Study

This study has three main aims:

1. Formulate the problem of linear search packet classification accurately and develop an efficient algorithm;

2. Develop an algorithm for determining the equivalence of rule lists;

3. Propose specialized data structures that are independent of the number of (arbitrary bitmask) rules.

We discuss the above tasks individually. As described above, it is necessary to check whether the reordering or reconstructing rules retain the original policy.

For the second task, in the conventional ORO model, for overlap rules r_i and r_j, we can not place r_j ahead of r_i, even if those actions are the same. This condition is too strict, so Tanaka and Mikawa relaxed the constraint so that rules r_i and r_j can not be interchanged if they overlap and their actions are different. However, there is a case whereby interchanging such rules does not cause a policy violation. Aside from this problem, there is an issue with the dependency model, whereby interchanging overlap rules may cause some variation in the rule weights. Thus, most algorithms do not calculate objective values accurately. To solve these problems, we formulate this problem based on policy violations instead of binary relations on rules like *overlap* or *dependency*.

Following the growth of NFV, it is important develop an algorithm that is independent of the number of (arbitrary bitmask) rules. Although researchers have developed algorithms that

are independent of the number of rules or correspond to arbitrary bitmask rules, no algorithm covers both properties within practical memory constraints. Thus, in sections 5.3 and 5.4, we discuss fast classification and memory efficient algorithms.

1.3 Studies on Classification

Packet classification is a difficult task. The algorithms developed this task can largely be separated into distinct groups depending on their approach [14, 62, 79].

In [14], Chao classified existing algorithms into four categories: 1) Basic Data Structures, 2) Geometric Algorithms, 3) Heuristics, and 4) Hardware–based Algorithms; see Table 1.1. Taylor [79] divided them into four categories of 1) Exhaustive Search, 2) Decision Tree, 3) Decomposition, and 4) Tuple Space; see Table 1.2. Nagpal et al. [62] used a slightly narrower classification, with the categories of 1) Decision Tree, 2) Trie, 3) Geometrical, 4) Divide & Conquer, 5) Tuple Space, and 6) Hardware; see Table 1.3. These classifications are not exhaustive — there are many algorithms that do not appear in the above three taxonomies [8, 31, 49, 51, 52, 74, 82, 86, 87, 89].

Table 1.1: Taxonomy developed by Chao [14].

Approach	Algorithms
Basic Data Structure	Linear Search
	Hierarchical Trie [73]
	Set–Pruning Trie
	Grid of Tries [73]
Geometric Algorithms	Cross–Producting [73]
	2–D Classification Scheme [46]
	Area–Based Quadtree [12]
	Fat Inverted Segment Tree [20]
Heuristics	Recursive Flow Classification [25, 26]
	Hierarchical Intelligent Cuttings (HiCuts) [24]
	Tuple Space Search [72]
Hardware–Based Algorithms	Bitmap Intersection [46]
	Ternary CAMs

In contrast to the above researchers, we categorize packet classification algorithms based on what they do instead of how they do it. Figure 1.1 shows the resulting classification of existing algorithms. The problems addressed in this study are highlighted in gray. We roughly divide the algorithms into software– and hardware–based approaches, and then split them into static and dynamic. Software–based algorithms can be further classified into *Policy/Configuration Checking*, *Classification Rule List Optimization*, and *Specialized Data Structures*. Specialized data structures are of particular interest for *arbitrary bitmasks*, and we discuss the complexity of algorithms that are independent of the number of rules. Rule list optimization consists of rule

Table 1.2: Taxonomy developed by Taylor [79].

Approach	Algorithms
Exhaustive Search	Linear Search
	Ternary CAM
Decision Tree	Grid of Tries [73]
	Extended Grid–of–Tries (EGT) [5]
	Hierarchical Intelligent Cuttings (HiCuts) [24]
	Modular Packet Classification [85]
	HyperCuts [67]
	Extended TCAM (E–TCAM) [70]
	Fat Inverted Segment Tree [20]
Decomposition	Parallel Bit–Vectors (BV) [46]
	Aggregated Bit–Vector (ABV) [6]
	Cross–Producting [73]
	Recursive Flow Classification [25, 26]
	Parallel Packet Classification (P^2C) [83]
	Distributed Crossproducting of Field Labels (DCFL) [78]
Tuple Space	Tuple Space Search & Tuple Pruning [72]
	Rectangle Search [72]
	Conflict–Free Rectangle Search [84]
Caching [13, 15]	

list reconstruction and the reordering of rules, which can be formulated as Optimal Rule List (ORL) and Optimal Rule Ordering (ORO) problems, respectively. As the reordering condition in most conventional OROs is too strict, we divide ORO into that used in overlap models and *Relaxed* ORO (RORO). Most previous research on configuration checking considers the complicated problems of conflict analysis and configuration modeling and analysis. Although researchers who have tackled ORL and RORO need *policy checkers* that efficiently determine whether a reconstructed rule list maintains the original policy or not, there are no algorithm specifically designed for this problem. Thus, we consider this as a separate algorithm from those for configuration modeling and analysis. The above methods are detailed in the following sections. Of course, there are hybrid methods and algorithms that can not be classified by this diagram, such as decision diagram schemes [1, 10, 11, 37, 38, 58, 59].

1.3.1 Policy and Configuration Checking

Network configurations for access control, QoS, load balancing, and virtual private networks (VPNs) are the complex and error–prone. For example, checking reachability is an **NP**–hard problem [53]. As any misconfiguration may trigger a service stop or result in insecure transmission and, researchers have developed frameworks and validation techniques for network config-

Table 1.3: Taxonomy developed by Nagpal et al. [62].

Approach	Algorithms
Decision Tree	Hierarchical Intelligent Cuttings (HiCuts) [24]
	HyperCuts [67]
Trie	Hierarchical Trie [73]
	Set–Pruning Trie
	Grid of Tries [73]
Geometrical	Area–Based Quadtree [12]
	Fat Inverted Segment Tree [20]
	Grid of Tries [73]
Divide & Conquer	Lucent Bit Vector [46]
	Aggregated Bit–Vector (ABV) [6]
	Cross–Producting [73]
	Recursive Flow Classification [25, 26]
Tuple Space	Tuple Space Search & Tuple Pruning [72]
Hardware	Ternary CAMs
	Bitmap Intersection [46]

Table 1.4: Taxonomy for algorithms that do not appear in [14, 62, 79].

Approach	Algorithms
Decision Tree	Smart Split [31]
	Boundary Cutting [52]
	EffiCuts [82]
	HybridCuts [49]
	Adaptive Grouping Factor(AGF) [74]
	Independent Sub-Rule(ISR) Leaf Structure [74]
Trie	Quad–Trie [51]
	Clustering Boundary Cutting [8]
Tuple Space	Partition Sort [89]
Geometrical	CAching in Buckets (CAB) [87]
	Adaptive Wildcard Rule Cache Management [86]

urations [2, 3, 7, 18, 28, 33, 37, 38, 42, 43, 53, 88, 91, 92].

In addition to the network configuration, anomalies in the classification policy of an individual rule list, such as *Shadowing*, *Generalization*, *Correlation*, and *Redundancy*, are defined and analyzed [4, 28, 35, 91]. Most of these algorithm use the binary decision diagram (BDD) [1, 10, 11] and can be modified to determine the equivalence of the rule list policies. However, as the classification rule list specifies only parts of header spaces, zero–suppressed BDD (ZDD) [58, 59] is more efficient than BDD for sparse combination set.

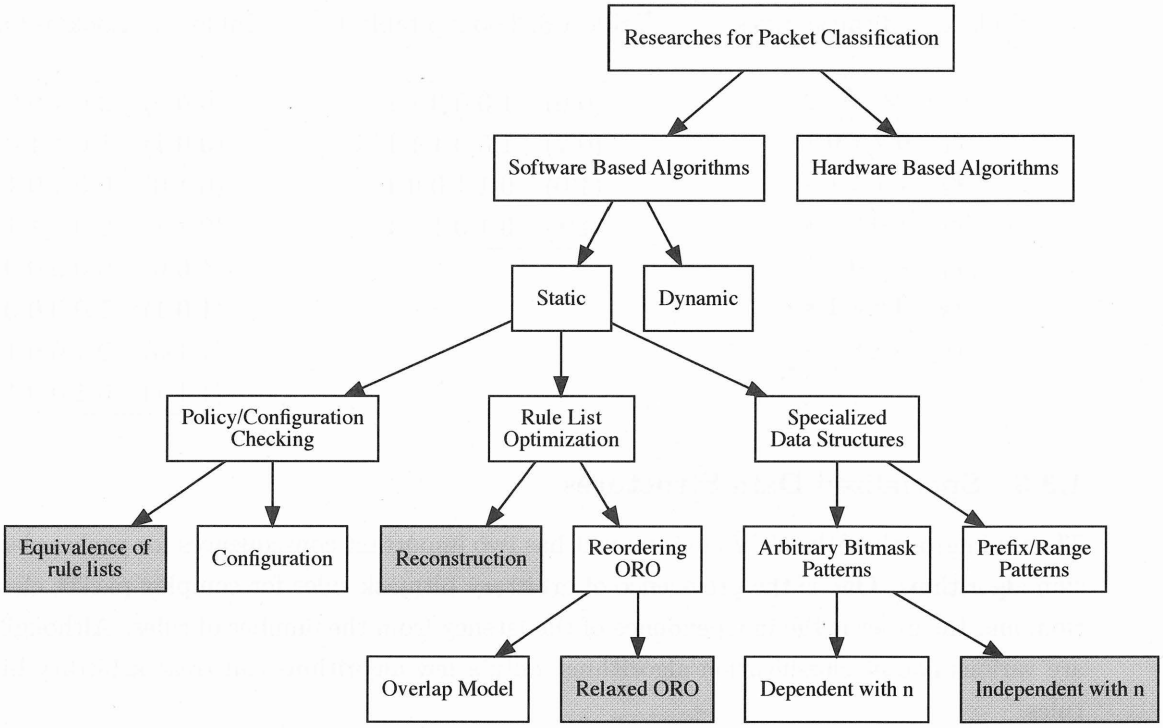

Figure 1.1: Research classification overview.

1.3.2 Rule List Optimization

To reduce the packet classification latency, packet classification is modeled and formulated as an optimization problem. Researchers have developed various rule reordering algorithms [19,22,29, 30,32,36,56,60,61,65,66,75–77], most of which are designed for the overlap model [19,22,29,30, 32,61,77]. In the overlap model, for two different rules that match the same packet, the posterior rule cannot be placed higher than the prior rule. However, even if rules r_i and r_j match the same packet p, r_j can be placed before r_i when the actions of r_i and r_j are the same. Thus, Tanaka and Mikawa introduced the *dependency model* [36,65,66,75,76], whereby even if rules r_i and r_j overlap, r_j can be placed ahead of r_i if these actions are the same. Although several algorithms have been developed for this model [21, 36, 56, 60, 65, 66, 75, 76], variations in the rules weights mean that most do not accurately calculate the objective value. This phenomenon is discussed in the following section. In contrast to these algorithms, Misherghi et al. [60] formulated ORO as an integer programming problem that accounts for the above problems, and Fuchino et al. [21] proposed a fast rule reordering algorithm. The method presented [21] is discussed in detail in Section 2.6.

Table 1.5: Bitmask rules.

Filter $\mathcal{R}$	
r_1	$0 * * 0\ 1$
r_2	$1 * 1\ 1 *$
r_3	$1\ 0\ 0 * *$
r_4	$* 1\ 0 * 1$
r_5	$0 * * 1 *$
r_6	$* * * * *$

Table 1.6: Lookup table 1.

(0 0)	1 0 0 0 1 1
(0 1)	1 0 0 1 1 1
(1 0)	0 1 1 0 0 1
(1 1)	0 1 0 1 0 1

Table 1.7: Lookup table 2.

(0 0 0)	0 0 1 0 0 1
(0 0 1)	1 0 1 1 0 1
(0 1 0)	0 0 1 0 1 1
(0 1 1)	0 0 1 1 1 1
(1 0 0)	0 0 0 0 0 1
(1 0 1)	1 0 0 0 0 1
(1 1 0)	0 1 0 0 1 1
(1 1 1)	0 1 0 0 1 1

1.3.3 Specialized Data Structures

The rapid growth of the NFV environment has two important consequences for packet classification algorithms. One is the prevalence of arbitrary bitmask rules for complex packet classification, and the other is the independence of the latency from the number of rules. Although there are various packet classification algorithms, only a few algorithms can treat arbitrary bitmask rules.

Kobayashi et al. proposed an algorithm based on the matching order of bits [44,45]. For each internal node v, they add the highest priority $priority(v) \in \{1, \ldots, \}$ to the sub–trie rooted at v. This allows us to traverse sub–tries that have high priority and achieve fast packet classification. Grouper [50] generates t lookup tables from an input rule list and uses them to classify a packet. Each lookup table consists of $2^{\lfloor w/t \rfloor}$ or $2^{\lceil w/t \rceil}$ rows and $\lfloor w/t \rfloor$ or $\lceil w/t \rceil$ columns. Each lookup table takes a sub–bitstring of length $\lfloor w/t \rfloor$ or $\lceil w/t \rceil$ and returns an n–length bitmap indicating which of the n rules match the sub–bitstring. Grouper classifies a packet as follows: It divides a packet into t groups and applies t lookup tables to those sub–bitstrings to find the t bitmaps. Intersecting all bitmaps indicates which of the n rules match the packet. The rule corresponding to the left–most 1 bits in the final bitmap is the highest priority rule for the packet. For example, given the rule list $\mathcal{R}$ in Table 1.5 and $t = 2$, Grouper generates the lookup tables in Tables 1.6 and 1.7. For instance, packet $p = 01010$ is classified as follows: Applying lookup tables 1 and 2 to 01 and 010 gives bitmaps 100111 and 001011. Intersecting them generates 000011, and so the highest priority rule is r_5.

The multi–valued decision diagram (MDD) [71] is a data structure that can be used for manipulating a function $f : \{0,1\}^w \rightarrow \{0, 1, \ldots, n\}$. The MDD for a function f is obtained by applying reduction rules to a binary decision tree representing f. The deletion of a redundant node and sharing of an identical node are illustrated in Figures 1.2 and 1.3. For the rule list in Table 1.5, Figure 1.4 shows the MDD. Circles and boxes denote non–terminal and terminal nodes, respectively. A numeral i associated with a non–terminal node represents a Boolean variable of a function. Non–terminal nodes have edges with values of $0 \ldots, m$. In Figure 1.4, m

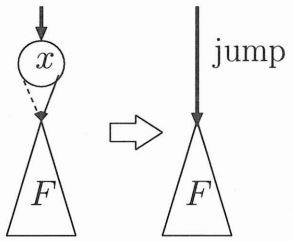

Figure 1.2: Node deletion.

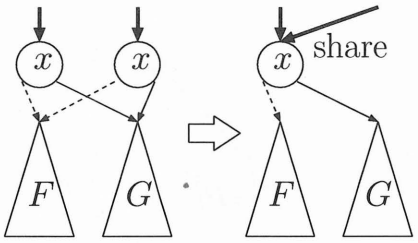

Figure 1.3: Node sharing.

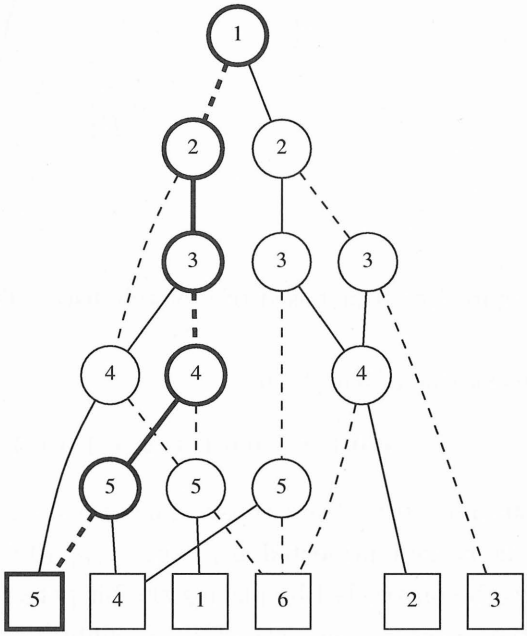

Figure 1.4: MDD for rule list in Table 1.5.

is 1; 0 and 1 edges from a non–terminal node with a numeral i represent that the ith variable takes a value of 0 and 1, respectively. Terminal nodes denote the value of a function. By traversing from the root node to a terminal node according to the bitstring of a packet, we can obtain the rule number of the highest priority rule. For instance, a packet (bitstring) 01010 traverses the heavy arrows in Figure 1.4 and reaches the terminal node labeled 5.

