CHAPTER 5

Hierarchical Intelligent Cuttings: A Dynamic Multi-dimensional Packet Classification Algorithm

1 Introduction

We saw in the previous chapter that real-life classifiers exhibit structure and redundancy that can be exploited by simple algorithms. One such algorithm RFC, was described in the previous chapter. RFC enables fast classification in multiple dimensions. However, its data structure (reduction tree) has a fixed *shape* independent of the characteristics of the classifier.

This chapter is motivated by the observation that an algorithm capable of adapting its data structure based on the characteristics of the classifier may be better suited for exploiting the structure and redundancy in the classifier. One such classification algorithm, called *HiCuts* (Hierarchical Intelligent Cuttings), is proposed in this chapter.

HiCuts discovers the structure while preprocessing the classifier and adapts its data structure accordingly. The data structure used by HiCuts is a decision tree on the set of rules in the classifier. Classification of a packet is performed by a traversal of this tree followed by a linear search on a bounded number of rules. As computing the optimal decision tree for a given search space is known to be an NP-complete problem [40], HiCuts uses simple heuristics to partition the search space in each dimension.

Configuration parameters of the HiCuts algorithm can be tuned to trade-off query time against storage requirements. On 40 real-life four-dimensional classifiers obtained from ISP and enterprise networks with 100 to 1733 rules,[1] HiCuts requires less than 1 Mbyte of storage. The worst case query time is 12, and the average case query time is 8 memory accesses, plus a linear search on 8 rules. The preprocessing time can be sometimes large — up to a minute[2] — but the time to incrementally update a rule in the data structure is less than 100 milliseconds on average.

1.1 Organization of the chapter

Section 2 describes the data structure built by the HiCuts algorithm, including the heuristics used while preprocessing the classifier. Section 3 discusses the performance of HiCuts on the classifier dataset available to us. Finally, Section 4 concludes with a summary and contributions of this chapter.

1. The dataset used in this chapter is identical to that in Chapter 4 except that small classifiers having fewer than 100 rules are not considered in this chapter.

2. Measured using the *time()* lynx system call in user level ‘C’ code on a 333MHz Pentium-II PC, with 96 Mbytes of memory and 512 Kbytes of L2 cache.

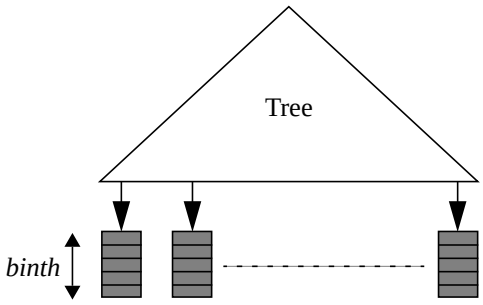

Figure 5.1 This figure shows the tree data structure used by HiCuts. The leaf nodes store a maximum of *binth* classification rules.

2 The Hierarchical Intelligent Cuttings (HiCuts) algorithm

2.1 Data structure

HiCuts builds a decision tree data structure (shown in Figure 5.1) such that the internal nodes contain suitable information to guide the classification algorithm, and the external nodes (i.e., the leaves of the tree) store a small number of rules. On receiving an incoming packet, the classification algorithm traverses the decision tree to arrive at a leaf node. The algorithm then searches the rules stored at this leaf node sequentially to determine the best matching rule. The tree is constructed such that the total number of rules stored in a leaf node is bounded by a small number, which we call *binth* (for 'bin-threshold'). The shape characteristics of the decision tree — such as its depth, the degree of each node, and the local search decision to be made by the query algorithm at each node — are chosen while preprocessing the classifier, and depend on the characteristics of the classifier.

Next, we describe the HiCuts algorithm with the help of the following example.

Example 5.1: Table 5.1 shows a classifier in two 8-bit wide dimensions. The same classifier is illustrated geometrically in Figure 5.2. A decision tree is constructed by recursively partitioning[1] the two-dimensional geometric space. This is shown in Figure 5.3 with a binth of 2. The root node of the tree represents the complete two-dimensional space of size $2^8 \times 2^8$. The algorithm partitions this space into four, equal

