Dynamic Algorithms with Worst-case Performance for Packet Classification

Pankaj Gupta[1] and Nick McKeown[1]

Computer Systems Laboratory, Stanford University
Stanford, CA 94305-9030
{pankaj, nickm}@stanford.edu

Abstract. Packet classification involves — given a set of rules — finding the highest priority rule matching an incoming packet. When designing packet classification algorithms, three metrics need to be considered: query time, update time and storage requirements. The algorithms proposed to-date have been heuristics that exploit structure inherent in the classification rules, and/or trade off one or more metrics for others. In this paper, we describe two new simple dynamic classification algorithms, *Heap-on-Trie* or *HoT* and *Binarysearchtree-on-Trie* or *BoT* for general classifiers. The performance of these algorithms is considered in the worst-case, i.e., without assumptions about structure in the classification rules. They are also designed to perform well (though not necessarily the "best") in each of the metrics simultaneously.

1 Introduction

Internet routers perform packet classification to identify the flow to which arriving packets belong, and hence the action or service that the packet should receive. More formally, packet classification can be defined as:
Packet Classification: Given a classifier with N rules, $\{R_k\}_{k=1}^{N}$, where rule R_k consists of three entities: (1) A *d-tuple of ranges* $([l_1^k : r_1^k], [l_2^k : r_2^k], ..., [l_d^k : r_d^k])$, (2) A number indicating the *priority* of the rule in the classifier, referred to as $pri(R_i)$, and (3) An *action*, referred to as $action(R_i)$: for an incoming packet P with the relevant fields considered as a d-tuple of points $(P_1, P_2, ..., P_d)$, the **d-dimensional packet classification problem** is to find a specific value of j, say j^*, such that $pri(R_{j^*}) > pri(R_j)\ \forall j \neq j^*$ and $l_i^j \leq P_i \leq r_i^j, \forall i :\ 1 \leq i \leq d$ in order to identify $action(R_{j^*})$ to be applied to the packet P.

Packet classification functions have started to appear in routers to provision services [1] such as access control in firewalls, load balancing across web servers, policy-based routing, virtual private networks, network address translation, quality of service differentiation, and traffic accounting and billing.

We evaluate a packet classification algorithm for a classifier with N rules on the basis of the following metrics: (1) *Query time* – the amount of time taken to classify each arriving packet, also called the lookup or search time; (2) *Storage requirement* – memory required by the data structure used by the algorithm to hold the classification rules; (3) *Update time* – the amount of time needed to

incrementally update the data structure on insertion or deletion of classification rules. An algorithm that supports incremental updates is said to be *dynamic*. In contrast, the whole data structure has to be recomputed in a *static* algorithm, whenever a rule is added or deleted.

Fast update time is important in many applications – for example, rules need to be inserted or deleted online as flows become active or inactive in a router providing flow-based quality of service. If there is a large discrepancy between the update and query times of an algorithm, incoming packets may need to be buffered before lookup, if an update operation is in progress. Therefore, in order to avoid buffering, delay variation and head-of-line blocking problems, we seek algorithms with update time comparable to the query time.

In this paper, we present two novel dynamic algorithms and their worst-case query time, update time and storage complexities for general classifiers. These algorithms are primarily of interest in that they simultaneously have attractive worst-case complexities for each metric for any set of classification rules.

2 Related Work

The simplest algorithm is a linear search which sequentially compares the packet with each rule in turn, starting with the highest priority rule. The query time and storage complexities are both $O(N)$. Updates can be made incrementally by a simple binary search in the sorted list of rules in $O(\log N)$ time. Thus, linear search is an example of an algorithm that performs well in two metrics (storage and update time), but poorly in the query time metric. This makes the algorithm impractical for all but the smallest set of rules.

Frequently, there is a trade-off between the query and update times. Algorithms typically achieve fast query times by pre-computation to carefully organize the data structure. This, in turn, usually renders updates inefficient. Examples of solutions that ignore update times in order to have fast query times in reasonable amount of space include a ternary CAM based solution and the bitmap intersection approach of Lakshman and Stiliadis [2]. Update time complexity is $O(N)$ in each case.