Mikawa et al. proposed a data structure called a run–based trie (RBT) [57]. They define a run as a bitstring of maximal length and that does not contain any wild–cards. A run is defined as follows:

Definition 1.3.1. *(run form) Let $r_i \in \{0,1,*\}^w$ be a bitmask rule of length w. A substring $b_i b_{i+1} \cdots b_j$ $(1 \leq i \leq j \leq w)$ of r that satisfies the following two conditions is called a run,*

i) $b_k = 0 \vee b_k = 1 \quad (i \leq k \leq j)$

ii) $(i \geq 2 \Rightarrow b_{i-1} = *) \ \wedge \ (j \leq w-1 \Rightarrow b_{j+1} = *)$.

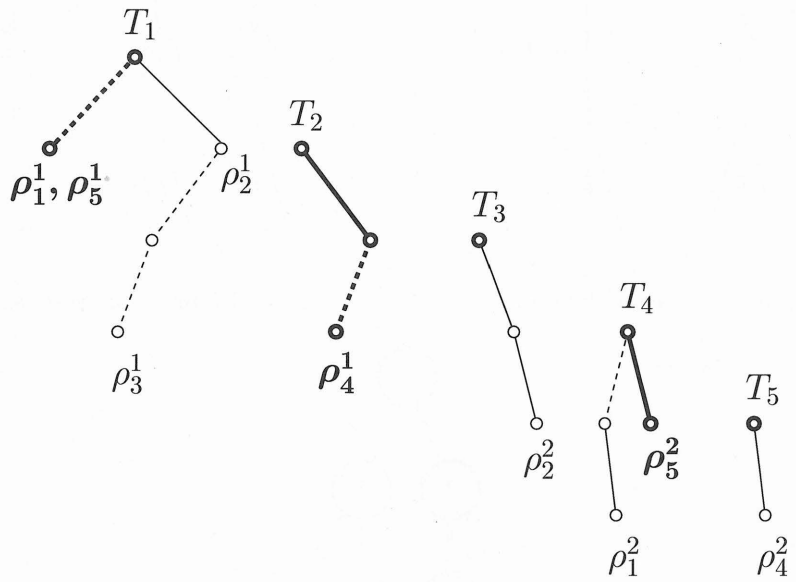

Figure 1.5: Run–based trie for rule list in Table 1.5.

For instance, a bitmask rule of length 16

$$* * 0 1 * * 0 0 1 * * * 1 0 1 0$$

consists of 3 runs. 01, 001, and 1010. These runs begin at the third, 7th, and 13th bits in the rule, respectively. Runs in rule r_i are represented as $\rho_i^1, \rho_i^2, \ldots, \rho_i^k$ $(0 \leq k \leq \lceil w/2 \rceil)$. RBT consists of w tries $T_1, T_2, \ldots, T_w$, each constructed by placing the bit pattern of the run beginning at the k–th bit of $r_i \in \mathcal{R}$ on the corresponding path of T_k. In addition, we mark ρ_i^j on the path if the run is the j–th run of r_i. RBT for the rule list in Table 1.5 is shown in Figure 1.5. A simple RBT search [57] traverses tries $T_1, T_2, \ldots, T_w$ with the bit patterns of the packet beginning at the k–th bit, and collects the runs that match the pattern. The matched rules from the collected runs are then calculated, and the highest priority rule in the matched rules is returned. If there are no matching rules, the default rule r_n is returned. For example, packet 01010 traverses the heavy lines in Figure 1.5 and collects runs ρ_1^1, ρ_5^1, ρ_4^1, and ρ_5^2. Because 01010 only matches rule r_5, the highest priority rule for 01010 is r_5.

Mikawa et al. also proposed decision tree algorithm constructed from RBT [57]. Because the patterns of the runs collected for each trie T_i in the RBT search are limited, they enumerate the patterns as $S_1, S_2, \ldots, S_w$ and take the Cartesian product $S_1 \times S_2 \times \cdots \times S_w$. The decision tree reflects the structure of this process. Each path from the root to a leaf of the decision tree is equivalent to a search path obtained by traversing the RBT from T_1 to T_w. Computing the highest priority rule for each path on the decision tree in advance, we can determine the highest priority rule by traversing the decision tree using RBT.

Table 1.8 shows the time and space complexities of algorithms for arbitrary bitmask rules, where w is the rule length and n is the number of rules. The algorithms at lines 7 and 8 In Table

Table 1.8: Comparison of various packet classification schemes with arbitrary bitmask rules.

Algorithm	Worst–case Time	Worst–case Space
Linear Search	$O(nw)$	$O(nw)$
Grouper [50]	$O(tn/w)$	$O(2^{w/t} \cdot tn)$
MDD [71]	$O(w)$	$O(2^w)$
RBT Search [57]	$O(nw + w^2)$	$O(nw)$
RBT Decision Tree [57]	$O(w^2)$	$O(n^w)$
MOB [44]	$O(nw)$	$O(nw^2)$

1.8 are discussed in the following sections. From Table 1.8, it is apparent that only MDD [71] and the RBT decision tree [57] are independent of the number of rules. As these two algorithms can consume a lot of memory, it is vital to develop a memory–efficient algorithm that is independent of the number of (arbitrary bitmask) rules.

1.4 Organization of This Thesis

This thesis consists of two main parts. Chapters 2–4 focus on problems related to rule lists and Chapter 5 discusses fast packet classification techniques based on specialized data structures.

In Chapter 2, we introduce the conventional optimization problem of *optimal rule ordering* and highlight its defects. To model the latency caused by classification exactly, we introduce an optimization problem called *relaxed optimal rule ordering* and prove that this problem is **NP**–hard. Furthermore, the counting problem related to RORO is defined and we show that this problem is #**P**–complete complexity. We propose algorithms for these problems.

Chapter 3 formulates an optimization problem constructs the optimal rule list so as to minimize the classification latency. We then propose reconstruction algorithms for rule lists based on the feasible property that the optimal rule reordering problem is solvable in polynomial time.

The algorithms for reordering rules and reconstructing rule lists should ensure that the resulting rule list maintains the classification policy. In Chapter 4, to determine whether those algorithms satisfy this property, we propose an algorithm determining the equivalence of the rule list policies.

In sections 5.3 and 5.4, we present algorithms that classify packets in constant time, independent of the number of rules in the rule list.

Finally, Chapter 6 summarizes this paper and discusses tasks for future work.

Chapter 2

Optimal Rule Ordering

Packet classification is achieved by performing a linear search on a classification rule list. A larger number of rules will result in a longer communication delay. To solve this problem, the packet classification problem can be generalized as optimal rule ordering (ORO), which aims to find the rule ordering that minimizes the latency due to packet classification. The decision problem corresponding to ORO is known to be **NP**–complete [29], and various heuristic methods have been developed [19, 22, 29, 61, 75–77].

In most ORO problems, for two different rules that match the same packet, the posterior rule cannot be placed higher than the prior rule if the packet classification policy is to hold. However, there are many cases in which we can actually interchange such rules without any policy violation. Even if rules r_i and r_j match the same packet p, r_j can still be placed before r_i when (1) the actions of r_i and r_j are the same [75, 76] or (2) there is a rule r_k matching p that is placed before r_i [60]. According to this property, the classification latency can become lower than that of the conventional model by relaxing the condition of interchanging rules. Based on this, we formulate an optimization problem that aims to exhaustively search for the most efficient packet classification. We refer to this problem as *relaxed ORO (RORO)*. In RORO, interchanging rules may vary the number of packets for some rules, and so both a rule list and a packet arrival distribution are required as inputs. In this chapter, we prove the computational complexity of a decision problem corresponding to RORO. We refer to this problem as *relaxed rule ordering (RRO)*. RRO is shown to be **NP**–hard. The formulation is a novel foundation for developing the heuristics for an optimization problem that minimizes the classification latency.

Section 2.1 defines RORO and presents some terminology. In section 2.2, we prove that RRO is **NP**–hard. A rule pairing algorithm, and an algorithm based on the method [75] are presented in sections 2.6 and 2.7, respectively. The effectiveness of these algorithms is also confirmed.

2.1 Relaxed ORO

In this section, we formalize the process of packet classification and define the RORO problem.

Packet classification on network devices is modeled as shown in Fig. 2.1. Each rule consists

Table 2.1: The rule list.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R}, i)\rvert_{\mathcal{U}}$
$r_1^P = * \; 0 \; * \; 1$	4
$r_2^P = 0 \; 0 \; 0 \; 0$	1
$r_3^P = 0 \; * \; 0 \; 0$	1
$r_4^D = 0 \; * \; 1 \; *$	3
$r_5^P = * \; 1 \; * \; 1$	3
$r_6^P = * \; * \; * \; 1$	0
$r_7^D = * \; * \; * \; *$	4
$L(\mathcal{R}, \mathcal{U}) = 60$	

Table 2.2: Reordering according to (2.2).

Filter $\mathcal{R}_\sigma$	$\lvert E(\mathcal{R}_\sigma, i)\rvert_{\mathcal{U}}$
$r_1^P = * \; 0 \; * \; 1$	4
$r_4^D = 0 \; * \; 1 \; *$	3
$r_5^P = * \; 1 \; * \; 1$	3
$r_3^P = 0 \; * \; 0 \; 0$	2
$r_2^P = 0 \; 0 \; 0 \; 0$	0
$r_6^P = * \; * \; * \; 1$	0
$r_7^D = * \; * \; * \; *$	4
$L(\mathcal{R}_\sigma, \mathcal{U}) = 51$	

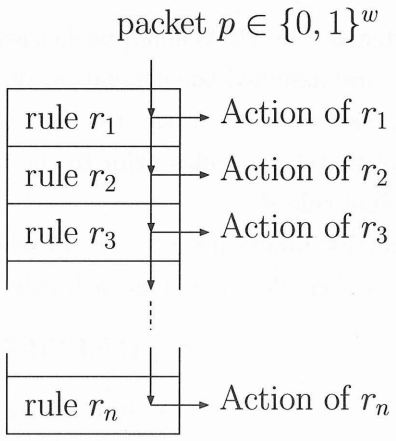

Figure 2.1: Packet classification model.

of a rule number $i \in \mathbb{N}$, a condition string on $\{0, 1, *\}^w$, and an evaluation type $\{P, D\}$, where w is the length of a condition and $*$ is a *don't care term* denoting that any bit can be matched. A rule list consists of n rules. P and D denote whether the device accepts or denies incoming packets, respectively. A packet p is a bit string of length w, i.e., $p \in \{0, 1\}^w$. A rule is defined as shown in (2.1). An example of a rule list is provided in Table 2.1.

Definition 2.1.1. *(rule form)*

$$r_i^e = b_1 b_2 \cdots b_w, \;\; b_k \in \{0, 1, *\}, \;\; e \in \{P, D\} \tag{2.1}$$

Table 2.3: Latency under (2.4).

Filter $\mathcal{R}$	$\lvert E(\mathcal{R}, i)\rvert_{\mathcal{F}}$
$r_1^P = * \; 0 \; * \; 1$	4
$r_2^P = 0 \; 0 \; 0 \; 0$	20
$r_3^P = 0 \; * \; 0 \; 0$	10
$r_4^D = 0 \; * \; 1 \; *$	0
$r_5^P = * \; 1 \; * \; 1$	9
$r_6^P = * \; * \; * \; 1$	0
$r_7^D = * \; * \; * \; *$	13
$L(\mathcal{R}, \mathcal{F}) = 197$	

Table 2.4: Latency under (2.2) and (2.4).

Filter $\mathcal{R}_\sigma$	$\lvert E(\mathcal{R}_\sigma, i)\rvert_{\mathcal{F}}$
$r_1^P = * \; 0 \; * \; 1$	4
$r_4^D = 0 \; * \; 1 \; *$	0
$r_5^P = * \; 1 \; * \; 1$	9
$r_3^P = 0 \; * \; 0 \; 0$	30
$r_2^P = 0 \; 0 \; 0 \; 0$	0
$r_6^P = * \; * \; * \; 1$	0
$r_7^D = * \; * \; * \; *$	13
$L(\mathcal{R}_\sigma, \mathcal{F}) = 229$	

A set of packets is denoted by $\mathcal{P}$. When a packet arrives at a network device, it is compared with each rule in order, and assigned the evaluation type of the first matching rule. Because all packets must match at least one rule in the rule list, a default rule is added to the end of the list. If a packet does not match any rules prior to the nth rule, it is automatically assigned the evaluation type of the final rule r_n^e.

An ordering is a bijective function $\sigma : [n] \to [n]$, where $[n] = \{1, 2, \ldots, n\}$. In this paper, we denote an ordering as $\sigma = (x_1, x_2, \ldots, x_n)$ as each rule k moves to x_k. For example,

$$\sigma = (1\,5\,4\,2\,3\,6\,7) \tag{2.2}$$

signifies $1 \to 1$, $2 \to 5$, $3 \to 4$, $4 \to 2$, $5 \to 3$, $6 \to 6$ and $7 \to 7$. In this case, $\sigma(2) = 5$ means that the rule in the second position moves to the fifth position, and $\sigma^{-1}(5) = 2$ means that the rule that moved to the fifth position was previously in the second position. Informally, the domain and codomain of the function σ represent a set of rule numbers and a set of positions for rules. Let $\mathcal{R}$ be a rule list and σ be an ordering. $\mathcal{R}_\sigma$ denotes the rule list reordered by σ. With the above ordering σ, the rule list

$$\mathcal{R} = [r_1^e, r_2^e, r_3^e, r_4^e, r_5^e, r_6^e, r_7^e]$$

is reordered as follows:

$$\mathcal{R}_\sigma = [r_1^e, r_4^e, r_5^e, r_3^e, r_2^e, r_6^e, r_7^e].$$

We use $\mathcal{R}(p)$ to denote an evaluation type for p as the classification result. For instance, given the rule list $\mathcal{R}$ in Table 2.1, $\mathcal{R}(0111) = D$. The rule list in Table 2.1 denotes the function $f : \{0,1\}^4 \to \{P, D\}$ given in (2.3).

$$
\begin{array}{llll}
0000 \mapsto P, & 0001 \mapsto P, & 0010 \mapsto D, & 0011 \mapsto P, \\
0100 \mapsto P, & 0101 \mapsto P, & 0110 \mapsto D, & 0111 \mapsto D, \\
1000 \mapsto D, & 1001 \mapsto P, & 1010 \mapsto D, & 1011 \mapsto P, \\
1100 \mapsto D, & 1101 \mapsto P, & 1110 \mapsto D, & 1111 \mapsto P
\end{array}
\tag{2.3}
$$

If there exists a packet p such that $\mathcal{R}(p) \neq \mathcal{R}_\sigma(p)$, we say that order σ violates the policy or that a policy violation occurs.

Let $id : [n] \to [n]$ be the identity ordering, i.e., $id(i) = i$ for all $i \in [n]$. $\mathcal{R}_{id}$ means that the rule list is not reordered and $\mathcal{R}_{id}$ equals $\mathcal{R}$. In the following, id is omitted from a rule list inscription when the order of the rule list is id.

Let $M(r_i)$ denote a set of packets that can match rule r_i^e, i.e., $M(r_i)$ is a set of binary sequences generated by changing each '$*$' on the condition of r_i^e to 0 or 1. For example, for r_5^P (Table 2.1),

$$M(r_5) = \{\ 0101, 0111, 1101, 1111\ \}.$$

As the evaluation type e is redundant for $M(r_i)$, e is omitted from the inscription of the set of packets that can match rule r_i^e as long as the absence of e causes no confusion.