1. We will use the terms 'partition' and 'cut' synonymously throughout this chapter.

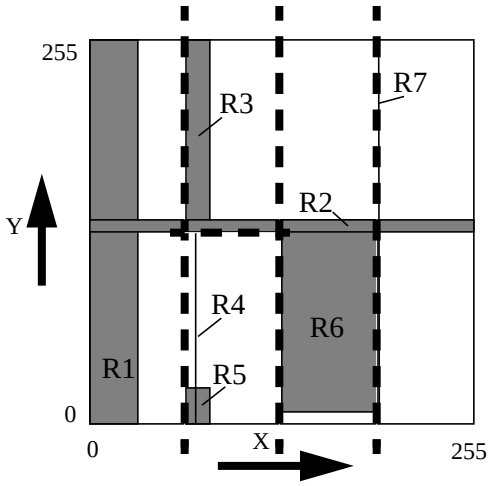

Figure 5.2 An example classifier in two dimensions with seven 8-bit wide rules.

sized geometric subspaces by cutting across the X dimension. The subspaces are represented by each of the four child nodes of the root node. All child nodes, except the node labeled A, have less than or equal to *binth* rules. Hence, the algorithm continues with the tree construction only with node A. The geometric subspace of size $2^6 \times 2^8$ at node A is now partitioned into two equal-sized subspaces by a cut across dimension Y. This results in two child nodes, each of which have two rules stored in them. That completes the construction of the decision tree, since all leaf nodes of this tree have less than or equal to *binth* rules

TABLE 5.1. An example 2-dimensional classifier.

Rule	Xrange	Yrange
R1	0-31	0-255
R2	0-255	128-131
R3	64-71	128-255
R4	67-67	0-127
R5	64-71	0-15
R6	128-191	4-131
R7	192-192	0-255

We can now generalize the description of the HiCuts algorithm in d dimensions as follows. Each internal node, v, of the tree represents a portion of the geometric search space

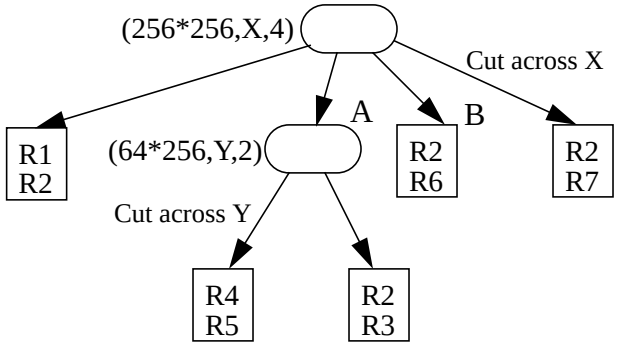

Figure 5.3 A possible HiCuts tree with binth = 2 for the example classifier in Figure 5.2. Each ellipse denotes an internal node v with a tuple $\langle B(v), dim(C(v)), np(C(v))\rangle$. Each square is a leaf node which contains the actual classifier rules.

— for example, the root node represents the complete geometric space in d dimensions. The geometric space at node v is partitioned into smaller geometric subspaces by cutting across one of the d dimensions. These subspaces are represented by each of the child nodes of v. The subspaces are recursively partitioned until a subspace has no more than *binth* number of rules — in which case, the rules are allocated to a leaf node.

More formally, we associate the following entities with each internal node v of the HiCuts data structure for a d-dimensional classifier:

- A hyperrectangle $B(v)$, which is a d-tuple of ranges (i.e., intervals): $([l_1,r_1], [l_2,r_2], ..., [l_d,r_d])$. This rectangle defines the geometric subspace stored at v.
- A cut $C(v)$, defined by two entities. (1) $k = dim(C(v))$, the dimension across which $B(v)$ is partitioned. (2) $np(C(v))$, the number of partitions of $B(v)$, i.e., the number of cuts in the interval $[l_d,r_d]$. Hence, the cut, $C(v)$, divides $B(v)$ into smaller rectangles which are then associated with the child nodes of v.
- A set of rules, $CRS(v)$. If v is a child of w, then $CRS(v)$ is defined to be the subset of $CRS(w)$ that 'collides' with $B(v)$, i.e., every rule in $CRS(w)$ that spans, cuts or is contained in $B(v)$ is also a member of $CRS(v)$. $CRS(root)$ is the set of all

rules in the classifier. We call $CRS(v)$ the colliding rule set of v, and denote the number of rules in $CRS(v)$ by $NumRules(v)$.

As an example of two W-bit wide dimensions, the root node represents a rectangle of size $2^W \times 2^W$. The cuttings are made by axis-parallel hyperplanes, which are simply lines in the case of two dimensions. The cut C of a rectangle B is described by the number of equal-sized intervals created by partitioning one of the two dimensions. For example, if the algorithm decides to cut the root node across the first dimension into D intervals, the root node will have D children, each with a rectangle of size $\left(2^W / D\right) \times 2^W$ associated with it.

