Packet Classification Using Two-Dimensional Multibit Tries

Wencheng Lu and Sartaj Sahni
Department of Computer and Information Science and Engineering,
University of Florida, Gainesville, FL 32611
{wlu, sahni}@cise.ufl.edu

September 21, 2004

Abstract

We develop fast algorithms to construct space-optimal contrained two-dimensional multibit tries for Internet packet classifier applications. Experimental evidence suggests that using the same memory budget, space-optimal two-dimensional multibit tries require 1/4 to 1/3 the memory accesses required by two-dimensional one-bit tries for table lookup. A heuristic for two-dimensional multibit tries with switch pointers also is proposed.

Keywords
Packet classification, two-dimensional tries, multibit tries, switch pointers, prefix expansion, fixed-stride tries, variable-stride tries, dynamic programming.

1 Introduction

An Internet router classifies incoming packets based on their header fields using a classifier, which is a table of rules. Each classifier rule is a pair (F, A), where F is a filter and A is an action. If an incoming packet matches a filter in the classifier, the associated action specifies what is to be done with this packet. Typical actions include packet forwarding and dropping. A d-dimensional filter F is a d-tuple $(F[1], F[2], \cdots, F[d])$, where $F[i]$ is a range that specifies destination addresses, source addresses, port numbers, protocol types, TCP flags, etc. A packet is said to match filter F, if its header field values fall in the ranges $F[1], \cdots, F[d]$. Since it is possible for a packet to match more than one of the filters in a classifier, a tie breaker is used to determine a unique matching filter.

In one-dimensional packet classification (i.e., $d = 1$), $F[1]$ is usually specified as a destination address prefix and lookup involves finding the longest prefix that matches the packet's destination address. Data structures for longest-prefix matching have been extensively studied (see [13, 14], for surveys). Although 1-dimensional prefix filters are adequate for destination based packet forwarding, higher dimensional filters are required for firewall, quality of service, and virtual private network applications, for example. Two-dimensional prefix filters, for example, may be used "to represent host to host or network to network or IP multicast flows" [9] and higher dimensional filters are required if these flows are to be represented "with greater granularity." Eppstein and Muthukrishnan [6] state that "Some proposals are underway to specify many fields ... while others are underway which seem to preclude using more than just the source and destination IP addresses ... (in IPsec for example, the source or destination port numbers may not be revealed)." Kaufman et al. [20] also point out that in IPsec, for security reasons, fields other than the source and destination address may not be available to a classifier. *Thus two-dimensional prefix filters*

represent an important special case of multi-dimensional packet classification. Data structures for multi-dimensional (i.e., $d > 1$) packet classification are developed in [1, 2, 3, 5, 6, 7, 8, 9, 11, 12, 16, 17, 25, 21, 22], for example.

Our focus in this paper is trie structures for d-dimensional packet classification, $d \geq 2$, Srinivasan and Varghese [25] proposed using two-dimensional one-bit tries for destination-source prefix filters. The proposed two-dimensional trie structure takes $O(nW)$ memory, where n is the number of filters in the classifier and W is the length of the longest prefix. Using this structure, a packet may be classified with $O(W^2)$ memory accesses. The basic two-dimensional one-bit trie may be improved upon by using pre-computation and switch pointers [25]. The improved version classifies a packet making only $O(W)$ memory accesses. Srinivasan and Varghese [25] also propose extensions to higher-dimensional one-bit tries that may be used with d-dimensional, $d > 2$, filters. Baboescu et al. [3] suggest the use of two-dimensional one-bit tries with buckets for d-dimensional, $d > 2$, classifiers. Basically, the destination and source fields of the filters are used to construct a two-dimesional one-bit trie. Filters that have the same destination and source fields are considered to be equivalent. Equivalent filters are stored in a bucket that may be serached serially. Baboescu et al. [3] report that this scheme is expected to work well in practice because the bucket size tends to be small. They note also that switch pointers may not be used in conjunction with the bucketing scheme.

Srinivasan et al. [18] explored the idea of using multibit tries (i.e., tries whose degree is more than 2) for one-dimensional packet classification. They proposed the use of fixed- (FST) and variable-stride (VST) one-dimensional tries and developed dynamic programming algorithms to construct space-optimal FSTs and VSTs whose height is k, where k is a design parameter[1]. Sahni and Kim [23] develop inproved algorithms for space optimal one-dimensional FSTs and VSTs. In this paper, we develop fast polynomial-time algorithms to construct space-optimal constrained 2DMTs. The constructed 2DMTs may be searched with at most k memory accesses, where k is a design parameter. Our space-optimal constrained 2DMTs may be used for d-dimensional filters, $d > 2$, using the bucketing strategy proposed in [3]. For the case $d = 2$, switch pointers may be employed to get multibit tries that require less memory than required by our space-optimal constrained 2DMTs and that permit packet classification with at most k memory accesses. We develop a fast heuristic to construct good multibit tries with switch pointers. Experiments conducted by us indicate that, given the same memory budget, our proposed space-optimal constrained 2DMT structures perform packet classification using 1/4 to 1/3 as many memory accesses as required by the one-bit two-dimensional tries of [25, 3].

We begin, in Section 2, by reviewing basic concepts related to the trie data structure. This section also describes multibit fixed- and variable-stride tries and prefix expansion [18]. Section 3 develops basic concepts related to two-dimensional multibit tries. Dynamic programming algorithms for space-optimal constrained 2DMTs are developed in Sections 6. Two algorithms needed in support of these dynamic programming algorithms are developed in Sections 4 and 5. Our heuristic for two-dimensional multibit tries with switch pointers is developed in Section 7 and our experimental results are presented in Section 8.

2 Tries

2.1 1-bit Tries

A *1-bit trie* is a binary tree-like structure in which each node has two element fields, *le* (left element) and *re* (right element) and each element field has the components *child* and *data*. Branching is done based on the bits in the search key. A left-element child branch is followed at a node at level i (the root is at

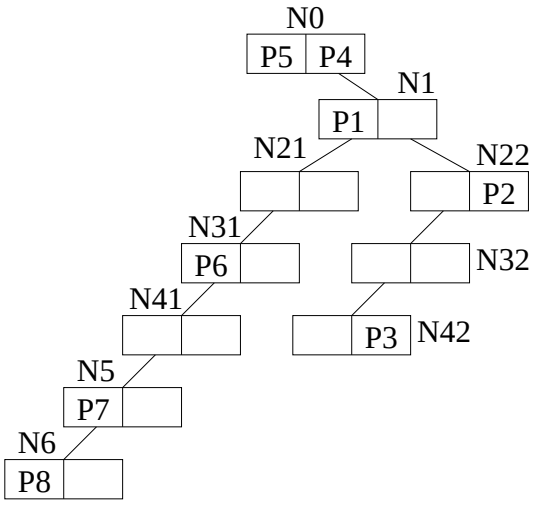
[1]We are interested in height-bounded multibit tries because the number of memory accesses needed to search a multibit trie is bounded by its height. Hence, worst-case classifier performance is determined by the height of the multibit trie.

level 0) if the ith bit of the search key is 0; otherwise a right-element child branch is followed. Level i nodes store prefixes whose length is $i+1$ in their data fields. A prefix that ends in 0 is stored as $le.data$ and one whose last bit is a 1 is stored as $re.data$. The node in which a prefix is to be stored is determined by doing a search using that prefix as key. Let N be a node in a 1-bit trie and let E be an element field (either left or right) of N. Let $Q(E)$ be the bit string defined by the path from the root to N followed by a 0 in case E is a left element field and 1 otherwise. $Q(E)$ is the prefix that corresponds to E. $Q(E)$ is stored in $E.data$ in case $Q(E)$ is one of the prefixes to be stored in the trie.

Figure 1 shows a set of 8 prefixes and the corresponding 1-bit trie. The $*$ shown at the right end of each prefix is used neither for the branching decsribed above nor is in the length computation. So, the length of $P1$ is 2.

Original prefixes
P5=0*
P1=10*
P2=111*
P3=11001*
P4=1*
P6=1000*
P7=100000*
P8=1000000*

(a) 8-prefix example of [18]



(b) Corresponding 1-bit trie

Figure 1: Prefixes and corresponding 1-bit trie [23]

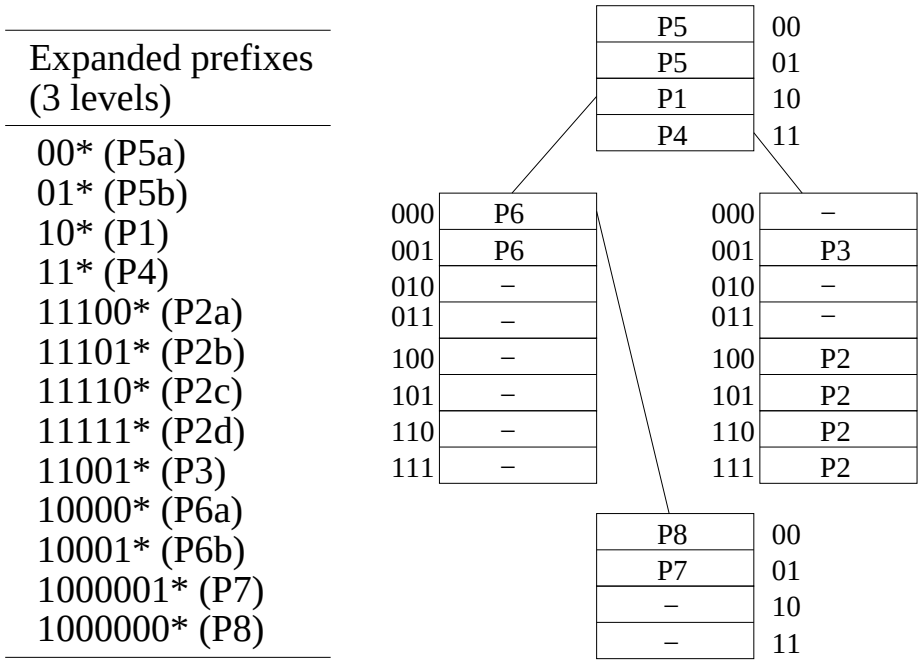
2.2 Multibit Tries

The *stride* of a node is defined to be the number of bits used at that node to determine which branch to take. A node whose stride is s has 2^s element fields (corresponding to the 2^s possible values for the s bits that are used). Each element field has a data and a child component. A node whose stride is s requires 2^s memory units (one memory unit being large enough to accomodate an element field). Note that the stride of every node in a 1-bit trie is 1.

In a *fixed-stride trie* (FST), all nodes at the same level have the same stride; nodes at different levels may have different strides. In a *variable-stride trie* (VST), nodes may have different strides regardless of their level.