Given a rule list $\mathcal{R}$ and an ordering σ, a set of packets evaluated by rule r_i is defined. This set is denoted as $E(\mathcal{R}_\sigma, i)$. Similarly to $M(r_i)$, e is omitted. For example, given the rule list in Table 2.1, the set of packets evaluated by rule r_5^P is expressed as

$$E(\mathcal{R}, 5) = \{\ 0101, 1101, 1111\ \}.$$

Note that $E(\mathcal{R}, 5)$ is different from $M(r_5)$. As the packet 0111 is evaluated by rule r_4^D, 0111 is not in $E(\mathcal{R}, 5)$.

Let $\mathcal{F} : \{0,1\}^w \to \mathbb{N}$ be a packet arrival frequency distribution and let $|\mathcal{P}|_\mathcal{F}$ denote $\sum_{p \in \mathcal{P}} \mathcal{F}(p)$. As an example, for $\mathcal{P} = \{\ 0101, 1101, 1111\ \}$ and $\mathcal{F}$ in (2.4),

$$|\mathcal{P}|_\mathcal{F} = |\{\ 0101, 1101, 1111\ \}|_\mathcal{F} = \mathcal{F}(0101) + \mathcal{F}(1101) + \mathcal{F}(1111) = 9.$$

$$
\begin{array}{llll}
0000 \mapsto 20, & 0001 \mapsto 0, & 0010 \mapsto 0, & 0011 \mapsto 3, \\
0100 \mapsto 10, & 0101 \mapsto 2, & 0110 \mapsto 0, & 0111 \mapsto 0, \\
1000 \mapsto 0, & 1001 \mapsto 1, & 1010 \mapsto 13, & 1011 \mapsto 0, \\
1100 \mapsto 0, & 1101 \mapsto 0, & 1110 \mapsto 0, & 1111 \mapsto 7
\end{array}
\tag{2.4}
$$

Given a packet arrival distribution $\mathcal{F}$, a rule list $\mathcal{R}$, and an order of rules σ, the number of packets evaluated by r_i under $\mathcal{F}$ can be defined. We denote this number as $|E(\mathcal{R}_\sigma, i)|_\mathcal{F}$ and call it the *weight of* r_i. For example, under the uniform distribution $\mathcal{U}$,

$$
\begin{array}{llll}
0000 \mapsto 1, & 0001 \mapsto 1, & 0010 \mapsto 1, & 0011 \mapsto 1, \\
0100 \mapsto 1, & 0101 \mapsto 1, & 0110 \mapsto 1, & 0111 \mapsto 1, \\
1000 \mapsto 1, & 1001 \mapsto 1, & 1010 \mapsto 1, & 1011 \mapsto 1, \\
1100 \mapsto 1, & 1101 \mapsto 1, & 1110 \mapsto 1, & 1111 \mapsto 1
\end{array}
\tag{2.5}
$$

the number of evaluated packets r_3 in Table 2.2 is $|E(\mathcal{R}_\sigma, 3)|_{\mathcal{U}} = 2$.

Considering that the comparison of a packet with a rule has latency 1, under the order of rules σ and the packet arrival distribution $\mathcal{F}$, the classification latency $L(\mathcal{R}_\sigma, \mathcal{F})$ of rule list $\mathcal{R}$ is defined as follows:

Definition 2.1.2. *(Classification latency)*

$$L(\mathcal{R}_\sigma, \mathcal{F}) = \sum_{i=1}^{n-1} i|E(\mathcal{R}_\sigma, \sigma^{-1}(i))|_{\mathcal{F}} + (n-1)|E(\mathcal{R}_\sigma, \sigma^{-1}(n))|_{\mathcal{F}}. \tag{2.6}$$

In other words, latency can be expressed as

$$L(\mathcal{R}_\sigma, \mathcal{F}) = \sum_{i=1}^{n} |E(\mathcal{R}_\sigma, i)|_{\mathcal{F}} \cdot \sigma(i) - |E(\mathcal{R}_\sigma, \sigma^{-1}(n))|_{\mathcal{F}}$$

in terms of the rule number. As a packet is not compared with the last rule $r_{\sigma^{-1}(n)}$, the second term is necessary. For example, the classification latency for the rule list in Table 2.1 with uniform distribution $\mathcal{U}$ is expressed as

$$L(\mathcal{R}, \mathcal{U}) = 1{\cdot}4 + 2{\cdot}1 + 3{\cdot}1 + 4{\cdot}3 + 5{\cdot}3 + 6{\cdot}0 + 6{\cdot}4 = 60.$$

By reordering the rules in Table 2.1 according to σ while maintaining the classification policy denoted by f, the latency decreases from 60 to

$$L(\mathcal{R}_\sigma, \mathcal{U}) = 1{\cdot}4 + 2{\cdot}3 + 3{\cdot}3 + 4{\cdot}2 + 5{\cdot}0 + 6{\cdot}0 + 6{\cdot}4 = 51.$$

As described above, by reordering the rules, the classification latency of a rule list can be decreased. In addition, for each rule r_i, reordering the rules may vary the number of packets evaluated by r_i. Therefore, the optimal order of rules actually varies according to the packet arrival distribution. For example, the rule weights and latency for the packet arrival distribution $\mathcal{F} : \{0,1\}^4 \to \mathbb{N}$ given by (2.4) are listed in Table 2.3. Comparing them with the results for σ and $\mathcal{F}$ in Table 2.4, we find that the latency increases from 197 to 229. In this way, the optimal order for rules is dependent on the packet arrival distribution. To clarify the optimal order for rules, we now define RORO with a given packet arrival distribution.

Definition 2.1.3. *(RORO)*

Input: *Rule list $\mathcal{R}$ and packet arrival distribution $\mathcal{F}$*
Output: *Order of rules σ that minimizes $L(\mathcal{R}_\sigma, \mathcal{F})$*
s.t. $\forall p \in \mathcal{P},\ \mathcal{R}(p) = \mathcal{R}_\sigma(p)$

In the above definition, $\forall p \in \mathcal{P},\ \mathcal{R}(p) = \mathcal{R}_\sigma(p)$ means that order σ does not violate the policy represented by rule list $\mathcal{R}$, i.e., order σ is a feasible solution.

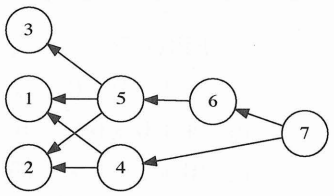

Figure 2.2: The precedence graph $G_{\mathcal{R}}$ for $\mathcal{R}$ in Table 2.5.

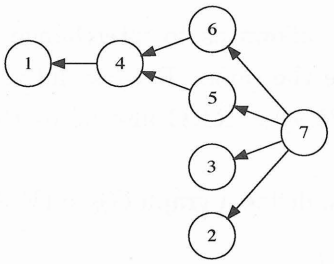

Figure 2.3: The precedence graph $G'_{\mathcal{R}}$ for $\mathcal{R}$ in Table 2.1.

2.2 RRO is NP–hard

In this section, we show that the decision problem corresponding to RORO, i.e., ***RRO*** is **NP**–hard. We define two decision problems for ORO and RORO in advance.

Definition 2.2.1. ***(RULE ORDERING (RO))***

Instance: *Rule list $\mathcal{R}$ and positive integer B*
Question: *Is there an order σ, s.t. $\sum_{i=1}^{n} w_i \cdot \sigma(i) < B$ and $\forall r_i, r_j \in \mathcal{R},\ O(r_j, r_i) \wedge i < j \Rightarrow \sigma(i) < \sigma(j)$,*

where $O(r_j, r_i)$ denotes that a packet matching both r_i and r_j exists and w_i is the weight of r_i. w_i corresponds to $|E(\mathcal{R}, i)|_{\mathcal{F}}$. Note that the above subscript i represents the rule number, not the position of the rule. If r_i and r_j hold for $O(r_j, r_i)$, we say that r_i and r_j overlap. For example, r_4^D and r_5^P overlap in Table 2.1, because the packet 0111 matches both r_4^D and r_5^P.

Definition 2.2.2. ***(RRO)***

Instance: *Rule list $\mathcal{R}$, packet arrival distribution $\mathcal{F}$, and positive integer B*
Question: *Is there an order σ, s.t. $L(\mathcal{R}_\sigma, \mathcal{F}) < B$ and $\forall p \in \mathcal{P}, \mathcal{R}(p) = \mathcal{R}_\sigma(p)$.*

Table 2.5: Rule list $\mathcal{R}$ of **RO**.

	Filter $\mathcal{R}$	w_i
r_1	$*\,1\,*\,0\,1\,0$	3
r_2	$*\,*\,0\,*\,0\,0$	5
r_3	$0\,*\,1\,0\,0\,*$	19
r_4	$1\,1\,*\,0\,*\,0$	37
r_5	$0\,1\,*\,0\,*\,*$	13
r_6	$0\,1\,0\,*\,*\,*$	29
r_7	$*\,*\,*\,*\,*\,*$	43

As opposed to **RO**, **RRO** allows us to interchange rules r_i and r_j when $O(r_j, r_i)$ holds and these evaluation types are the same. Because interchanging such rules may vary the set of packets $|E(\mathcal{R}, i)|_{\mathcal{F}}$ and $|E(\mathcal{R}, j)|_{\mathcal{F}}$, **RRO** also needs the packet arrival distribution $\mathcal{F}$ as an input.

For a rule list $\mathcal{R}$ in **RO**, we define a graph $G_{\mathcal{R}} = (V, A)$ as

$$\begin{aligned} V &= \{\ 1, 2, \ldots, n\ \} \\ A &= \{\ ki \mid i, k \in V, i < k, O(r_k, r_j), \\ &\qquad \neg\exists j \in V, i < j < k \wedge O(r_j, r_i) \wedge O(r_k, r_j)\ \}, \end{aligned} \tag{2.7}$$

and for a rule list $\mathcal{R}'$ in **RRO**, we define a graph $G'_{\mathcal{R}'} = (V', A')$ as

$$\begin{aligned} V' &= \{\ 1, 2, \ldots, n\ \} \\ A' &= \{\ ki \mid i, k \in V, i < k, D(r_k, r_i) \\ &\qquad \neg\exists j \in V, i < j < k \wedge D(r_k, r_j) \wedge D(r_j, r_i)\ \}, \end{aligned} \tag{2.8}$$

where $D(r_j, r_i)$ denotes that $O(r_j, r_i)$ holds and the evaluation types of r_i and r_j are different. For example, for the rule list $\mathcal{R}$ in Table 2.5, the graph $G_{\mathcal{R}}$ is shown in Fig. 2.2. For the rule list $\mathcal{R}$ in Table 2.1, the graph $G'_{\mathcal{R}}$ is shown in Fig. 2.3. In graph $G'_{\mathcal{R}}$ in Fig. 2.3 for the rule list in Table 2.1, because $D(r_7, r_5)$ holds and there is no rule r_j such that $D(r_7, r_j)$ and $D(r_j, r_5)$, there is edge $(7, 5)$. If we avoid the evaluation types of the rules in Table 2.1 and make graph G instead of G', there is no edge $(7, 5)$ in G, because we have r_6 such that $O(r_7, r_6)$ and $O(r_6, r_5)$.

To prove that RRO is **NP**–hard, we first present several lemmas.

Lemma 2.2.1. *The graph $G'_{\mathcal{R}}$ for a rule list $\mathcal{R}$ in **RRO** is two–colorable.*

Proof. The vertices of the graph $G'_{\mathcal{R}}$ can be divided into two sets, U and V, where the actions of rules corresponding to a vertex $u \in U$ and $v \in V$ are P and D, respectively. For all rules r_i^e and r_j^f corresponding to the vertices in $U(V)$, as those actions are the same, $D(r_j^f, r_i^e)$ does not hold. Thus, the sets U and V are independent sets and the graph $G'_{\mathcal{R}}$ is bipartite. If a graph G is a bipartite, the graph G is two–colorable. Thus, the graph $G'_{\mathcal{R}}$ for a rule list $\mathcal{R}$ in **RRO** is two–colorable. □

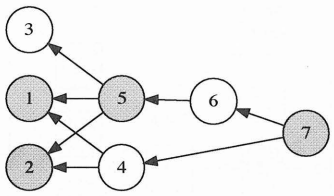

Figure 2.4: Coloring vertices from source vertex 7.

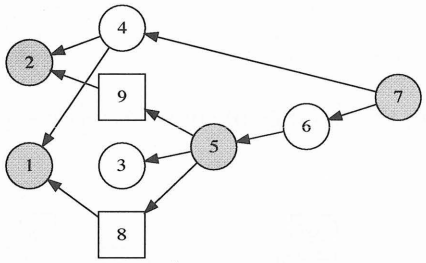

Figure 2.5: Inserting a vertex between same color vertices.

Lemma 2.2.2. *For a directed acyclic graph $G = (V, A)$ and a positive integer $k \leq |V|^2$, we can generate a bipartite graph $G' = (V', A')$ in $O(n^2 + nm)$, where the out–degree of a vertex $v' \in V' \cap V$ is $deg^+(v) + k$.*

Proof. We show that there is a method that takes graph G as input and outputs a graph such as G'.

1. Assign one of two colors to each vertex via a depth–first search, starting from the source vertex. For example, for the graph shown in Fig. 2.2, a colored graph is shown in Fig. 2.4.

2. Insert a vertex between vertices that have the same color. For the graph in Fig. 2.4, insert vertices 8 and 9, as shown in Fig. 2.5. The inserted vertices are boxed, and the original vertices are circled.

3. For each $v \in V$, insert vertices before v such that the number of squared vertices is k. For the graph in Fig. 2.5 and $k = 2$, inserting squared vertices results in the graph in Fig. 2.6.

As the complexity of the depth–first search is $O(n+m)$, the complexity for the step 1 is $O(n+m)$, where m is the number of edges of the input graph. Steps 2 and 3 insert at most k ($\leq n^2$) vertices and have $O(n^2 + nm)$ complexity. □

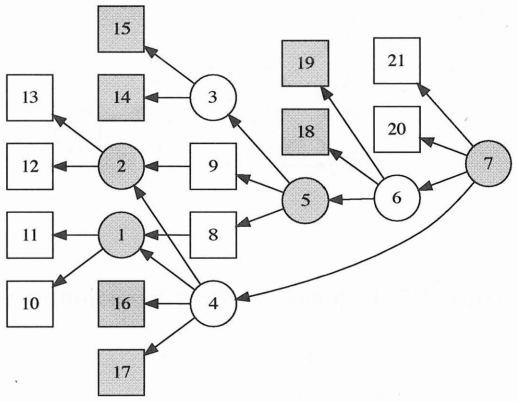

Figure 2.6: Adjusting the number of squared vertices preceding each circled node.

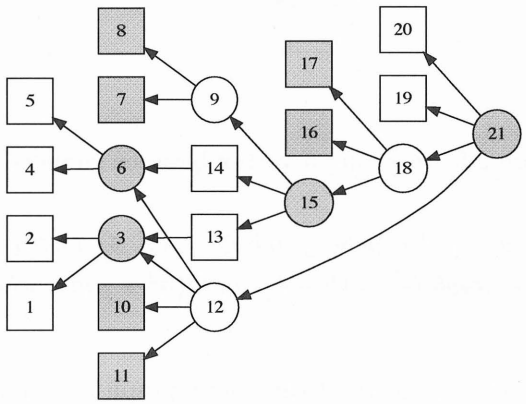

Figure 2.7: Renumbering.

Lemma 2.2.3. *We can construct a rule list $\mathcal{R}$ and a packet arrival distribution $\mathcal{F}$ from a weighted directed acyclic graph $G = (V, A)$ that is bipartite and has just one source vertex in $O(|V|^2 + |V||A|)$.*

Proof. We show that Algorithm 1 takes such a directed acyclic graph and returns a rule list and a packet arrival distribution.

Firstly, Algorithm 1 computes the longest distances for each vertex from the source vertex. In Algorithm 1, r_{u_c} denotes the condition of r_u. On lines 6–14, the algorithm makes rules whose longest distances are equal to 1. To be independent of these rules, the algorithm inserts '1' diagonally. In Table 2.6, r_{12}, r_{18}, r_{19}, and r_{20} make a 4×4 unit–like matrix. The binary operation ++ on lines 10, 11, 25, 26, 38, and 39 concatenates two strings. For example, "0010"

++ "10001" gives "001010001".

