HyperCuts

Packet Classification Using Multidimensional Cutting

Viktor Masicek

Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic
cabadaj@gmail.com

Abstract

This paper describes classification algorithms called HyperCuts and PTree. Both methods will be compared. HyperCuts and PTree are based on a decision tree structure. HyperCuts is based on range matched and PTree is based on fixed matched. So, it is the biggest difference. Next difference is number of fields in nodes. HyperCuts can use more field but PTree not. PTree can saved rules into arbitrary node, but HyperCuts only into leaves, it is important different too. HyperCuts is in comparison with PTree better about number of rules. PTree end by breakdown with much more rules, but HyperCuts is not as show some tests in [1].

1 Introduction

Demand for speed of packet classification is growing up in core and edge today. Need and scientific interest are motivation for next working on Packet Classification. We have some new ideas, as multidimensional cutting, pulling rules up in Decision Tree or somewhere linear searching.

1.1 Introduction to classification problem

The Packet Classification problem is to determine the first matching rule for each incoming message at the router. The rule consists of values which can be one of three kinds of matches: exact match, prefix match or range match. Lots of algorithms for Packet Classification exist, now. For example: linear searching, algorithm with pre-computed earliest rules, RFC or Hierarchical Cuts (HiCuts [2]) or HyperCuts [1].

1.2 PTree vs. HyperCuts

PTree is one of implemented methods. A comparison between the HyperCuts (HC) method, described in [1], and the PTree (PT) method, described in [3], is very intricate. At first, the HC method is based on range match. Unlike HC, the PT method is based on fixed match. At second, the PT method takes account of rules' form to a great extent and it is not very efficacious in global application use. However, the HC method looks for only theoretical solution and optimization. Format of rules is very important for the HC, but it does not affect speed and memory as in the PT method.

2 Description of PTree

The basis of PTree [3] is building a tree with information taken from rules. We build the tree only at the beginning. If a new packet comes, we only traverse through the tree. And we need rebuild the tree to add some new rules. Traversing the tree in PTree and in HyperCuts is very similar. However building the tree is not.

Firstly, PTree uses only one field in all nodes in the tree. It is as in HiCuts [2]. HiCuts is older then HyperCuts, but it is easier. On the other hand, in PTree matched rules can be in all nodes and leaves in the tree. But in HiCuts and HyperCuts matched rules can be only in leaves.

At first, a set of all fields which are included in all rules is prepared. Fields have to be ordered by their first bit in packet. Some fields are not determined, because their place in packet depends on other fields. If a cycle does not come into existence in dependence fields, we have no problem. In the opposite case it is impossible to continue in building the tree. After preparing a list of fields without a cycle, we can start with building the tree.

2.1 Building the tree

Each node in the tree has three sets of information:

- the node position - this position in packet is same as field's position which is used in this node
- a list of branches - each branch matches some value
- a list of rules that match at the node.

First, we determine the "nearest" (closest to the start of the packet) field in the ruleset. This determines the position of the node currently being generated in the tree.

Now, we have checked all rules in the ruleset whether they comprise "nearest" field or not. If yes, then check if there is already a branch with the rule's comparison value. Rules with a comparison value are included into the right branch. New branches are created for rule without a comparison value. If rule does not compare at the "nearest" field, then the rule is included into a special wildcard branch. Special case is empty rule. This rule is included into the match list in the node. When all rules from ruleset are checked, all rules form wildcard branch are copied into all other branches. Because the rules in the wildcard branch do not compare at the current node's position, they must be available in every child node for comparison. While rules are added to some branch "nearest" field is done off of the rule. It is done, because this field is resolved in all rules.

Finally, we used recursion to create child nodes. For each branch, same procedures with their set of rules are called.

This process continues while some rule comprises any field.

2.2 Traversing in the tree

For finding matches rules tree is traversed. In each node matched list are copied into appropriate array. Next, we continue in matched branch. If node has not any child, then traversing during the tree is stopped. Finally, rules saved in appropriate array are printed.

