

Research Article

Pruned Kd-tree: a memory-efficient algorithm for multi-field packet classification



M. Rafiee¹ · M. Abbasi¹

Received: 3 September 2019 / Accepted: 29 October 2019 / Published online: 4 November 2019 © Springer Nature Switzerland AG 2019

Abstract

Packet classification is a basic process in most network-based packet processing systems. The key operation in this process is to match the packet header against the rules defined in a rule-set and, finally, to find the best matching rule. One of the well-known algorithms for packet classification is the Kd-tree algorithm. This algorithm produces a binary tree using the tuples created by the length of the prefix of the source and destination IP addresses of the rules. The tree is intended to classify the packets. The efficiency of the mechanism for producing and searching the binary tree in this algorithm is affected by two major disadvantages, namely, redundant nodes and redundant accesses during the search. The former increases memory consumption and the latter slows down packet classification. The proposed idea in this article is to prune redundant nodes by sorting the rules corresponding to each tuple in the tree nodes. The experimental results suggest that the throughput rate of the pruned Kd-tree is 43.64% higher than the Kd-tree. Also, by pruning the structure of the Kd-tree, our proposed method could at best reduce 24% of the memory consumed for the storage of the data structure of the Kd-tree.

Keywords Packet classification · Acceleration · IP networks · Kd-tree · Pruning · Memory

1 Introduction

The process of dividing network packets into different flows in Internet routers and firewalls is called packet classification. Packet classification specifies to which flow each packet belongs. Classification is based on matching one or more fields from the packet header against the corresponding values from the rules of the classifier. The most important fields of packet headers that are used for packet classification include source IP address, destination IP address, protocol type, source port number, and destination port number. When a packet is received, the table of rules is searched for a matching rule in accordance with the fields of the packet header. In addition to the valid values for the header fields, each rule specifies the action to be applied to the packet [1, 2]. The action determined

by the matching rule will be applied to the packet. Since a packet may be matched by several rules, the action determined by the rule with the highest priority will be performed. For example, one priority can be the length of the prefix of the source or destination IP addresses or the order of the appearance of a rule in a rule set [3–5].

One of the well-known packet classification algorithms is the Kd-tree algorithm [6]. This algorithm has a balanced binary tree structure. The tree represents the rules. Each node in this tree, therefore, contains rules that are partly similar to each other. The rules that are mapped onto a node in the tree have identical tuples. That is, the prefixes of the source and destination IP address fields of the rules mapped onto a tree node will have the same length. It should be noted that the Kd-tree algorithm is a

M. Abbasi, abbasi@basu.ac.ir; M. Rafiee, m.rafiee@alumni.basu.ac.ir | ¹Department of Computer Engineering, Engineering Faculty, Bu-Ali Sina University, Hamedan, Iran.



SN Applied Sciences (2019) 1:1537 | https://doi.org/10.1007/s42452-019-1592-z

combination of tree-based and tuple space methods. In other words, a binary search is performed on the tuple space.

This study proposes an optimized version of the Kd-tree packet classification algorithm by pruning the redundant nodes of the tree. The most important criteria for the efficiency of the algorithm are packet classification time and the space required to store the structure of the Kd-tree algorithm. For this purpose, we first examine the tree structure of the Kd-tree algorithm as well as how this algorithm classifies packets. Then, we will show how pruning the tree structure created by the rules and changing the packet classification model in the pruned tree will reduce the time for traversal as well as packet classification. Also, we closely examine how to reduce the space needed to store the tree structure of the Kd-tree algorithm.

The structure of the article is organized as following. First, we shall review related works on the structure of the Kd-tree algorithm as used in tuple space search as well as the use of markers in this algorithm. In the third section, the proposed method for pruning the Kd-tree and producing a pruned Kd-tree is examined. Implementation and evaluation of the proposed method are explained in Sect. 4. The final section of the paper is dedicated to concluding and presenting solutions for further development of research in this field.

2 Related works

2.1 Decision tree based packet classification algorithms

Decision-tree based algorithms [5, 7, 8] are the most popular algorithms for packet classification. Their success is owned by the key idea of recursively cutting the search space into smaller sub-spaces, each of which corresponding to a child of node in a decision-tree. Such a recursive terminates when the number of rules in tree nodes becomes lower than a predefined threshold.

To classify an incoming packet, information of certain fields of its header are extracted and then used to traverse the tree. During traversal from root to leaf nodes, the algorithm stores the best matching rule based on its specific policy.

There are a few variations of the decision tree that differ on the method of constructing the tree via cutting the search space and the way of traversing the tree for classifying packets. For example, HiCuts [7] constructes the tree via multiple evenly-spaced cuttings on a single dimension at each iteration. HyperCuts [8] differs from HiCuts only in allowing multiple dimensions to be cut concurrently to moderate the height of the constructed tree. To resolve

the memory blow up problem caused by rule replication of HyperCuts, HyperSplit [5] uses non-equal cuts for controlling the memory usage. As shown in Fig. 1a, HiCuts algorithm cuts the search space into two equal-sized subspaces. The Rule 2 and Rule 3 are replicated in the respective subspaces. By aligning the cuttings at the edges of the rules, the HyperSplit algorithm is able to reduce rule replication (Fig. 1b). However, HyperSplit still cannot eliminate all rule replication, especially for complex rule sets.

However, this algorithm is not as fast as HyperCuts. A key solution to this problem is to implement the algorithm on FPGA [9]. In this hardware implementation of HyperSplit, a pipelined architecture accelerates the classification process. However, due the limited resources of the programmable devices limits the extensibility and customizability of the hardware packet classifiers. Also, the considerable design costs of hardware classifiers makes their performance to cost ratio smaller than that of the corresponding software classifiers. For this reasons, software packet classifiers are more interested.

The common issue in using software packet classifiers is their inability in achieving maximum speed and minimum memory usage, simultaneously. Therefore, the memory usage of fast tree-based packet classification algorithms may grow exponentially as the number of rules increases. In this paper we show how using simple but effective tricks in pruning redundant tree nodes would reduce the memory consumption of the algorithm and increase the classification speed.

2.2 Kd-tree algorithm based on tuple space search

In spite of the large number of rules in a classifier, often the majority of prefixes have the same length. Using this fact, the rules of a classifier are divided into separate groups based on their prefix length.

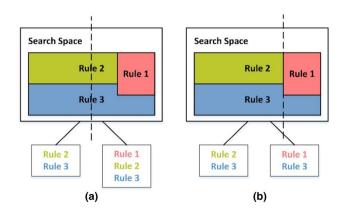


Fig. 1 a Unaligned cut point (causing replication of Rule 2); **b** Aligned cut point to the boundary of Rule 1 and Rule 2 (reducing replication)

It is clear that the number of rules that are placed in a category with the same prefix length is less than the total number of rules. For example, consider a classifier in a traditional router that sends packets based on the destination IP address. Such a classifier, regardless of the number of rules, can have up to 32 distinct groups. In a two-dimensional classifier in which rules are defined based on the source and destination IP addresses, a maximum of 1024 separate groups can be conceived. However, many of the tuples may have no specific rule. In general, if you consider a d-dimensional classifier, each rule is mapped onto a vector with a set of d integers in which the ith integer denotes the length of the prefix of the ith field of the rule. A vector with d integers is called a tuple. A set of tuples created by the classifier is called tuple space. The modest algorithm for tuple space search is to check all the tuples.

2.3 Definitions

A tuple T is a vector of d integers that is defined as T.vec[1], T.vec[2], ..., T.vec[d]. In this definition, T.vec[i] denotes the integer value corresponding to the ith item in T. An f rule matches a tuple if and only if $\forall i$, $1 \le i \le d$ and the length of the ith field of f is exactly T.vec[i].

The space of tuple T can be divided into three partitions, i.e., LongerTuple, ShorterTuple, and IncomparableTuple. Suppose two tuples T and T_a and let $T_a \neq T$ ($\exists i, 1 \leq i \leq d, T_a.vec[i] \neq T.vec[i]$). T_a is a tuple in the LongerTuple partition of T if $\forall i, 1 \leq i \leq d, T_a.vec[i] \geq T.vec[i]$. Also, if $\forall i, 1 \leq i \leq d, T_a.vec[i] \leq T.vec[i]$, then T_a is a tuple in the ShorterTuple partition of T. Otherwise, T_a is a tuple in the IncomparableTuple partition of T.