On lines 15–32, the algorithm makes rules whose longest distances are equal to k. As stated above, the rule is first constructed from the rules depending on r_i using $\oplus$. The binary operation $\oplus$ on line 20 in Algorithm 1 takes two strings, x and y, on $\{0, 1, *\}$ and returns a string z as follows:

$$z_i = \begin{cases} x_i & (\text{if } y_i = \epsilon \ \vee \ x_i = y_i) \\ y_i & (\text{if } x_i = \epsilon \ \vee \ x_i = y_i) \\ * & (\text{otherwise}), \end{cases}$$

where ϵ is an empty string. For example, let

$$\begin{aligned} x &= 1\ 0\ 0\ *\ 1\ *\ 0\ *\ , \\ y &= 0\ 1\ 0\ 0\ 1\ 1\ \epsilon\ \epsilon\ , \end{aligned}$$

$z = x \oplus y = * * 0 * 1 * 0 *$.

For instance, on lines 15–32, we show how r_3 is made. As the longest distance of r_3 from the source vertex r_{21} is 4, before making r_3, those rules whose longest distance is 3 are already made. Thus, rules r_{12} and r_{13} depending on r_3 are already made. c ("") $\oplus r_{12_c}$ and $c(r_{12_c}) \oplus r_{13_c}$ on line 20 generate the string $**0000100010$. As there are 4 rules whose longest distance is 4, the 4×4 unit–like matrix is added for rules r_3, r_6, r_7, r_8 on lines 34–41. As a result, the string $**00001000101000$ is generated.

Next, similar to the lines 6–14, '1's are inserted diagonally.

On lines 34–41, the algorithm adds a matrix with '$*$'s on the main diagonal and '0's elsewhere. From this matrix, we can make $E(\mathcal{R}', i) \neq \emptyset$ and $|E(\mathcal{R}', i)|_{\mathcal{F}} = 0$ for any rule list and packet arrival distribution.

Algorithm 1 makes the rule list in Table 2.6 from the graph in Fig. 2.7. We change the font of the condition of the rules in Table 2.6 to align the columns.

We construct the packet arrival distribution $\mathcal{F}$ from $\mathcal{R}$ and weighted graph G as follows: For each rule r_i with $w_i \neq 0$, the algorithm makes packet p by changing '$*$' on r_{i_c} to '0' and $\mathcal{F}(p) = w_i$, where w_i is the weight of vertex $i \in V$. Let the weights of the vertices in the Graph in Fig. 2.7 be

$$w_i = \begin{cases} 3, & (\text{if } i = 3) \\ 5, & (\text{if } i = 6) \\ 19, & (\text{if } i = 9) \\ 37, & (\text{if } i = 12) \\ 13, & (\text{if } i = 15) \\ 29, & (\text{if } i = 18) \\ 43, & (\text{if } i = 21) \\ 0, & (\text{otherwise}). \end{cases} \tag{2.9}$$

For rule list $\mathcal{R}$ in Table 2.6 and weighted graph G in Fig. 2.7, the arrival distribution is expressed as shown in (2.10).

$$\mathcal{F}(p) = \begin{cases} 3, & (\text{if } p = 10011001101000001111001000000000000000000000) \\ 5, & (\text{if } p = 10011001010100001111000001000000000000000000) \\ 19, & (\text{if } p = 10010010001111111111000000001000000000000000) \\ 37, & (\text{if } p = 10001001001111111111000000000001000000000000) \\ 13, & (\text{if } p = 10010011111111111111000000000000010000000) \\ 29, & (\text{if } p = 10011111111111111111000000000000000010000) \\ 43, & (\text{if } p = 11111111111111111111000000000000000000010) \\ 0, & (\text{otherwise}). \end{cases} \tag{2.10}$$

The complexity of Algorithm 2 is a function of the size of the graph, and making the rule for the corresponding node is a function of the size of A. Thus, Algorithm 1 has complexity $O(|V|^2 + |V||A|)$. □

Theorem 2.2.1. ***RRO** is **NP**–hard.*

Proof. As the decision problem **RO** is **NP**–complete [29], to prove that **RRO** is **NP**–hard, we reduce **RO** to **RRO**.

Let $\mathcal{R}$ be a rule list in **RO**, i.e., $r \in \mathcal{R}$ has a weight, and let B be a positive integer. We construct a rule list $\mathcal{R}^{'}$ in **RRO**, i.e., $r \in \mathcal{R}^{'}$ has an evaluation type, a packet arrival distribution $\mathcal{F}$, and a positive integer $B^{'}$ such that

$$\begin{aligned} &\exists\sigma,\ \sum_{i=1}^{n} w_i{\cdot}\sigma(i) < B \ \wedge\ \forall r_i, r_j \in \mathcal{R}, O(r_j, r_i) \wedge i < j \Rightarrow \sigma(i) < \sigma(j) \\ \text{iff } &\exists\tau,\ L(\mathcal{R}_{\tau}^{'}, \mathcal{F}) < B^{'} \ \wedge\ \forall p \in \mathcal{P}, \mathcal{R}(p) = \mathcal{R}_{\tau}(p). \end{aligned} \tag{2.11}$$

First, we construct the weighted graph $G_{\mathcal{R}}$ from rule list $\mathcal{R}$ based on definition (2.7), where the vertex i and r_i have the same weight. This process has $O(wn^2)$ complexity, because we need only to compare all pairs of rules r_i and r_j to create that graph, where w is the length of the rule condition. Based on the method in Lemma 2.2.2, we construct graph $G^{'}$ from $G_{\mathcal{R}}$ with $k = \max\{deg^{+}(1), deg^{+}(2), \ldots, deg^{+}(n)\}$ in $O(n^3 + n^2m)$ time. Next, rule list $\mathcal{R}^{'}$ and packet arrival distribution $\mathcal{F}$ are determined using Algorithm 1. Finally, we set $B^{'} = B \cdot k$.

According to the above explanation, the transformation of $(\mathcal{R}, B)$ into $(\mathcal{R}^{'}, \mathcal{F}, B^{'})$ can be done in polynomial time, $O(n^4)$. Proving that (2.11) holds is the only task that is yet to be performed. We show (2.11) by separately proving

$$\begin{aligned} &\exists\sigma,\ \sum_{i=1}^{n} w_i{\cdot}\sigma(i) < B \ \wedge\ \forall r_i, r_j \in \mathcal{R}, O(r_j, r_i) \wedge i < j \Rightarrow \sigma(i) < \sigma(j) \\ \Rightarrow\ &\exists\tau,\ L(\mathcal{R}_{\tau}^{'}, \mathcal{F}) < B^{'} \ \wedge\ \forall p \in \mathcal{P}, \mathcal{R}(p) = \mathcal{R}_{\tau}(p). \end{aligned} \tag{2.12}$$

and

$$\begin{aligned} &\exists\tau,\ L(\mathcal{R}_\tau^{'},\mathcal{F}) < B^{'} \ \wedge\ \forall p \in \mathcal{P}, \mathcal{R}(p) = \mathcal{R}_\tau(p). \\ \Rightarrow\ &\exists\sigma,\ \sum_{i=1}^{n} w_i{\cdot}\sigma(i) < B \ \wedge\ \forall r_i, r_j \in \mathcal{R}, O(r_j, r_i) \wedge i < j \Rightarrow \sigma(i) < \sigma(j) \end{aligned} \tag{2.13}$$

in parts I) and II) below. In the following, n and $n^{'}$ denote the number of rules in $\mathcal{R}$ and $\mathcal{R}^{'}$, respectively.

I) Suppose that there is an order σ such that $\sum_{i=1}^{n} w_i\sigma(i) < B$ and $\forall r_i, r_j \in \mathcal{R},\ O(r_j, r_i) \wedge i < j \Rightarrow \sigma(i) < \sigma(j)$. We define $\tau : [n^{'}] \rightarrow [n^{'}]$ as

$$\tau(i) = \begin{cases} n^{'} & (\text{if } i = n^{'}) \\ k{\cdot}\sigma(\lceil i/k \rceil) - k + i \bmod k & (\text{otherwise}). \end{cases}$$

From this definition, in the order τ, the inserted rules $r_{i-1}, r_{i-2}, \ldots, r_{i-k}$ according to rule r_i are ordered right before r_i. For example, for $k = 3$, $\sigma = (2\,1\,4\,3\,5\,6)$,

$$\tau = (4\,5\,6\,1\,2\,3\,10\,11\,12\,7\,8\,9\,13\,14\,15\,16\,17\,18\,19),$$

where $n = 6$ and $n^{'} = 19$. As $G_{R^{'}}^{'}$ and τ are based on $G_{\mathcal{R}}$ to ensure that the preceding relation holds, $\forall p \in \mathcal{P}, \mathcal{R}^{'}(p) = \mathcal{R}_\tau^{'}(p)$ holds.

The latency of $R_\tau^{'}$ under $\mathcal{F}$ is expressed as follows:

$$\begin{aligned} L(\mathcal{R}_\tau^{'},\mathcal{F}) &= \sum_{i=1}^{n^{'}-1} i|E(\mathcal{R}_\tau^{'}, \tau^{-1}(i))|_{\mathcal{F}} + (n^{'}-1)|E(\mathcal{R}_\tau^{'}, \tau^{-1}(n^{'}))|_{\mathcal{F}} \\ &= \sum_{i=1}^{n^{'}} i|E(\mathcal{R}_\tau^{'}, \tau^{-1}(i))|_{\mathcal{F}} - |E(\mathcal{R}_\tau^{'}, \tau^{-1}(n^{'}))|_{\mathcal{F}} \end{aligned}$$

According to the construction method of the packet arrival distribution $\mathcal{F}$, $|E(\mathcal{R}_\tau^{'}, \tau^{-1}(n^{'}))|_{\mathcal{F}} = 0$, $|E(\mathcal{R}_\tau^{'}, \tau^{-1}(l))|_{\mathcal{F}} = 0$, and $(n^{'} - 1)/k = n$, where l is not a multiple of k. Therefore,

$$\begin{aligned} L(\mathcal{R}_\tau^{'},\mathcal{F}) &= \sum_{i=1}^{n^{'}-1} i|E(\mathcal{R}_\tau^{'}, \tau^{-1}(i))|_{\mathcal{F}} \\ &= \sum_{i=1}^{n} ki|E(\mathcal{R}_\tau^{'}, \tau^{-1}(ki))|_{\mathcal{F}} \end{aligned}$$

As $i|E(\mathcal{R}_\tau^{'}, \tau^{-1}(i))|_{\mathcal{F}}$ is equal to $\sigma(i) \cdot w_i$, we obtain

$$\begin{aligned} L(\mathcal{R}_\tau^{'},\mathcal{F}) &= \sum_{i=1}^{n} k \cdot w_i \cdot \sigma(i) \\ &= k \sum_{i=1}^{n} w_i \cdot \sigma(i) \\ &< k \cdot B = B^{'}. \end{aligned}$$

Thus, we have proved statement (2.12).

II) Suppose that there is an order τ such that $L(\mathcal{R}_{\tau}^{'}, \mathcal{F}) < B^{'}$ and $\forall p \in \mathcal{P}, \mathcal{R}(p) = \mathcal{R}_{\tau}(p)$. We define $\sigma : [n] \rightarrow [n]$ as $\sigma(i) = \tau(ik)/k$. As $G_{R'}^{'}$ and τ are based on $G_{\mathcal{R}}$ to ensure that the preceding relation holds, $\forall r_i, r_j \in \mathcal{R}$, $O(r_j, r_i) \wedge i < j \Rightarrow \sigma(i) < \sigma(j)$ holds.

$$\begin{aligned}
\sum_{i=1}^{n} w_i \sigma(i) &= \frac{1}{k} \sum_{i=1}^{n} k w_i \sigma(i) \\
&= \frac{1}{k} \sum_{i=1}^{n} ki | E(\mathcal{R}^{'}, \tau^{-1}(ki)) |_{\mathcal{F}} \\
&= \frac{1}{k} \sum_{i=1}^{n^{'}-1} i | E(\mathcal{R}^{'}, \tau^{-1}(i)) |_{\mathcal{F}} \\
&< \frac{1}{k} B^{'} = B
\end{aligned}$$

Thus, we have proved statement (2.13).

From the above, statement (2.11) holds.

As stated above, for any positive integers $B, B^{'}$ and rule lists $\mathcal{R}, \mathcal{R}^{'}$, there is an order σ such that $\sum_{i=1}^{n} w_i \cdot \sigma(i) < B \wedge \forall r_i, r_j \in \mathcal{R}$, $O(r_j, r_i) \wedge i < j$ that implies $\sigma(i) < \sigma(j)$ if and only if there is an order τ such that $L(\mathcal{R}_{\tau}^{'}, \mathcal{F}) < B^{'} \wedge \forall p \in \mathcal{P}, \mathcal{R}(p) = \mathcal{R}_{\tau}(p)$. □

As **RRO** is **NP**–hard, its optimization version RORO is also **NP**–hard. The fact that RORO is **NP**–hard implies that we should develop a polynomial time heuristic instead of an exact algorithm.

Algorithm 1: *GraphToRulelist*$(G = (V, A))$

// input graph $G = (V, A)$ is two–colorable;
// $n \in V$ is the only source vertex in G;
$\mathcal{R} \leftarrow$ an empty list;
add r_n^P whose condition r_{n_c} is the list of '$*$' of length $|V| - 1$ to $\mathcal{R}$;
$d \leftarrow$ LongestDistances(n, G);
$V_k \leftarrow \{\ v \mid d_v = k\ \}$;
$i \leftarrow 1$;
while $i \leq |V_1|$ **do**
  $j \leftarrow 1$;
  $c \leftarrow$ "" // empty string ;
  while $j \leq |V_1|$ **do**
    if $i + j = |V_1| + 1$ **then** $c \leftarrow c$ ++ "1";
    else $c \leftarrow c$ ++ "0";
    $j \leftarrow j + 1$;
  end
  add r_u^P whose condition r_{u_c} is c for the present to $\mathcal{R}$;
  $i \leftarrow i + 1$;
end
$D \leftarrow \max\{d_1, d_2, \ldots, d_n\}$;
$k \leftarrow 2$;
while $k \leq D$ **do**
  $m \leftarrow |V_1| + \cdots + |V_{k-1}|$;
  foreach $u \in V_k$ **do**
    $c \leftarrow$ "";
    foreach $v \in V$ **do**
      if $(v, u) \in A$ **then** $c \leftarrow c \oplus r_{v_c}$;
    end
    $i \leftarrow 1$;
    while $i \leq |V_k|$ **do**
      $j \leftarrow m + 1$;
      while $j \leq |V_k|$ **do**
        if $i + m - j = |V_k| + 1$ **then** $c \leftarrow c$ ++ "1";
        else $c \leftarrow c$ ++ "0";
        $j \leftarrow j + 1$;
      end
      $e \leftarrow D$;
      if k *is even* **then** $e \leftarrow P$;
      add r_u^e whose condition r_{u_c} is c for the present to $\mathcal{R}$;
      $i \leftarrow i + 1$;
    end
  end
  $k \leftarrow k + 1$;
end
foreach $i \in \{1, 2, \ldots, |V|\}$ **do**
  if $|r_{i_c}| < |V|$ **then** pad a shortage with $*$;
end
$i \leftarrow 1$;
while $i \leq |V|$ **do**
  $j \leftarrow 1$;
  while $j \leq |V|$ **do**
    if $i = j$ **then** $r_{i_c} \leftarrow r_{i_c}$ ++ "$*$";
    else $r_{i_c} \leftarrow r_{i_c}$ ++ "0";
    $j \leftarrow j + 1$;
  end
  $i \leftarrow i + 1$;
end
add the default rule $r_{|V|+1}^D$.