3 Description of HyperCuts

In HyperCuts [1], we build a multi-tree and all rules are only in leaves. All rules consist of fields. These fields are used for decision which branch we have to continue in tree. Multiple field's tests can be used in each node.

3.1 Building the tree

Each field represents an interval. If field is used in a node the interval must be split to elements (smaller intervals). We have to identify set of fields for each node. At first, we select fields which have the largest number of elements together. To avoid choosing all fields, we only use fields with number of elements

greater than return value of function, depending on set of selected rules. Other possibility is to use the ration of the number of unique elements to the total number of possible values covered by the range representing the dimension as a measure.

When the set of fields is selected we must identify the number of cuts for each field. It means how many elements each field is split to. We pick set of the number of cuts for set of field. The set depends on the special constant. Considering every possible combination is computationally infeasible in reasonable time. Then we use greedy approach. It is based on finding local optimum of cuts for every field.

It is possible to use some optimization. In [1] are described four possible optimization: Node Merging, Rule Overlap, Region Compaction, Pushing Common Rule Subsets Upwards. Of course, some other optimization are applicable.

3.2 Traversing in the tree

Traversing in decision tree is very easy. We select correct array's elements by binary shift. It is much less time-consuming. In leaf we use linear searching. Special constant limits number of rules in leaf. So, the time need for linear searching in leaf are inconsiderable.

4 Comparison

4.1 Rules and Fields in PTree and HyperCuts

At first, we must compare rules and fields in HyperCuts and PTree. In HyperCuts all rules have same set of fields. So, some "unneeded" fields have wildcard value in some rules. On the other hand, in PTree all fields do not have to be used in all rules.

Second difference are values in fields. In PTree a field can comprise only one value (it means one number). On the other hand, in HyperCuts a field can represent some interval. So, if we take some rules from PTree with same set of fields we can make only one rule for HyperCuts. If some fields are used in other rules and are not used in this made rule, then unused fields can be added with wildcard values.

4.2 Data Structure Size

Nowadays, there is no implementation of HyperCuts available. But in [1] in **Figure 18** we can see values which compare three types of databases (ER-edge router, CR-core router and FW-firewall). Values are counted for different number of rules. We get similar results for ER and CR. Memory space is linear dependent on number of rules. But in FW case, we get quadratic dependence. It is possible, that these results are not perfect, but there are top estimates.

In PTree method the situation is very different. In the worst case, we can have a great deal of rules and all these rules include some field. It is possible, that all rules include only this one field. In this case, tree will include only one node with too much branches. But it is not the main problem. Main problem is in number of rules. If we make too much rules, then PTree method end by breakdown. And memory for less rules is not very interest.

4.3 Search Speed

4.3.1 Search Speed - PTree

In PTree the search speed depends the depth of tree. The depth of tree is evident. If we take a rule with maximum fields considering other rules, then the depth of tree cannot be bigger, because in each node some field is done off. And the depth of tree cannot be smaller, because we must use all fields included in rules. So, the depth of tree (marked DT) is

$$DT = \max_{r \in Ruleset} (NumF(r)),$$

where $NumF(r)$ is number of fields included in rule r. Appropriate rule, which had maximum number of field is marked R_{DT}. Of course, we can find more rules with maximum fields, but it is possible to mark whichever rule as R_{DT}.

However, the PTree method has one big handicap. Each node can include many branches. In the worst case, we must compare all values which correspond with branches. In each node can be as much branches as are possible values for selected field in this node. So, this property is very important for the search speed. In the worst case, in each node, which we have traversed during the searching, we have all possible branches. That the search speed (marked SS) is

$$SS = \sum_{f \in R_{DT}} IntF(f),$$

where $IntF(f)$ is number of possible value of field f.

