Decision Tree based Network Packet Classification Algorithms
T S URMILA
Department of Computer Science, Sourashtra College, Madurai-4  Email: urmi_ts@yahoo.co.in
Dr R BALASUBRAMANIAN
Dean - MCA, KVCET, Mathuranthagam.  Email: drrb_1951@gmail.com

ABSTRACT

The Internet is comprised of a mesh of routers interconnected by links. Communication among nodes on the Internet (routers and end-hosts) takes place using the Internet Protocol, commonly known as IP. IP datagrams (packets) travel over links from one router to the next on their way towards their final destination. Each router performs a forwarding decision on incoming packets to determine the packet’s next-hop router. IP router forwards the packets and also chooses to perform special processing on incoming packets. Such special processing requires that the router classify incoming packets into one of several flows which is called Packet Classification. Packet Classification may be based on single Dimension or Multidimensional. The Packet classification is based on set of rules among which one of them is matched by the incoming packet and the corresponding action is taken. Decision Tree is one of the best datastructure available to store and find the best matching rules. HiCut, HyperCut, Hypersplit, Layered cutting, Dimcut, Efficut are some of the packet classification algorithms based on Decision tree Data structure.

Keywords – Packet Classification, HiCut, HyperCut, Hypersplit, Dimcut, Efficut.

1. INTRODUCTION

A packet-switch in a router must perform a forwarding decision on each arriving packet for deciding where to send it next. An IP router does this by looking up the packet’s destination address in a forwarding table. This yields the address of the next-hop router and determines the outlet port through which the packet should be sent. This lookup operation is called a route lookup or an address lookup operation. Second, the packet-switch must transfer the packet from the way in to the way out port identified by the address lookup operation. This is called switching, and also involves physical movement of the bits carried by the packet.

1.1 Architecture of a packet-by-packet router

The Packet by packet router consists of one line card for each port and a switching fabric (such as a crossbar) that interconnects all the line cards. Typically, one of the line cards houses a processor functioning as the central controller for the router. The path taken by a packet through a packet-by-packet router consists of two main functions on the packet: (1) performing route lookup based on the packet’s destination address to identify the outgoing port, and (2) switching the packet to the output port.

The routing processor in a router performs one or more routing protocols such as RIP by exchanging protocol messages with neighboring routers. This enables it to maintain a routing table that contains a representation of the network topology state information and stores the current information about the best known paths to destination networks. The router typically maintains a version of this routing table in all line cards so that lookups on incoming packets can be performed locally on each line card, without loading the central processor. This version of the central processor’s routing table is referred o as the line card’s forwarding table because it is directly used for packet forwarding.

1.2 Flow-aware IP router and Packet Classification

One main reason for the existence of flow-aware routers stems from an ISP’s desire to have the capability of providing differentiated services to its users. Traditionally, the Internet provides only a “best-effort” service, treating all packets going to the same destination identically, and servicing them in a first-come-first-served manner. In order to provide differentiated services, routers require additional mechanisms. These mechanisms — admission control, conditioning (metering, marking, shaping, and policing), resource reservation (optional), queue management and fair scheduling require, first of all, the capability to distinguish and isolate traffic belonging to different users based on service agreements negotiated between the ISP and its customer. This has led to service agreements, express them in terms of rules or policies configured on incoming packets, and isolate incoming traffic according to these rules. The collection of rules or policies is called a policy database, flow classifier, or simply a classifier. Each rule specifies a flow that a packet may belong to based on some criteria on the contents of the packet header. All packets belonging to the same flow are treated in a similar manner. The identified flow of an incoming packet specifies an action to be applied to the packet. For example, a firewall
router may carry out the action of either denying or allowing access to a protected network. The determination of this action is called packet classification — the capability of routers to identify the action associated with the “best” rule an incoming packet matches. Packet classification allows ISPs to differentiate from their competition and gain additional revenue by providing different value-added services to different customers.

Flow-aware routers perform a superset of the functions of a packet-by-packet router. It consists of four main functions on the packet: (1) performing route lookup to identify the outgoing port, (2) performing classification to identify the flow to which an incoming packet belongs, (3) applying action (as part of the provisioning of differentiated services or some other form of special processing) based on the result of classification, and (4) switching to the output port.