Algorithm 2: *LongestDistances*(s, G)

```
    // input graph G = (V, A) is a directed acyclical graph;
    // parameter s is the source vertex of G;
    // ∀v ∈ V there exists a directed path from s to v;
1   ∀v ∈ V, d_v ← 0;
2   σ ← a topological ordering of V, where σ(s) = 1;
3   i ← 1;
4   while i ≤ |V| do
5   |   v ← σ^{-1}(i);
    |   foreach (v, u) ∈ E do
6   |   |   if d_u < d_v + 1 then
7   |   |   |   d_u = d_v + 1;
    |   |   end
    |   end
8   |   i ← i + 1 ;
9   |   return d ;
    end
```

Table 2.6: The rule list generated from the graph in Fig. 2.7.

r_1^D	* * 0 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
r_2^D	* * 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 1 0 0 0 * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
r_3^P	* * 0 0 0 0 1 0 0 0 1 0 1 0 0 0 * * * * 0 0 * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
r_4^D	* * 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
r_5^D	* * 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
r_6^P	* * 0 0 0 0 1 0 0 0 0 1 0 1 0 0 * * * * 0 0 0 0 0 * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
r_7^P	0 1 0 0 0 0 1 0 0 1 0 0 0 0 1 0 * * * * 0 0 0 0 0 0 * 0 0 0 0 0 0 0 0 0 0 0 0 0 0
r_8^P	0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 1 * * * * 0 0 0 0 0 0 0 * 0 0 0 0 0 0 0 0 0 0 0 0 0
r_9^D	0 1 0 0 0 0 1 0 0 1 0 0 * * * * * * * * 0 0 0 0 0 0 0 0 * 0 0 0 0 0 0 0 0 0 0 0 0
r_{10}^P	1 0 0 0 1 0 0 0 0 * * * * * * * * * * 0 0 0 0 0 0 0 0 0 * 0 0 0 0 0 0 0 0 0 0 0 0
r_{11}^P	1 0 0 0 0 1 0 0 0 * * * * * * * * * * 0 0 0 0 0 0 0 0 0 0 * 0 0 0 0 0 0 0 0 0 0 0
r_{12}^D	1 0 0 0 * * * * * * * * * * * * * * 0 0 0 0 0 0 0 0 0 0 0 * 0 0 0 0 0 0 0 0 0 0 0
r_{13}^D	0 1 0 0 0 0 1 0 0 0 1 0 * * * * * * * * 0 0 0 0 0 0 0 0 0 0 0 0 * 0 0 0 0 0 0 0 0
r_{14}^D	0 1 0 0 0 0 1 0 0 0 0 1 * * * * * * * * 0 0 0 0 0 0 0 0 0 0 0 0 0 * 0 0 0 0 0 0 0
r_{15}^P	0 1 0 0 0 0 1 0 0 * * * * * * * * * * 0 0 0 0 0 0 0 0 0 0 0 0 0 * 0 0 0 0 0 0 0
r_{16}^P	0 1 0 0 0 0 0 1 0 * * * * * * * * * * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 * 0 0 0 0 0 0
r_{17}^P	0 1 0 0 0 0 0 0 1 * * * * * * * * * * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 * 0 0 0 0 0
r_{18}^D	0 1 0 0 * * * * * * * * * * * * * * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 * 0 0 0 0
r_{19}^D	0 0 1 0 * * * * * * * * * * * * * * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 * 0 0 0
r_{20}^D	0 0 0 1 * * * * * * * * * * * * * * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 * 0 0
r_{21}^P	* * * * * * * * * * * * * * * * * * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 * 0
r_{22}^D	* *

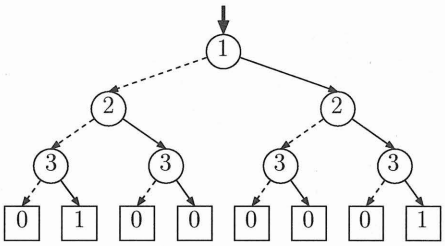

Figure 2.8: Binary decision tree.

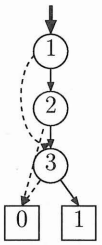

Figure 2.9: ZDD

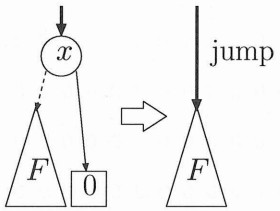

Figure 2.10: Node deletion.

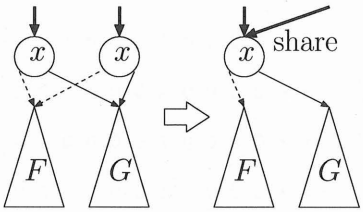

Figure 2.11: Node sharing.

2.3 Zero–Suppressed Binary Decision Diagrams

As the algorithms discussed in the following sections use the zero–suppressed binary decision diagram (ZDD) [58], we explain the mechanism and process of ZDD in this section.

A combination of w items can be represented by a w–bit vector $(b_1, b_2, \ldots, b_w)$, where each b_k expresses whether or not the combination contains the item. A set of combinations can be represented by a set of w–bit vectors. A set of evaluated packets can also be regarded as a set of combinations.

The ZDD data structure was proposed by Minato to manipulate a set of combinations efficiently [58]. A ZDD is obtained by applying reduction rules to a binary decision tree representing a set of combinations. The deletion of a redundant node and sharing of an identical node are illustrated in Figs. 2.10 and 2.11, respectively. Figures 2.8 and 2.9 represent the same set of combinations $\{001, 111\}$. Circles denote non–terminal nodes and boxes indicate terminal nodes. A numeral i associated with a non-terminal node represents a Boolean variable of item i. Non-terminal nodes have edges with values of 1 and 0. The 1 and 0 edges of node i express whether or not this node contains item i. In ZDDs, the variables are ordered. On the path from the root node to a terminal node indicated by a bold arrow, a skipped variable i indicates that the combination does not contain item i.

2.4 Computing The Number of Evaluated Packets

As mentioned in section 2.1, computing the packet classification latency for an order σ with a given packet arrival distribution involves computing each number of evaluated packets $|E(\mathcal{R}_\sigma, i)|_F$ for every rule in a given list. However, in general, computing the number of evaluated packets $|E(\mathcal{R}_\sigma, i)|_F$ for rule r_n is very difficult. As the algorithms described in the following sections must compute the number of evaluated packets, this section defines the problem of *computing the number of evaluated packets.* We show that this problem is #**P**–complete under the uniform distribution.

2.4.1 Complexity of Computing the Number of Evaluated Packets

Definition 2.4.1. *(Computing the number of evaluated packets)*

Input: *rule list* $\mathcal{R}$, *order* σ, *distribution* F, $i \in \mathbb{Z}^+$

Output: $|E(\mathcal{R}_\sigma, i)|_F$ *under* $F, \sigma, \mathcal{R}$

We show that the problem is #**P**–complete if $i = n$ and F is the uniform distribution [63].

To prove the following theorem, we introduce some terminology. A binary relation $S(u, v)$ is polynomially balanced if there exists a Turing machine that can determine $S(u, v)$ in polynomial time. A binary relation $S(u, v)$ is polynomially decidable if there exists some k such that $S(u, v)$ implies $|v| \leq |u|^k$ [63].

Theorem 2.4.1. *The problem of computing the number of evaluated packets is #**P**–complete if* $i = n$ *and* F *is the uniform distribution.*

In the following, the term *uniform distribution* is omitted.

Proof. First, we show that the problem of computing the number of evaluated packets of r_n is in class #**P**. Next, we present a reduction algorithm from #SAT to the problem of computing the number of evaluated packets of r_n; then, we show that this reduction maintains the number of solutions.

1. Let x be a formula $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_{n-1}$ constructed from the conditions of rules r_i, where ϕ_i is the disjunction of variables z_j; z_j is avoided if the jth character of the condition of r_i is '$*$', and z_j is set to z_j or $\bar{z_j}$ if the jth character of the condition of r_i is '0' or '1', respectively ($1 \leq j \leq w$). Let $S(x, y)$ be the binary relation denoting that y satisfies x. The problem of computing the number of evaluated packets of r_n is an enumeration problem defined by the binary relation $S(x, y)$. As $S(x, y)$ is polynomially balanced and polynomially decidable, the problem defined by $S(x, y)$ is in class #**P**.

2. We demonstrate a reduction from #SAT to the problem of computing the number of evaluated packets of r_n, as shown in Algorithm 3. In the algorithm, ++ denotes the string concatenation operation and the rule evaluation type is *don't care.* As the number of solutions of $x \in$ #SAT is clearly the same as the number of solutions of $Red(x)$ and the

Algorithm 3: *Red*

input : formulra $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_{n-1}$
output: rule list $\mathcal{R}$

```
1  n ← the number of variables ;
2  m ← the number of clauses ;
3  R ← an empty list ;
4  i ← 0 ;
   while i < m do
5      φ_i = ⊥ ;
6      cond ← "" ;
7      j ← 1 ;
       while j ≤ n do
8          if x_j ∈ c_i then cond ← cond ++ "0" ;
9          else if x̄_j ∈ c_i then cond ← cond ++ "1" ;
10         else cond ← cond ++ "*" ;
       end
11     set R_i's condition cond ;
12     add R_i to the last of R ;
13     i ← i + 1 ;
   end
   return R
```

order of *Red* is $O(mn)$, *Red* is a parsimonious algorithm from #SAT to the problem of computing the number of evaluated packets of r_n.

From the above, the problem of computing the number of evaluated packets of r_n is #**P**–complete. □

We present an example of a reduction from #SAT to the problem of computing the number of evaluated packets of r_n. Applying Algorithm 3 to the formula in (2.14) yields the rule list in Table 2.1. The number of assignments satisfying (2.14) is the same as $|E(\mathcal{R}_{id}, n)|_U$ in Table 2.1.

$$\begin{aligned} &(x_2 \vee \bar{x_4}) \wedge (x_1 \vee x_2 \vee x_3 \vee x_4) \\ \wedge\ &(x_1 \vee x_3 \vee x_4) \wedge (x_1 \vee \bar{x_3}) \\ \wedge\ &(\bar{x_2} \vee \bar{x_4}) \wedge \bar{x_4} \end{aligned} \tag{2.14}$$

2.5 Manipulating $M(r_i)$ and $E(\mathcal{R}_\sigma, i)$ via ZDDs

The problem of computing the number of evaluated packets of r_i is #**P**–complete, meaning it is hard to solve. RORO requires the efficient manipulation of a set of packets. As the range of packets reaching a network device is smaller than all possible packets, in practice, the

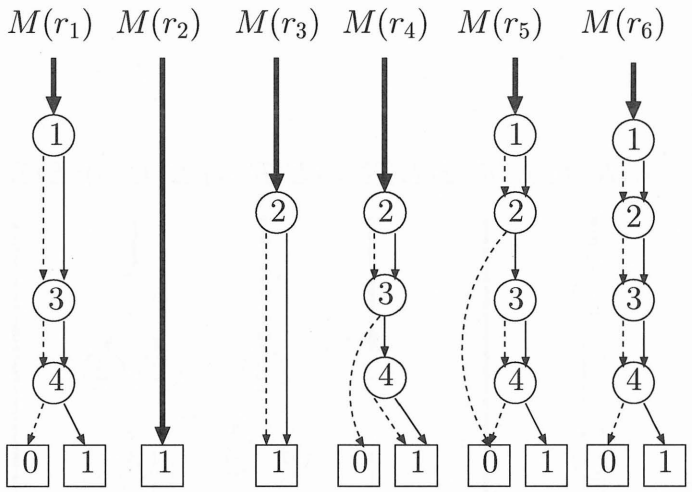

Figure 2.12: ZDDs for $M(r_1), M(r_2), \ldots, M(r_6)$.

manipulated data in packet classification are regarded as a sparse set of all combinations. In this section, we describe an efficient method that manipulates a set of evaluated packets $E(\mathcal{R}_\sigma, i)$ for an order σ using ZDDs.

The ZDDs for the set of packets $M(r_i)$ of each single rule and the rule list $\mathcal{R}$ in Table 2.1 are shown in Fig. 2.12, where $M(r_4)$ expresses the set of combinations $\{0111, 0110, 0011, 0010\}$. The set of packets evaluated by r_i using rule list $\mathcal{R}$ and order σ, $E(\mathcal{R}_\sigma, i)$, does not match any of rules $1, 2, \ldots, (\sigma(i) - 1)$, but does match rule r_i. That is, $E(\mathcal{R}_\sigma, i)$ is the set of packets that does not match rule numbers $\sigma^{-1}(1), \sigma^{-1}(2), \ldots, \sigma^{-1}(\sigma(i) - 1)$ and matches r_i, where σ^{-1} is the inverse function of σ. Thus, to find $E(\mathcal{R}_\sigma, i)$, we compute the following:

$$M(r_i) \setminus M(r_{\sigma^{-1}(1)}) \setminus M(r_{\sigma^{-1}(2)}) \setminus \cdots \setminus M(r_{\sigma^{-1}(\sigma(i)-1)}). \tag{2.15}$$

To compute $E(\mathcal{R}_\sigma, i)$, it is necessary to determine $M(r_1)$, $M(r_2)$, ..., $M(r_n)$. As this set of combinations is frequently used in reordering methods, we can use ZDDs to manipulate them efficiently. The ZDDs of $E(\mathcal{R}_{id}, i)$ for the rule list in Table 2.1 are shown in Fig. 2.13, where $E(\mathcal{R}_{id}, 4)$ expresses the set of combinations $\{0111, 0110, 0010\}$. Note that, comparing $M(r_4)$ with $E(\mathcal{R}_{id}, 4)$, 0011 has been removed. For brevity, $\mathcal{R}_{id}$ is now abbreviated as $\mathcal{R}$. In practice, the ZDDs for Figs. 2.12 and 2.13 are stored in a computer by sharing identical nodes, as in Fig. 2.14. This sharing of nodes provides a compact representation of the set of combinations and allows for efficient manipulation. We show the efficiency of manipulating the sets of evaluated packets by ZDDs in section 2.7.4.

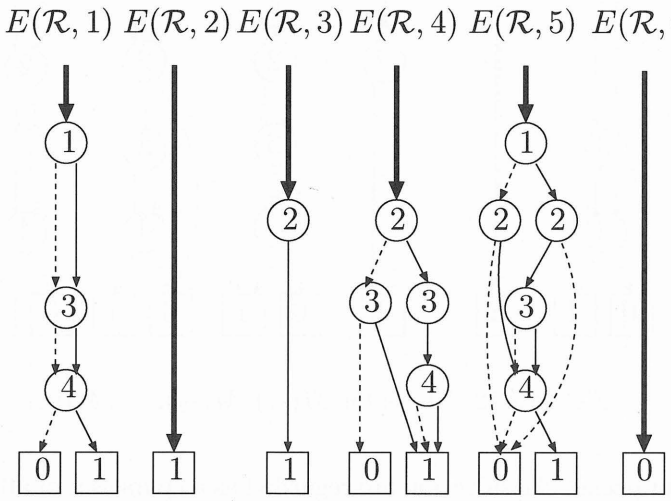

Figure 2.13: ZDDs for $E(\mathcal{R}_{id},1), E(\mathcal{R}_{id},2), \ldots, E(\mathcal{R}_{id},6)$.

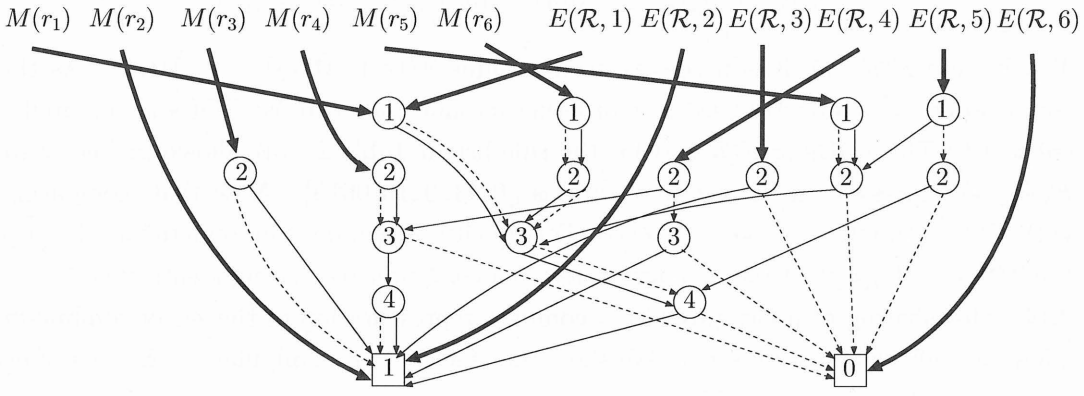

Figure 2.14: ZDDs for $M(r_i)$s and $E(\mathcal{R}_{id},i)$s.