To perform a binary search in the tuple space, a binary tree should be produced using tuples. For this purpose, each tuple is mapped onto a unique value called *Super-Key* which specifies the position of the tuple on the tree. The SuperKey of tuple T is generated by joining all the elements in T.vec in a rotational order. For this purpose, a discriminator is used to determine the place where the elements should be joined. $SK_{T.dis}$ represents the SuperKey of T and dis index represents the discriminator. Therefore, $SK_{T.dis} = T.vec [dis] T.vec [dis + 1], ..., T.vec [d] T.vec [1] T.vec [2], ..., T.vec [dis - 1]. For example, <math>SK_{(3,0).1} = 30$ and $SK_{(3,0).2} = 03$. Since there are d elements in T.vec, tuple T can have d SuperKeys that may begin at any element 1, 2, 3, ..., d.

As SuperKeys are integers, they can be easily sorted. As a result, the tuples can be sorted by their SuperKeys. Assuming dis as the discriminator, T_a is smaller than T if $SK_{T_a,dis} < SK_{T,dis}$. Otherwise, T_a would be greater than T. For example, let dis = 1. Then the tuple (3, 0) is greater than the tuple (2, 2) because $SK_{(3,0),1} > SK_{(2,2),1}(30 > 22)$. If the discriminator of all the tuples equals 2, the tuple

(3, 0) would be smaller than the tuple (2, 2) because of $SK_{(3,0),2} < SK_{(2,2),2}(03 < 22)$.

2.4 The structure of the Kd-tree algorithm

Rules that are mapped onto a tuple can be stored in a hash table. Tuples can be considered as points in a multidimensional space. Over the past three decades, many data structures have been developed for organizing multidimensional objects, including Kd-tree [10], KDB-tree [11], R-tree [12], R+-tree [13], and R * -tree [14]. Among these data structures, Kd-tree provides a simple and convenient method. More information on this structure can be found in several recent studies [15–18].

In our proposed method, each node of a Kd-tree holds a tuple T, and two pointers are connected to the right and left sub-trees. According to the explanations given in Sect. 2.3, all the tuples of the left-hand sub-tree of a tuple T are smaller than T itself whereas all the tuples in the right-hand sub-tree of T are greater than T. Note that the tuples of the left-hand sub-tree of T belong to the ShorterTuple or IncomparableTuple of T. Also, the right-hand tuples of T belong to the LongerTuple or IncomparableTuple of T.

To simplify the formation of a Kd-tree algorithm, assume that only the fields of source and destination IP address of the rules are checked. The sample classifier works with the ten rules in Table 1. In this table, the first column shows the number of rules. The second and third columns represent the source and destination IP addresses, respectively. The fourth column represents the tuple corresponding to the rule. For example, in the case of R0 rule, the number of the bits of the source IP address prefix is 4 and that of the destination IP address prefix is 3. Therefore, the tuple corresponding to this rule is (4, 3).

Table 1 An example of the rules of a classifier

Rule	Source IP address	Destination IP address	Tuple
R0	1010*	011*	(4, 3)
R1	1010*	0000*	(4, 4)
R2	100*	101*	(3, 3)
R3	1*	010*	(1,3)
R4	10110*	1000*	(5, 4)
R5	11*	011*	(2, 3)
R6	10111*	01001*	(5, 5)
R7	*	0*	(0, 1)
R8	101*	01*	(3, 2)
R9	11*	1111*	(2, 4)

^{*}Symbol in all patterns of source IP address and destination IP address denotes a part of the address which is unimportant to the classifier engine

To select the root node or a sub-tree of the Kd-tree, SuperKeys are created to sort the tuples. The tuples are first sorted according to their SuperKeys, and then the middle tuple in the sorted list is selected as the root node or a sub-tree of the Kd-tree. Tuples with SuperKeys smaller than that of the selected tuple are assigned to the left-hand sub-tree of the tuple, and the rest of the tuples are assigned to the right-hand sub-tree of the root tuple. This process is repeated for the tuples assigned to the left-hand and right-hand sub-trees. In other words, to select the root of the left-hand and right-hand tree, the existing tuples are once again sorted using a new discriminator, and the middle tuple of the list is selected as the root of the sub-tree. The following formula is used to determine the discriminator field in creating the SuperKey of tuples at the Lth level:

$$dis = (L \, mod \, d) + 1 \tag{1}$$

In the above equation, d denotes the number of dimensions of the classifier. In fact, with the help of Eq. (1), fields 1, 2, 3, ..., d are selected in rotation as the discriminator. The tree shown in Fig. 2 is based on the source and destination IP addresses in Table 1. In this tree, the elliptical nodes correspond to the tuples in the fourth column of Table 1. Also, the rules belonging to each tuple are shown with the symbol R[]. For example, tuple (3, 2) has an R8 rule.

In this example, the tuples are initially sorted by the discriminator dis=1. Accordingly, the SuperKeys corresponding to the tuples in Table 1 are obtained and sorted in the form of (0,1), (1,3), (2,3), (2,4), (3,2), (3,3), (4,3), (4,4), (5,4), (5,5). The middle tuple (3,2) is selected as the root node of the Kd-tree. The tuples (0,1), (1,3), (2,3), (2,4) fall under the left-hand sub-tree of the root and the tuples (3,3), (4,3), (4,4), (5,4), (5,5) under the right-hand sub-tree. Then the discriminator dis=2 is selected, and the tuples in the left-hand and right-hand sub-trees are sorted again. The sorted sequences are (0,1), (1,3), (2,3), (2,4) and (3,3), (4,3), (4,4), (5,4), (5,5). The middle tuples (1,3) and (4,4) are

selected as the root nodes of the left-hand and right-hand sub-trees, respectively. By repeating this process, the tree of the Kd-tree algorithm is constructed.

2.5 Search in Kd-tree algorithm and the misjudged problem

This section examines an example of searching in the Kdtree. Assume a packet with the source IP address (1001 0011 1010 1111 1111 1111 0011 0100) and the destination IP address (0010 0111 0000 1011 0011 0101 1111 1111). Given the rules in Table 1 and the corresponding Kd-tree depicted in Fig. 2, the act of classification for this packet will be as following. In the search process, BstMatch variable is used. This variable holds the first matching rule until the search is finished and a leaf node is reached.

First, traversal starts from the root node of the Kd-tree. which is the tuple (3, 2) in this example. Therefore, this tuple should be checked. To check (3, 2), if it contains a rule according to the number of the prefix bits of the source and destination IP addresses, it will be matched against the input packet. Therefore, the packet matches the rules in the form of (100*, 00*). In this example, the input packet does not conform to any rule because the prefixes of R8 cannot be matched with the prefixes of the packet. Since this tuple has returned the result of mismatch, the tree is further traversed on its left-hand child. In the next step which consists of matching against the tuple (1, 3), the search algorithm uses 1 bit of the prefix of the source IP address and 3 bits of the prefix of the destination IP address for examining the rules in the tuple (1, 3). At this stage, the algorithm cannot find any matching rule. Due to the lack of matching rules, therefore, traversal is led to the left-hand sub-tree of this tuple. Then the rules in the last tuple, i.e. (0, 1), are checked. In this step, Rule 7 is returned as the best matching rule because the algorithm has already reached a leaf node.

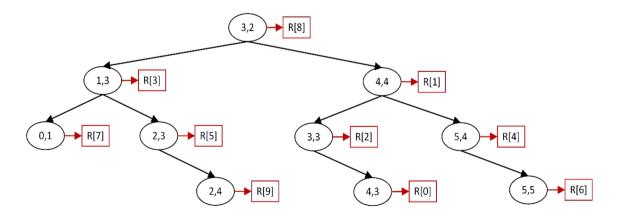


Fig. 2 The basic Kd-tree produced from the rules in Table 1

As the above description shows, performing a traversal for packet classification by the Kd-tree is not flawless. For example, suppose an input packet for the matching operation has a source IP address of (1100 0011 1011 0010 0111 0110 1111 1111) and a destination IP address of (1111 1011 0100 1110 1111 1111 0100 0000). At the root of the tree, the packet has not been matched by any of the rules of (3, 2) and, as a result, traversal continues at the left-hand sub-tree. In the next step, the input packet does not match any of the rules in the tuple (1, 3) and again traversal continues at the left-hand sub-tree of the tuple (1, 3). The root tuple of this sub-tree is (0, 1) whose rules do not match the packet. Finally, the search is over without any result. In a linear search, however, the packet would easily match the rule R9. A solution to this problem are markers. The next section explains how a marker is used in the basic algorithm.