Other solutions such as Grid-of-Tries proposed by Srinivasan et al [3], and the fractional cascading solution by Lakshman and Stiliadis [2] are static, and so are not relevant here as we are only considering dynamic algorithms. Heuristic solutions which attempt to take advantage of the structure of real-life classifiers are proposed in [3][4][5][6]. While these solutions seem to work well with real-life classifiers today, they have prohibitive $O(n^2)$ or higher storage requirements in the worst-case. A notable exception is the tuple space search scheme proposed by Srinivasan, Suri and Varghese [7]. This scheme has $O(N)$ worst-case storage requirement, but queries and updates can require $O(N)$ hashed memory accesses in the worst case.

Two dynamic algorithms with sub-linear worst-case bounds have been recently proposed. The first algorithm, proposed by Buddhikot, Suri, and Waldvogel [8] uses a novel prefix-partitioning technique which helps implement fast

incremental updates. This scheme achieves $O(\alpha W)$ search time, $O(N)$ space and $O(\sqrt[\alpha]{N})$ update time where α is a tunable integer parameter greater than 1. However the scheme does not readily extend to more than two dimensions or to general rules that have non-prefix field specifications. The second algorithm, proposed by Feldmann and Muthukrishnan [9], is based on a novel data structure, which they call an *FIS* tree. An FIS tree is similar to an inverted multi-ary segment tree and is essentially a static data structure. The authors also extend it to handle incremental updates. However, they state results for only a small number of updates ($O(n^{1/l})$ in one dimension). Their scheme in one-dimension has storage complexity of $O(n^{1+1/l})$, update time complexity of $O(ln^{1/l}\log n)$ and lookup complexity of $O(\log^2 n)+l$ memory accesses where l is a constant suitably chosen to trade-off lookup time versus storage requirement. The authors also make suggestions on how to handle larger number of updates, but have to "sacrifice" either space or lookup or update time to achieve that.

Table 1. Worst-case bounds obtained in this paper for dynamic d-dimensional packet classification.

Algorithm	*Query*	*Space*	*Update*
Heap-on-Trie (HoT)	$O(\log^d N)$	$O(N\log^d N)$	$O(\log^{d+1} N)$
Binarysearchtree-on-Trie (BoT)	$O(\log^{d+1} N)$	$O(N\log^d N)$	$O(\log^d N)$

In this paper, we focus on algorithms rather than implementation details, even though we favor readily-implemented data structures and algorithms. For a discussion of the implementation-related goals of a practical algorithm, please see references [2] and [4]. We propose two dynamic algorithms for one-dimensional classification with arbitrary classifiers and extend them to higher dimensions. We obtain the worst case bounds for both algorithms as shown in Table 1. To the best of our knowledge, these are the first published simultaneous worst-case bounds for search time, update time and space consumption for general multi-dimensional classifiers, where none of the worst-case metrics is "sacrificed" in favor of the other two.

3 Heap-on-Trie (HoT)

3.1 One-dimensional classifiers

If the field specifications in a one-dimensional classifier are restricted to prefixes, a trie is a good dynamic data structure supporting inserts, deletes and searches in $O(W)$ time, where W is the width of the field in bits, and therefore the depth of the trie (see [10] for an example of algorithms with a trie-like data structure). The space complexity of the trie is $O(NW)$.

If the field specifications in the rules of a classifier are not restricted to be prefixes but allowed to be arbitrary contiguous ranges, one method of storing a rule is to split a range into several maximal prefixes and to then use a trie. For example, a range $[0, 10]$ (with $W = 4$) can be split up into three maximal prefixes: (1) *0**** denoting the range $[0, 7]$; (2) *100** denoting the range $[8, 9]$ and (3) *1010* denoting the single element range $[10, 10]$. Note that any range in $[0, 2^W - 1]$ can be split into a maximum of $(2 * W - 2)$ such prefixes. We call the set of constituent prefixes of a range, G, as $C(G)$. We say that a range G is allocated to a particular trie node z if $C(G)$ has a prefix represented by the trie node z.

Some constituent prefixes of different ranges might be identical because ranges can overlap. Hence multiple rules may be allocated to the same trie node. In a static solution one simply pre-computes the highest priority of these rules and store it in the corresponding trie node. However, this is unacceptable in a dynamic solution where rules can be inserted and deleted because the identities of individual rules need to be maintained. The *Heap-on-Trie* data structure uses a heap at each trie node to maintain the rules allocated to that trie node. A heap is a data structure for storing keys with the following property – the key value at every non-leaf node is higher than the key values of its two children nodes.