2.6 Rule Pairing Algorithm

In this section, we propose a rule reordering algorithm and calculate its time complexity.

If there is no dependent rule (r_i, r_j), where $i < j$ and the weight of r_j is greater than that of r_i, we can simply sort the rules with those weights to find a better ordering. For example, the rule list in Fig. 2.15 can be sorted as shown in Fig. 2.16. In contrast to the rule list in Fig. 2.15, that in Table 2.1 cannot be simply reordered with those weights, as shown in Fig. 2.17.

This is because r_7^D preceding r_6^P, r_7^D preceding r_5^P, r_7^D preceding r_2^P, and r_7^D preceding r_3^P violate the classification policy. These relations are represented by heavy lines in Fig. 2.17, except relation r_7^D, r_5^P. By gathering rules together as one in advance to satisfy

$$\text{if } ji \in A' \text{ then the weight of } r_i \text{ is greater than that of } r_j, \tag{2.16}$$

we can achieve a better ordering for rule list $\mathcal{R}$ by simply sorting. We focus on this property.

Our algorithm aims to impart the above property on the precedence graph of the rule list. It recursively pairs the rules causing the policy violation until there are no such rules. As an example, for the rule list in Table 2.1, the weighted precedence graph $G_{\mathcal{R}}$ is shown in Fig. 2.18. Because pairs (r_7, r_6), (r_7, r_3), and (r_7, r_2) do not have property (2.16), the algorithm selects one of the pairs and applies property (2.16). Pairing (r_7, r_6), the weighted graph in Fig. 2.18 becomes that in Fig. 2.19. The weight of gathered rules r_{i_1}, r_{i_2}, ..., r_{i_k} is the mean of weights $(w_{i_1} + w_{i_2} + \ldots, w_{i_k})/k$.

Next, pairing (r_6', r_3), the weighted graph becomes that in Fig. 2.20.

Then, pairing (r_3', r_2), the weighted graph is transformed to that shown in Fig. 2.21. The resulting weighted graph has property (2.16).

Finally, the algorithm sorts the rules with those weights and decomposes the paired rules. The rule list in Table 2.1 is reordered as shown in Fig. 2.22.

The complete pairing and sorting procedure is described by Algorithm 4. In the algorithm, w_i is the weight of rule r_i. To pair the rules, the algorithm searches for the heaviest rule on line 4. To explain why the algorithm searches for the heaviest rule, we consider the weighted graph in Fig. 2.23. For this graph, there can be two pairing processes, presented as follows:

$$[r_i, r_j, r_k] \rightarrow [(r_i, r_j), r_k] \rightarrow [((r_i, r_j), r_k)],$$
$$[r_i, r_j, r_k] \rightarrow [(r_i, r_k), r_j] \rightarrow [((r_i, r_k), r_j)].$$

In this example, the algorithm first selects pair (r_i, r_k), to decrease the latency. As shown above, in most cases, if there are rules to be paired, the pairing algorithm should select the pair containing the heaviest rule.

Another key process in the algorithm is the search for the rule to be paired with r_{max} on line 13. Generally speaking, this process selects the minimum weight rule from the rules that directly precede r_{max}. To explain why the algorithm selects the minimum weight rule, we consider the weighted graph in Fig. 2.24. Pairing r_a with r_b means that there will be no rule between r_a and r_b in the resulting ordering. Because the rules directly preceding r_{max} are independent, we can

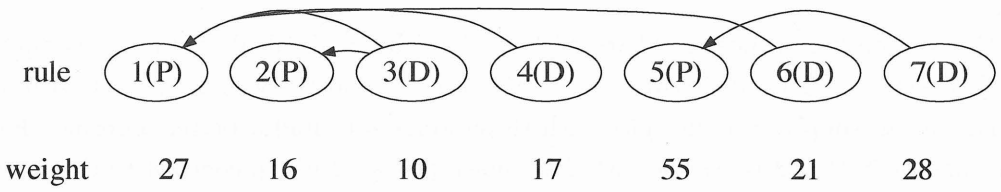

Figure 2.15: Example of a rule list that can be simply reordered.

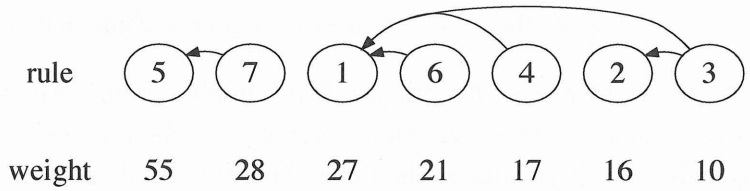

Figure 2.16: Example of a rule list simply reordered by the weights of the rules.

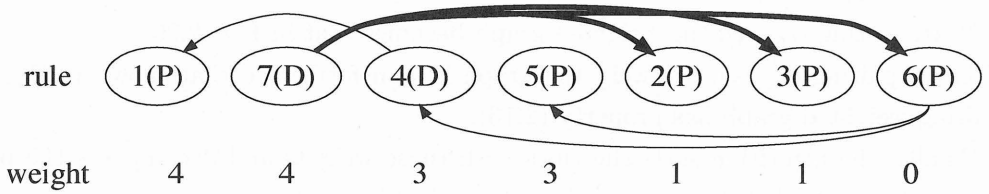

Figure 2.17: Reordering causing policy violations at the red edges.

simply sort these according to their weights. Thus, in the case shown in Fig. 2.24, the pairing process selects r_j to be paired with $r_{max}(r_h)$.

We now consider the process searchPairingRule() in detail. First, it computes the average weight of all vertices to which there is a path from r_{max}. Then, the process selects the minimum average weight rule r_{min}. Finally, it reversely traverses the edges from r_{min} to r_{max} by selecting the smaller average weight rule.

For example, we consider the graph shown in Fig. 2.25. The procedure uses r_7 as r_{max} because w_7 is the maximum weight. To select one of r_4, r_5, and r_6, the average weights are computed for rules r_1, r_2, ..., r_6 as shown in Fig. 2.26. Here, the upper, bottom–left, and bottom–right numbers show the rule number i, rule weight w_i, and average weight A_i, respectively. In this example, because the average weight A_2 is the minimum, r_2 is selected, and then

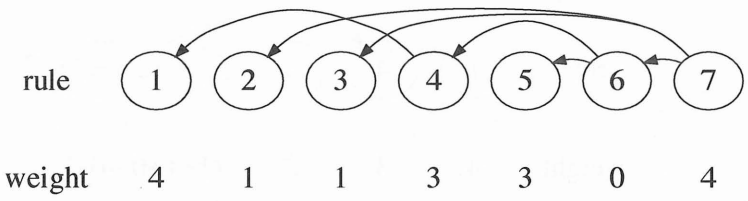

Figure 2.18: Weighted precedence graph for the rule list in Table 2.1.

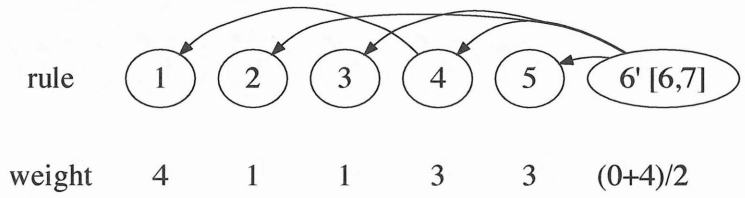

Figure 2.19: Pairing (r_7, r_6).

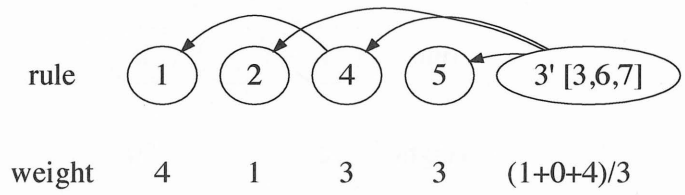

Figure 2.20: Pairing $(r_6{}', r_3)$.

r_4, because $A_4 < A_5$. Thus, r_4 is paired with r_7.

2.6.1 Complexity Analysis

In this section, we describe the time complexity for the rule pairing procedure presented in Algorithm 4. In the following, let m denote the number of pairs of dependent rules and n be the number of rules. Searching for the maximum weight rule in S at line 4 has $O(n)$ complexity. Checking whether there is no rule on which r_{max} depends (line 5) is an $O(1)$ process. The time complexity for searchPairingRule(r_{max}, S) is $O(n + m)$. Sorting the rules in T on line 16 can be done in $O(n \log n)$ time. As lines 4–17 repeat at most n times, their overall complexity is $O(n^2 + nm)$. Thus, the time complexity for the proposed algorithm is $O(n^2 + nm)$.

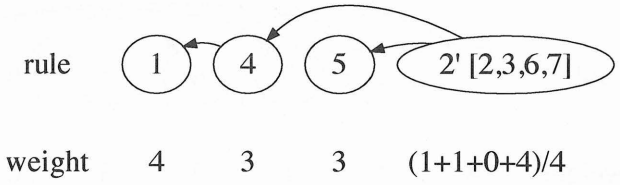

Figure 2.21: Pairing (r_3', r_2).

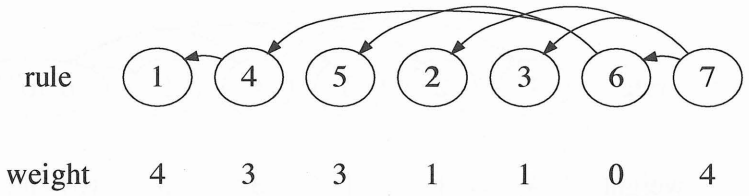

Figure 2.22: Rules reordered by pairing.

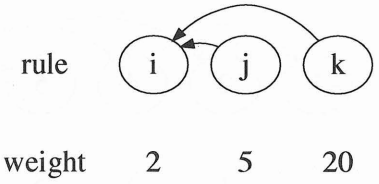

Figure 2.23: Selecting the heaviest rule r_k.

The time complexity for Sub–Graph Merging (SGM) is $O(n^3)$ [77]. Because the precedence graph for a rule list is actually sparse, the proposed algorithm is faster than SGM in most cases.

2.6.2 Experiments

The efficiency of the proposed algorithm is demonstrated through experiments based on both conventional ORO and RORO. We implemented the swapping–window based paradigm (SWBP) [61], the algorithm of Tanaka et al. [76], SGM [77], and the proposed method in Java under the Cent OS Release 6.5(Final) on an Intel Core i5–2400 3.10 GHz CPU with 4 GB main memory. We generated the rules and headers with the standard benchmark for packet classification algorithms ClassBench [80]. For RORO experiments, we added an evaluation type P or D to each rule of a rule list generated by ClassBench with a probability of $1/2$, and added the default rule. A packet

Algorithm 4: PairingAndSortingAlgorithm

input : rule list $\mathcal{R}$, order σ and packet arrival distribution F
output: rule list $\mathcal{R}'$

```
1  make a set S of rules for R ;
2  prepare an empty list T ;
3  while S ≠ ∅ do
4      r_max := maximum weight rule in S ;
5      if there is no rule on which r_max depends then
6          add r_max into T ;
7          delete r_max from S ;
8          continue ;
       end
9      else if for all r ∈ S, D(r_max, r) ∧ w_{r_max} < w_r then
10         add r_max into T ;
11         delete r_max from S ;
12         continue ;
       end
       else
13         r_target := searchPairingRule(r_max, S) ;
14         r_new := pair r_max with r_target ;
15         add r_new into S ;
16         delete r_max from S ;
17         delete r_target from S ;
       end
   end
18 sort the rules in T with those weights ;
19 decompose the paired rules in T ;
20 return T ;
```

header generated by ClassBench consists of source/destination addresses, source/destination port number and protocol number. Because the lengths of these components are 32, 32, 16, 16, and 8 bits, respectively, the length of the condition of the rule and header was 104 bits. The number of headers was about $1M$.

ClassBench has three kinds of seed files, namely Access Control List (acl), Fire Wall (fw), and IP chains (ipc). Because the SGM algorithm did not terminate within 1 h for fw and ipc, we only utilized the acl seed files in the experiments. Because the resulting graphs for fw and ipc can be dense, the graph updating phase and subgraphs construction phase in SGM are impractically slow.

Using the generated rule lists and header lists, we measured the time required to reorder

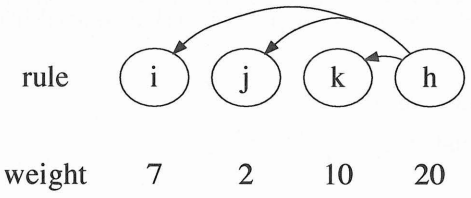

Figure 2.24: Selecting the minimum weight rule.

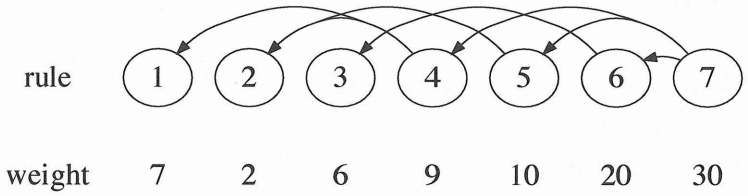

Figure 2.25: Weighted precedence graph.

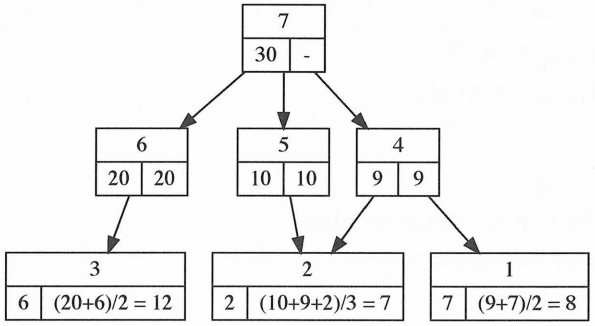

Figure 2.26: Computation of the average weight.

the rules and the latency of the rule list for every algorithm. The units of measurement are seconds. The mean values over 10 trials for RORO are shown in Figs. 2.27 and 2.28. Note that the reordering times are plotted on a logarithmic scale in Fig. 2.28.

As shown in Fig. 2.27, the proposed algorithm decreases the latency by about 9% compared with SWBP [61] and the algorithm of Tanaka et al. [76]. Detailed results are presented in Table 2.7. The latency of the proposed algorithm is less than or equal to that of SGM. Figure 2.28 shows that the reordering times for SGM and the proposed algorithm are 44 and 3 s, respectively,

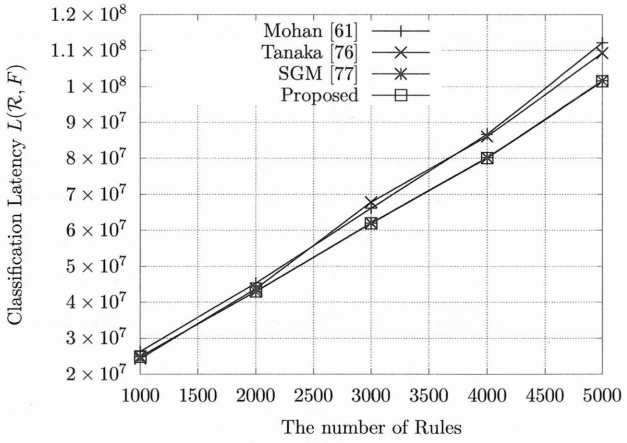

Figure 2.27: Latency of dependency relation.