2.6 Use of markers in the Kd-tree algorithm

When a search is not successful on a tuple, the search algorithm continues the traversal on the left-hand subtree and removes all the tuples of the right-hand subtree from the traversal. To avoid this problem, which is called the misjudged problem, every tuple should retain information about the rules of its right-hand sub-tree. For example, the tuple (1, 3) must have a marker that produces R5 rule in the tuple (2, 3). Also, the tuple (1, 3) must have a marker of R9 rule related to the tuple (2, 4). Figure 3 shows the Kd-tree of the rules in Table 1 along with added markers. Markers in the tuples are indicated by M [...].

In the search for the best matching rule from the previous example, traversal cannot find any matches in the tuple (3, 2) and continues on the left. When matched against (1, 3), the search is successful due to the presence of the R9 marker. Finally, the search algorithm continues

to traverse the right-hand sub-tree and reports R9 rule in the tuple (2, 4) as the best matching rule.

3 Pruned Kd-tree algorithm

In this section, we describe the pruned Kd-tree method which significantly improves the Kd-tree algorithm. Assume an input packet with fields that can be best matched by R6 in Fig. 3 (a source IP address of (1011 1111 0010 0101 0001 1111 1001 1101) and a destination IP address of (0100 1101 1010 0001 0110 1111 1011 1111)). Figure 4 shows the tree traversal path to find the best matching rule. The tree nodes to be examined are shown in shaded shapes. Traversal starts from the root node. To search for the best matching rule for this packet by means of the Kd-tree algorithm, in each node a certain length of the source and destination IP address of the packet is compared with the corresponding prefixes from the rules mapped onto the tuple of the node. For example, in the root node which contains the tuple (3, 2), the input packet is compared with only 3 bits of the prefix of the source IP address and 2 bits of the prefix of the destination IP address of the existing rules. Therefore, as the figure shows, all the rules in a tuple must be checked. In this example, traversal starts from the root towards the right-hand child of (4, 4). Then the tuple (5, 4) and, finally, the tuple (5, 5) are traversed. Finally, as the traversal path reaches a leaf node, the rules of that node are also examined and R6 rule is reported as the result. In this example, 14 nodes are checked in the traversal path. In the following, we will show that the idea of pruning a Kd-tree can significantly reduce the number of accessed nodes. Thus, the search speed in this algorithm increases significantly.

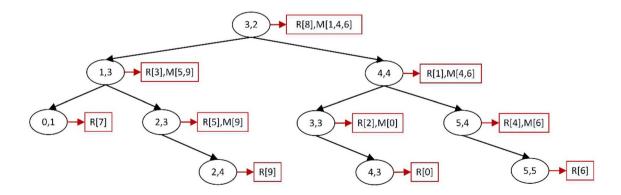


Fig. 3 The Kd-tree with markers of the rules from Table 1

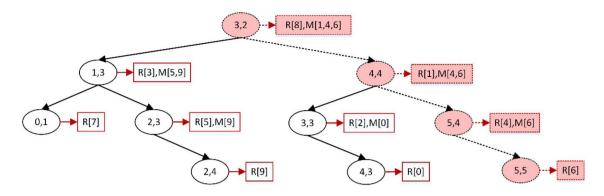


Fig. 4 Traversal of the Kd-tree

3.1 Suggested techniques for pruning a Kd-tree

Our examination of Kd-trees created by various rules resulted in the following ideas for tree pruning.

Technique 1 Pruning the right-hand leaves

In the Kd-tree data structure, by replacing the markers of a node with the rules that correspond to the tuple of the right child of the node and matching the entire prefix bits against the input packet, we can prune the right child if it is a leaf node.

In other words, all the rules in the right child node will also exist in the parent as markers. If the right child node is a leaf node, it can be pruned by transferring the rules to the parent; therefore, instead of matching the marker, it is necessary to match the rules against the input packet.

Obviously, pruning some of the tree's redundant nodes will reduce the memory needed to store the tree structure. On the other hand, sometimes the tree's depth will decrease with the removal of leaf nodes. Another advantage of pruning a tree is to reduce the number of memory accesses during the traversal.

Technique 2 Sorting by the priority of rules

If in each node corresponding to a tuple all the existing rules have been sorted in descending order according to the prefix length, memory access will be significantly reduced when searching for the best matching rule. The rules are sorted based on the length of the sum of the source and destination IP address prefixes.

Indeed, sorting the rules by prefix length makes it possible that, when the input packet is matched with the first rule during the traversal, the search could be finished at that tuple and continue on the right-hand sub-tree of the node, if any, to find matching rules corresponding to longer tuples. Because this method does not need to necessarily check all the nodes in the tuple, the number

of memory accesses and the time for packet classification is reduced.

3.2 The Structure of pruned Kd-tree algorithm

In this section, using the techniques mentioned above, we prune a Kd-tree. Algorithm 1 describes how to create a pruned Kd-tree. The input and output of this algorithm are Kd-tree and pruned Kd-tree. To prune the tree, all nodes are checked (lines 1 through 7). If the examined node is a leaf node, it will be pruned as long as it is the right child of its parent (lines 2 to 6).

```
Algorithm 1. Creating a pruned kd-tree
Input: kd-tree
Output: Pruned kd-tree
  Pruning:
1: for all Nodes do
2:
      while Node == leaf do
3:
          if the Node is the rightchild of its parent then
4:
            Delete the tuple
5:
          end if
6:
     end while
7: end for
  Sorting:
8: for all tuples do
   Sort rules in the tuple by the prefix
10:end for
```

After the tree is pruned, it is necessary to sort the rules in the nodes corresponding to each tuple. For this purpose, all the nodes of the tuples are examined, and their rules are sorted by the prefix length in descending order (lines 8 to 10).

Figure 5 represents the Kd-tree corresponding to Table 1. In this tree, the leaf nodes that are the right child of their parent are marked in gray. In the next step, these

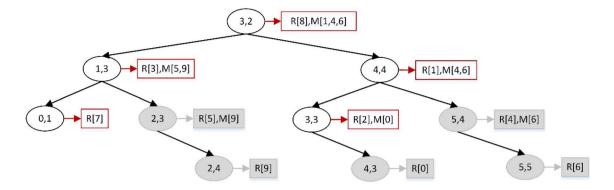


Fig. 5 Nodes to be pruned by Kd-tree algorithm

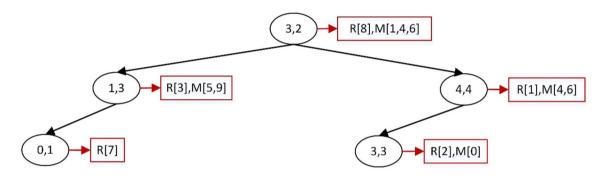


Fig. 6 The Kd-tree after pruning some nodes

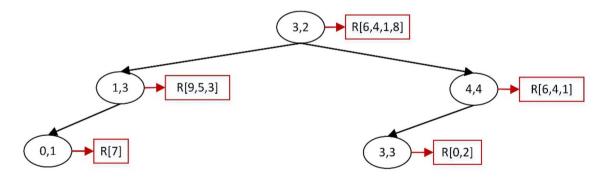


Fig. 7 The tree produced by the implementation of the proposed method of pruned Kd-tree on the rules in Table 1

nodes are eliminated, resulting in the removal of 12 nodes from the tree. Figure 6 shows the structure of the Kd-tree after pruning.

Figure 7 shows the final version of the tree shown in Fig. 6. In this trie, all of the markers in each node are added to the rules set R. Next, these rules are sorted descending

by their prefix length. For example, the root node which has the markers 1, 4, 6 and the rule R8, is converted to R [1, 4, 6, 8]. In this node, rule R6 has the longest prefix as compared to others. R4 is the second of the longest-prefix rules. The rules are sorted in this way in all of the tree nodes.

3.3 Search in a pruned Kd-tree

Search in a pruned Kd-tree is done as follows. Traversal starts from the root and the rules in each tuple node are checked linearly. The rules are examined in their entirety, i.e. the full length of the prefix of source and destination IP addresses, source and destination port numbers, and protocol. If the packet matches a search rule, the search in the rules of that tuple is stopped and directed to the right-hand sub-tree of the tuple.