Suppose we wish to represent the prefixes of Figure 1(a) using an FST that has three levels. Assume that the strides are 2, 3, and 2. The root of the trie stores prefixes whose length is 2; the level one nodes store prefixes whose length is 5 (2 + 3); and level three nodes store prefixes whose length is 7 (2 + 3 + 2). This poses a problem for the prefixes of our example, because the length of some of these prefixes is different from the storeable lengths. For instance, the length of P5 is 1. To get around this problem, a prefix with a nonpermissible length is expanded to the next permissible length [18]. For example, P5 = 0*

is expanded to P5a = 00* and P5b = 01*. If one of the newly created prefixes is a duplicate, dominance rules are used to eliminate all but one occurrence of the prefix. Because of the elimination of duplicate prefixes from the expanded prefix set, all prefixes are distinct. Figure 2(a) shows the prefixes that result when we expand the prefixes of Figure 1 to lengths 2, 5, and 7. Duplicate prefixes, following expansion, are eliminated by favoring longer length original prefixes. Figure 2(b) shows the corresponding FST whose height is 2 and whose strides are 2, 3, and 2.



(a) Expanded prefixes (b) Corresponding fixed-stride trie

Figure 2: Prefix expansion and fixed-stride trie

Since the trie of Figure 2(b) can be searched with at most 3 memory accesses, it represents a time-performance improvement over the 1-bit trie of Figure 1(b), which requires up to 7 memory accesses to perform a search. However, the space requirements of the FST of Figure 2(b) are more than that of the corresponding 1-bit trie. For the root of the FST, we need 4 units; the two level 1 nodes require 8 units each; and the level 3 node requires 4 units. The total is 24 memory units. Note that the 1-bit trie of Figure 1 requires only 20 memory units.

Let N be a node at level j of a multibit trie. Let s_0, s_1, $\cdots$, s_j be the strides of the nodes on the path from the root of the multibit trie to node N. Note that s_0 is the stride of the root and s_j is the stride of N. With node N we associate a pair $[s, e]$, called the *start-end* pair, that gives the start and end levels of the corresponding 1-bit trie covered by this node. By definition, $s = \sum_{i=0}^{j-1} s_j$ and $e = s + s_j - 1$. In the case of an FST all nodes at the same level of the FST have the same $[s, e]$ values. In the FST of Figure 2(b), the $[s, e]$ values for the level 0, level 1, and level 2 nodes are $[0, 1]$, $[2, 4]$ and $[5, 6]$, respectively.

Starting with a 1-bit trie for n prefixes whose length is at most W, the strides for a space-optimal FST with at most k levels may be determined in $O(nW + kW^2)$ time[2] [18, 23]. For a space-optimal VST

[2]The complexity of $O(kW^2)$ given in [18, 23] assumes we start with data extracted from the 1-bit trie; the extraction of this data takes $O(nW)$ time.

Filter	Dest	Source	Cost	Action
F1	0*	1100*	1	Allow inbound mail
F2	0*	1110*	2	Allow DNS access
F3	0*	1111	3	Secondary access
F4	000*	10*	4	Incoming telnet
F5	000*	11*	5	Outgoing packets
F6	0001*	000*	6	Return ACKs okay
F7	0*	1*	7	Block packet

Table 1: An example of seven dest-source filters

whose height[3] is constrained to k, the strides may be determined in $O(nW^2k)$ time [18, 23].

2.3 Two-Dimensional 1-Bit Tries

A *two-dimensional 1-bit trie* (2D1BT) is a one-dimensional 1-bit trie (called the top-level trie) in which the data field of each element[4] is a pointer to a (possibly empty) 1-bit trie (called the lower-level trie). So, a 2D1BT has 1 top-level trie and potentially many lower-level tries.

Consider the 7-rule two-dimensional classifier of Table 1. For each rule, the filter is defined by the Dest (destination) and Source prefixes. So, for example, $F1 = (0*, 1100*)$ matches all packets whose destination address begins with 0 and whose source address begins with 1100. When a packet is matched by two or more filters, the rule with least cost is used. The classifier of Table 1 may be represented as a 2D1BT in which the top-level trie is constructed using the destination prefixes. In the context of our destination-source filters, this top-level trie is called the *destination trie*. Let N be a node in the destination trie and let E be one of the element fields (either *le* or *re*) of N. If no dest prefix equals $Q(E)$, then $N.E.data$ points to an empty lower-level trie. If there is a dest prefix D that equals $Q(E)$, then $N.E.data$ points to a 1-bit trie for all source prefixes S such that (D, S) is a filter. In the context of destination-source filters, the lower-level tries are called *source tries*. We say that N has two source-tries (one or both of which may be empty) *hanging* from it. Figure 3 gives the 2D1BT for the filters of Table 1.

3 Two-Dimensional Multibit Tries

3.1 Definition

Although two-dimensional multibit tries (2DMTs) do not appear to have been studied in the literature, they are a natural extension of 2D1BTs to the case when nodes of the dest and source tries may have a stride that is more than 1. As in the case of one-dimensional multibit tries, prefix expansion is used to accomodate prefixes whose length lies between the $[s, e]$ values of a node on its search path. Let $D_1, D_2, \cdots, D_u$ be the distinct destination prefixes in the rule table. For the rules of Table 1, $u = 3$, $D_1 = 0*$, $D_2 = 000*$ and $D_3 = 0001*$. The destination trie of the 2DMT for our 7 filters is a multibit trie for D_1–D_3 (see Figure 4). In the destination trie of this figure element fields that correspond to a D_i or the expansion of a D_i (in case D_i is to be expanded) are marked with '+'. The remaining element

[3]The height of a trie is the number of levels in the trie. Height is often defined to be 1 less than the number of levels. However, the definition we use is more convenient in this paper.

[4]Recall that each node of a 1-bit trie has two element fields.

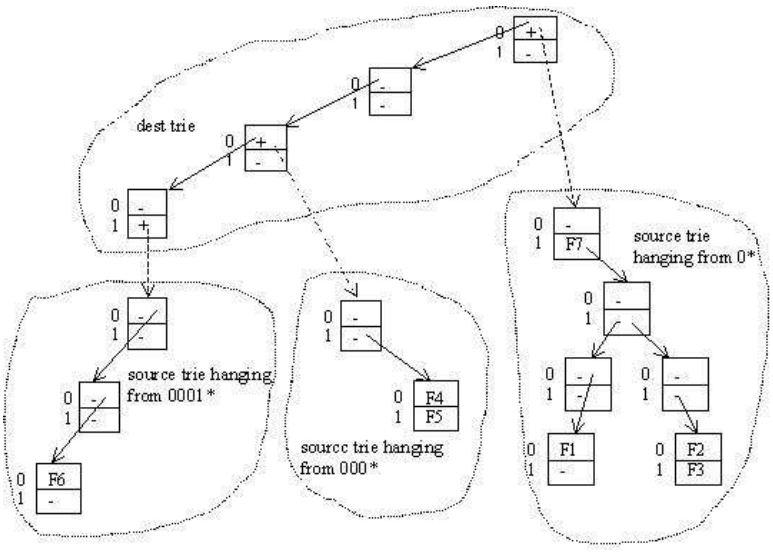


Figure 3: Two-dimensional 1-bit trie for Table 1

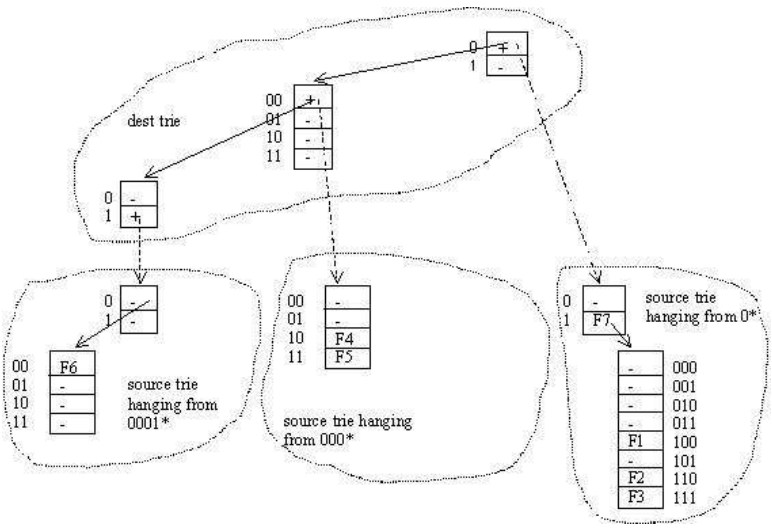


Figure 4: 2DMT for filters of Table 1

fields are marked with '−'. Element fields marked with '+' have a non-empty source trie hanging from them, the remaining element fields have an empty (or null) source trie hanging from them.

Let $L(p)$ be the length of the prefix p. To define the source tries of a 2DMT, we introduce the following terminology:

- $D_{i,j} = \{D_q | D_q$ is a prefix of D_i and $L(D_i) - L(D_q) \leq j\}$
- $S_{i,j} = \{S_q | (D_q, S_q)$ is a filter and $D_q \in D_{i,j}\}$

We see that $D_{i,0} = \{D_i\}$ for all i. For our example filter set, $D_{1,0} = \{0*\}$, $S_{1,0} = \{1*, 1100*, 1110*, 1111*\}$, $D_{2,0} = D_{2,1} = \{000*\}$, $S_{2,0} = S_{2,1} = \{10*, 11*\}$, $D_{2,2} = \{0*, 000*\}$, $S_{2,2} = \{1*, 10*, 11*, 1100*, 1110*, 1111*\}$,

$D_{3,0} = \{0001*\}$, $S_{3,0} = \{000*\}$, $D_{3,1} = D_{3,2} = \{0001*, 000*\}$, $S_{3,1} = S_{3,2} = \{000*, 10*, 11*\}$, $D_{3,3} = \{0001*, 000*, 0*\}$, and $S_{3,3} = \{000*, 10*, 11*, 1100*, 1110*, 1111*\}$.

Let N be a node in the destination trie of a 2DMT. Let $[s, e]$ be the start-end pair associated with N and let E be an element field of N. Let D_i be the longest destnation prefix, $s < L(D_i) \leq e + 1$, that expands (note that a prefix may expand to itself, a trivial expansion) to $Q(E)$. If there is no such D_i, the source trie that hangs from E is empty. If D_i exists, the source trie that hangs from E is a multibit trie for $S_{i,j}$, where $j = L(D_i) - s - 1$. When multiple filters expand to the same element field of the multibit source trie, the least-cost filter is recorded. For our example of Figure 4, the source tries hanging from the '+' fields of the three dest-trie nodes (top to bottom) are $S_{1,0}$, $S_{2,1}$ and $S_{3,0}$. *Note that, in a 2DMT, several Es may have the same source trie hanging from them. To conserve space, only one copy of each distinct source trie is retained.*

A 2DMT may be searched for the least-cost filter that matches a given pair of destination and source addresses (da, sa) by following the search path for da in the destination trie of the 2DMT. All source tries encountered on this path are searched for sa. The least-cost filter recorded on these source-trie search paths is the least-cost filter that matches (da, sa). Suppose we are to find the least-cost filter that matches $(00010, 11101)$. Searching the 2DMT of Figure 4 for 00010 takes us through the 0 field of the root, the 00 field of the root's left child, and the 1 field of the remaining dest-trie node. Each of these three fields has a non-empty source-trie hanging from it. So, three source tries are searched for 11101. When the root source-trie is searched, the element fields with $F2$ and $F7$ are encountered. When the source trie in the left child of the root is searched, the element field with $F5$ is encountered. The search in the remaining source trie encounters no matching filters. The least-cost matching-filter is determined to be $F2$.