### 1.3 Packet Classification

A packet classifier must compare header fields of every incoming packet against a set of rules in order to assign a flow identifier. A rule must specify a set of headers and the policy to be in use. A Rule Space is a collection of rules, specified as a table (flat data base) with, Columns as RuleID, ‘D’ header field specification as $f_1, f_2, \ldots, f_n$ and an Action column. Each record (Row) specifies a rule. Number of rules is said to be the Dimension of the rule space. Process of classification requires, applying the rules from the top. Attributes of the packet to be classified are matched with values in the Header columns, and if successful the Action is called access to a protected network. The determination of this classification relies on a set of rules in a rule table.

Table 1 is a 2-D rule table. There are 5 rules and R1’s priority is highest. Fig 3 show the decision trees built by Hicuts, Hypercuts, Hypersplit. In Figure 3, Hicuts employs the equal-sized subspace partition, and chooses only one dimension to cut for every internal node. Therefore, Hypersplit only cuts a single dimension in an internal node, but it employs end-point to find out the “cut bits” in decision tree’s internal nodes instead of the keys (or cutpoint). The other method is to divide the rule table by using cutting endpoints. Each rule in the filters generates a range (or interval) between two endpoints. Only endpoints of ranges are used as cut-points. Choosing end-points has more flexibility than choosing prefix.

#### 2.1 Hicut, Hypercuts and Hypersplit

Hicuts and Hypercuts both employ equal-sized cuts. They use a heuristic to decide how many cuts should be employed. The most important difference between Hicuts and Hypercuts is that Hicuts only cuts one dimension in an internal node but Hypersplits cuts multiple dimensions. Therefore, Hypersplit’ tree depth is shorter than Hicuts. Hypersplit only cuts a single dimension in an internal node, but it employs end-point to find out the cut-point. Fig 3 show the decision trees built by Hypersplits. Hicuts employs the equal-sized subspace partition, and chooses only one dimension to cut for every internal node. Therefore, Hypersplit only cuts a single dimension in an internal node, but it employs end-point to find out the cut-point. Fig 3 show the decision trees built by Hypersplits.
exists in 4 leaf nodes. It wastes lots of memory to store those duplicated rules. In Figure 3, Hypersplit chooses cut-points. At the first level, the rules in each of three intervals at field-x are 2, 1, and 2. So, value 10 is used as the cut-point which divides the rule table into two groups, \{R1, R2, R3\} and \{R4, R5\}. At level 2, left internal node’s rules in each of two intervals at field-x are 3 and 1. So, selecting 01 as cut-point can divide rules into \{R1, R2\} and \{R3\}. The right internal node’s rules in each of three intervals at field-y are 1, 1, ad 2. So, choosing 11 as cut-point can divide rules into \{R4\} and \{R5\}. By this rules it could completes the decision tree. This cut-point selection algorithm of Hypersplit reduces the rule duplications effectively.

2.2 Layered Cutting Scheme

In a Layered cutting Scheme for a packet classification algorithm picks up multiple dimensions and cutting with end-point to make the height of decision tree much shorter. Then a layered mechanism is proposed to reduce the memory consumption dramatically. The algorithm focuses on two aspects. The first aspect is to pick up the dimensions and the second aspect is to decide the cut-point.

A. Select the cut dimensions:

The Cut dimensions are chosen based on The set of dimensions with Larger Distinct field values, the dimensions with value smaller than the average value of all dimension and dimensions whose number of end-points is greater than average number of endpoints of all dimensions.

B. Space decomposition:

For Space decomposition Weighted Segment balanced scheme and ½ end point schemes are chosen. In ½ end point scheme the cut-point m is selected such that the number of intervals at m’s leftside is equal to that of m’s right side .i.e . 1/2(lowbound endpoint + upbound end-point)

In the Layered cutting scheme for selecting the cut dimension, distinct field values heuristics, and for select cut-point weighted segment-balanced heuristic is chosen to obtain the best results of memory consumption and number of memory accesses.

Optimization

Rule duplication is a very serious problem in packet classification. It will cause a rule replicated many times and use a lot of memory to keep them.
2.3 DimCut Algorithm

DimCut algorithm adds some modifications and improvements on the HiCuts algorithm. Consider the following definitions:

Definition: Let $wc(H)$ be the count of wild card entries in the column $H$ in the whole of the rule set.