2.2 Heuristics for decision tree computation

There are many ways to construct a decision tree for a given classifier. During preprocessing, the HiCuts algorithm uses the following heuristics based on the structure present in the classifier:

1. A heuristic that chooses a suitable number of interval cuts, $np(C)$, to make at an internal node. A large value of $np(C)$ will decrease the depth of the tree, hence accelerating query time at the expense of increased storage. To balance this trade-off, the preprocessing algorithm follows a heuristic that is guided and tuned by a pre-determined space measure function $spmf()$. For example, the definition $spmf(n) = spfac \times n$, where *spfac* is a constant, is used in the experimental results in Section 3. We also define a *space measure* for a cut $C(v)$ as:
   $sm(C(v)) = \sum_{i=1}^{np(C(v))} NumRules(child_i) + np(C(v))$, where $child_j$ denotes the j^{th} child node of node v. HiCuts makes as many cuttings as $spmf()$ allows at a certain node, depending on the number of rules at that node. For instance, $np(C(v))$ could be chosen to be the largest value (using a simple binary search) such that $sm(C(v)) < spmf(NumRules(v))$. The pseudocode for such a search algorithm is shown in Figure 5.4.
2. A heuristic that chooses the dimension to cut across, at each internal node. For example, it can be seen from Figure 5.2 that cutting across the Y-axis would be

```
/* Algorithm to do binary search on the number of cuts to be made at node v. When the number of cuts are
such that the corresponding storage space estimate becomes more than what is allowed by the spacemea-
sure function spmf(), we end the search. Note that it is possible to do smarter variations of this search algo-
rithm.*/
n = numRules(v);
nump = max(4, sqrt(n)); /* arbitrary starting value of number of partitions to make at this node */
for (done=0;done == 0;)
{
sm(C) = 0;
 for each rule r in R(v)
   { sm(C) += number of partitions colliding with rule r; }
 sm(C) += nump;
 if (sm(C) < spmf(n))
 {
   nump = nump * 2; /* increase the number of partitions in a binary search fashion */
 }
 else { done = 1;}
}
/* The algorithm has now found a value of nump (the number of children of this node) that fits the storage
requirements */
```

Figure 5.4 Pseudocode for algorithm to choose the number of cuts to be made at node v.

less beneficial than cutting across the X-axis at the root node. There are various methods to choose a good dimension: *(a)* Minimizing $max_j(NumRules(child_j))$ in an attempt to decrease the worst-case depth of the tree. *(b)* Treating $\left\{NumRules(child_j) / \left(\sum_{i=1}^{np(C(v))} NumRules\,(child_i) \right)\right\}$ as a probability distribution with $np(C)$ elements, and maximizing the entropy of the distribution. Intuitively, this attempts to pick a dimension that leads to the most uniform distribution of rules across nodes, so as to create a balanced decision tree. *(c)* Minimizing $sm(C)$ over all dimensions. *(d)* Choosing the dimension that has the largest number of distinct components of rules. For instance, in the classifier of Table 5.1, rules $R3$ and $R5$ share the same rule component in the X dimension. Hence, there are 6 components in the X dimension and 7 components in the Y dimension. The use of this heuristic would dictate a cut across dimension Y for this classifier.

3. A heuristic that maximizes the reuse of child nodes. We have observed in our experiments that in real classifiers, many child nodes have identical colliding rule sets. Hence, a single child node can be used for each distinct colliding rule set,

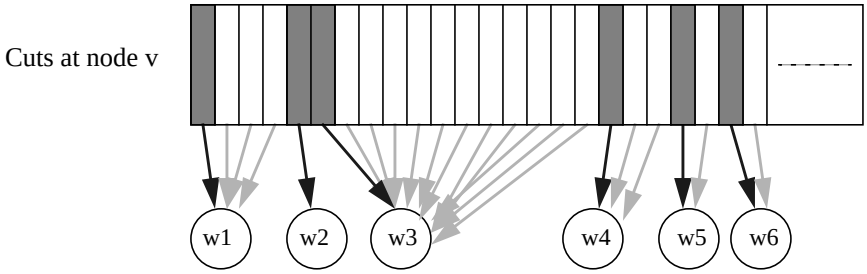

Figure 5.5 An example of the heuristic maximizing the reuse of child nodes. The gray regions correspond to children with distinct colliding rule sets.

while other child nodes with identical rule sets can simply point to this node. Figure 5.5 illustrates this heuristic.