Let P be any root to leaf path in the destination trie of a 2DMT T. Let the sum of the heights of the source tries hanging from the element fields on this path be $H(P)$. The maximum number of memory accesses (MNMA) needed to find the least-cost matching filter for a given destination-source address pair is

$$MNMA(T) = \max_P\{H(P) + \text{number of nodes on } P\}$$

We can reduce the MNMA needed for a search of a 2DMT through the use of switch pointers [16]. Although [16] describes the use of switch pointers only in the context of 2D1BTs, the concept is extended easily to 2DMTs (though not to 2DMTs augmented with the bucketing scheme of [3]).

4 The Algorithm *dominatingLevel*

In this section we describe an algorithm that runs on the 1-bit trie O of distinct destination prefixes of a two-dimensional classifier. Let D_1, D_2, $\cdots$, D_u be these distinct destination prefixes. Each D_i is stored in exactly one of the element fields of exactly one of the nodes of O. Let N be a node of O. Let E be one of the element fields of N. If $Q(E)$ is one of the D_is, this D_i is stored in $E.data$. Otherwise, $E.data$ is empty. We say that $Q(E)$ is *dominated* by D_i iff $L(Q(E)) < L(D_i)$ and $Q(E)$ is a prefix of D_i. The *dominating level* for E is the smallest l such that every downward path in O from $E.child$ encounters a D_i by the time a node at level l is reached. The dominating level for E is ∞ in case there is no such l. In particular if $E.child$ is null, the dominating level for E is ∞. Figure 5 gives the recursive algorithm $dominatingLevel(N)$, which computes the dominating level for each element (field) in the subtree rooted at N. The initital invocation is $dominatingLevel(root(O))$. $E.dl$ denotes the dominating level of E. The time compexity of $dominatingLevel$ is $O(uW)$. Its space complexity is $O(uW)$ as O has this many nodes.

```
Algorithm dominatingLevel(N)
{
    if (N == null) return ∞;
    N.le.dl = lDL = dominatingLevel(N.le.child);
    N.re.dl = rDL = dominatingLevel(N.re.child);

    if (N.le.data ≠ null)
        lDL = level# of N in O;
    if (N.re.data ≠ null)
        rDL = level# of N in O;
    return max(lDL, rDL);
}
```

Figure 5: Algorithm for calculating dominating levels.

```
Algorithm sourceTries(N)
{
    if (N == null) return;
    E' = parentElement(N);
    l = level# of N in O;

    for each element field E of N do
    {
        if (E.data = D_i for some i)
            E.S[0] = S_{i,0};
        else E.S[0] = null;
        for (i = 1; i ≤ l; i++)
            E.S[i] = E.S[0] ∪ E'.S[i − 1];

    }
    sourceTries(N.le.child);
    sourceTries(N.re.child);
}
```

Figure 6: Algorithm to compute 1-bit source tries

5 The Algorithm *sourceTries*

Let N be a node in the 1-bit trie O of distinct destination prefixes in a two-dimensional classifier. Let E be an element field of N. If $E.data = D_i$, then $E.S[j]$ is the source trie for $S_{i,j}$. If $E.data = null$ and N is the root, then $E.S[j]$ is *null* (i.e., the empty source trie). If $E.data = null$ and N is not the root, then $E.S[j]$ is $parentElement(E).S[j-1]$. Note that $parentElement(N).child = N$. Figure 6 gives a recursive algorithm to compute the $S[]$ arrays for all elements of all nodes of the 1-bit destination trie O. The initial invocation is $sourceTries(root(O))$. The correctness of this algorithm follows from the definitions of $S_{i,j}$ and $S[]$.

Note that when there are u distinct dest prefixes, exactly u element fields have a non-null $S[0]$. So, the union between tries done in the second **for** loop needs to be done only for u element fields. At the remaining element fields, we simply copy the $S[i-1]$ values, which are pointers to source-trie roots, from the parent element. Therefore, the number of union operations is $O(uW)$. When the total number of filters in the two-dimensional classifier is n, each $S[j]$ has $O(nW)$ nodes and a union takes $O(nW)$ time. So, the time complexity of *sourceTries* is $O(unW^2) = O(n^2W^2)$. The space complexity is $O(n^2W^2)$ because there are $O(uW) = O(nW)$ source tries and each source trie has $O(nW)$ nodes.

6 Space-Optimal Constrained 2DMTs

When developing a 2DMT for a packet classifier, we begin with a specification of the desired performance (i.e., number of packets to be classified per second) and the memory access rate of the computer to be used for packet classification. By dividing these two numbers, we get the maximum number, k, of memory accesses permissible to classify a single packet. Our task then becomes one of developing a 2DMT T that uses the least amount of memory and has $MNMA(T) \leq k$. Such a 2DMT is called a *space-optimal 2DMT*. In this paper, we do not develop a fast algorithm for space-optimal 2DMTs. Rather, we conider space-optimal constrained 2DMTs. We consider four different contrained 2DMTs. In the first, 2DMTa, we limit ourseleves to 2DMTs in which the destination trie is an FST and the height of each source trie is at most z, where z is a design parameter. Let $2DMTa(y, z)$ be the set of 2DMTas in which the height of the destination trie is y and each source trie has a height at most z. Every T, $T \in 2DMTa(y, z)$, may be searched with at most $y * (z + 1)$ memory accesses (at each level of the destination trie we need to access at most one dest-trie node and search at most one source trie). By requiring $z * (y + 1) \leq k$, we obtain the desired performance.

Our second constrained 2DMT is called 2DMTb. In a 2DMTb the destination trie also is constrained to be an FST. All source tries that hang from elements at the same level l of the dest-trie are contrained by a common height bound $h(l)$. Let P be a root to leaf path in the destination trie of a 2DMTb T. Let $H(P)$ be the sum of the heights of the source tries that hang from the elements on this path. Let $nodes(P)$ be the number of nodes/elements on the path P. In a 2DMTb, $H(P) = \sum_{l=0}^{nodes(P)-1} h(l)$. $T \in 2DMTb(z)$ iff

$$\max_P\{H(P) + nodes(P)\} \leq z \tag{1}$$

Note that every T, $T \in 2DMTb(z)$, can be searched with at most z memory accesses per packet. So, every T in $2DMTb(k)$ provides the desired performance.

In the third variant, 2DMTc, the destination trie is a VST and all source tries that hang from the same destination-trie node obey the same height constraint. A 2DMTc T is in $2DMTc(z)$ iff Equation 1 is satisfied for every root-to-leaf path P in T. As is the case for $2DMTb(z)$s, every T, $T \in 2DMTc(z)$, can be searched with atmost z memory accesses per packet. So, every T in $2DMTc(k)$ provides the desired performance.

In the fourth and final variant, 2DMTd, the destination trie is a VST and no constraint is placed on the source tries that hang off of a node. A 2DMTd T is in $2DMTd(z)$ iff Equation 1 is satisfied for every root-to-leaf path P in T. As is the case for $2DMTb(z)$s and $2DMTd(z)$s, every T, $T \in 2DMTd(z)$, can be searched with atmost z memory accesses per packet. So, every T in $2DMTd(k)$ provides the desired performance.

Let $2DMTa(k)$ be the union of $2DMTa(y, z)$ over all y and z satisfying $y * (z + 1) \leq k$ and let $2DMT(k)$ be the set of 2DMTs that can be searched with at most k memory accesses. Note that even though no constraints are placed on the source tries of a 2DMTd, a space-optimal $2DMTd(k)$ may not

be a space-optimal $2DMT(k)$. This is because, $2DMT(k)$ may contain tries that have root-to-leaf paths P that do not satisfy Equation 1. Of course, every path P that violates Equation 1 must be a path for which no search query is possible.

From the definitions just provided, it follows that:

$$2DMTa(k) \subset 2DMTb(k) \subset 2DMTc(k) \subset 2DMTd(k) \subset 2DMT(k)$$

In the remainder of this section, we develop dynamic programming algorithms to determine space optimal $2DMTa(k)$s, $2DMTb(k)$s, $2DMTc(k)$s and $2DMTd(k)$s. Although, our algorithms explicitly determine only the space/memory required by these space-optimal constrained 2DMTs, these algorithms are easily extended to construct the space-optimal constrained 2DMTs as well. We use the number of element fields in a 2DMT as a measure of its space requirement. Each element field of the space-optimal constrained 2DMT will have two components–source-trie pointer and child pointer in case of an element field of the destination trie and filter data and child pointer in case of an element of the source trie. We assume that in both cases, an element field requires the same number of bytes. In reality, this may not be the case as for each filter in the source trie we will need to store the filter priority and corresponding action. Our algorithms are modified easily to handle the situation when an element field of the source trie requires a different amount of memory than that required by an element of the destination trie.

6.1 2DMTa

We develop a dynamic programming algorithm to determine a space-optimal $2DMTa(k)$. Let $[s, e]$ be the start-end pair associated with the nodes at some level l of a 2DMTa. Let $nodes(s)$ be the number of nodes at level s of the 1-bit trie O of the distinct destination prefixes. The number of element fields at level l of the 2DMTa is $nodes(s) * 2^{e-s+1}$. Since only one copy of each different source trie is actually stored in the 2DMTa, the space required by all the source tries that hang from the level l elements of the 2DMTa may be determined by examining elements E in O for which $Q(E)$ is a dest prefix and $s < L(Q(E)) \leq e+1$. If the length of $Q(E)$ is $L(Q(E))$ and the dominating level of E is greater than e, then $E.S[L(Q(E))-s-1]$ hangs from one or more of the level l elements of the 2DMTa. Let $sourceSum(s, e, z)$ be the total number of elements in the optimal VSTs of all of these source tries; these VSTs are constrained to have at most z levels each. Let $f(s, e, z) = nodes(s) * 2^{e-s+1} + sourceSum(s, e, z)$. If level l of a 2DMTa has the start-end pair $[s, e]$, then $f(s, e, z)$ gives the minimum number of elements at level l of the 2DMTa (this count includes elements in the source tries that hang from the level l dest-trie elements). For $s > e$, define $f(s, e, z) = 0$.

Algorithm $f2DMTa$ (Figure 7) computes $f(s, e, z)$, $0 \leq s \leq e < W$, $z \leq Z$. In this algorithm $opt(S, Z)$ is the algorithm of Sahni and Kim [23] to compute the memory requirements of the space-optimal VSTs for the 1-bit trie S, the VSTs are constrained to have at most 1, 2, $\cdots$, Z levels.