Definition: Let $gd(H)$ be the geometric distance associated with column $H$ in the whole of the rule set.

Some of the guidelines and principles followed in this algorithm are:

i. Dimension Selection: Select the two fields $H_a, H_b$ which have the least $wc(\cdot)$ values, as the two selected dimensions or alternatively select $H_a, H_b$ which have least $gd(\cdot)$ values.  

ii. Number of cuts and Bucket size: Compute the number of cuts as $NC = \lceil 20 + \frac{N}{1000} \rceil$ and the bucket size threshold as $B = \lceil \frac{N}{20 + \frac{N}{1000}} \rceil$. Here $N$ is the total number of rules in the complete rule set.

iii. Separate those rules in the same chosen field as cut dimension which have wildcard value and shift them to the bucket and reject their use for making the decision tree.

iv. Building index tables to facilitate search within: Build an index table for each bucket. The framework will contain two stages: an index table and rule buckets. Use the same field of the input packet to search in the index table. If the specific field matches, the matching filter will be selected out of the set in the bucket via linear search (using smaller set of rules). All incoming packets need to check at the fields selected during preprocessing. The decision tree traverses to find the buckets that cover the incoming packet. There is priority sorting of all rules. When first match index is found a packet will traverse all regions of possible belonging. The packet will check all the header fields of governing rules linearly. The most prioritized packet is picked up via those that match completely. So the final action (Accept/Deny) will be taken for that incoming packet and the search will end. It supports incremental update but in case of significant decreasing performance it needs reconstruction. Updating will work in the same manner as the search algorithm. For firewalls a very slow update rate would suffice and entries can be added manually or infrequently.

The Briefed Preprocessing Algorithm:

Read rules and create a link list to store them. ii. Find the cut dimension by using any of the 2 heuristics (any dimension that has the smallest geometric length/any dimension that has the smallest number of wildcards). iii. Calculate the number of cuts by using of $(NC= \lceil 20 + \frac{N}{1000} \rceil)$ and calculate the threshold $T = \frac{N}{NC}$. iv. Separate those rules that has wildcard value in the same chosen field as cut dimension and shift them to the bucket and reject their use for making the decision tree.

2.4 EFFICUTS
EffiCuts: Algorithm implements the new ideas of separable trees combined with selective tree merging to tackle the variation in the size of overlapping rules and equi-dense cuts to tackle the variation in the rule-space density. EffiCuts also leverages equi-dense cuts to achieve fewer accesses per node than HiCuts and HyperCuts by co-locating parts of information in a node and its children.

Separable Trees
Placing small and large rules in different trees would reduce the replication. Large rules are identified easily as those that have wildcards in many fields. There is a possibility of two trees — one for rules with many wildcard fields and the other for the rest. Another factor called separability, is more fundamental than rule size, which determines the extent of replication. While the above scheme ignores the rule space’s dimensions, separability considers variability of rule size in each dimension. Separability enables the solution to avoid assigning and optimizing arbitrary percentages of the rules to distinct trees. To eliminate overlap among small and large rules, all small and large rules are separated by defining a subset of rules as separable if all the rules in the subset are either small or large in each dimension. A distinct tree is built for each such subset where each dimension can be cut coarsely to separate the large rules, or finely to separate the small rules without incurring replication.

Identifying Separable Rules
Separability implies that all the rules in a tree are either wildcard or non-wildcard in each field; otherwise, cuts separating the non-wildcard rules would replicate the wildcard rules. The categories assuming the standard, five-dimensional IPv4 classifier are:
- Category 1: rules with four wildcards
- Category 2: rules with three wildcards
- Category 3: rules with two wildcards
- Category 4: rules with one or no wildcards

To capture separability, each category is broken into sub-categories where the wildcard rules and non-wildcard rules are put in different sub-categories on a per-field basis. Accordingly, Category 1 has a sub-category for each non-wildcard field, for a total of $C_1 = 5$ sub-categories. Category 2 has a sub-category for each pair of non-wildcard fields for a total of $C_2 = 10$ sub-categories. Category 3 has a sub-category for each triplet of non-wildcard fields for a total of $C_3 = 10$ sub-categories. Because Category 4 contains mostly small rules, the further sub-categories are unnecessary.