4.3.2 Search Speed - HyperCuts

In HyperCuts method the search speed is intricate. We must look at constant bounding maximum number of branches in a node (mark as B) and maximum number of rules marked as L) in each leaf. Number of all rules is marked as R. Suppose that $B = 3$. Now, we are in a node and we use only one field f. In the worst case, HyperCuts recursively creates one branch from rules which include the field f as wildcard. In the worst case this set of "wildcard-rules" included all rules except for two rules. Each of these two rules is used, with set of wildcard-rules, for recursive creation of other two branches.

So, in "wildcard-branch" we reduced two rules and in other branches we reduced one rule. If we continue the same way, we will reduce one rule in each node. In a leaf more rules can be included. So we can determine Search speed (marked SS)

$$SS = R - L.$$

In general case (B is arbitrary) reducing of rules is faster. So, the search speed can be determined too

$$SS = \frac{R - L}{B - 2}.$$

But, if we consider very much rules, that $R - L \sim R$, because L is imponderable on R. As well we can vanishes $B - 2$. So, after both vanish we get

$$\frac{R - L}{B - 2} \sim R \Longrightarrow SS = R.$$

In connection with the search speed, we can consider next property. It is maximal length of all prefixes used in all fields. We marked this variable W. If we have determined number of fields in rules, then number of rules is limited. If we do not use any prefixes in fields, then all rules included only exact matches. Because possible values in fields are limited, the number of rules are limited too. Thus, we can say that W hangs together with R.

4.4 Build/Update Complexity

In both methods we have two main results. Time needed to build one node and number of nodes in the tree. For better comparison we confront both results separately.

4.4.1 Building HyperCuts

To count up the time needed to build one node, we can use the pseudo code from [1] on page 220.

	1	CreateNodeP($l_1, r_1, l_2, r_2, ..., l_k, r_k$, Ruleset);
c_1	2	if (R < bucketSize) return;
R	3	for i $\longleftarrow$ 1 to k do
R	4	$N_i \longleftarrow$ numberOfUniqueValuesOnDim(Ruleset,i);
d_1	5	Mean $\longleftarrow$ mean($N_1 ... N_k$);
R	6	for i $\longleftarrow$ 1 to k do
c_2	7	if $N_i >$ Mean then Dims $\longleftarrow$ Dims$\bigcup${i};
$\|D\|$	8	for i $\in$ Dims do
d_2	9	NC(i) $\longleftarrow$ optimumNoCutsOnDimension(i, l_i, r_i,Ruleset);
c_3	10	N $\longleftarrow \prod_{i \in Dims} NC_i$;
$spfac$	11	for i $\longleftarrow$ 1 to N do
d_3	12	$(l_1^i, r_1^i, ..., l_k^i, r_k^i, R^i) \longleftarrow$ createCut();
recursion	13	CreateNode($l_1^i, r_1^i, ..., l_k^i, r_k^i, R^i$);
	14	return;

All lines with constants c_i or d_i are only some base instructions or we can count this line by some simple formula. Constants c_i represent some small constants and d_i some medial constants. The for-cycle on lines 3 and 4 runs through all rules, in the worst case. The function in cycle on line 4 has to use all rules. And a result should be counted after each rule is read once. So, the for-cycle on lines 3 and 4 takes R^2 time to build the node. The for-cycle on lines 6 and 7 is more simple. This cycle takes $R * c_2$ time for build the node. The for-cycle on lines 8 and 9 runs through all dimensions and for each dimensions it count the optimal number of cuts. The number should be counted by some formula. So, the for-cycle takes $|D| * d_2$ time to build the tree. The for-cycle on line 11, 12 and 13 runs through all children. But, the number of children is limited by the constant (marked $spfac = space\ factor$). So, assume that $N = spfac$. Creating cuts for children should be counted by some formula. So, the for-cycle takes $spfac * d_3$ time to build the tree. In the cycle, recursive function is called. Because the function calls itself, it is not included into the time needed to build the node.