4. A heuristic that eliminates redundancies in the colliding rule set of a node. As a result of the partitioning of a node, rules may become redundant in some of the child nodes. For example, in the classifier of Table 5.1, if $R6$ were higher priority than $R2$, then $R2$ would be made redundant by $R6$ in the third child of the root node labeled B (see Figure 5.3). Detecting and eliminating these redundant rules can decrease the data structure storage requirements at the expense of increased preprocessing time. In the experiments described in the next section, we invoked this heuristic when the number of rules at a node fell below a threshold.

3 Performance of HiCuts

We built a simple software environment to measure the performance of the HiCuts algorithm. Our dataset consists of 40 classifiers containing between 100 and 1733 rules from real ISP and enterprise networks. For each classifier, a data structure is built using the heuristics described in the previous section. The preprocessing algorithm is tuned by two parameters: (1) *binth*, and (2) *spfac* — used in the space measure function *spmf()*, defined as $spmf(n) = spfac \times n$.

Figure 5.6 shows the total data structure storage requirements for $binth = 8$ and $spfac = 4$. As shown, the maximum storage requirement for any classifier is approximately 1 Mbyte, while the second highest value is less than 500 Kbytes. These small stor-

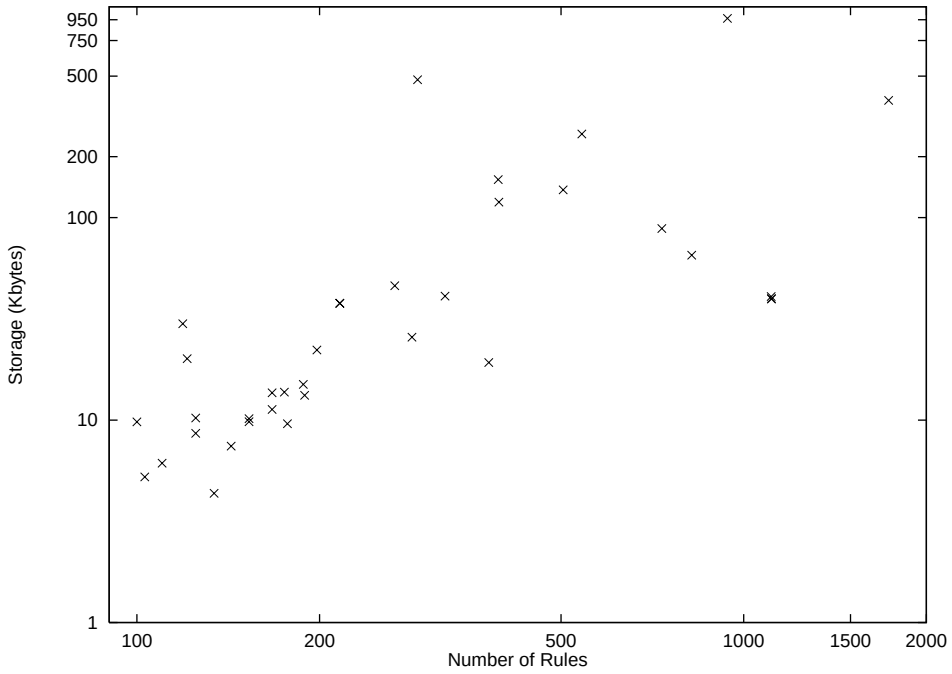

Figure 5.6 Storage requirements for four dimensional classifiers for binth=8 and spfac=4.

age requirements imply that in a software implementation of the HiCuts algorithm, the whole data structure would readily fit in the L2 cache of most general purpose processors.

For the same parameters ($binth = 8$ and $spfac = 4$), Figure 5.7 shows the maximum and average tree depth for the classifiers in the dataset. The average tree depth is calculated under the assumption that each leaf is accessed in proportion to the number of rules stored in it. As shown in Figure 5.7, the worst case tree depth for any classifier is 12, while the average is approximately 8. This implies that — in the worst case — a total of 12 memory accesses are required, followed by a linear search on 8 rules to complete the classification. Hence, a total of 20 memory accesses are required in the worst case for these parameters.