Algorithm 2 shows how to search and classify packets based on the data structure of the pruned tree. The input of this algorithm is the rule set R, the pruned Kd-tree T, the tuples, and the header of input packets. The output is rulesIndexArray which holds the index of the best matching rule of every incoming packet. Packet classification starts from the root node (line 2). Initially, BMR variable that holds the best matching rule for a packet is set to Null (line 3). The traversal of the tree continues until a leaf node is reached (lines 4 through 13). On the traversal path, the different fields of the rules in the tuples are compared with the input packet. If the packet matches a rule, the number of that rule will be stored in rldx (line 6). Then, if the packet matches a rule in the tuple, the result will be stored in BMR and traversal will be directed towards the right child; otherwise, the left child will be traversed (lines 7 to 12). At the end of the traversal, the result of packet classification is stored in rulesIndexArray which is an array used for holding the results (line 14).

```
Algorithm 2. Search using the pruned kd-tree
```

```
Input: Rules R, Pruned kd-tree T, headers H
Output: rules Index Array
1: for all |H| do
2.
           Node \leftarrow root \ of \ T
3.
           BMR \leftarrow Null
4:
          while Node! = leaf do
5.
              rIdx \leftarrow Null
6:
              rIdx \leftarrow Match(rules \ of \ Node, H, R)
7:
              if \ rIdx ! = Null \ then
8:
                  BMR \leftarrow rIdx
Q.
                  Node \leftarrow node's \ right \ child \ in \ T
10:
               else
11:
               Node \leftarrow node's left child in T
12:
              end if
          end while
13:
          rulesIndexArray(H,BMR)
14:
15: end for
```

Table 2 compares the number of memory accesses required for classifying same packets using basic Kd-tree and its enhanced version.

Figure 8 illustrates searching this prund Kd-tree for first packet of Table 2. It is assumed that the packet matches R6. Multiple nodes and rules that are examined on the traversal path are shown in red. This search, which would otherwise need 14 memory accesses in the Kd-tree algorithm, is performed with only four memory accesses in the proposed method.

Table 2 Number of memory-accesses required for Kd-tree and pruned Kd-tree

Input packet	Algorithm	Algorithm		
Source IP address	Destination IP address	Kd-tree	Pruned Kd-tree	
10111111001001010001111110011101	01001101101000010110111110111111	14	4	R6
10110010000110011001100000000000	100000000000110101001100111001	9	6	R4
110000111011001001110110111111111	111110110100111011111111101000000	14	7	R9

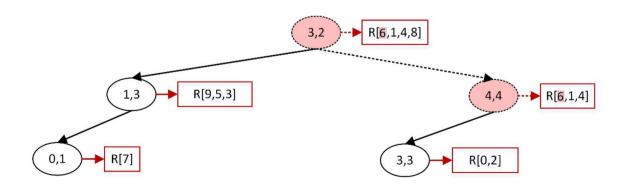


Fig. 8 Search using the pruned Kd-tree algorithm

Table 3 System specifications

Specification	Processor
Name	Intel Core i7-740QM
Clock speed	1733 MHz
L1 data cache	4×32 KB
L1 inst. cache	4×32 KB
L2 cache	4×256 KB
L3 cache	6 MB
Main memory	4 GB DDR3
Operation system	Windows 7 ultimate, 64-bit

In classifying the second packet of Table 2, the Kdtree algorithm examines rule 8 in the tuple (3, 2) and for its inconsistency checks the marker of this tuple. Hence, the packet is examined against the markers of rule 1 and rule 4, and due to matching with the marker of rule 4, the subsequent search is directed to the right subtree. Then, the packet is compared with rule 1 and marker 4 in the tuple (4, 4), which finally is matched with rule 4. The number of memory accesses in classifying this packet using Kd-tree is nine. But, in pruned Kd-tree, the packet is first examined against rules 4 and 6 in the tuple (3, 2), and then is compared with rules 4 and 6, with only six memory accesses. Similarly, the number of required memory accesses for classifying the third packet of Table 2 is computed and presented. In these examples, the number of memory accesses of the proposed algorithm is lower than that of the basic algorithm.

Table 4 The number of memory accesses with the ACL rule set

Packet	Access memory	Kd-tree algo	orithm			Pruned Kd-t	ree algorithm		
		1k	5k	10k	20k	1k	5k	10k	20k
512	Max	544	3142	5699	12,030	426	2319	3980	9381
	Min	185	948	1587	2924	77	91	98	350
	Sum	130,841	596,352	964,795	1,872,710	122,359	523,232	882,270	1,678,095
1k	Max	544	3142	5699	12,030	375	2319	4497	9021
	Min	185	948	1576	2924	28	91	98	280
	Sum	255,201	1,207,024	1,957,944	3,576,870	239,219	1,048,327	1,738,695	3,224,380
2k	Max	544	3142	5699	12,030	426	2319	4497	9465
	Min	185	937	1576	2893	14	14	14	70
	Sum	517,771	2,246,115	4,057,069	10,246,641	477,216	2,026,514	3,523,529	5,340,693
4k	Max	544	3142	5699	12,030	438	2319	4553	9393
	Min	185	937	1577	2893	14	14	14	70
	Sum	1,047,587	4,689,638	7,985,342	13,772,088	964,999	4,134,084	7,036,544	12,523,765
8k	Max	544	3142	5699	12,030	432	2364	4553	9489
	Min	185	937	1577	2893	14	14	14	49
	Sum	2,063,609	9,196,415	14,508,307	28,847,845	1,935,299	8,192,319	13,869,101	25,545,303
16k	Max	544	3142	5699	12,030	438	2364	4553	9423
	Min	185	937	1576	2895	14	14	14	49
	Sum	4,078,028	17,468,936	30,468,442	55,962,934	3,839,218	16,175,042	17,852,162	50,669,497
32k	Max	544	3142	5699	12,030	438	2364	4560	9495
	Min	185	937	1576	2924	14	14	14	42
	Sum	8,098,792	38,481,691	63,569,548	125,007,952	7,680,932	32,501,442	56,179,615	106,443,702
64k	Max	544	3142	5699	12,030	438	2364	4560	9495
	Min	185	937	1576	2893	14	14	14	21
	Sum	15,646,914	77,127,273	134,153,228	231,739,723	15,517,797	67,965,278	113,712,252	205,476,450
128k	Max	544	3142	5699	12,030	438	2364	4560	9495
	Min	185	937	1576	2893	14	14	14	21
	Sum	33,141,660	148,833,606	262,013,018	433,884,540	30,668,615	132,008,892	222,438,007	402,884,488
256k	Max	544	3,142,937	5699	12,030	438	2364	4560	9495
	Min	185	937	1576	2893	14	14	14	14
	Sum	71,756,336	301,774,440	501,780,227	922,711,260	57,690,021	260,490,897	444,624,839	820,648,686

Table 5 The number of memory accesses with the FW rule set

Packet	Access memory	Kd-tree algor	rithm			Pruned Kd-tre	ee algorithm		
		1k	5k	10k	20k	1k	5k	10k	20k
512	Max	1199	3130	5159	12,454	1033	2581	4965	9991
	Min	422	1039	1931	4124	65	65	70	305
	Sum	356,443	1,136,854	1,874,853	3,557,947	255,322	853,432	1,457,881	3,221,409
1k	Max	1199	3130	6186	12,454	1033	2581	6163	9991
	Min	422	1039	1931	4215	20	65	70	150
	Sum	668,357	2,023,881	3,210,080	7,090,479	549,917	1,840,454	3,052,962	6,285,888
2k	Max	1199	3130	6186	12,503	1086	2581	6181	12,496
	Min	422	1039	1910	4124	10	65	70	50
	Sum	1,341,260	3,982,774	6,783,093	14,179,408	1,072,978	3,686,741	6,188,172	12,377,521
4k	Max	1199	3130	6186	12,503	1086	3057	6179	12,503
	Min	422	1039	1910	4124	10	10	10	50
	Sum	2,678,963	7,991,672	14,517,075	28,159,817	2,154,155	7,326,447	12,763,918	24,901,899
8k	Max	1199	3130	6186	12,503	1084	3064	6176	12,503
	Min	422	1006	1910	4124	10	10	10	35
	Sum	5,460,157	16,415,274	31,978,990	55,019,262	4,301,367	14,517,171	19,193,383	50,439,626
16k	Max	1199	3130	6186	12,503	1086	3055	6180	12,503
	Min	422	1006	1910	4124	10	10	10	35
	Sum	10,894,541	37,703,706	50,566,605	110,212,121	8,396,090	32,417,792	49,128,771	99,489,160
32k	Max	1199	3130	6186	12,503	1086	3064	6180	12,502
	Min	422	1006	1910	4124	10	10	10	30
	Sum	23,157,719	65,556,229	100,929,684	223,898,085	16,296,924	57,797,109	98,517,382	200,488,788
64k	Max	1199	3130	6186	12,503	1086	3064	6186	12,503
	Min	422	1006	1910	4124	10	10	10	26
	Sum	42,492,153	131,326,833	223,530,631	455,442,469	34,756,397	115,950,625	192,205,381	398,400,842
128k	Max	1199	3130	6186	12,503	1086	3064	6186	12,503
	Min	422	1006	1910	4124	10	10	10	15
	Sum	85,833,341	262,031,623	446,157,629	891,605,570	71,234,413	232,741,927	376,538,129	788,539,632
256k	Max	1199	3130	6186	12,503	1086	3064	6186	12,503
	Min	422	1006	1910	4124	10	10	10	10
	Sum	175,974,550	504,174,306	874,611,717	1,787,756,828	140,608,710	484,286,547	775,062,307	1,603,978,979