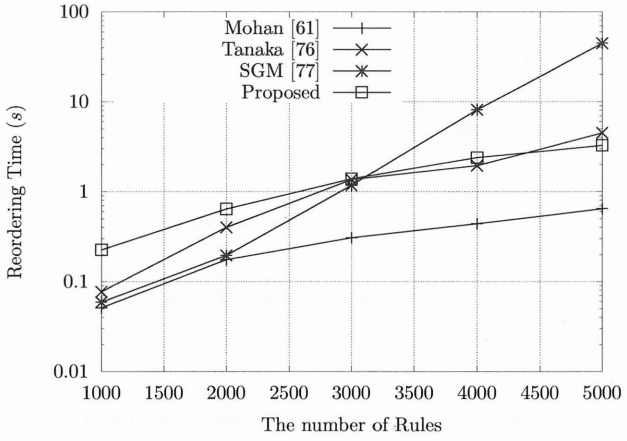

Figure 2.28: Reordering time (s) of dependency relation.

for 5000 rules. That is, the reordering time of the proposed algorithm is 15 times faster than that of SGM for 5000 rules. The results in Figs. 2.27, and 2.28 and Table 2.7 indicate that the proposed algorithm is quite effective in terms of processing latency the time required to reorder the rule list.

The effectiveness of the proposed algorithm using the conventional ORO model is shown in Figs. 2.29 and 2.30. Because SGM applied to ORO did not terminate within 1 h for 4000 rules, we only show the results for up to 3000 rules. Note that, similar to Fig. 2.28, Fig. 2.30 uses a logarithmic scale. The latency of the proposed algorithm is about 10% lower than that of SWBP [61] and Tanaka et al. [76], as shown in Fig. 2.29. The reordering time of the proposed algorithm is 300 times faster than that of SGM for 3000 rules, as shown in Fig. 2.30. Because the graph of the overlap relation is denser than that of the dependency relation, the reordering time for SGM applied to ORO is much longer than for RORO, as shown in Figs. 2.28 and 2.30. The above results demonstrate that the proposed algorithm is remarkably efficient for both RORO

Table 2.7: Latency for SGM [77] and the proposed algorithm.

# of rules	SGM [77]	proposed algorithm
1000	2.4898e+07	2.4882e+07
2000	4.3011e+07	4.2977e+07
3000	6.2013e+07	6.1955e+07
4000	8.0241e+07	8.0115e+07
5000	1.0159e+08	1.0143e+08

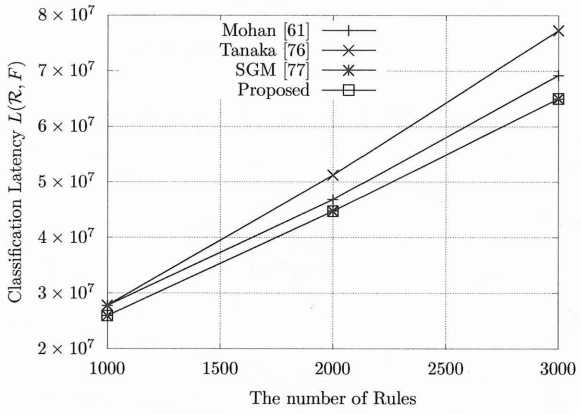

Figure 2.29: Latency on the overlap relation.

and conventional ORO.

Because the graph for the rule lists generated by ClassBench is actually sparse, the results show that the proposed algorithm is faster than SGM. As an actual rule list can be assumed to have the same characteristics as ClassBench, we conclude that the proposed algorithm would be sufficiently effective for real packet classification.

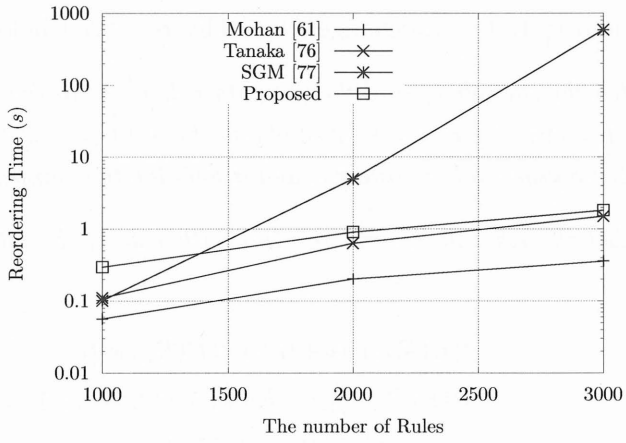

Figure 2.30: Reordering time (s) on the overlap relation.

2.7 Improving Reordering Methods based on [75]

The algorithm described in [75] neglected the variation in the evaluated packets and the packet arrival distributions. In this section, we propose a novel algorithm based on [75] that considers the variation of evaluated packets and the packet arrival distributions.

The reordering algorithm [75] consists of three modules: interchanging two consecutive rules, interchanging a single rule, and interchanging a set of rules.

2.7.1 Interchange Adjacent Rules

In this section, we show that the latency can be reduced by interchanging adjacent rules, and present an algorithm that repeatedly interchanges adjacent rules in a rule list.

Theorem 2.7.1. *Exchanging the ith rule r_ι^e and the $(i+1)$th rule r_κ^f in a rule list $\mathcal{R}$ maintains the classification policy if and only if $e = f \ \vee \ M(r_\iota) \cap M(r_\kappa) = \emptyset$.*

We consider r_i^e and r_j^f to be interchangeable if Theorem 2.7.1 holds for rules r_i^e and r_j^f.

Theorem 2.7.2. *Let the ith rule r_ι^e and the $(i+1)$th rule r_κ^f be interchangeable. If e is different from f, then if $|(E(\mathcal{R}_\sigma, \iota)|_F < |(E(\mathcal{R}_\sigma, \kappa)|_F$ holds, we have $L(\mathcal{R}_{\tau_{\iota,\kappa}\circ\sigma}, F) < L(\mathcal{R}_\sigma, F)$, where $\tau_{\iota,\kappa}$ is an order that interchanges only r_ι and r_κ, and $\circ$ denotes the composition of functions.*

Theorem 2.7.3. *Let the ith rule r_ι^e and the $(i+1)$th rule r_κ^f be interchangeable. If e is the same as f, then if*

$$\begin{aligned} & i|(E(\mathcal{R}_\sigma, \iota)|_F + (i+1)|(E(\mathcal{R}_\sigma, \kappa)|_F \\ > \ & i(|(E(\mathcal{R}_\sigma, \iota)|_F + |E(\mathcal{R}_\sigma, \iota) \cap M(r_\kappa)|_F) \\ & + (i+1)(|(E(\mathcal{R}_\sigma, \iota) \setminus M(r_\kappa)|_F) \end{aligned}$$

hold, we have $L(\mathcal{R}_{\tau_{\iota,\kappa}\circ\sigma}, F) < L(\mathcal{R}_\sigma, F)$.

We say that r_i^e and r_j^f are reducible if Theorem 2.7.2 or Theorem 2.7.3 holds for r_i^e, but not for r_j^f.

The repeated interchange of adjacent rules is described in Algorithm 5. Note that we use a flag $|\mathcal{R}_\sigma, 0|_F$ to terminate the algorithm.

This algorithm interchanges the ith and $(i+1)$th rules until Theorems 2.7.1, 2.7.2, and 2.7.3 hold. If the first i rules are interchangeable, the algorithm orders them according to

$$|E(\mathcal{R}_\sigma, \sigma^{-1}(1))|_F \geq |E(\mathcal{R}_\sigma, \sigma^{-1}(2))|_F \geq \cdots \geq |E(\mathcal{R}_\sigma, \sigma^{-1}(i))|_F. \tag{2.17}$$

Although the rules will ideally be listed in descending order after applying the algorithm, in many cases, this may not be the case because Theorem 2.7.1 does not hold.

Algorithm 5: InterchangeRandR

input : $\mathcal{R}, \sigma, F$

let $|E(\mathcal{R}_\sigma, 0)|_F = \infty$ for any F ;

$i := 2$;

while $i \leq n-1$ **do**

  $j := i - 1$;

  while $r_{\sigma^{-1}(j)}$ *and* $r_{\sigma^{-1}(j+1)}$ *are interchangeable and reducible* **do**

    interchange $r_{\sigma^{-1}(j)}$ and $r_{\sigma^{-1}(j+1)}$;

    $\sigma := \tau_{\sigma^{-1}(j),\sigma^{-1}(j+1)} \circ \sigma$;

    update ZDDs ;

    $j := j - 1$;

  end

  $i := i + 1$;

end

Table 2.8: A given rule list.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R}_{id}, i)\rvert_H$
$r_1^D = 1\ 0\ 0\ *$	2
$r_2^P = 1\ 1\ 1\ 1$	3
$r_3^P = 1\ 1\ 0\ *$	4
$r_4^P = *\ 0\ *\ *$	15
$r_5^D = *\ *\ *\ *$	28
$L(\mathcal{R}_{id}, H) = 192$	

Table 2.9: Reordering by the algorithm in Fig. 5.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R}_\pi, i)\rvert_H$
$r_3^P = 1\ 1\ 0\ *$	4
$r_2^P = 1\ 1\ 1\ 1$	3
$r_1^D = 1\ 0\ 0\ *$	2
$r_4^P = *\ 0\ *\ *$	15
$r_5^D = *\ *\ *\ *$	28
$L(\mathcal{R}_\pi, H) = 188$	

Table 2.10: Reordering by grouping rules.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R}_\tau, i)\rvert_H$
$r_1^D = 1\ 0\ 0\ *$	2
$r_4^P = *\ 0\ *\ *$	15
$r_3^P = 1\ 1\ 0\ *$	4
$r_2^P = 1\ 1\ 1\ 1$	3
$r_5^D = *\ *\ *\ *$	28
$L(\mathcal{R}_\tau, H) = 168$	

2.7.2 Interchange of Single Rule and Consecutive Rules

We now describe the effect of interchanging adjacent rules on the classification latency. Consider the rule list in Table 2.8 and the packet arrival distribution H given by (2.18).

$$\begin{array}{llll} 0000 \mapsto 10, & 0001 \mapsto 0, & 0010 \mapsto 0, & 0011 \mapsto 0, \\ 0100 \mapsto 7, & 0101 \mapsto 2, & 0110 \mapsto 8, & 0111 \mapsto 13, \\ 1000 \mapsto 2, & 1001 \mapsto 0, & 1010 \mapsto 0, & 1011 \mapsto 5, \\ 1100 \mapsto 1, & 1101 \mapsto 3, & 1110 \mapsto 0, & 1111 \mapsto 3 \end{array} \tag{2.18}$$

To decrease the classification latency, we place r_4^P, which has a large number of evaluated packets, in the uppermost position without violating the classification policy. Algorithm 5 orders the first

rule in descending order of the number of evaluated packets, as in Table 2.9. However, because r_4^P depends on r_1^D, r_4^P cannot be placed in the uppermost position. To overcome this, an algorithm that interchanges a single rule and rule groups is proposed [75]. For example, for order π in Table 2.9, r_1^D and r_4^P are interchangeable with r_2^P and interchanging the set of rules r_1^D, r_4^P and the single rule r_2^P to give $[r_3^P, r_1^D, r_4^P, r_2^P, r_5^D]$ reduces the latency to 173. The latency can be decreased to 168 by interchanging the set of rules r_1^D, r_4^P and the single rule r_3^P, as in Table 2.10.

The method described in [75] places the ith rule in an upper position, without any policy violation if the ith rule depends on the $(i-1)$th rule, by grouping and regarding them as a single rule.

$$i|E(\mathcal{R}_\sigma, \iota)|_F + \sum_{k=1}^{j} |E(\mathcal{R}_\sigma, \iota_k)|_F - \sum_{k=1}^{j} (i+k-1)|M(r_{\iota_k}) \cap (E(\mathcal{R}_\sigma, \iota) \setminus M(r_{\iota_1}) \setminus \cdots \setminus M(r_{\iota_{k-1}}))|_F \\ - (i+j)|E(\mathcal{R}_\sigma, \iota) \setminus M(r_{\iota_1}) \setminus M(r_{\iota_2}) \setminus \cdots \setminus M(r_{\iota_j})|_F \tag{2.19}$$

Theorem 2.7.4. *Interchanging the ith rule $r_\iota^{e_\iota}$ and the set of $(i+1)$th to $(i+j)$th rules $r_{\iota_1}^{e_{\iota_1}}, \ldots, r_{\iota_j}^{e_{\iota_j}}$ in a rule list $\mathcal{R}$ maintains the classification policy if and only if $r_\iota^{e_\iota}$ is interchangeable with all rules $r_{\iota_1}^{e_{\iota_1}}, \ldots, r_{\iota_j}^{e_{\iota_j}}$.*

We say that a single rule $r_\iota^{e_\iota}$ and a set of rules $r_{\iota_1}^{e_{\iota_1}}, \ldots, r_{\iota_j}^{e_{\iota_j}}$ are interchangeable when they satisfy Theorem 2.7.4.

Theorem 2.7.5. *Let the ith rule $r_\iota^{e_\iota}$ and the set of rules from the $(i+1)$th rule to the $(i+j)$th rule $r_{\iota_1}^{e_{\iota_1}}, \ldots, r_{\iota_j}^{e_{\iota_j}}$ be interchangeable. If $r_\iota^{e_\iota}$ and $r_{\iota_1}^{e_{\iota_1}}, \ldots, r_{\iota_j}^{e_{\iota_j}}$ satisfy (2.19), then*

$$L(\mathcal{R}_{\pi\circ\sigma}, F) < L(\mathcal{R}_\sigma, F),$$

where π is the order that interchanges the ith rule $r_\iota^{e_\iota}$ and the set of $(i+1)$th to $(i+j)$th rules $r_{\iota_1}^{e_{\iota_1}}, \ldots, r_{\iota_j}^{e_{\iota_j}}$.

We say that a single rule $r_\iota^{e_\iota}$ and a set of rules $r_{\iota_1}^{e_{\iota_1}}, \ldots, r_{\iota_j}^{e_{\iota_j}}$ are reducible when they satisfy Theorem 2.7.5.

Proof. Without loss of generality, we assume that $\iota=1$, $\iota_1=2, \ldots, \iota_j=j+1$, i.e., σ is the identity. We describe $\mathcal{R}_\sigma$ and $\mathcal{R}_{\pi\circ\sigma}$ as $\mathcal{R}$ and $\mathcal{R}_\pi$ for brevity. From the assumption that $\mathcal{R}_\pi$ is given by interchanging the first rule r_1 and the set of rules $r_2, \ldots, r_{1+j}$, $L(\mathcal{R}, F)$ and $L(\mathcal{R}_\pi, F)$ become

$$L(\mathcal{R}, F) = \sum_{i=1}^{j+1} i|E(\mathcal{R}), i|_F \\ + \sum_{i=j+2}^{n-1} i|E(\mathcal{R}), i|_F + (n-1)|E(\mathcal{R}), n|_F \tag{2.20}$$

$$L(\mathcal{R}_\pi, F) = \sum_{i=1}^{j+1} \pi(i)|E(\mathcal{R}_\pi, i)|_F \\ + \sum_{i=j+2}^{n-1} \pi(i)|E(\mathcal{R}_\pi, i)|_F + (n-1)|E(\mathcal{R}_\pi), n|_F. \tag{2.21}$$

Algorithm 6: InterchangeRandRs

input : $\mathcal{R}, \sigma, F, i$

1 let $|E(\mathcal{R}_\sigma, 0)|_F = \infty$ for any F ;

2 $j := 2$;

while $r_{\sigma^{-1}(j)}$ *and* $r_{\sigma^{-1}(i+1)}, \ldots r_{\sigma^{-1}(i+j)}$ *hold (2.19)* **do**

  if $r_{\sigma^{-1}(j)}$ *and* $r_{\sigma^{-1}(i+1)}, \ldots r_{\sigma^{-1}(i+j)}$ *are interchangeable* **then**

3 interchange $r_{\sigma^{-1}(j)}$ and $r_{\sigma^{-1}(i+1)}, \ldots r_{\sigma^{-1}(i+j)}$;

4 update σ and ZDDs for each $E(\mathcal{R}_\sigma, i)$;

  end

  else

5 $j := j + 1$

  end

6 $i := i - 1$;

end

□

From the definition of the latency, $\sum_{i=j+2}^{n-1} \pi(i)|E(\mathcal{R}_\pi, i)|_F$ is $\sum_{i=j+2}^{n-1} i|E(\mathcal{R}, i)|_F$ and $E(\mathcal{R}_\pi, n)$ is $E(\mathcal{R}, n)$ in (2.21). As $L(\mathcal{R}, F) - L(\mathcal{R}_\pi, F)$ can be written as in (2.22), this proves Theorem 2.7.5.

$$\begin{aligned} L(\mathcal{R}, F) - L(\mathcal{R}_\pi, F) &= \sum_{i=1}^{j+1} i|E(\mathcal{R}), i|_F - \sum_{i=1}^{j+1} \pi(i)|E(\mathcal{R}_\pi, i)|_F \\ &= \sum_{i=1}^{j+1} i|E(\mathcal{R}), i|_F - \sum_{k=2}^{j+1} (k-1)|E(\mathcal{R}, k)|_F \\ &\quad - \sum_{k=2}^{j+1} (k-1)|M(r_k) \cap (E(\mathcal{R}, 1) \setminus M(r_2) \setminus \cdots \setminus M(r_{k-1}))|_F \\ &\quad - (j+1)|E(\mathcal{R}, 1) \setminus M(r_2) \setminus \cdots \setminus M(r_{k-1})|_F \end{aligned} \tag{2.22}$$

The procedure for moving a group of rules to an upper position is described in Algorithm 6. The first part of the algorithm interchanges $r_{\sigma^{-i}(i)}$ and $r_{\sigma^{-1}(i+1)}, \ldots, r_{\sigma^{-1}(i+j)}$ repeatedly until Theorem 2.7.5 holds, with the aim of moving toward the first position in a rule group of j rules $r_{\sigma^{-1}(i+1)}, \ldots, r_{\sigma^{-1}(i+j)}$. This part is implemented in lines 2, 3, and 6 in Algorithm 6. The second part of the algorithm moves rule $r_{\sigma^{-1}(i+j)}$ by adding $r_{\sigma^{-i}(i)}$ to the group of rules $r_{\sigma^{-1}(i+1)}, \ldots, r_{\sigma^{-1}(i+j)}$ to form a rule group of $(j+1)$ rules when $r_{\sigma^{-i}(i)}$ and $r_{\sigma^{-1}(i+1)}, \ldots, r_{\sigma^{-1}(i+j)}$ are not interchangeable.