The number of different source tries for which $opt(S, Z)$ is invoked equals the number, $O(uW) = O(nW)$, of union operations done by Algorithm $sourceTries$ (Figure 6). Each invocation of $opt(S, Z)$ takes $O(nW^2Z)$ time [23]. So, the time for all invocations of opt is $O(n^2W^3Z)$. The remainder of Algorithm $f2DMTa$ takes $O(nW^2Z)$ time. So, the overall complexity of this algorithm is $O(n^2W^3Z)$. Note that if we use optimal FSTs (rather than VSTs) as our source tries, the time for all invocations of opt as well as the overall time drops to $O(nW^3Z)$.

Let $Opt1(j, y, z)$ be the minimum number of elements in a space-optimal $2DMTa(y, z)$ that covers only levels 0 through j of O. Note that the destination trie of this 2DMTa is constrained to have at most y levels and each of its source tries is constrained to have at most z levels. Note that $Opt1(W-1, y, z)$ gives the minimum number of elements in a $2DMTa(y, z)$ for the given classifier.

```
Algorithm f2DMTa(Z)
{
    // O is the 1-bit dest-trie for the classifier.
    dominatingLevel(root(O));
    sourceTries(root(O));
    for each different 1-bit src-trie S computed by sourceTries
       run opt(S, Z);
    for (s = 0; s < W; s++)
       for (e = s; e < W; e++)
          for (y = 1; y < Z; y++){
             f(s, e, y) = nodes(s) * 2^(e-s+1);
             for each element E such that Q(E) is a
               destination prefix && s < L(Q(E)) ≤ e + 1 && E.dl > e {
                 S = E.S[L(Q(E)) - s - 1];
                 f(x, y, Y) += opt(S, y);
               }
          }
}
```

Figure 7: Algorithm to compute $f(s, e, z)$

When $y = 1$, there is at most one level. The start-end pair for this level is $[0, j]$ and the minimum number of elements is $f(0, j, z)$. When $y > 1$, we may or may not have more than 1 level. If we have only 1 level, $Opt1(j, y, z) = f(0, j, z)$. If we have more than 1 level, then let $[i + 1, j]$, $i < j$, be the start-end pair for the last level. The number of elements in the last level of the space-optimal $2DMTa(y, z)$ is $f(i + 1, j, z)$. The number in the remaining levels is $Opt1(i, y - 1, z)$.

The preceding discussion leads to the following dynamic programming recurrence for $Opt1$.

$$Opt1(j, y, z) = \min_{0 \le i \le j} \{Opt1(i, y - 1, z) + f(i + 1, j, z)\} \tag{2}$$

$$Opt1(j, 1, z) = f(0, j, z) \tag{3}$$

Using these equations, we may compute all $Opt1(j, y, z)$, $0 \le j < W$, $1 \le y \le k$, $1 \le z \le k$ values in $O(W^2k^2)$ time.

The number of elements in the space-optimal $2DMTa(k)$ is given by

$$\min_{1 \le z < k} \{Opt1(W - 1, \frac{k}{z + 1}, z)\}$$

So, once the $Opt1$ values have been computed the number of elements in the space-optimal $2DMTa(k)$ may be determined expending an additional $O(k)$ time. The overall complexity of our algorithm to compute the number of elements in a space-optimal $2DMTa(k)$ is $O(n^2W^3k + W^2k^2) = O(n^2W^3k)$ (under the assumption $n > \max\{k, W\}$). Its space complexity is $O(n^2W^2)$ (the space complexity is determined by the space requirement of *sourceTries*). Notice that when the source tries are constrained to be optimal FSTs (rather than VSTs) the overall complexity becomes $O(nW^3k + n^2W^2 + W^2k^2) = O(nW^3k + n^2W^2)$.

6.2 2DMTb

In a 2DMTb the destination trie is an FST and all source tries that hang from the same level of the destination trie obey a common height bound. The development of the dynamic programming algorithm for space-optimal $2DMTbs(k)$ has much in common with that for $2DMTa(k)$s. Let $f(s,e,z)$ be as in Section 6.1. Let $Opt2(j,z)$ be the minimum number of elements in a space-optimal $2DMTb(z)$ that covers only levels 0 through j of O.

When the destination trie has only 1 level, $Opt2(j,z) = f(0,j,z)$. When the destination trie has more than 1 level, the start-end pair for the last level is $[i+1,j]$ for some i in the range $[0,j-1]$. If the source tries hanging from the last level of the destination trie have at most y levels, the last level contributes $y+1$ to the value of $H(P)+$ number of nodes on P for some path P. So, we get the following recurrence for $Opt2$.

$$Opt2(j,z) = \min\{f(0,j,z) + X(j,z)\} \tag{4}$$

where

$$X(j,z) = \min_{0 \le i < j, 0 \le y < z} \{Opt2(i, z-y-1) + f(i+1,j,y)\} \tag{5}$$

Notice that this recurrence yields $Opt2(0,z) = f(0,0,z)$. Also, $Opt2(W-1,k)$ gives the number of elements in the space-optimal $2DMTb(k)$. The time required to compute $Opt2(W-1,k)$ is dominated by the time needed to compute $f(s,e,z)$, $0 \le s \le e < W$, $1 \le z < k$. This time is $O(n^2W^3k)$ when the source tries are VSTs and $O(nW^3k + W^2k^2)$ when the source tries are FSTs. The space complexity is $O(n^2W^2)$.

6.3 2DMTc

Let N be a level l node in the 1-bit trie O of distinct destination prefixes. Consider the subtree $ST(N)$ of O rooted at N. If levels 0 through q of this subtree are compacted into a single node M of a 2DMTc, the stride of this compacted node is $q+1$ and this compacted node has 2^{q+1} element fields. Since only one copy of each different source trie is actually stored in the 2DMTc, the space required by all the source tries that hang from the compacted node may be determined by examining elements E in $ST(N)$ for which $Q(E)$ is a destination prefix and $l < L(Q(E)) \le l+q+1$; $E.S[L(Q(E)) - l - 1]$ hangs from one or more of the elements of the M iff the dominating level of E is greater than $l+q$. Let $sourceT(N,q)$ denote the set of these source tries. Let $sourceSum(N,q,z)$ be the total number of elements in the optimal VSTs of all of these source tries; these VSTs are constrained to have at most z levels each. Let $g(N,q,z) = 2^{q+1} + sourceSum(N,q,z)$. $g(N,q,z)$ gives the minimum number of elements in a 1-level $2DMTc(z+1)$ for levels 0 through q of $ST(N)$ (this count includes elements in the source tries that hang from the single node of the dest-trie of the 2DMT).

Let $Opt3(N,z)$ be the minimum number of elements in a $2DMTc(z)$ for $ST(N)$. $Opt3(root(O),k)$ is the number of elements in the space-optimal $2DMTc(k)$ for the classifier. Let $D_t(N)$ denote the descendents of N that are at level t of $ST(N)$. Let $Opt3(N,t,z)$ be $\sum_{R \in D_t(N)} Opt3(R,z)$ when $t > 0$ and $Opt3(N,z)$ when $t = 0$.

It is easy to see that for $t > 0$

$$\begin{aligned} Opt3(N,t,z) &= \sum_{R \in D_{t(N)}} Opt3(R,z) \\ &= Opt3(N.le.child, t-1, z) + Opt3(N.re.child, t-1, z) \end{aligned} \tag{6}$$

$$Opt3(null, t, z) = 0 \tag{7}$$

Next consider the case $t = 0$. If the destination trie of the space-optimal $2DMTc(z)$ for $ST(N)$ has just 1 level, it has $g(N, height(ST(N)) - 1, z - 1)$ elements. If this destination trie has more than 1 level, then the root of the destination trie covers levels 0 through q of $ST(N)$ for some q in the range $[0, height(ST(N)) - 2]$, the source tries hanging from the dest-trie root can be searched with at most y memory accesses, $0 \leq y < z-1$, and the child fields of the root point to space-optimal $2DMTc(z-y-1)$s (the -1 accounts for the root of the dest-tries of the $2DMTc(z)$ of $ST(N)$). for the subtrees rooted at $D_{q+1}(ST(N))$.

The preceding discussion leads to the following dynamic programming recurrence for $Opt3(N, 0, z)$, $N \neq null$.

$$Opt3(N, 0, z) = \min\{g(N, height(ST(N)) - 1, z - 1), X(N, 0, z))\} \tag{8}$$

where

$$X(N, 0, z) = \min_{0 \leq q \leq height(ST(N))-2, 0 \leq y < z-1} \{Opt3(N, q + 1, z - y - 1) + g(N, q, y)\} \tag{9}$$

Using a strategy similar to that used to compute f in Section 6.1, we can compute the needed g values in $O(n^2W^3k)$ time when the source tries are VSTs and in $O(nW^3k)$ time when the source tries are FSTs. There are $O(nW^2k)$ $Opt3$ values to compute using Equation 6. Each of these is computed in $O(1)$ time (exclusive of the time required to compute the needed $Opt3(*, 0, *)$ values), for a total time of $O(nW^2k)$. We also have $O(nWk)$ $Opt3$ values to compute using Equation 9. Each of these takes $O(Wk)$ time. So, the total time needed to compute these $O(nWk)$ $Opt3$ values is $O(nW^2k^2)$. Hence the time neded to determine the number of elements, $Opt3(root(O), k) = Opt3(root(O), 0, k)$, in the space-optimal $2DMTc(k)$ for the classifier is $O(n^2W^3k + nW^2k^2) = O(n^2W^3k)$ when the source tries are VSTs and $O(nW^3k + nW^2k^2)$ when the source tries are FSTs. The space complexity is $O(n^2W^2)$.

6.4 2DMTd

We use the same notation as used in Section 6.3 except that M now is a node of the 2DMTd. The following additional notation is used:

$sA(N, q) \cdots$ This denotes the subset of $sourceT(N, q)$ that hang from elements of M that have a non-null child.

$sB(N, q) \cdots$ This is $sourceT(N, q) - sA(N, q)$.

$A_{q+1}(N, s) \cdots$ This set comprises all nodes $R \in D_{q+1}(N)$ for which there is an element E in M with the properties (a) $E.data = s$ and (b) $E.child$ is the 2DMTd for $ST(R)$.

$D'_{q+1}(N) \cdots$ This is $D_{q+1}(N) - A_{q+1}(N)$.