4 Implementation and evaluation

To implement the proposed packet classification algorithm, C++ language was used on a system with the specifications in Table 3.

The rule set and packets needed to evaluate and test the proposed algorithm was created using ClassBench tool [19]. Three general rule sets were created by ClassBench, including Access Control List (ACL), Firewall (FW), and IP Chain (IPC). We used rule sets of varying sizes for our evaluations. Each generated rule set was named according to its type and size. For example, ACL2_3K refers to an ACL rule set with 3000 rules.

For our evaluations, we used rule sets of 1k to 20k and packets of 512 to 256k. In the following, the results

of testing the proposed algorithm and its comparison with the typical Kd-tree algorithm are presented and discussed.

4.1 Number of memory accesses

One of the criteria for evaluating packet classification algorithms is the number of memory accesses. When a packet is being classified by our method, after reading the packet header fields, the nodes of the tree are traversed. Traversal starts from the root of the tree and proceeds to the leaf node according to the search mechanism of the proposed method. If a node has rules, those rules are also checked linearly. Of course, this part of the search can be done using a hash table which reduces access time in

Table 6 The number of memory accesses with the IPC rule set

Packet	Access	Kd-tree algor	ithm			Pruned Kd-tre	ee algorithm		
	Memory	1k	5k	10k	20k	1k	5k	10k	20k
512	Max	1177	4505	8902	18,300	1035	4197	8387	16,768
	Min	518	2272	4571	9258	24	52	56	244
	Sum	345,996	1,386,816	2,760,527	5,594,128	255,068	1,166,789	2,373,728	4,426,525
1k	Max	1177	4505	8902	18,300	1035	4201	8393	16,736
	Min	518	2272	4571	9258	16	52	56	120
	Sum	672,580	2,933,798	5,664,583	11,160,473	496,039	2,251,819	4,478,732	9,294,388
2k	Max	1177	4505	8902	18,300	1035	4197	8397	16,768
	Min	518	2272	4571	9258	8	8	8	40
	Sum	1,499,426	5,612,115	10,982,065	22,795,948	1,131,262	4,640,361	9,054,679	19,039,423
4k	Max	1177	4505	8902	18,300	1035	4206	8394	16,768
	Min	518	2272	4571	9258	8	8	8	40
	Sum	2,831,981	10,907,906	21,219,865	46,511,349	1,903,639	9,228,581	18,675,042	36,361,780
8k	Max	1177	4505	8902	18,300	1035	4209	8399	16,768
	Min	518	2272	4571	9258	8	8	8	28
	Sum	5,518,273	23,191,884	43,251,250	89,222,546	4,368,599	18,740,797	36,891,300	73,558,586
16k	Max	1177	4505	8902	18,300	1035	4209	8399	16,754
	Min	518	2272	4571	9258	8	8	8	24
	Sum	11,186,350	46,906,430	90,059,127	179,097,795	7,937,502	63,000,357	72,764,365	148,563,167
32k	Max	1177	4505	8902	18,300	1035	4209	8399	16,769
	Min	518	2272	4571	9258	8	8	8	24
	Sum	23,033,832	89,320,318	179,833,731	370,009,708	16,239,111	65,876,786	145,825,021	291,522,203
64k	Max	1177	4505	8902	18,300	1035	4210	8399	16,769
	Min	518	2272	4571	9258	8	8	8	12
	Sum	46,402,561	179,393,966	358,063,296	719,578,148	32,402,640	146,457,305	291,276,079	585,812,370
128k	Max	1177	4505	8902	18,300	1035	4210	8399	16,769
	Min	518	2272	4571	9258	8	8	8	12
	Sum	85,383,655	359,159,114	715,062,556	1,554,271,563	61,234,320	291,887,858	577,484,603	1,302,536,756
256k	Max	1177	4505	8902	18,300	1035	4210	8399	16,769
	Min	518	2272	4571	9258	8	8	8	8
	Sum	177,781,262	762,191,145	1,444,735,988	2,984,245,224	130,884,008	579,482,047	1,164,547,320	2,275,663,443

examining the rules of a tuple. In this research, we have used the linear method in searching the rules of each tuple to illustrate the effect of the pruned Kd-tree. The number of memory accesses can be defined as the total number of traversed nodes along with the number of examined rules. In the following, we will examine the number of memory accesses in both Kd-tree classification algorithm and the proposed method with regard to the different rule sets and different numbers of rules and packets. Table 4 represents the number of memory accesses related to the ACL rule set. The table shows the lowest and highest amount of memory accesses per packet, and the total number of memory accesses for all the packets.

In Table 4, it can be seen that the total number of memory accesses has increased in both algorithms with the increase in the number of packets. As the number of rules increases, the number of memory accesses will increase too. In the worst case of classification, the number of accesses equals the depth of the tree plus the total number of rules available in the tuple nodes on the path from the root to the leaf. For example, with 512 packets and 1k rules, the Kd-tree algorithm has a maximum number of 544 memory accesses. In our method, however, the number of memory accesses decreased to 426.

On the other hand, the minimum number of memory accesses in the proposed method is less than that of the Kd-tree algorithm. For example, in classifying 2k packets according to 5k rules, the proposed method has at least 923 less accesses to system memory than the Kd-tree algorithm. In this case, the Kd-tree algorithm makes 937 accesses, and the proposed method makes 14 accesses.

Table 7 Classification time with the ACL rule set

Packets	Rules	Rules				
	1k	5k	10k	20k		
Kd-tree						
512	52.9	200.1	426.2	713		
1k	101.4	405.1	889.5	1442.4		
2k	201.2	760.8	1823.2	3596.2		
4k	396.5	1534.5	3614.5	5242		
8k	803.3	3127.1	6540	10,324.8		
16k	1580.5	5726.7	13,749.6	20,172.1		
32k	3157.2	12,477.5	28,455.6	44,922.5		
64k	6436.6	25,275.4	59,320.5	84,939.4		
128k	11,682.5	48,804.5	115,642	165,533		
256k	24,690.8	135,354	218,268	352,803		
Pruned Kd	-tree					
512	46	174.9	396.3	644.1		
1k	93.4	350.9	784.1	1113.5		
2k	177.5	669.2	1579.9	2078		
4k	373.1	1399.5	3168	4609.1		
8k	720	2698.2	6222.8	9177.8		
16k	1532.7	5332.9	12,485.9	18,873.7		
32k	2957.4	10,803.6	25,162.8	39,105.3		
64k	5804.4	22,255.2	50,568.6	73,975.3		
128k	11,023.2	47,713	99,622.1	158,810		
256k	19,848.4	118,386	196,271	318,850		