Selective Tree Merging
Selective tree merging, which merges two separable trees mixing rules that may be small or large in at most one dimension. For instance, a Category 1 tree that contains rules with non-wildcards in field A (and wildcards in the other fields) is merged with Category 2 tree that contains rules with non-wildcards in fields A and B, and wildcards in the rest of the fields. This choice ensures that wildcards (of Category 1) are merged with non-wildcards (of Category 2) in only field B; in each of the rest of the fields, either non-wildcards are merged with non-wildcards (field A) or wildcards with wildcards (the rest). This significantly reduces the number of lookups while incurring only modest rule replication. One exception is the single Category 4 tree which is not broken into sub-categories, and hence, already mixes wildcard and non-wildcards in multiple fields. As such, merging this tree with other Category 3 trees would cause such mixing in additional fields and would lead to significant rule replication. Therefore, do not merge the Category 4 tree with any other tree.

In EffiCuts, copy of rules, instead of a pointer to, each rule at the leaf, forcing the rules to be in contiguous memory locations. However, if a rule is not replicated then this strategy requires less memory as it stores only the rule, and not a pointer and the rule. Because EffiCuts’ rule replication is minimal, these two effects nearly cancel each other resulting in little extra memory.

Equi-dense Cuts
Recall that HyperCuts’ equi-sized cuts, which are powers of two in number, simplify identification of the matching child but result in redundancy due to rule-space density variation. Fine cuts to separate densely-clustered rules needlessly partition the sparse parts of the rule space resulting in many ineffectual tree nodes that separate only a few rules but incur considerable memory overhead. This redundancy primarily adds ineffectual nodes and also causes some rule replication among the ineffectual nodes. The child-pointer redundancy enlarges the node’s child-pointer array which contributes about 30-50% of the total memory for the tree. Consequently, reducing this redundancy significantly reduces the total memory. Similarly, the partial redundancy in siblings’ rules manifests as rule replication which is rampant in HyperCuts even after employing node merging and moving up. To tackle both the child-pointer redundancy and partial redundancy in siblings’ rules, we propose equi-dense cuts which are unequal cuts that distribute a node’s rules as evenly among the children as possible. Equi-dense cuts achieve fine cuts in the dense parts of the rule space and coarse cuts in the sparse parts. Unequal cuts are constructed by fusing unequal numbers of HyperCuts’ equi-sized cuts. By fusing redundant equi-sized cuts, our unequal cuts (1) merge redundant child pointers at the parent node into one pointer and (2) remove replicas of rules in the fused siblings.

Fusion Heuristics
For the fusion of equi-sized cuts to produce unequal cuts, the simple and conservative heuristic is to fuse contiguous sibling leaves (i.e., corresponding to contiguous values of the bits used in the cut) if the resulting node remains a leaf (i.e., has fewer than $binth$ rules). This fusion does not affect the tree depth but reduces the number of nodes in the tree and reduces rule replication among siblings. This heuristic serves to remove fine cuts in sparse regions along with the accompanying rule replication. To capture rule replication in denser regions, the moderate heuristic fuses contiguous, non-leaf siblings if the resulting node has fewer rules than (1) the sum of the rules in the original nodes, and (2) the maximum number of rules among all the siblings of the original nodes (i.e., including those siblings that are not being fused). The first constraint ensures that the original nodes share some rules so that the heuristic reduces this redundancy. The second constraint decreases the chance of the tree becoming deeper due to the fusion. However, there is no guarantee on the tree depth because the resultant node could have a different set of rules than the original nodes which may lead to a deeper tree. The aggressive heuristic is
to fuse non-leaf nodes as long as the resulting node does not exceed some percentage (e.g., 40%) of the number of rules in the sibling with the maximum number of rules. This heuristic always reduces the number of children and thereby shrinks the child-pointer array.