Let $Opt4(N, z)$ be the minimum number of elements in a 2DMTd for $ST(N)$, this 2DMTd may be searched with at most z memory accesses. Let M be the root of this 2DMTd and assume that M covers levels 0 through q of $ST(N)$. We see that the number of elements in every $s \in sB(N, q)$ is $opt(s, z - 1)$, where $opt(s, z - 1)$ is the minimum number of elements in any VST for s that can be searched with at most $z - 1$ memory accesses. For $s \in sA(N, q)$, if the VST for s requires i memory accesses then the 2DMTds that are the children of M must each be searchable with at most $z - i - 1$ memory accesses. (It

is easy to see that in a 2DMTd there is only one VST for each $s \in sA(N,q)$.) Thus we are motiviated to define the function $h(s,N,q,z-1)$ to be $opt(s,z-1)$ for $s \in sB(N,q)$ and

$$\min_{1 \le i < z} \{opt(s,i) + \sum_{R \in A_{q+1}(N,s)} Opt4(R, z-i-1)\}$$

for $s \in sA(N,q)$. The following dynamic programming recurrence for $Opt4(N,z)$ now follows:

$$\begin{aligned} Opt4(N,z) &= \min_{0 \le q < height(ST(N))} \{2^{q+1} + \sum_{s \in sourceT(N,q)} h(s,N,q,z-1) \\ &\quad + \sum_{R \in D'_{q+1}(N)} Opt4(R, z-1)\} \end{aligned} \tag{10}$$

To compute $Opt4(root(O)),k)$, we must compute $O(nWk)$ $Opt4(.,.)$ values. It takes $O(nW)$ time to compute each $Opt4(.,.)$ value exclusive of the time needed to compute the $opt(.,.)$ and $h(.,.,.,.)$ values. So, the total time needed is $O(n^2W^2k)$ plus the time for the opt and h values. For each s, it takes $O(nW^2k)$ time [23, 18] to compute $opt(s,*)$. Since there are $O(nW)$ s values, all $opt(*,*)$ values may be computed in $O(n^2W^3k)$ time. For $h()$, we see that there are $O(nW)$ s values, $O(nW)$ N values, $O(W)$ q values and $O(k)$ z values. However, for any triple (N,q,z), only $O(n)$ s values are possible. So, the total number of $h()$ values that are computed is $O(n^2W^2k)$. To compute a single $h(s,N,q,z)$ value, we need to take the min of z items and we need to sum some number of $Opt4()$ values. So, the time spent taking the mins for all $O(n^2W^2k)$ h values is $O(n^2W^2k^2)$. For any triple (N,q,z) at most all nodes in $D_{q+1}(N)$ are involved in all the summations for all the s values; no node is involved in the summations for 2 s values. So, all $h(*,N,q,z)$ values may be computed with $O(|D_{q+1}(N)|$ summations. Hence, all $h(*,N,*,z)$ values require $O(|ST(N)|)$ summations. Since each node of the top-level tries is in $O(W)$ subtrees, the number of summations required for all $h(*,*,*,z)$ computations is $O(nW^2)$. Finally, as there are $O(k)$ z values, the total number of summations is $O(nW^2k)$. So, the total time spent performing these $Opt4$ summations is $O(nW^2k)$. Adding in the time required for the remaining tasks (e.g., *dominatingLevel* and *sourceTries*), we get $O(n^2W^3k + n^2W^2k^2)$ as the overall time needed to determine $Opt4(root(O),k)$. In typical applications, we expect $W > k$ and so the complexity for these applications is $O(n^2W^3k)$. Its space complexity is $O(n^2W^2)$. Note that the h values may be computed as needed and discarded once used. Hence, we need no additional space to store h values. If we limit the source tries to be FSTs, the time complexity drops to $O(nW^3k + n^2W^2k^2) = O(n^2W^2k^2)$.

6.5 Postprocessing

The number of elements in a space-optimal $2DMTa(k)$, $2DMTb(k)$, and $2DMTc(k)$ may be reduced without increasing the worst-case number of memory cases for a search beyond k via a postprocessing step. Let T be a space-optimal 2DMT constructed using one of the algorithms of Sections 6.1–6.3. If a source trie of T is on no search path P with

$$H(P) + nodes(P) = k$$

then this source trie may be replaced with an optimal source trie of larger height. This replacement, of course, is done only if the larger-height source trie has a smaller number of elements.

7 2DMTs With Switch Pointers

The number of memory accesses required to search a 2D1BT may be reduced by the use of switch pointers[25, 3]. Switch pointers also may be used with 2DMTs to reduce the number of memory accesses

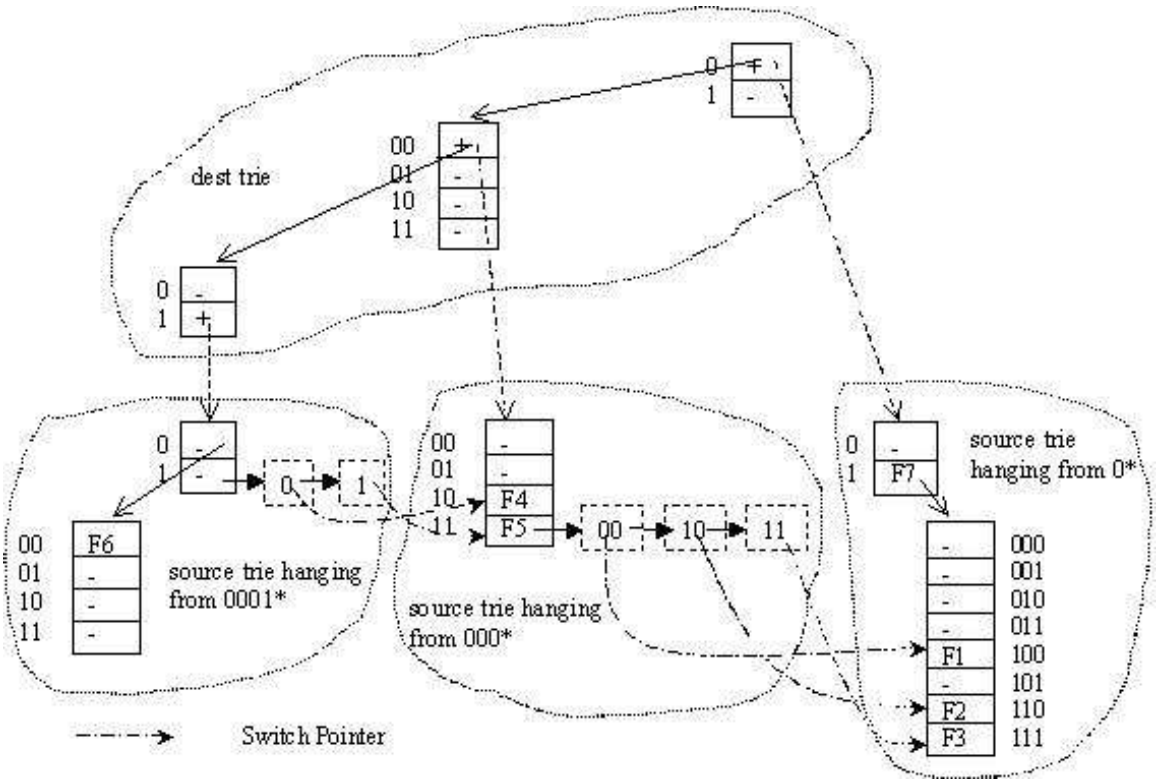


Figure 8: 2DMT of Figure 4 augmented with switch pointers

during the search for a least-cost matching filter. Let E be an element field in the dest-trie of a 2DMT T and let $T' = E.data$ be the source trie hanging from E. Assume that $T' \neq null$. Let E' be an element field in T' such that $E'.child = null$. Let E'' be the nearest ancestor of E such that the source trie $E''.data$ has an element field E''' such that $Q(E')$ is a proper prefix of $Q(E''')$ and $E'''.data \neq null$. If there is no such E'' the switch pointer list for E' is $null$. Assume that such an E'' exists and let F be an E''' in $E''.data$ such that $L(Q(E'''))$ is least. Let N be the node of $E''.data$ that contains the element field F. The node E' maintains a list of pointers to all element fields G of N such that $Q(E')$ is a (proper) prefix of $Q(G)$ and either the data or child (or both) field of G is not $null$. The nodes on this pointer list, which is called the *switch-pointer list*, are indexed by bit sequeneces b such that $Q(E')||b = Q(G)$. Figure 8 shows the 2DMT of Figure 4 augmented with switch-pointer lists.

In a 2DMT with switch pointers (2DMTS), source tries are augmented with switch-pointer lists. Additionally, every element field E' in every source trie stores the least-cost filter (D,S) that matches the destination prefix $Q(E)$ and the source prefix $Q(E')$. Here E is the element field from which the source trie with E' hangs. Note that $L(D) \leq L(Q(E))$ and $L(S) \leq L(Q(E'))$. To search a 2DMTS for the least-cost filter that matches the destination- source-address pair da, sa), we first search for da in the dest-trie of the 2DMTS. This search terminates at the element E such that $Q(E)$ is a destination prefix (so, $Q.data \neq null$) and $Q(E)$ is the longest destination-prefix that matches da. We then move into the source trie $E.data$ and search for the longest source-prefix in this source trie that matches sa. We use a switch pointer in the element at which this search terminates to move to an element in an ancestor source-trie from where the search for sa continues and so on. This continues until a $null$ switch pointer is reached.

As an example, consider the case (da, sa) = (00010, 11101) and the 2DMTS of Figure 8. We first search the dest-trie for 00010. This search stops at the element with $Q()$ = 0001. We continue by searching for 11101 in the source trie hanging from 0001. This search terminates at and stops at the element with $Q()$ = 1. This element has both the least-cost filter that matches (0001,1) and a list of switch pointers. Since the switch pointers on this list are indexed with 1-bit indexes, we use the next bit of sa to move into an ancestor source trie. The search continues at the element field with $Q()$ = 11 of the source trie hanging from the destination-trie element with $Q()$ = 000. The search for sa = 11101 terminates at this element. This element stores the least-cost filter that matches (000,11) as well as a list of switch pointers. Since these switch pointers use 2-bit indexes, we use the next two bits of sa (i.e., 10) to move to an element in an ancestor source trie. We get to the element with $Q()$ = 1110 in the source trie hanging from the root of the destination trie. When we search from here for sa, the search terminates at this element. Since this element has no switch pointer, we are done. By examining the least-cost filters encountered on this search path, we may determine the least-cost filter that matches (da, sa).

When implementing a 2DMTS, we do not actually store switch-pointer lists explicitly. This is because of the excessive space required by these lists and because of the excessive number of memory accesses required to search these lists for the proper switch pointer to follow. In a typical 2DMTS implementation, elements of a node are stored sequentially in an array. So, if a source-trie element stores the first switch pointer together with the number of bits in a switch-pointer index, we can compute the location of the desired element to switch to.

Although we may convert the space-optimal $2DMTa(k)$, $2DMTb(k)$, $2DMTd(k)$ and $2DMTd(k)$ tries obtained using the algorithms of Sections 6.1–6.4 into 2DMTSs and improve search performance, there appears to be no easy way to modify the algorithms of Sections 6.1–6.4 to construct space-optimal constrained 2DMTSs that may be searched with at most k memory accesses.

Let T be an l-level FST. Let s_i be the stride for nodes at level i of T. $(s_0, \cdots, s_l)$ is the *stride sequence* for T. A set of FSTs is said to be *uniform* iff all FTSs in the set have the same stride sequence. For purposes of this definition we regard two sequences σ and τ the same even when σ is a proper prefix of τ. Let $2DMTSa(k)$ be a 2DMTS in which the destination trie is an FST or a VST and all source tries define a uniform set of FSTs. Figure 9 shows a 2DMTSa for the data of Table 1. In this 2DMTSa, the stride sequence for the source tries is 2, 1, 1. When a uniform set of source tries is used, the switch pointer scheme is simplified as shown in Figure 9. It is easy to see that the number of memory accesses needed to search a 2DMTSa is at most (height of destination trie + maximum height of a source trie). So, the 2DMTSa of Figure 9 is a $2DMTSa(6)$.