A *Heap-on-Trie (HoT)* is therefore a two-level data structure where the set of ranges associated with a particular node is arranged in a heap, ordered by the priority of the ranges. The heap property ensures that the maximum priority of all the rules associated with a trie node is stored in the root node of the heap at that trie node and is hence available in $O(1)$ time. Given this data structure, a classification query proceeds downwards starting from the root of the trie, returning the maximum priority rule stored at the root node of each of the heaps associated with the nodes traversed in the trie. The total query time is $O(W)$. A given range to be inserted or deleted is first split into $O(W)$ prefixes. Each of these prefixes is then inserted/deleted separately to/from the corresponding heaps. The heaps are then readjusted to maintain heap order such that the highest priority range is available in its root node. Since an insert or delete takes $O(\log N)$ time in a heap, the total update time is $O(W \log N)$.[1] The total storage requirement is $O(NW)$ because there are $O(W)$ constituent prefixes of a range and the total number of trie nodes that can be contributed by these constituent prefixes is also $O(W)$. Hence each range consumes $O(W)$ space for a total complexity of $O(NW)$. Note that this is the same space complexity as a classifier with prefix-only field specifications.

Simplifications to the data structure are possible in restricted environments. For example, when it is known that overlaps among ranges are small, we could simply maintain a linked list at each node instead of a heap. Similarly, if an application is such that only inserts need to be supported, once the data structure

[1] For the delete operation, we need to have access to the pointers to the $O(W)$ nodes in different heaps containing the constituent maximal prefixes of a particular range. This can be easily maintained in a separate table indexed by the rule identifier at an extra cost of $O(NW)$ space.

has been built, insert time can be decreased to $O(1)$ at each node by simply checking the priority in front of the list. We insert the new rule in front of the list if this is lower than the priority of the new rule, and after the first rule otherwise. This takes a total of $O(W)$ time for a rule insertion. Likewise, if only deletes need to be supported, time for a delete operation could be brought down to $O(W)$ by using a sorted list instead of a heap and storing the pointers to the constituent prefixes of a range in different lists corresponding to each range present in the classifier. To delete a range, one would access the corresponding prefixes by these pointers and delete them from their respective sorted lists in $O(1)$ time each, to obtain a total delete time of $O(W)$. A summary of these bounds is shown in Table 2.

Table 2. Bounds for the different types of dynamic algorithms using the HoT data structure.

Algorithm-HoT	*Query*	*Space*	*Insert-Time*	*Delete-Time*
Dynamic	$O(W)$	$O(NW)$	$O(W \log N)$	$O(W \log N)$
Semi-dynamic (only inserts)	$O(W)$	$O(NW)$	$O(W)$	–
Semi-dynamic (only deletes)	$O(W)$	$O(NW)$	–	$O(W)$

3.2 Multi-dimensional classifiers

The dynamic data structure for d-dimensions can be obtained by building hierarchical multi-level tries, one level for each dimension except for the last, on which a *Heap-on-Trie* is built. It is not difficult to see that the total space consumed is then $O(NW^d)$ with query and update times being $O(W^d)$ and $O(W^d \log N)$ respectively. We do not use fractional cascading in these multi-level tries, as it is essentially a static technique for searching among similar lists. Mehlhorn and Naher have proposed a dynamic version of fractional cascading [11] where updates to cascaded lists can be made such that query time complexity only increases from $O(\log N)$ to $O(\log N \log\log N)$. However, the constants hidden in the *O(.)* notation are so high that a simpler $O(\log^2 N)$ solution that does not use fractional cascading is usually better for all practical values of N [12].

4 Binarysearchtree-on-Trie (BoT)

The motivation in this section is to develop an algorithm that executes updates faster than the *HoT* algorithm described above. As is often the case, faster updates are obtained at the expense of increased query time.

4.1 One-dimensional classifiers

We start with defining the terminology used in this section and then describe the BoT data structure and algorithms.