Table 8 Classification time with the FW rule set

Packets	Rules			
	1k	5k	10k	20k
Kd-tree				
512	132.5	420.9	703.8	1540.9
1k	248.6	780.2	1206.8	3068.1
2k	515.3	1716.8	2532.9	5897.3
4k	1003.1	3369.3	5423.6	11,591.8
8k	2188.8	6424.8	12,028.9	20,572.5
16k	4201.4	14,302.9	19,089.8	37,940.3
32k	8975	25,120.4	39,453.4	71,972.3
64k	16,773.1	47,973.5	86,182.2	144,052
128k	33,901.9	95,255.8	178,078	355,075
256k	70,610.1	190,010	322,950	792,204
Pruned Kd	-tree			
512	95.3	325.9	549.9	1472
1k	203.9	703.5	1148.7	2603.7
2k	408.7	1526.8	2344.4	5067.9
4k	865.2	2874.6	4776.4	10,498.2
8k	1694	5603.6	9161.1	17,849.7
16k	3248.1	12,302.7	18,752.2	35,544.4
32k	6385.6	21,266.7	38,853.2	65,392.3
64k	14,620.1	42,197.7	74,233.1	137,763
128k	27,829	86,457.6	144,751	317,864
256k	54,544.8	181,531	274,718	720,966

Table 5 shows the number of memory accesses for classifying the packets corresponding to the FW rule set in the two algorithms. In our evaluations, the number of required accesses to system memory for packet classification was examined. Similar to the results in Table 4, for this rule set the pruned Kd-tree method makes fewer accesses to the system memory than the Kd-tree algorithm. It should be noted that memory access has a direct relationship with the time of classification. In other words, increased memory access would increase the time of classification and decreased access would decrease it.

Similarly, Table 6 lists the number of memory accesses in packet classification with IPC rule set. According to the results, with an increased number of packets and rules, the number of memory accesses has also increased. The reason why an increased number of rules augmented memory access is the enlargement of the tree of rules. As the rules increase in number, the number of tuples and, accordingly, the number of rules in each tuple will also increase. According to Table 6, for example, for 16k packets and 1k rules in the Kd-tree algorithm, the number of memory accesses is 10894541, but it reaches 50566605 with increasing the number of rules to 10k.

From the results in Tables 4, 5 and 6 it can be concluded that, in the classification of different numbers of input packets with rules of varying type and size, memory access in the pruned Kd-tree algorithm is less than in the typical Kd-tree algorithm. These results are indicative of the better performance of the proposed method. For example, given an IPC rule set of 20k rules and 256k packets, the total number of memory accesses for the Kd-tree algorithm is 2984245224. In this case, the number of memory accesses by the proposed method is 2275663443. Therefore, our method reduces memory access by 23.74%. The main reason for this reduction is the fact that right-hand leaves are pruned and the rules in every tuple are sorted.

4.2 Packet classification time

The time interval from the moment the packets arrive at the classifier until they are classified by the algorithm is called packet classification time. In this section, the packet classification time for different rule sets is examined. Packet classification time in this experiment was measured in milliseconds

Table 7 shows the classification time with the ACL rule set. According to this table, with an increase in the

Table 9 Classification time with the IPC rule set

Packets	Rules			
	1k	5k	10k	20k
Kd-tree				
512	148.8	459.9	1236	2122
1k	284.4	972.7	2540.5	4719.8
2k	589.5	1798.5	4890.7	9237.7
4k	1091.9	3507.2	9352.5	18,086.1
8k	2134.1	8296.3	19,648.5	35,600.5
16k	4409.8	14,584.2	33,420.1	72,090.5
32k	9116.4	28,521.4	64,648.1	143,146
64k	17,619	61,756.7	148,144	260,777
128k	34,130	123,424	274,478	532,779
256k	66,588.8	341,769	548,962	1,080,750
Pruned Kd	-tree			
512	112.5	379.2	1073.6	1787.3
1k	197.4	718	2015.5	3812.2
2k	449.9	1506.5	4101.3	7638.7
4k	744.7	3000.8	8376.3	14,331.5
8k	1643.4	6022	16,213.2	29,138.5
16k	3110.6	11,856.4	26,599.6	57,428
32k	6324.8	21,253.3	59,890.8	113,045
64k	12,109.5	52,257.6	118,218	223,633
128k	23,178.5	112,349	211,450	443,647
256k	51,236.3	261,962	464,276	992,319

number of packets, packet classification time in both the basic and pruned algorithms would increase. With any number of packets, the proposed method has a shorter classification time than the Kd-tree algorithm. For example, the classification time for 32k packets is 3157.2 ms in the Kd-tree algorithm and 2957.4 ms in the proposed method. Therefore, our method reduces the time by 199.8 ms.

Also, with all the different rule sets, the proposed algorithm has consumed less time than the Kd-tree algorithm. With 10k rules, for example, the speed of the proposed algorithm for classifying 64k packets is 14.75% higher than the Kd-tree algorithm. With 20k rules, the proposed algorithm has classified 2k packets in 2078 ms whereas the typical Kd-tree algorithm has classified this number of packets in 2596.2 ms. In other words, the Kd-tree algorithm is 24.94% slower than the proposed algorithm.

In the next experiment, packet classification time with FW and IPC rule sets was measured. Tables 8 and 9 represent the results for 1k, 5k, 10k, and 20k rules. In this rule set, too, the packet classification time of the proposed method in all cases is much less than that of the Kd-tree algorithm. For example, with an IPC rule set of 1k rules, 256k packets are classified in 66,588.8 ms by the Kd-tree

algorithm and in 51,236.3 ms by our proposed algorithm. Therefore, the proposed method functions 15,352.5 ms faster. This difference is also seen in other results.

Overall, the proposed method provided us with a shorter classification time with all rule sets (i.e., ACL, FW, and IPC) in comparison with the typical Kd-tree algorithm. The results obtained in this section are consistent with the results of examining the number of memory accesses, which is indicative of the superiority of the proposed method.

4.3 Throughput

Throughput refers to the number of packets that are classified in the unit of time. Increased throughput means that more packets have been classified in a second.

Figure 9a–d depict the throughput of algorithms on the ACL rule set with 1k, 5k, 10k, and 20k rules, respectively. As the created tree enlarges with an increase in the number of rules, the packet classification time will also increase. As a result, the number of packets classified in a second will be reduced. Our results show that throughput with smaller rule sets is higher than throughput with larger rule sets. For example, the throughput of the Kd-tree algorithm in the classification of 32k packets is 10,135 packets per second with 1k rules and 1124 packets per second with 10k rules. With ACL rule set, our method had a higher throughput than the Kd-tree algorithm in all classification scenarios. In Fig. 9a, the throughput of classifying 8k packets is 9958.92 packets per second for the Kd-tree algorithm and 11,111.11 packets per second for the proposed method.

Figure 10a–d show the results of the throughput of the algorithms for classification of packets based on FW rule set with 1k, 5k, 10k, and 20k rules, respectively. In this rule set, too, the proposed method has a higher throughput than the Kd-tree algorithm. With all the numbers of packets and rules, the proposed method classifies more packets than the Kd-tree algorithm in one second. For example, Fig. 10a shows that the Kd-tree algorithm has resulted in a throughput of 3864.15 packets per second in classifying 512k packets with 1k rules. However, our proposed method has achieved a rate of 5372.51 packets per second in the same scenario. In other words, our method classifies 39.03% more packets in the same time interval.

The maximum throughput obtained by our method with FW rule set was 5372.51 packets per second with 1k rules, 1571.03 packets per second with 5k rules, 931.86 packets per second with 10k rules, and 489.35 packets per second with 20k rules. As discussed above, decreased throughput is due to the increased complexity of the Kd-tree or its corresponding pruned tree which increases traversal time. The throughput obtained by the Kd-tree algorithm in classifying 4k packets was 3987.64 packets

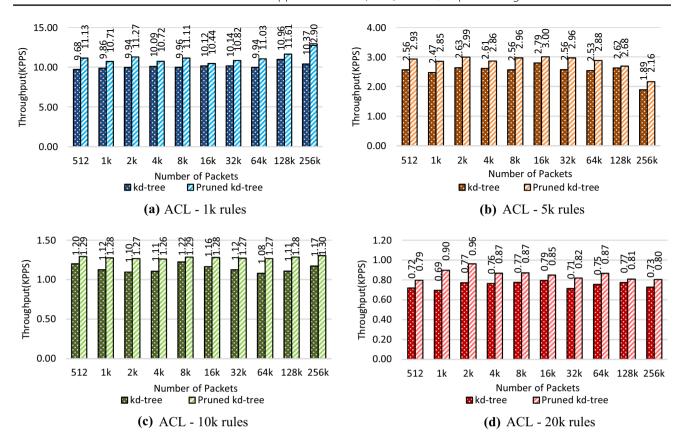


Fig. 9 Throughput of the kd-tree algorithm and the proposed method with different rule sets in ACL rules

per second with 1k rules, 1187.19 packets per second with 5k rules, 737.52 packets per second with 10k rules, and 345.07 packets per second with 20k rules.