Figure 10 gives an algorithm to determine an approximation to the minimum number of elements in a space-optimal $2DMTSa(k)$. This algorithm essentially tries all combinations of x and $k - x$, where x is the height of the destination VST and $k - x$ is the maximum height of the source FSTs in a uniform set of FSTs. The VSTs used for the destination trie are required to be space-optimal VSTs. $opt()$ refers to the VST algorithm of [23]. Once this VST algorithm is run from the statement outside the **for** loop, we have the number of elements in the optimal height l VST, $1 \leq l < k$, for the destination prefixes of our classifier. So, when we use $opt(O, x)$ within the **for** loop the needed values are known and accessed in $O(1)$ time each. Hence, the time for all invocations of opt is $O(nW^2k)$. $sourceElements(S, k - x)$ computes the minimum number of elements in any uniform set of FSTs for the 1-bit source tries of S. This algorithm is simply the FST algorithm of [23] started with $nodes(i)$ being the total number of nodes on level i of all tries of S, $0 \leq i < W$. Since each invocation of $sourceElements$ takes $O(nW + W^2k)$ time, the time spent on all invocations is $O(nWk + W^2k^2)$. Adding in the time for $dominatingLevel$, $sourceTries$ and opt, we get $O(n^2W^2 + nW^2k + W^2k^2) = O(n^2W^2)$ (assuming $n > k$) as the overall complexity of Algorithm $2DMTSa$. When the destination trie is an FST, the complexity of $opt(O, x)$ is

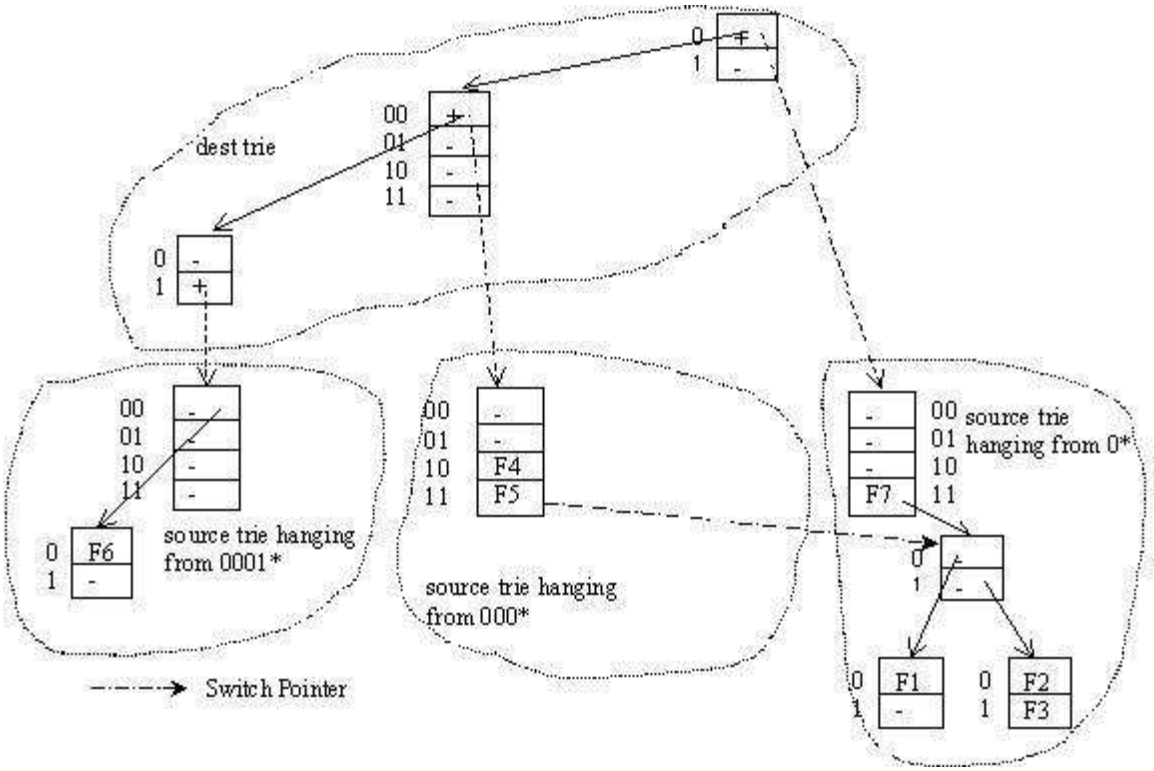


Figure 9: A 2DMTSa for the filters of Table 1

Algorithm $2DMTSa(k)$
{
// Let O be the 1-bit dest-trie for the classifier.
$dominatingLevel(root(O))$;
$sourceTries(root(O))$;
run $opt(O, k-1)$;
$minElements = \infty$;
for $(x = 1; x < k; x++)$ {
$destElements = opt(O, x)$;
Let S be the set of 1-bit src-tries hanging from
the optimal VST for O with at most x levels.
$minElements = \min(minElements, destElements + sourceElements(S, k-x))$;
}
return $minElements$;
}

Figure 10: Heuristic to determine a good $2DMTSa(k)$.

$O(Wk)$ and the total complexity is $O(n^2W^2)$.

8 Experimental Results

We programmed our algorithms in C++ and compiled all codes using Microsoft Visual C++ 6.0 with optimization level `O2`. The compiled codes were run on a 3.06 GHz Pentium 4 PC. Since there are no publicly available destination-source databases, we used the strategy of [15] to generate destination-source database from a publically available database of destination prefixes. We started from the Paix database (85987 prefixes) obtained from the IPMA project on Sep 13, 2000 and randomly selected a set of m prefixes. These prefixes became the destination prefixes for our destination-source database. Each of these destination prefixes was paired with 20 source prefixes that were randomly chosen from Paix. The result was a database with $20m$ destination-source filters. This database generation process was done 10 times for each of the m values 50, 100, 250, 500 and 1000. The constructed 50 databases were grouped into 5 data sets, DS1 through DS5, by their m value. DS1 is the set of 10 databases with $m = 50$ and DS5 is that with $m = 1000$. Table 2 gives the characteristics of our 5 data sets as well as of the corresponding 2D1BTs. For example, each of the 10 databases of DS2 have 2000 filters. The number of elements in these 10 databases ranges from 61,082 to 62,040 and the average is 61,429. When switch pointers are not used, the maximum number of memory accesses (MNMA) needed to do a search ranges from 48 to 61 and the average is 50. When switch pointers are used, the MNMA range is 48 to 52 and the average is 49.

Data Set	#Filters	Corresponding 2D1BTs					
		#Elements		MNMA(NoSW)		MNMA(SW)	
		Range	AVG	Range	AVG	Range	AVG
DS1	1000	30508-31140	30839	48-52	48	48-52	48
DS2	2000	61082-62040	61429	48-61	50	48-52	49
DS3	5000	151832-153054	152373	48-67	59	48-52	49
DS4	10000	302382-304732	303552	52-68	63	49-53	51
DS5	20000	602434-606014	604390	65-73	69	50-53	52

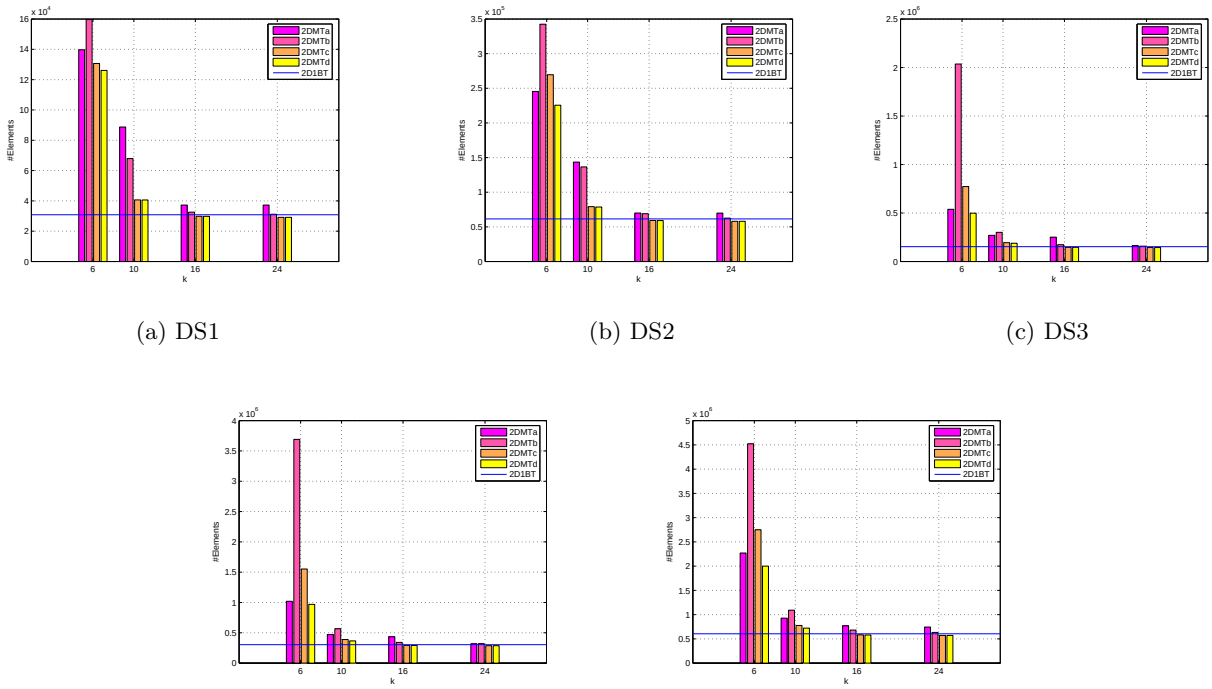
Table 2: Synthetic Data Sets

For the postprocessing step (Section 6.5), we examined each source trie T in the constructed two-dimensional trie. Let E be the element of the destination trie from which T hangs. If $E.child \neq null$, T was not changed. If $E.child = null$, T was redetermined permitting its height to be as much as $k - H(P) - nodes(P)$, where P is the path from the root of the destination trie to E and k, $H(P)$ and $nodes(P)$ are as defined in Section 6.

Postprocessing was very effective on 2DMTa and 2DMTb and had almost no impact on 2DMTc. For 2DMTas with VST source tries, postprocessing reduced the number of elements between 12% and 95% while on 2DMTbs with VST source tries, this reduction was between 0% and 46%. For FST source tries, the reduction in number of elements ranged from 15% to 96% for 2DMTa and from 0% to 50% for 2DMTb. In the case of VST source tries postprocessing increased the total run time by up to 2% and for FST source tries, postprocessing increased the run time by up to 100%. There was significant variation in the effectiveness of postprocessing within each of our 5 data sets.