Definitions

- For a node z in a trie T, we let *parent(z)*, *lchild(z)* and *rchild(z)* denote its parent, left and right children nodes. Also, we let *root(T)* denote the root node of trie T.
- For a node z of a W-bit trie, we define *prefix(z)* to be the prefix that z represents in the trie. The length of *prefix(z)*, denoted as *length(prefix(z))*, is defined to be the distance of the node z from the root of the trie, We define the root's children to be at a distance of 1 from it. We can write *prefix(z)* by the W-bit string $z_1 z_2 \ldots z_l * * \ldots *$, where $l = length(prefix(z))$, or simply by an $(l+1)$-bit string $z_1 z_2 \ldots z_l *$ where the trailing $(W-l)$ '*' wildcard-bits are implicit.
- *prefix(z)* defines a contiguous range, called *prefixRange(z)* of size $2^{W-length(prefix(z))}$ on the integer number line. The starting point of this range is denoted by *st(z)* and equals $z_1 z_2 \ldots z_l 0 \ldots 0$, while the ending point of this range is denoted by *en(z)* and equals $z_1 z_2 \ldots z_l 1 \ldots 1$. Thus, *prefixRange(z)* = [*st(z)*, *en(z)*].
- We associate a distinguished point, $Pt(z)$, with every trie node, z, and refer to it as the *d-point*. The d-point is the midpoint of *prefixRange(z)*. Equivalently, $Pt(z) = \lfloor (st(z) + en(z))/2 \rfloor = z_1 z_2 \ldots z_l 10 \ldots 0$.
- $G(z)$ denotes the set of ranges allocated to trie node z. The algorithm for allocation of ranges to nodes is described below.
- $PGL(z)$ (respectively $PGR(z)$) is defined to be the set of the leftmost (rightmost) endpoints of the ranges in $G(z)$. If e is any such endpoint in PGL or in PGR, we let $range(e)$ denote the range for which e is one of the endpoints.

Table 3. *An example one dimensional 4-bit classifier consisting of arbitrary ranges. The priority of R_i is assumed to be i.*

Rule	Range	Maximal Prefixes
R_5	[3,11]	0011, 01**, 10**
R_4	[2,7]	001*, 01**
R_3	[4,11]	01**, 10**
R_2	[4,7]	01**
R_1	[1,15]	0001, 001*, 01**, 10**, 110*, 1110

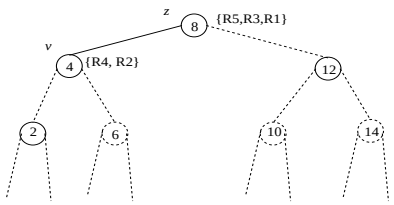

Fig. 1. *Showing the range allocations to the trie nodes for rules in Table 3. The number inside a trie node v represents Pt(v) associated with it. In this particular example, all ranges are allocated to the root node and its left child. The remaining nodes are actually not present in the trie – they are only shown here to illustrate the calculation of the distinguished points.*

Table 4. Showing the allocated ranges to the trie nodes.

Trie Node,w	$Pt(w)$	$G(w)$	$PGL(w)$	$PGR(w)$
z	8	$\{R_5, R_3, R_1\}$	$\{1, 3, 4\}$	$\{11, 11, 15\}$
v	4	$\{R_4, R_2\}$	$\{2, 4\}$	$\{7, 7\}$

The *BoT* Data Structure Similar to *HoT*, we use a trie as the underlying data structure. The basic difference is that we now allocate a range to only one trie node rather than to $O(W)$ nodes. We allocate a range H to a trie node z, if z satisfies the following two conditions: (1) H contains $Pt(z)$; and (2) If z is not the root node, H does not contain $Pt(parent(z))$. For example, range $R_5[0011, 1011]$ in Table 3 is allocated to the root node of the trie as it contains the point $1000 = Pt(root(T))$; while range $R_4[0010, 0111]$ is allocated to the left child of $root(T)$ as it does not contain $Pt(root(T))$ but contains $Pt(lchild(root(T))) = 0100$. We allocate every range in the one-dimensional classifier to exactly one trie node in this manner. The set of allocations for the example classifier in Table 3 is shown in Figure 1. This is also shown in tabular form along with the sets PGL and PGR in Table 4.