The proposed method and the Kd-tree algorithm were implemented on IPC rule set. The results of this experiment on 1k, 5k, 10k, and 20k rules are shown in Fig. 11a–d, respectively. With this rule set, too, the proposed method obtained a higher throughput rate than the Kd-tree algorithm. According to Fig. 11a, with 64k packets, our method classified 5285.11 packets per second while the Kd-tree algorithm classified 3632.44 packets per second. In other words, our method classified 1652.67 more packets per second. Here again, the number of rules increased due to the enlarged and more complicated tree, thereby reducing the throughput rate.

Overall, the maximum throughput value obtained by the proposed method with ACL, FW, and IPC rule sets was 12897.77, 5372.51, and 5522.36 packets per second. However, the maximum throughput obtained by the basic algorithm with the same rule sets was 10,956.56, 4022.53, and 3844.49, respectively. Moreover, in all scenarios, the number of packets classified by the proposed

method was much more than those classified by the Kdtree algorithm at the same time.

4.4 Memory usage for the tree structure

One of the important criteria in comparing the efficiency of the proposed method and the basic algorithm is the amount of memory needed to store the tree structure created by the rules. The data structure used to hold the tree is composed of the following parts:

- Nodes that hold the tuples.
- Nodes that hold the number of member tuples and the number of markers assigned to the tuple.

Figure 12 shows the memory consumption in bytes of the Kd-tree algorithm and the proposed method with ACL, FW, and IPC rule sets. In the graphs of Fig. 12, the memory needed to hold tuple nodes and rules separately, together with the sum of the two, is shown for both algorithms. Graphs (a) through (d) show the

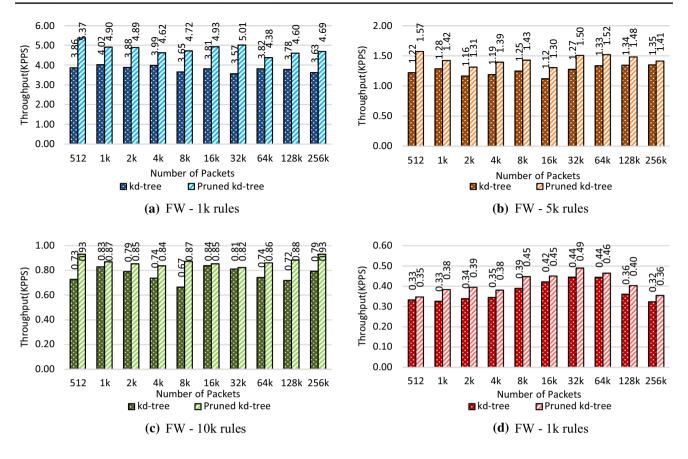


Fig. 10 Throughput of the kd-tree algorithm and the proposed method with different rule sets in FW rules

amount of memory needed to store the data structure corresponding to 1k, 5k, 10k, and 20k rules, respectively.

According to the results of this test, the proposed method requires less memory in all scenarios. For example, with an ACL rule set of 1k rules, the memory needed to store the Kd-tree is 23,815.9 bytes while it is 19,120.4 bytes in the proposed method. Therefore, the proposed method was able to reduce memory usage by 4695.5 bytes through pruning the tree nodes. With an FW rule set of 10k rules, the Kd-tree algorithm needs 0.74 KB for storing nodes that contain tuples and 156.09 KB for storing nodes that contain rules. By pruning the redundant nodes, however, the proposed method needs only 0.54 KB for tuple nodes and 127.21 KB for nodes that contain rules. With IPC rule set, too, our method used less memory than the Kd-tree algorithm. This confirms the success of the method in pruning the Kd-tree data structure and making better use of memory.

Overall, our method consumed less memory than the Kd-tree algorithm with all rule sets and all numbers of rules. In other words, our method for pruning the Kd-tree

was efficient and could have a positive effect on reducing the size of the tree.

Table 10 shows the comparison of the proposed method with other packet classification algorithms. The algorithms in the first column are compared according to three evaluation criteria including lookup time, memory usage, and dimension scalability.

First, the lookup time is examined. Most of the existing designs have a time complexity of either O(W) or $O(\log N)$. Four algorithms provide better search efficiency. The time complexity of RFC [20] and HiCuts [21] are constant. The binary search scheme [22] has a time complexity of $O(\log^2 W)$. RFC and HiCuts have lower time complexity but at the high cost of considerable memory requirement, i.e. $O(N^d)$. Excluding RFC and HiCuts, our design has the lowest time complexity $O\left(d\log\left(\frac{W}{\alpha}\right)\right)$ with controlled storage space requirement $O\left(N\log\left(\frac{W}{\alpha}\right)\right)$. Here, $1 \ll \alpha \ll W$ represents the minimum ratio of the reduction in any of the distinct traversal patches from the root to any leaf node of the tree.

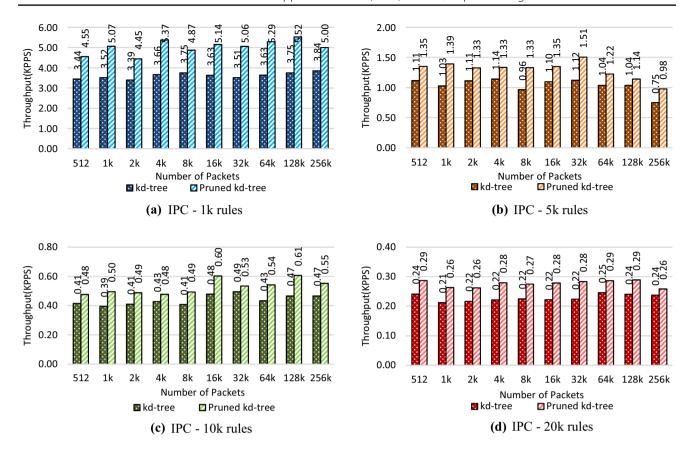


Fig. 11 Throughput of the kd-tree algorithm and the proposed method with different rule sets in IPC rules

Second, we compare the proposed design with Rectangle Search and Binary Search, which are both based on tuple space search. The two schemes can only be applied to two-dimensional classifiers, that is d=2. Rectangle Search requires O(W) hashes and Binary Search needs $O\left((\log_2 W)^2\right)$ hashes. The proposed design only requires $O(2\log W)$ hashes. From the storage perspective, Rectangle Search requires a storage space of O(NW) and Binary Search requires a space of $O(N*\log^2 W)$. In contrast, our design only needs $O\left(N\log\left(\frac{W}{a}\right)\right)$ of memory.

5 Conclusion

Packet classification plays a crucial role in the efficiency of many Internet-based processing devices such as routers and firewalls. Different software and hardware algorithms perform packet classification. One of the well-known algorithms for packet classification is the Kd-tree algorithm. By dividing the rules based on the length of the prefix of source and destination IP addresses, this algorithm places them in tuples. Then it sorts tuples with a special technique and creates a binary tree using the sorted tuples.

The input packets of the classifier can be classified using a search algorithm on this tree. The Kd-tree algorithm has several disadvantages including:

- Lack of order in the rules of the tuples and the necessity to examine all the rules of a tuple during classification.
- Existence of redundant nodes in the tree.
- Repetitive checking of rules during packet classification.

Given the defects in the Kd-tree algorithm, we attempted to develop an optimized version of the algorithm by close analysis of its structure as well as how it traverses the tree structure. The proposed method improves the Kd-tree algorithm in terms of both time and memory consumption. The key idea behind the optimized version is to prune redundant nodes that do not affect the process of classification. This can reduce memory access as well as the memory required for storing the tree structure. Also, in the proposed algorithm, sorting the rules in a tuple according to their priority has led to a further reduction in the classification time.