2.7.3 Adaptive Reordering Algorithm

In this section, we present an algorithm composed of Algorithms 5 and 6.

Algorithm 7: reorderingRuleList

input : $\mathcal{R}, \sigma, F$

1 let $|E(\mathcal{R}_\sigma, 0)|_F = \infty$ for any F ;

2 $i := 2$;

while $i \leq n-1$ **do**

3 | $j := i - 1$;

| **while** $r_{\sigma^{-1}(j)}$ *and* $r_{\sigma^{-1}(j+1)}$ *are interchangeable and reducible* **do**

4 | | interchange $r_{\sigma^{-1}(j)}$ and $r_{\sigma^{-1}(j+1)}$;

5 | | $\sigma := \tau_{\sigma^{-1}(j),\sigma^{-1}(j+1)} \circ \sigma$;

6 | | update ZDDs ;

7 | | $j := j - 1$;

| **end**

| **if** $j > 0$ **then**

8 | | call InterchangeRandRs($\mathcal{R}, \sigma, F, j-1$) ;

| **end**

9 | $i := i + 1$;

end

10 call InterchangeRandR($\mathcal{R}, \sigma, F$) ;

In Algorithm 7, the individual algorithms are implemented on lines 8 and 10. First, rule r_i^e is moved to the uppermost position for $i=2,3,\ldots,n-1$ (see Algorithm 7, lines 3 and 4). When the algorithm breaks out of the inner loop at line 3, if $j > 0$ (i.e., rule r_i^e cannot be placed in the first position), the algorithm calls the procedure in Algorithm 6 at line 10 to group the jth rule and r_i^e. When the algorithm breaks out of the outer loop at line 3, it terminates after finally calling the procedure in Algorithm 5.

2.7.4 Experiments

Experiments were conducted to demonstrate the need to consider the variation in the number of evaluated packets and determine the efficiency of the proposed method. We implemented the proposed method in C++ under the Cent OS Release 6.5(Final) on an Intel Core i5-2400 3.10 GHz CPU machine with 4 GB main memory. We used the CUDD ZDD package [68]. The rules and headers were generated with the benchmark packet classification algorithms of ClassBench [80], and evaluation type P or D was added to each rule in the rule list with a probability of 1/2. The length of the conditions for the rules and the headers was 120 characters and the number of rules was about $1k$. The number of headers was about $1M$. We generated 30 rule lists from seed files acl, fw, and ipc. With the generated rule lists and header lists, we measured the latency using the method described in [75] for both a fixed-weight model and a variable-weight model. The mean values of 10 trials are presented in Table 2.11. The latency of the fixed-weight model (row 1) is higher than that of the variable-weight model (row 2) by 18, 1 and 5% for acl, fw,

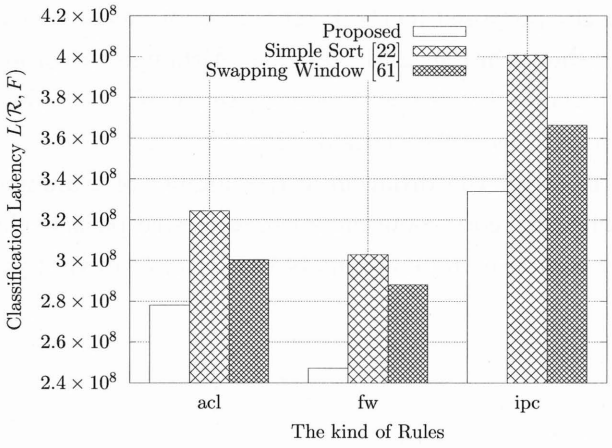

Figure 2.31: Latency.

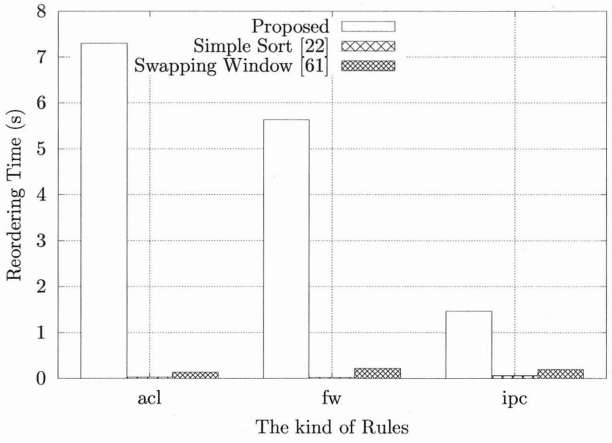

Figure 2.32: Reordering time (s).

and ipc, respectively. Thus, to solve the RORO problem, the variation of rule weights should be considered. By comparing rows 2 and 3 (proposed method) in Table 2.11, we find that the proposed method decreases the latency by about 6, 5, and 3% for acl, fw, and ipc, respectively, against the method of [75].

To confirm the efficiency of the proposed method, we measured the time required to reorder the rules and the latency using Simple Rule Sorting [22] and Swapping Window Based Paradigm

Table 2.11: Latency of proposed model and old model

	acl	fw	ipc
fixed weights (old model)	299843000	260785000	346560000
varying weights	294295000	260503606	344709000
proposed method	278112000	247181000	333953000

Algorithm [61], and the proposed method. The units of measurement are seconds. The mean values of 10 trials are shown in Figs 2.31 and 2.32. Although the proposed method takes longer than the methods of [22] and [61], it has the lower latency. This means that the proposed method represents an improvement over the methods [22] and [61].

In this program, to adapt the variation in the number of evaluated packets, we used ZDDs that were modified when the corresponding set of evaluated packets changed. Figure 2.32 shows that ZDDs efficiently compute and manipulate the set of evaluated packets.

Chapter 3

Optimizing Rule List

To this point, we have only discussed methods for *reordering* rules. However, to reduce the classification latency, reconstructing the rule list should also be considered. We formalize this latency problem as another optimization scheme in which the aim is to identify the rule list that minimizes the latency while preserving the original classification policy. We call this the Optimal Rule List (ORL) problem.

Although ORO is **NP**–hard, if the graph of preceding relations on rules is a forest of oriented trees and there is no variation in rule weights, ORO is solvable (under a certain assumption) in *polynomial time* using single–machine job sequencing algorithm [34]. Focusing on this point, this chapter presents a rule list reconstruction algorithms that uses the solution of single–machine job scheduling. The algorithm first rewrites an input rule list so that its precedence graph is a forest of oriented trees, and then optimizes the order of the rules.

3.1 Single Machine Job Sequencing Problem

The single–machine job sequencing problem is an optimization problem that attempts to identify the order of jobs that minimizes the value of an objective function subject to certain precedence constraints [48].

The input to the single machine job sequencing problem is the n jobs to be sequenced for processing by a single machine. The jobs have processing times of $p_1, p_2, \ldots, p_n$ and weights of $w_1, w_2, \ldots, w_n$ with precedence constraints on the jobs. The precedence constraints are given in the acyclic digraph $G = (V, A)$. Each vertex $v_i \in V$ corresponds to job i. If there is an edge $(t, s) \in A$, job s precedes t, i.e., job t can not be processed before job s. The output of the single–machine job sequencing problem is a feasible sequence of jobs that minimizes the weighted sum of the completion time $\sum_{i=1}^{n} w_i C_i$, where C_i is the completion time of job i. Lawler showed that single–machine job sequencing is **NP**–hard even if all $w_i = 1$ or all $p_i = 1$ [48].

An example of the single–machine job sequencing problem is shown in Fig. 3.1, where all $p_i = 1$.

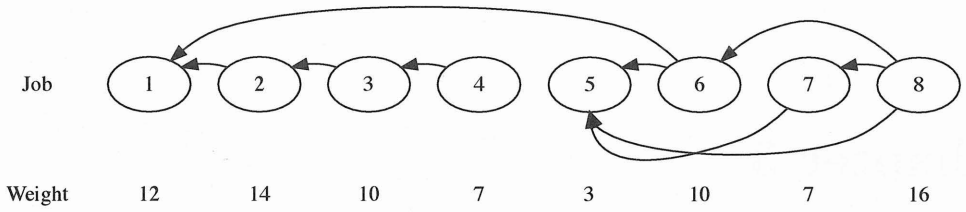

Figure 3.1: Example of job sequencing problem.

3.2 Rule List Reconstruction Method via Inclusive Rules

In this section, we present a rule list reconstruction algorithm based on an inclusive rule list.

If rule list $\mathcal{R}$ satisfies $\forall r_i, r_j \in \mathcal{R},\ O(r_i, r_j) \ \wedge\ i < j \Rightarrow M(r_i) \subset M(r_j)$, then $\mathcal{R}$ is said to be an *inclusive rule list*. An example of an inclusive rule list is given in Table 3.2.

Rule r_j that satisfies $\exists i,\ i < j \ \wedge\ M(r_j) \subset M(r_i)$ is redundant and can easily be removed from the rule list. In this study, an inclusive rule list does not contain such rules.

We prove that the graph of an inclusive rule list forms an oriented tree.

For rule list $\mathcal{R}$, graph $G_{\mathcal{R}}$ is defined as

$$\begin{aligned} &V = \{\ 1, \ldots, |\mathcal{R}|\ \}, \\ &E = \{\ ji \mid i, j \in V \ \wedge\ M(r_i) \subset M(r_j) \ \wedge \neg \exists k \ \big(M(r_i) \subset M(r_k) \wedge\ M(r_k) \subset M(r_j)\big)\}. \end{aligned} \tag{3.1}$$

An example of a graph for a rule list is shown in Fig. 3.2.

Definition 3.2.1. *(Oriented tree) If graph G satisfies the following three conditions, then G is said to be an oriented tree:*

1. $\forall v \in V\Big(v \neq r \ \Rightarrow\ \exists! e \in E\ v = end(e)\Big)$,
2. $\forall e \in E\ \Big(r \neq end(e)\Big)$,
3. *For all $v \in V$, there exists a path p from r to v,*

where r is the root of G and $end(e)$ is the end vertex of e.

Theorem 3.2.1. *For inclusive rule list $\mathcal{R}$, $G_{\mathcal{R}}$ is an oriented tree.*

Proof. For all r_v $(v \neq r)$, as $M(r_v)$ is a subset of $M(r_n)$, there is at least one edge e such that $v = end(e)$. We assume that

$$v = end(e_1) \ \wedge\ v = end(e_2) \ \wedge\ e_1 \neq e_2. \tag{3.2}$$

From Eq. (3.2) and the definition of overlap, $O(init(e_1), init(e_2))$, we have that $init(e)$ is a source vertex of e. Let $v_1 = init(e_1)$ and $v_2 = init(e_2)$. Because $\mathcal{R}$ is an inclusive rule list,

Table 3.1: Rule list.

Filter $\mathcal{R}$
$r_1 = 1\ 0\ *\ 1\ 1\ *$
$r_2 = 1\ 0\ *\ 0\ 0\ *$
$r_3 = *\ 0\ 0\ 0\ 1\ *$
$r_4 = 0\ *\ *\ 0\ 1\ *$
$r_5 = 0\ 1\ *\ *\ 1\ *$
$r_6 = *\ 0\ 0\ *\ *\ 0$
$r_7 = *\ *\ *\ *\ *\ *$
$L(\mathcal{R}) = 320$

Table 3.2: Inclusive rule list.

Filter $\mathcal{R}$
$r_1 = *\ 0\ 1\ 1\ *\ *$
$r_2 = 1\ 1\ 0\ *\ 0\ 0$
$r_3 = *\ 1\ 0\ *\ 0\ 0$
$r_4 = 1\ 0\ 0\ 1\ *\ *$
$r_5 = *\ 1\ 0\ *\ 0\ *$
$r_6 = *\ 0\ *\ 1\ *\ *$
$r_7 = *\ *\ *\ *\ *\ *$
$L(\mathcal{R}) = 358$

$M(r_1) \subset M(r_2)$ and $v_2v_1 \in E$ hold. Because v_1v and $v_2v_1 \in E$, $v_2v \notin E$. This is a contradiction for $v_2v \in E$. Thus, $\forall v \in V\Big(v \neq r \ \Rightarrow\ \exists! e \in E\ v = end(e)\Big)$ holds.

Because rule $r_n(= r_r)$ is not a subset of any rule, r is not an end vertex of any edge. Thus, $\forall e \in E\ \Big(r \neq end(e)\Big)$ holds.

For the set of rules $S_v = \{k \mid r_v \subset r_k\}$, we define the order $j <_{S_v} i \equiv r_i \subset r_j$ on S_v. Let P_v be a list of S_v that is sorted according to $<_{S_v}$. P_v with v added to it is a path from r to v. □

We now present a rule reconstruction algorithm that rewrites rule list $\mathcal{R}$ as inclusive rule list $\mathcal{R}'$ and applies the optimization method of [34] to $\mathcal{R}'$.

For rules r_i and r_j such that $O(r_i, r_j) \ \wedge\ M(r_i) \not\subset M(r_j)$, a rewriting algorithm searches position k such that $b_k^i =$ '*' $\wedge\ b_k^j \neq$ '*', generates $r_i^{'}$ and $r_i^{''}$, inserts them into the next r_i, and then removes r_i, where b_k of $r_i^{'}$ is expanded to 0 and b_k of $r_i^{''}$ is expanded to 1. The algorithm repeats this operation until the rule list becomes inclusive, and renumbers the rules. The rewriting of rule r_3 in Table 2.1 is shown in Fig. 3.3.

As there may be some variation in the rule weights in RORO, we can not always obtain the optimal order, even if the graph of preceding relations is a forest of oriented trees. Thus, the order obtained by reordering rules whose graph of precedence relation is a forest of oriented trees is near–optimal. However, in the case of ORO, such an order is always optimal.

3.2.1 Experiments

We demonstrate the efficiency of the proposed method through experiments. The rules and headers were generated with the benchmark packet classification algorithms ClassBench [80]. The number of headers was approximately $100k$. A total of 50 rule lists were generated from acl seed files. With the generated rule lists and header lists, we measured the latency using the proposed method, SGM [77], and SWBP [61]. Note that SGM and SWBP reorder the rules in ORO instead of RORO. Thus, r_j can not be placed ahead of r_i when $i < j$ and r_i overlaps r_j.