Figures 11 and reffig:MemCompare2 give the average number of elements in the space-optimal constrained 2DMTs for our 5 data sets. For each data set the average number of elements for the 10 databases in that dataset is histogrammed. Recall that the number of elements is a measure of the memory requirement of a 2DMT. For 2DMTa, 2DMTb and 2DMTc only the data with postprocessing included is histogrammed. The horizontal line in Figures 11 and 12 gives the average number of elements

in the 2D1BTs for the data set. The variance in the measured data within each data set was less than 1% for $k \geq 16$.



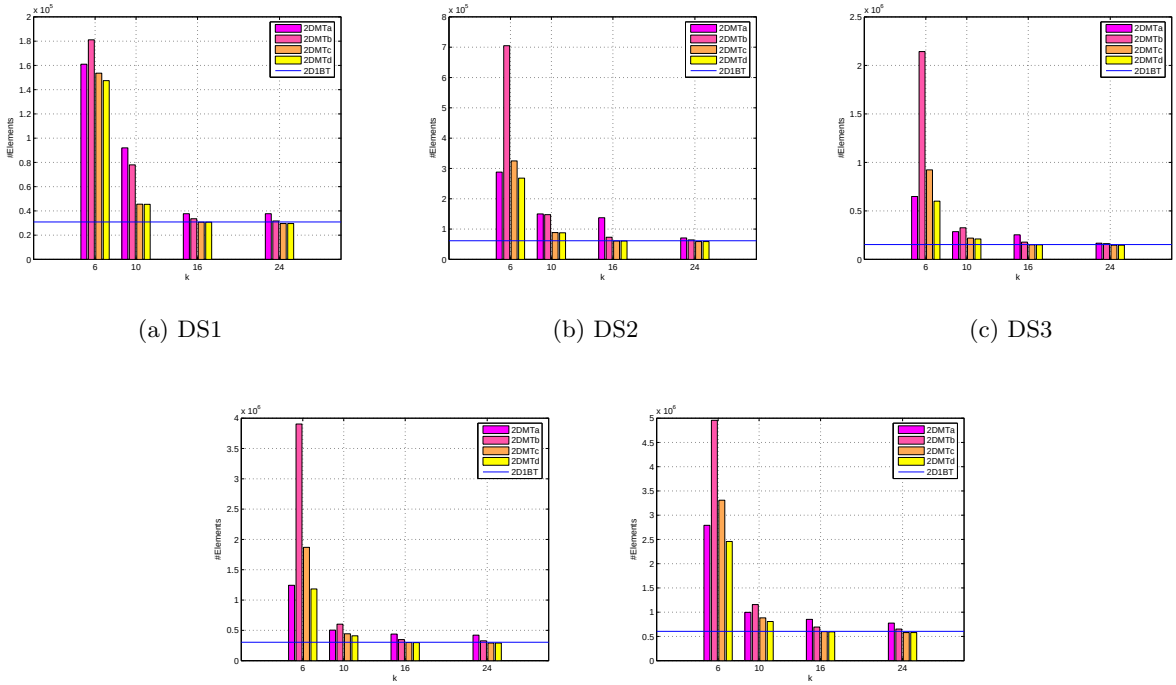
(a) DS1 (b) DS2 (c) DS3

(d) DS4 (e) DS5

Figure 11: Number of elements in space-optimal constrained 2DMTs with postprocessing and 2D1BTs. Source tries are VSTs.

Although prior to postprocessing space-optimal 2DMTas have significantly more elements than space-optimal 2DMTbs, which in turn have significantly more elements than space-optimal 2DMTcs, following postprocessing the difference was considerably less. In fact, following postprocessing, 2DMTas were, at times, superior to 2DMTbs and 2DMTcs for small values of k (e.g., $k = 6$ and 10). For large k (say $k \geq 16$), space-optimal 2DMTcs required almost the same number of elements as required by 2DMTds. Interestingly, for $k \geq 16$, the number of elements in space-optimal 2DMTcs with VST source tries is almost the same as in space-optimal 2DMTcs with FST source-tries. Even more interesting is the observation that for $k \geq 16$, space-optimal 2DMTcs and 2DMTds (with either VST or FST source tries) have about the same number of elements as in 2D1BTs. However, the memory accesses needed to search the 2D1BTs is between 3 and 4.5 times that needed to search a space-optimal 2DMTc(16) or 2DMTd(16)! So, for larger k, multibit tries significantly reduce the number of memory accesses while incurring no memory penalty!

Figures 13 and reffig:TimeCompare2 give the average execution times for each of our dynamic programming algorithms to determine the strides of space-optimal constrained 2DMTs. As can be seen, it takes longer to compute the strides for space-optimal 2DMTds than for space-optimal 2DMTcs. Further, the run time for 2DMTcs is slightly more than that for 2DMTbs and that for 2DMTBs is slightly more than that for 2DMTcs. There is, however, a significant difference in run time between using VSTs and FSTs as source tries. On some of our data sets (e.g., 2DMTb DS1) the VST version took more than 50



(a) DS1 (b) DS2 (c) DS3

(d) DS4 (e) DS5

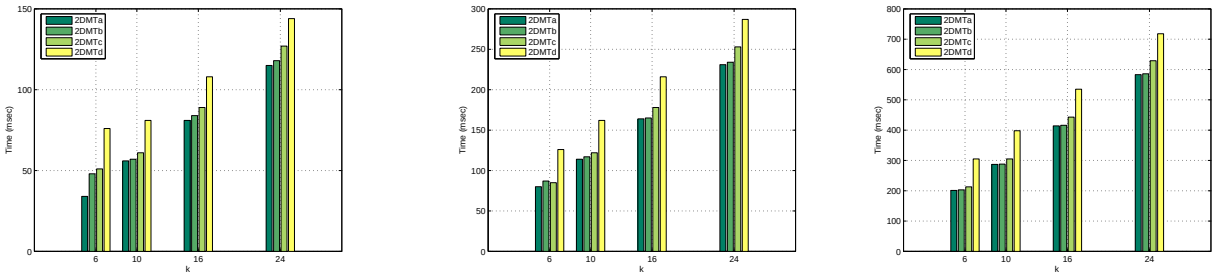
Figure 12: Number of elements in space-optimal constrained 2DMTs with postprocessing and 2D1BTs. Source tries are FSTs.

times the time taken by the FST version. Even for 2DMTds, the VST version often took more than 4 times the time taken by the FST version. Given the closeness of the memory requirements of 2DMTds with VST and FST source tries when $k \geq 16$, one may well opt to save run time by using the FST version. A further reduction in run time by a factor of about 3 is possible by using 2DMTcs with FST source tries rather than 2DMTds with FST source tries; the memory penalty is negligible.

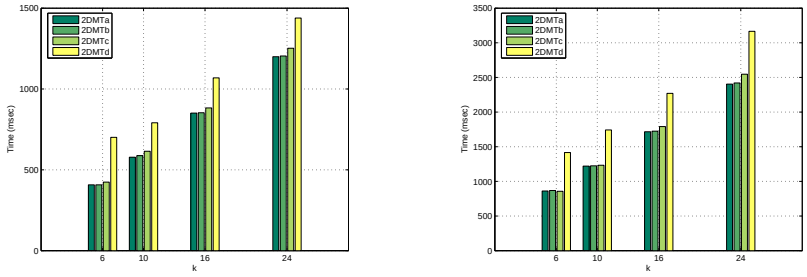
The run time data of Figures 13 and reffig:TimeCompare2 give only the time to compute the strides. To actually construct a two-dimensional multibit trie from this data takes additional time. For 2DMTds this construction time is much less than the time needed to determine the strides themselves. However, for 2DMTcs, this construction time is comparable to the time needed to determine strides when the source tries are FSTs and $k \approx 16$.

Next, we experimented with our heuristics for 2DMTs with switch pointers. We found that for small k (say $k = 6$), $2DMTSa(k)$s with FST destination tries have about about 10% more elements than $2DMTSa(k)$s with VST destination tries. For large k (say $k = 24$), this difference dropped to less than 1%. However, there is significant difference in the time needed to determine the strides between the two cases. The VST version took between 2 and 7 times the time taken by the FST version. Although the time to construct the 2DMT with switch pointers using the computed strides was generally smaller for the case when the destination trie is an FST, the difference isn't very significant.

Figures 15 and 16 compare the number of elements in the constructed $2DMTSa(k)$ with VST destination trie with that in $2DMTc(k)$s and $2DMTd(k)$s with VST source tries as well as the time needed to determine the strides. As can be seen, for large k, the number of elements in $2DMTSa(k)$ is slightly



(a) DS1 (b) DS2 (c) DS3



(d) DS4 (e) DS5

Figure 13: Time to determine strides required by space-optimal constrained 2DMTs with postprocessing. Source tries are VSTs.

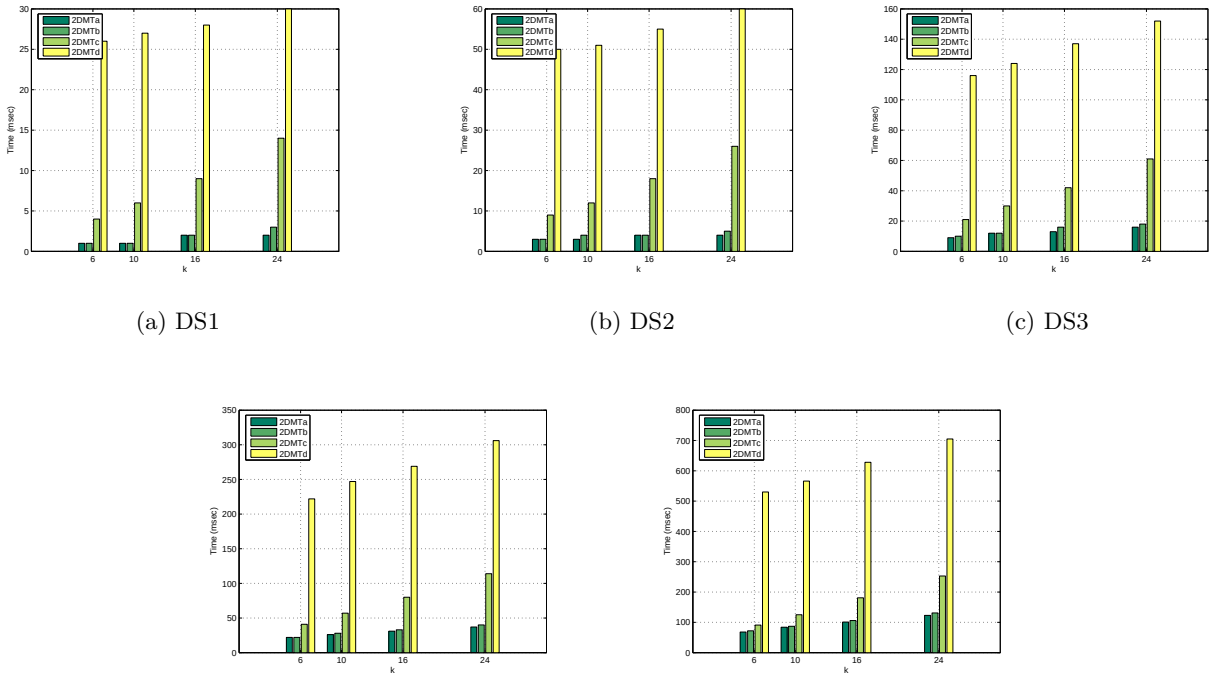
larger (about 2%) than in corresponding space-optimal $2DMTc(k)$s and $2DMTd(k)$s. For small k, the constructed $2DMTSa(k)$ with VST destination trie has up to 120% more elements than in the corresponding space-optimal $2DMTc(k)$ or $2DMTd(k)$. Since for any given k, the MNMA is the same regardless of whether we use a $2DMTc(k)$, $2DMTd(k)$ or $2DMTSa(k)$, there is no advantage (from the standpoint of improved search time or reduction in memory required) to using switch pointers unless we are able to develop a better heuristic to construct a 2DMT with switch pointers.