Both the proposed method and the Kd-tree algorithm were implemented on three rule sets, i.e., ACL, FW, and

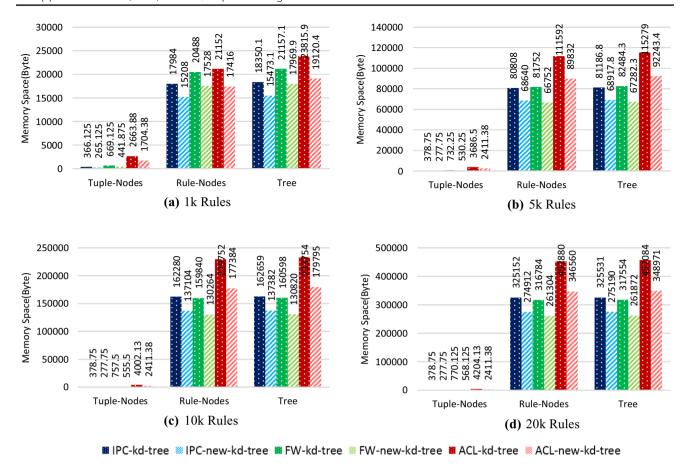


Fig. 12 Memory required for tree structure

Table 10 The complexity of different algorithms

Evaluation algorithms	Lookup time	Memory usage
Linear search on filter rules	O(N)	O(N)
Grid-of-tries [23]	O(W)	O(NW)
Cross-producting [24]	O(dW)	$O(N^d)$
Bit-parallelism [25]	$O(W \log N)$	O(NW)
Area-based QuadTree [26]	O(W)	O(NW)
Fat-Inverted segment tree [27]	O((L+1)W)	$O(LN^{(1+1/L)})$
Segment tree with fractional cascading [28]	$O(\log N)$	O(N * log N)
Recursive flow classification [20]	<i>O</i> (<i>d</i>)	$O(N^d)$
HiCuts [21]	<i>O</i> (<i>d</i>)	$O(N^d)$
Linear search on tuple [29]	$O(W^d)$	O(N)
Rectangle search [30]	O(W)	O(NW)
Binary search [22]	$O(log^2W)$	$O(Nlog^2W)$
Extended Grid-of-Trie [31]	O(W)	O(NW)
Kd-tree [6]	$O(d \log W)$	$O(Nd \log W)$
Proposed method	$O\left(d\log\left(\frac{W}{\alpha}\right)\right)$	$O\left(N\log\left(\frac{W}{\alpha}\right)\right)$

IPC. The rule sets were created using Classbench tool. This tool is used to generate experimental rules and packets for testing classification systems. The evaluation criteria we used include the number of tuples created by the rules, the number of memory accesses, packet classification time, throughput, memory consumption of the tree data structure, and the depth of the tree.

According to the results, the proposed method has been more efficient than the Kd-tree algorithm in every aspect. According to the results of the classification of 128k headers with IPC 1k rule set, our proposed algorithm could boost the classification speed of the basic Kd-tree algorithm by a ratio 1.47. Furthermore, our method needs less memory space than the Kd-tree algorithm for storing the tree data structure. The depth of the pruned tree is less than that of the Kd-tree algorithm.

One suggestion to continue the present research with the aim of reducing the classification time is to develop a method for searching the rules in a tuple. Currently, the rules of a tuple are checked linearly. In the worst case, this may require the input packet to be matched against all the rules. The time complexity of this search operation is in the worst case O(n), where n denotes the number of rules in the tuple. Therefore, if the time for checking the rules can be reduced, the classification time will also be reduced. For this purpose, like other methods derived from the basic algorithm, we can use hashing methods. Thus, the number of memory accesses to find the best matching rule in each tuple would decrease significantly.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

- Hager S, John P, Dietzel S, Scheuermann B (2018) RuleBender: Tree-based policy transformations for practical packet classification systems. Comput Netw 135:253–265
- Inoue T, Mano T, Mizutani K, Minato S-I, Akashi O (2018) Fast packet classification algorithm for network-wide forwarding behaviors. Comput Commun 116:101–117
- 3. Cheng Y-C, Wang P-C (2015) Packet classification using dynamically generated decision trees. IEEE Trans Comput 64:582–586
- Sun P, Lan J, Wang P, Ma T (2017) RFC: range feature code for TCAM-based packet classification. Comput Netw 118:54–61
- Qi Y, Xu L, Yang B, Xue Y, Li J (2009) Packet classification algorithms: from theory to practice. IEEE INFOCOM 2009:648–656
- Shieh S, Lee F-Y, Lin Y-W (2004) Accelerating network security services with fast packet classification. Comput Commun 27:1637–1646
- 7. Gupta P, McKeown N (1999) Packet classification using hierarchical intelligent cuttings. In: Hot interconnects VII

- 8. Singh S, Baboescu F, Varghese G, Wang J (2003) Packet classification using multidimensional cutting. In: Proceedings of the 2003 conference on applications, technologies, architectures, and protocols for computer communications, pp 213–224
- Qi Y, Fong J, Jiang W, Xu B, Li J, Prasanna V (2010) Multi-dimensional packet classification on FPGA: 100 Gbps and beyond. In: International conference on field-programmable technology, pp 241–248
- Bentley JL (1975) Multidimensional binary search trees used for associative searching. Commun ACM 18:509–517
- Robinson JT (1981) The KDB-tree: a search structure for large multidimensional dynamic indexes. In: Proceedings of the 1981 ACM SIGMOD international conference on management of data, pp 10–18
- 12. Guttman A (1984) R-trees: a dynamic index structure for spatial searching, vol 14. ACM, New York
- Sellis T, Roussopoulos N, Faloutsos C (1987) The R+-tree: a dynamic index for multi-dimensional objects
- Beckmann N, Kriegel H-P, Schneider R, Seeger B (1990) The R*-tree: an efficient and robust access method for points and rectangles. In: ACM sigmod record, pp 322–331
- Ahn HK, Mamoulis N, Wong HM (2001) A survey on multidimensional access methods. Technical report, Institute of Infomation and ComputingSciences, Utrecht University, The Netherlands
- Böhm C, Berchtold S, Keim DA (2001) Searching in highdimensional spaces: index structures for improving the performance of multimedia databases. ACM Comput Surv (CSUR) 33:322–373
- 17. Brown L, Gruenwald L (1998) Tree-based indexes for image data.

 J Vis Commun Image Represent 9:300–313
- Gaede V, Günther O (1998) Multidimensional access methods. ACM Comput Surv (CSUR) 30:170–231
- 19. Taylor DE, Turner JS (2007) Classbench: a packet classification benchmark. IEEE/ACM Trans Netw 15:499–511
- 20. Li X, Shao Y (2018) Memory compression for recursive flow classification algorithm in network packet processing devices. In: IEEE 3rd advanced information technology, electronic and automation control conference (IAEAC), pp 1502–1505
- Chang Y-K, Chen H-C (2018) Fast packet classification using recursive endpoint-cutting and bucket compression on FPGA. Comput J 62:198–214
- 22. Baboescu F, Warkhede P, Suri S, Varghese G (2006) Fast packet classification for two-dimensional conflict-free filters. Comput Netw 50:1831–1842
- 23. Nottingham A, Irwin B (2009) GPU packet classification using OpenCL: a consideration of viable classification methods. In: Proceedings of the 2009 annual research conference of the South African Institute of Computer Scientists and Information Technologists, pp 160–169
- 24. Pao D, Lu Z (2014) A multi-pipeline architecture for high-speed packet classification. Comput Commun 54:84–96
- Lakshman T, Stiliadis D (1998) High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In: ACM SIGCOMM computer communication review, pp 203–214
- 26. Lee J, Byun H, Mun JH, Lim H (2017) Utilizing 2-D leaf-pushing for packet classification. Comput Commun 103:116–129
- Feldman A, Muthukrishnan S (2000) Tradeoffs for packet classification. In: Proceedings IEEE INFOCOM 2000. Conference on computer communications. Nineteenth annual joint conference of the IEEE computer and communications societies (Cat. No. 00CH37064), pp 1193–1202
- Su C-F (2000) High-speed packet classification using segment tree. In: Globecom'00-IEEE. Global telecommunications conference. Conference record (Cat. No. 00CH37137), pp 582–586

- 29. Daly J, Bruschi V, Linguaglossa L, Pontarelli S, Rossi D, Tollet J et al (2019) TupleMerge: fast software packet processing for online packet classification. In: IEEE/ACM transactions on networking
- Srinivasan V, Suri S, Varghese G (1999) Packet classification using tuple space search. In: ACM SIGCOMM computer communication review, pp 135–146
- 31. Erdem O (2016) Pipelined hierarchical architecture for high performance packet classification. Comput Netw 103:143–164

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.