Each of the sets *PGL(z)* and *PGR(z)* is stored in a balanced binary search tree (BBST) — *PGL(z)* in *BBSTleft(z)* and *PGR(z)* in *BBSTright(z)*. We can choose one of various implementations of a BBST data structure, such as red-black trees [13] and 2-3 trees [14] for *BBSTleft(z)* and *BBSTright(z)*. We have three fields in a node, x, of the BBST:

1. *val(x)* which stores the value of one of the endpoints of the relevant range. Nodes in BBSTleft contain the left endpoint and those in BBSTright contain the right endpoint of the range associated with the node.
2. *pri(x)* which stores *pri(range(val(x)))*.
3. *augp(x)* which augments the BBST to enable fast search operations. *augp(x)* stores the priority of the highest priority rule among the rules in the subtree

rooted at x. Thus, it can be recursively defined as follows:

$$augp(x) = \begin{cases} 0 & \text{if } x \text{ is nil} \\ pri(x) & \text{if } x \text{ is a leaf} \\ max(augp(lchild(x)), augp(rchild(x)), pri(x)) & \text{otherwise} \end{cases}$$

If a BBST implementation, for example a 2-3 tree, is such that it stores data only in its leaf nodes, $pri(x)$ is considered to be 0 and is ignored in the calculation of $augp(x)$ for all internal nodes x. The augmented BBSTs for our running example are shown in Figure 2, with the *augp* field shown in bold and bigger font than the *val* and *pri* fields.

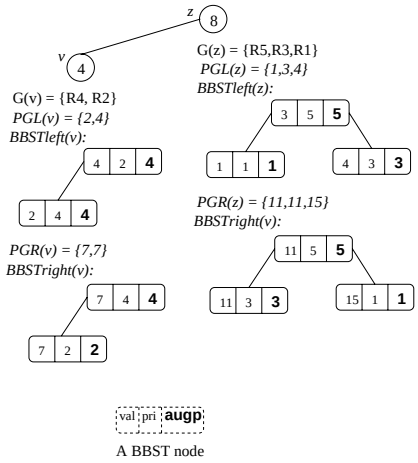

Fig. 2. *Showing the BBSTs associated with each trie node.*

Query Algorithm Consider *BBSTleft(v)*, which stores the set of left endpoints, *PGL(v)*, for the trie node v. Since $G(v)$ contains only those ranges that intersect $Pt(v)$ and *PGL(v)* has the left endpoints of these ranges, all points in *PGL(v)* lie to the left of $Pt(v)$ on the number line. Given a point Q representing an incoming packet, we define the following functions on the BBSTs at a trie node v:

1. *gethpLeft(B,Q)* — If the point Q is such that $Q < Pt(v)$, the function *gethpLeft(B,Q)* returns the priority of the highest priority range in $G(v)$ that contains the point Q. Equivalently, *gethpLeft()* calculates $max_j\{pri(range(e_j))|e_j \in PGL(v), e_j \leq Q\}$. Note that $e_j \leq Q$ if and only if $range(e_j)$ contains Q. This function is performed by an algorithm that traverses the BBST B from its root to a leaf node. The algorithm looks at the current node being traversed in B, say v. Either $Q < val(v)$ – in which case, the traversal descends to the

left-child of v, or $Q \geq val(v)$ – in which case, the highest priority value found so far in the traversal is compared with $max(augp(v), pri(v))$ and updated if found smaller than this value. The traversal descends to the right child of v in this case. On reaching a null node, the maximum priority found by the traversal algorithm is the desired result.

2. *gethpRight(B,Q)* — If $Q \geq Pt(v)$, the function *gethpRight(B,Q)* returns $max_j\{pri(range(f_j))|f_j \in PGR(v), f_j \geq Q\}$. The algorithm is similar to that for $gethpLeft(B, Q)$.
3. *gethpTrieNode(v,Q)* – This function obtains the priority of the highest priority range in $G(v)$ that contains the point Q. It first compares Q with $Pt(v)$ and returns *gethpLeft(BBSTleft(v), Q)* if $Q < Pt(v)$ or *gethpRight(BBSTright(v), Q)* otherwise.

The above functions are illustrated in Figure 3. Functions *gethpLeft* and *gethpRight* spend $O(1)$ time at each node on the traversal path in the corresponding BBST. Since the depth of each BBST is $O(\log |G(v)|)$, *gethpTrieNode* takes $O(\log |G(v)|)$ or $O(\log N)$ time for a classifier with N rules.