Although the time required to determine stride data is much less for $2DMTSa(k)$s than it is for $2DMTc(k)$s and $2DMTd(k)$s, the time to actually construct the 2DMT from this stride data is about the same whether or not switch pointers are used.

9 Conclusion

We have developed dynamic programming algorithms to construct space-optimal constrained two-dimensional multibit tries. Using a bucketing scheme, the constructed tries may be used for multidimensional packet classification even when filters have more than two dimensions. Our experiments indicate that space-optimal multibit two-dimensional tries may be searched with between 1/4 and 1/3 of the memory accesses required to search two-dimensional 1-bit tries when both varieties of tries are provided the same amount of memory.

We have proposed a heuristic to construct two-dimensional multibit tries with switch pointers. These tries may be used only with two-dimensional filters. However, even here, the constructed tries are inferior



(a) DS1 (b) DS2 (c) DS3

(d) DS4 (e) DS5

Figure 14: Time to determine strides required by space-optimal constrained 2DMTs with postprocessing. Source tries are FSTs.

to space-optimal two-dimensional tries without switch pointers as far as memory requirements go. The construction time (including the time to determine strides) is, however, smaller using our heuristic.

References

[1] F.Baboescu and G.Varghese, Scalable packet classification, *ACM SIGCOMM*, 2001.

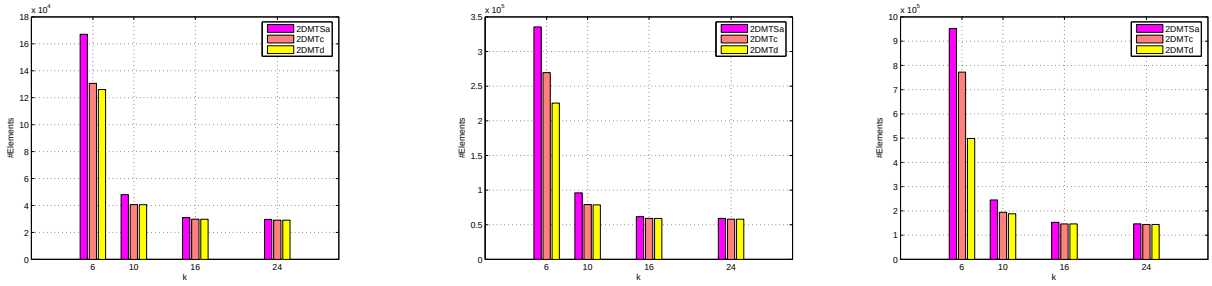
[2] F.Baboescu and G.Varghese, Fast and Scalable Conflict Detection for Packet Classifiers, *10th IEEE International Conference on Network Protocols (ICNP'02)*,2002.

[3] F.Baboescu, S.Singh and G.Varghese, Packet Classification for Core Routers: Is there an alternative to CAMs? *INFOCOM*, 2003.

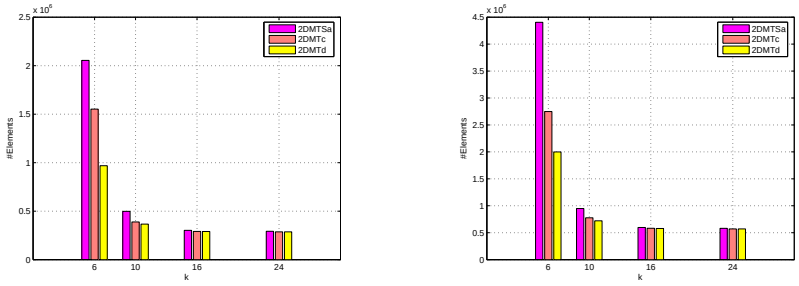
[4] J.L.Bentley and T.A.Ottmann, Algorithms for Reporting and Counting Geometric Intersections, *IEEE Transactions on Computers*, C-28, 9, 1979, 643-647.

[5] M. Buddhikot, S. Suri and M. Waldvogel, Space decomposition techniques for fast layer-4 switching, *Conference on High Speed Networks*, 1998.

[6] D. Eppstein and S. Muthukrishnan, Internet packet filter management and rectangle geometry, *12th ACM-SIAM Symp. on Discrete Algorithms*, 2001, 827-835.



(a) VST-DS1 (b) VST-DS2 (c) VST-DS3



(d) VST-DS4 (e) VST-DS5

Figure 15: Number of elements in $2DMTc(k)$s, $2DMTd(k)$s and $2DMTSa(k)$s

[7] A. Feldman and S. Muthukrishnan, Tradeoffs for packet classification, *INFOCOM*, 2000.

[8] P. Gupta and N. McKeown, Packet classification using hierarchical intelligent cuts, *ACM SIGCOMM*, 1999.

[9] A.Hari, S.Suri, G.Parulkar, Detecting and resolving packet filter conflicts, *INFOCOM*, 2000.

[10] E.Horowitz, S.Sahni, and S.Rajasekeran, Computer Algorithms/C++, W. H. Freeman, NY, 1997.

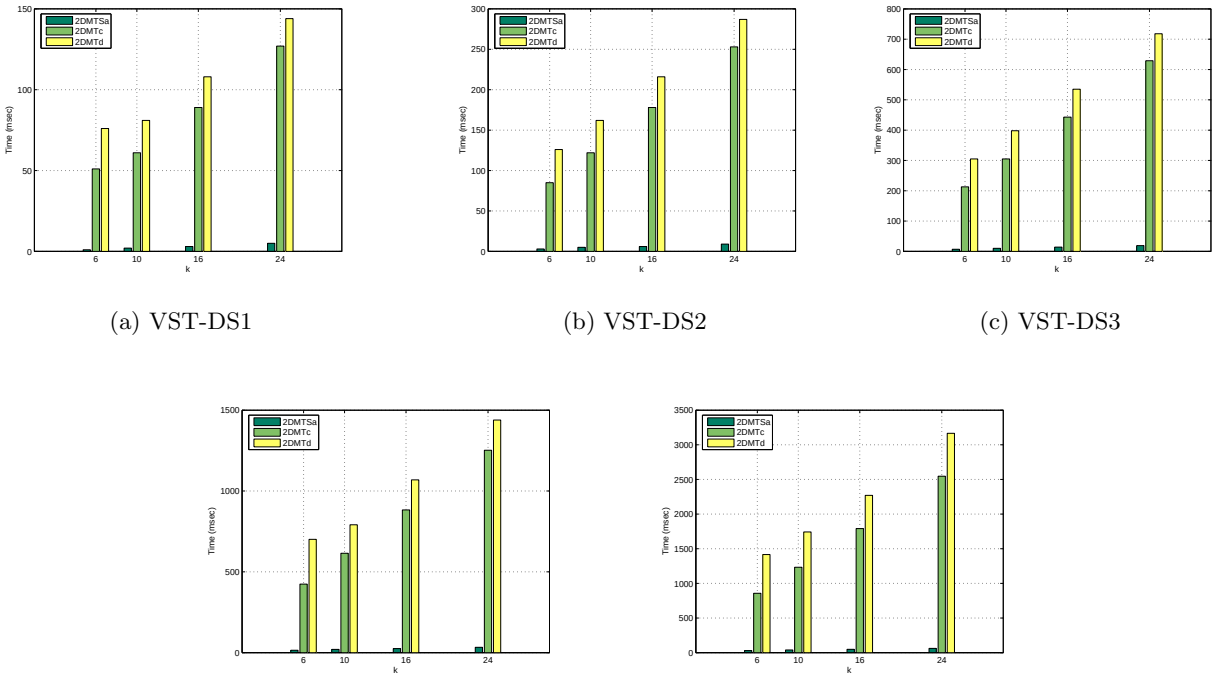
[11] T. Lakshman and D. Stidialis, High speed policy-based packet forwarding using efficient multi-dimensional range matching, *ACM SIGCOMM*, 1998.

[12] L. Qiu, G. Varghese and S. Suri, Fast firewall implementation for software and hardware based routers. *9th International Conference on Network Protocolos ICNP*, 2001.

[13] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, Survey and taxonomy of IP address lookup algorithms, *IEEE Network*, 2001, 8-23.

[14] S. Sahni, K. Kim, and H. Lu, Data structures for one-dimensional packet classification using most-specific-rule matching, *International Journal on Foundations of Computer Science*, 14, 3, 2003, 337-358.

[15] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, Fast and scalable layer four switching, *Proc. ACM SIGCOMM*, 1998.



(a) VST-DS1 (b) VST-DS2 (c) VST-DS3

(d) VST-DS4 (e) VST-DS5

Figure 16: Time (msec) to find strides for $2DMTc(k)$s, $2DMTd(k)$s and $2DMTSa(k)$s

[16] V. Srinivasan, S. Suri, and G. Varghese, Packet classification using tuple space search, *ACM SIGCOMM*, 1999.

[17] V. Srinivasan, A packet classification and filter management system, *INFOCOM*, 2001.

[18] V. Srinivasan and G. Varghese, Faster IP lookups using controlled prefix expansion, *ACM Transactions on Computer Systems*, Feb:1-40, 1999.

[19] C. W. Mortensen, Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time, *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, 2003.

[20] C. Kaufman, R. Perlman and M. Speciner, *Network Security: Private communication in a public world*, Second Edition, Chapter 17, Prentice Hall, NJ, 2002.

[21] H. Lu and S. Sahni, O(log W) multidimensional packet classification, paper in review.

[22] H. Lu and S. Sahni, Conflict detection and resolution in two-dimensional prefix router tables, paper in review.

[23] S. Sahni and K. Kim, Efficient construction of multibit tries for IP lookup, *IEEE/ACM Transactions on Networking*, 11, 4, 2003.

[24] X.Sun and Y.Zhao, Packet classification using independent sets, *IEEE Symposium on Computers & Communications*, 2003, 83-90.

[25] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, Scalable Algorithms for Layer four Switching, *Proceedings of ACM Sigcomm*, 8, 1998.