Then, the time needed to build one node (marked BN_{HC}) is

$$BN_{HC} = c_1 + R^2 + d_1 + R * c_2 + |D| * d_2 + c_3 + spfac * d_3.$$

However, the largest influence on BN_{HC} has R^2. So, we can vanish other summands

$$BN_{HC} = R^2.$$

Assume that the tree has not more leaves than rules. In another case the decision

tree is feckless. Then the depth of the tree is $DT_{HC} = \log_{spfac} R$. So, we can count the number of nodes in the tree (marked NN_{HC}), because the number of children is limited by $spfac$.

$$NN_{HC} = \sum_{i=1}^{DT_{HC}} spafc^i$$

The sum can be counted as

$$NN_{HC} = spfac * \frac{spfac^{\log_{spfac} R} - 1}{spfac - 1} = spfac * \frac{R - 1}{spfac - 1}.$$

In this expression R is greater than the $spfac$. So, we can write

$$NN_{HC} = R.$$

4.4.2 Building PTree

If we evaluate main parts in code of PTree to build the tree (function "PNODE buildNode(availRules *ruleset)", on lines 224 to 592 in "buildtree.c"), then we count the BN_{PT},

$$BN_{PT} = 5 * R + IntF(f) + 2 * R * IntF(f).$$

In this expression f is field used in this node. We can vanish $5 * R + IntF(f)$ in comparison with $R * IntF(f)$. So,

$$BN_{PT} = R * IntF(f).$$

This value in comparison with BN_{HC} is great, because in general cases

$$IntF(f) > R \quad \Longrightarrow \quad BN_{PT} = R * IntF(f) > R^2 = BN_{HC}.$$

Next, we evaluate the number of nodes in the tree. As we know, the depth of the tree in PTree is

$$DT_{PT} = \max_{r \in Ruleset}(NumF(r)).$$

We can limit the number of children in the tree by

$$CH = \min((\max_{f \in Fields} IntF(f)), R).$$

So, the number of nodes is

$$NN_{PT} = \sum_{i=1}^{DT_{PT}} CH^i.$$

This sum can be counted as in last paragraph

$$NN_{PT} = CH * \frac{CH^{DT_{PT}} - 1}{CH - 1}.$$

CH is great enough to vanish $CH/(CH-1)$. Thus,

$$NN_{PT} = CH^{DT_{PT}}.$$

Because $CH \geq R$ then

$$NN_{HC} < NN_{PT}.$$

Finally, building one node in HyperCuts needs less time and PTree should have more nodes. So, the tree should be built faster by HyperCuts method.

4.4.3 Update Complexity

The complexity of updating the tree is very simple. When the tree is updated, it needs be rebuilt in both methods.

Comparative Table

	HyperCuts	**PTree**
Matching	in leaves	in leaves and nodes
Decision Fields in Node	1 or more	1
Search Speed	R	$\sum_{f \in R_{DT}} IntF(f)$
Data Structure Size	R or R^2	uninteresting or breakdown
Build Complexity		
Build one node	R^2	$R * IntF(f)$
Number of nodes	R	$CH^{DT_{PT}}$
Update Complexity	Rebuild the Tree	Rebuild the Tree

4.5 Choosing fields in PTree

Choosing the best field included in all rules is based on packets' format. The algorithm takes field from first to final order by position in packet. It is good for linear reading of packet and for reducing the list of fields. But, we can't say it is the most optimal for height of the tree. Optimization depends on types of fields. And mainly, rules' format is very important for optimal work of PTree.

5 Conclusion

HyperCuts is better than PTree, because HC is independent on rules' format, unlike PT. And in the comparison, HC method has better results than PT method. PT isn't a bad program and algorithm, but it depends on the format of a rules. However, PTree is applicable for some sets of rules.

References

[1] Sumeet Singh, Florin Baboescu, George Varghese, Jia Wang, "Packet Classification Using Multidimensional Cutting", Proceeding of SIGCOMM, Karlsruhe, Germany, 2003

[2] P. Gupta, and N.McKeown, "Packet Classification using hierarchical intelligent cutting", in Proc. Hot Interconnects, 1999

[3] Derek Becker, Radivoje Todorovic, Qihen Wang, "PTREE: A System for Flexible, Efficient Packet Classification", CS524, Spring 2001