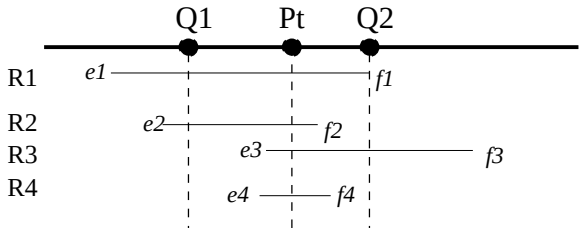

Fig. 3. *The set of ranges in $G(v)$ are shown in this figure. All ranges intersect $Pt = Pt(v)$. To calculate $gethp(v, Q1)$; we first find that $Q1 < Pt$, and therefore calculate gethpLeft(BBSTleft(v), Q1). This function considers ranges R_1 and R_2 returning the one with higher priority. Similarly for $gethp(v, Q2)$, gethpRight(BBSTright(v), Q2) considers ranges R_1 and R_3.*

An incoming packet Q is classified using the function *gethpTrieNode(v)* as follows: we traverse the trie nodes according to the bits in Q in the usual manner. For each node v in the trie traversal path, we call the function *gethpTrieNode(v)* and calculate the maximum of all the returned values. This maximum value is then the highest priority rule matching Q. The query algorithm makes at most W *gethpTrieNode()* calls and is therefore of complexity $O(W \log N)$.

The storage complexity of the BoT data structure is $O(NW)$ because a BBST is a linear space data structure in the number of nodes. Hence total space consumption equals $O(NW) + \sum_{v \in T} O(|G(v)|) = O(NW) + O(N) = O(NW)$. Comparing the HoT and BoT query algorithms, we see that their worst case storage complexity is identical. In practice, however, we expect a classifier to require a smaller amount of storage in the BoT algorithm because a range is

allocated to only one trie node as opposed to $O(W)$ trie nodes in the *HoT* algorithm.

Update algorithm We now describe the incremental update algorithms on a *BoT* data structure. We only describe the insertion algorithm – the deletion algorithm is similar and has the same complexity as the insertion algorithm. An incremental update needs to modify only the BBSTs of one trie node, the one that contains the rule to be updated. Modifying a BBST requires adding a new node to the BBST and appropriately updating the *augp* fields of all nodes. The addition of a new node has one non-trivial constraint – that the binary tree should be kept balanced. For concreteness, we describe the update algorithm under the assumption that a 2-3 tree is used for the implementation of BBSTs. The ideas are similar for other BBST implementations such as red-black trees.

A 2-3 tree stores the data only in its leaf nodes. Therefore, we assign a value of 0 to the *pri* field of each internal node and ignore this field in the calculation of the *augp* field for internal nodes. Every internal node of a 2-3 tree has either two or three children, and every path from the root node to a leaf node is of the same length. Insertion of a node with a new value, n, in a 2-3 tree is done in three phases:

- (Phase 1) The algorithm descends the tree and adds the node n as a child of some node, b, in the order determined by the relative ordering of n and the values in the existing children nodes of b.
- (Phase 2) The addition of a new child to node b may result in b having four children. In order to restore the 2-3 property, we create a new node b', make it the parent of two of the children nodes of b, and add b' as a child to the parent node of b. We recurse if the parent node of b now violates the 2-3 property. The recursive algorithm proceeds upward from b till it reaches the root of the tree.
- (Phase 3) The path from leaf n to the root is traced to update the value of the field *augp*.

The analysis of the insertion algorithm is simple. The depth of a 2-3 tree with N leaves is no more than $\log N$. Therefore phases 1 and 3 take $O(\log N)$ time. Phase 2 also takes $O(\log N)$ time because the recursive algorithm always moves up towards the root of the tree in one step, and $O(1)$ work is performed at each step. Combining, the total complexity of the insertion algorithm is $O(\log N)$. The deletion algorithm is similar and therefore has the same complexity. In summary, incremental updates take $O(\log N)$ time on a *BoT* data structure.