**Lookup by Packets**

Because equi-dense cuts are unequal, identifying the matching child at a tree node is more involved than simple indexing into an array. Equi-sized cuts, which are powers of two in number, result in a one-to-one, ordered correspondence between the index values generated from the bits of the appropriate field(s) of the packet and the entries in the child-pointer array at each node. This correspondence enables simple indexing into the array. In contrast, unequal cuts destroy this correspondence by fusing multiple equi-sized cuts into one equi-dense cut, causing multiple indices to map to the same array entry. Consequently, simple indexing would not work and an incoming packet needs to compare against all the array entries to find the matching child. To control the complexity of the comparison hardware, that the number of unequal cuts per node is constrained, and hence the number of comparators needed, not to exceed a threshold, called \( max_cuts \). For nodes that need more cuts, the algorithm falls back on equi-sized cuts, as in HiCuts and HyperCuts, with the accompanying redundancy. One bit per node is used to indicate whether the node uses equi-sized or equi-dense cuts. Each node using equi-dense cuts stores the number of unequal cuts and an array of the starting indices of the cuts.

**Node Co-location**

In EffiCuts’ nodes using equidense cuts, the first part additionally holds the table of starting indices of each cut. A packet has to look up the cut dimension and the number of cuts in each node’s first part to determine its index into the array in the second part, and then retrieve the child node pointer at the index. Consequently, each node requires at least two memory accesses. To enable each node to require only one access and thereby achieve better memory bandwidth, a node’s child-pointer array is co-located in contiguous memory locations (the second part) with all the children’s headers (their first parts). This co-location converts the array of pointers into an array of headers and pointers to the children’s arrays (rather than pointers to the child nodes themselves). Accessing each such collocated node retrieves the header of the indexed child node in addition to a pointer to the child node’s array (assuming the memory is wide enough), thereby combining the node’s second access with the child node’s first access. Thus, each node requires only one reasonably-wide access. (While narrower memories would require more than one access, the co-location would still reduce the number of accesses by one.)

With the co-location, the array now holds the children’s headers (and the pointers to the children’s arrays). The headers must be unique for each child node in order for the index calculated from the parent node’s header to work correctly. Consequently, the headers for identical children have to be replicated in the array, incurring some extra memory (though identical children may still share a single child node’s array). Fortunately, the redundancy is minimal for EffiCuts’ equi-dense cuts where the nodes are forced to have only a few children which are usually distinct (\( max_cuts \) is 8), making it worthwhile to trade-off small amounts of memory for significant bandwidth demand reduction. To reduce further the number of memory accesses per node, HyperCuts’ rule moving-up optimization in EffiCuts is eliminated because each moved-up rule requires two accesses: one for the pointer to the rule and the other for the rule itself whereas a rule that is not moved-up in EffiCuts would fall in a leaf where the rule may contribute only a part of a wide access. Rule moving-up reduces HyperCuts’ rule replication, which is minimal for EffiCuts, and therefore, the elimination makes sense. EffiCuts facilitates incremental updates in at least two ways. First, because separable trees drastically reduce replication, updates are unlikely to involve replication, and hence do not require many changes to the tree. Second, equi-dense cuts afford new flexibility that does not exist in HyperCuts. If a new rule falls in an already-full leaf (i.e., a leaf with \( bith \) rules) then equi-dense cuts provide two options: (1) the existing cuts can be nudged to create room for the new rule by moving some of the rules from the already-full leaf to a not-full sibling; or (2) if the leaf’s parent has fewer cuts than \( max_cuts \), then a cut can be added to accommodate the new rule.

**CONCLUSIONS**

The Hicut, Hypercut, Hypersplit algorithms are the early developed algorithms among the Decision tree based packet classification algorithms. They have their own advantages and disadvantages. The DImcut, Layered Cutting scheme and EffiCuts algorithms are improved from the Hicut and Hypercut algorithms by minimizing the memory requirements and access time for the Firewall databases and access control lists.

**REFERENCES**

3. Yaxuan Qi and Jun Li “Packet Classification with Network Traffic Statistics”
4. Pankaj Gupta and Nick McKeown, — Packet Classification using Hierarchical Intelligent Cuttings‖
5. Hediyeh AmirJahanshahi Sistani, Haridas Acharya —Comparative evaluation of Recursive Dimensional Cutting Packet Classification, DimCut, with Analyisls International Journal of Computer Science & Engineering Technology (IJCSET)
7. Bo Xu, Dongyi Jiang, —HSM: A Fast Packet Classification Algorithm
12. Yeim-Kuan Chang and Han-Chen Chen Layered Cutting Scheme for Packet Classification, SIGCOMM 2010, August 30-September 3, 2010, New Delhi, India.