4.2 Multi-dimensional classifiers

The *BoT* data structure can be extended to multiple dimensions in a manner similar to the *HoT* data structure by using multi-level tries, one level for each dimension except the last, on which a *BoT* is built. For example, in the *BoT* for a two dimensional classifier, the ranges of the rules in one dimension are

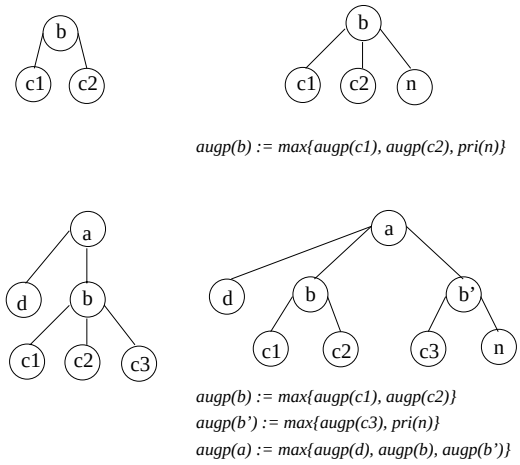

Fig. 4. *Showing the different cases of an update operation.*

allocated to $O(W)$ nodes of the first level trie and a one-dimensional BoT as explained above will be built on the nodes of this first level trie using the ranges of the rules in the second dimension. The total space consumed is then $O(NW^d)$ with query and update time complexities being $O(W^d \log N)$ and $O(W^{d-1} \log N)$ respectively.

5 Conclusions and open problems

This paper presented two dynamic algorithms for multi-dimensional packet classification. The complexities of these algorithms are as shown in Table 1 where W (the number of bits used in each dimension) has been approximated by $\log N$. The choice of algorithm in a practical application will be determined by the query and update time requirements.

There have been a number of proposed heuristics for packet classification, each with its pros and cons. In general, these algorithms exploit structure present in existing classifiers. Other non-heuristic algorithms trade off update time for reduced query time, or do not generalize to multiple dimensions. While these algorithms perform well today on relatively small, well-understood classifiers and on classifiers that change infrequently, they will perform less well in future when classifiers change rapidly (e.g., in a router providing service differentiation), or when classifiers evolve to have different structure.

As a first step towards future requirements, the *Hot* and BoT algorithms do not trade off among the metrics of query time, update time and space requirements. While still lacking in some implementation details, both algorithms have reasonable worst-case bounds in all three metrics for general multi-dimensional classifiers.

Finally, we pose two questions that are natural consequences of this paper:

1. What are non-trivial lower bounds to the query time in the d-dimensional packet classification problem, both static and dynamic version, in a given amount of space?
2. Can we improve upon *both* the search and update times of Table 1 in the same or better space complexity?

6 Acknowledgments

We wish to gratefully acknowledge Michael Lamoureux for useful discussions and for giving access to his bibliography on multi-dimensional search. We also wish to acknowledge Leo Guibas for useful discussions and Youngmi Joo for useful comments on a draft of this paper.

References

1. Cisco Systems white paper, "Advanced qos services for the intelligent internet," http://www.cisco.com/warp/public/cc/cisco/mkt/ios/qos/tech/qos_wp.htm.
2. T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proceedings of ACM SIGCOMM*, Sept. 1998, pp. 191–202.
3. V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Scalable level 4 switching and fast firewall processing," in *Proceedings of ACM SIGCOMM*, Sept. 1998, pp. 203–214.
4. P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proceedings of ACM SIGCOMM*, Sept. 1999, pp. 147–160.
5. D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, "Router plugins: A software architecture for next generation routers," in *Proceedings of ACM SIGCOMM*, Sept. 1998, pp. 229–240.
6. P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," *Hot Interconnects VII*, Aug. 1999.
7. V. Srinivasan, G. Varghese, and S. Suri, "Fast packet classification using tuple space search," in *Proceedings of ACM SIGCOMM*, Sept. 1999, pp. 135–146.
8. M. M. Buddhikot, S. Suri, and M. Waldvogel, "Space decomposition techniques for fast layer-4 switching," *Protocols for High Speed Networks*, vol. 66, no. 6, pp. 277–283, Aug. 1999.
9. A. Feldmann and S. Muthukrishnan, "Tradeoffs for Packet Classification," in *Proceedings of INFOCOM*, Mar. 2000.
10. W. Doeringer, G. Karjoth, and M. Nassehi, "Routing on longest-matching prefixes," *IEEE/ACM Transactions on Networking*, vol. 4, no. 1, pp. 86–97, Feb. 1996.
11. K. Mehlhorn and S. Naher, "Dynamic fractional cascading," *Algorithmica*, vol. 5, pp. 215–241, 1990.
12. L. J. Guibas, "Private communication," .
13. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.
14. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.