The preprocessing time required to build the decision tree is plotted in Figure 5.8. This figure shows that the highest preprocessing time is 50.5 seconds, while the next highest

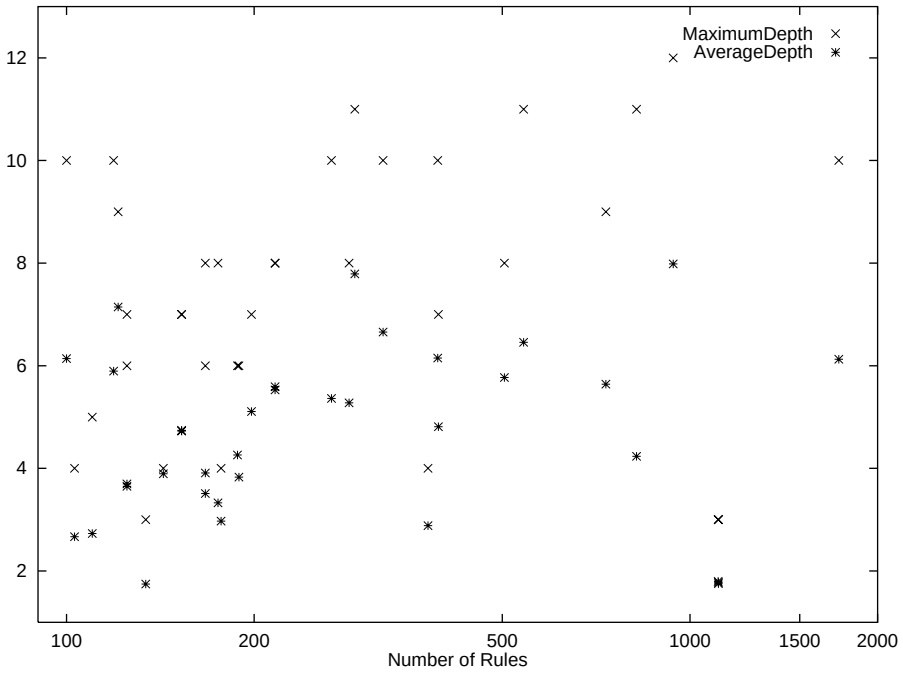

Figure 5.7 Average and worst case tree depth for binth=8 and spfac=4.

value is approximately 20 seconds. All but four classifiers have a preprocessing time of less than 8 seconds.

The reason for the fairly large preprocessing time is mainly the number and complexity of the heuristics used in the HiCuts algorithm. We expect this preprocessing time to be acceptable in most applications, as long as the time taken to incrementally update the tree remains small. In practice, the update time depends on the rule to be inserted or deleted. Simulations indicate an average incremental update time between 1 and 70 milliseconds (averaged over all the rules of a classifier), and a worst case update time of nearly 17 seconds (see Figure 5.9).

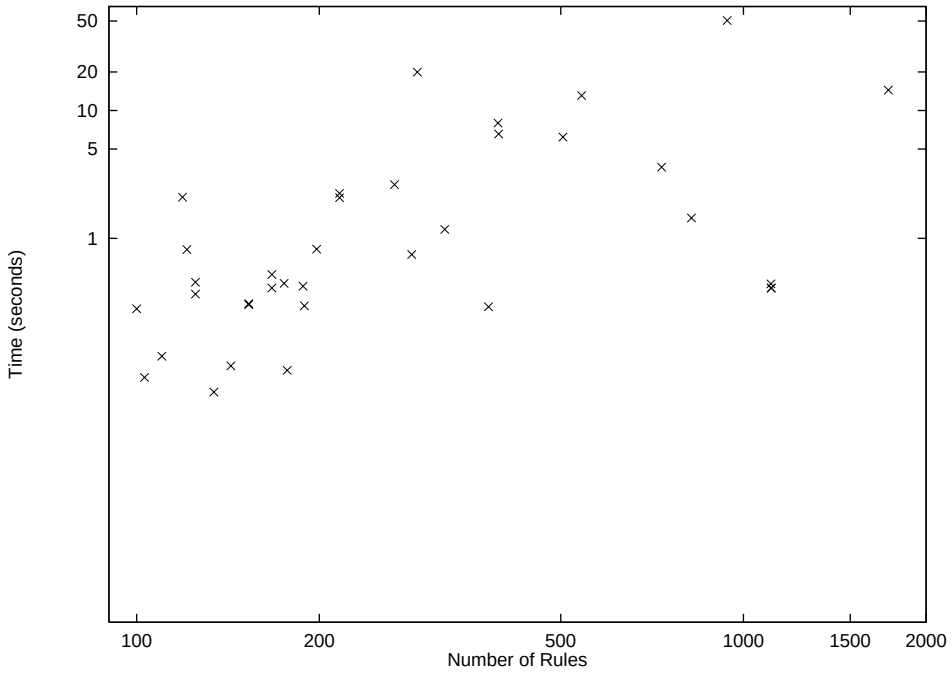

Figure 5.8 Time to preprocess the classifier to build the decision tree. The measurements were taken using the *time()* linux system call in user level 'C' code on a 333 MHz Pentium-II PC with 96 Mbytes of memory and 512 Kbytes of L2 cache.

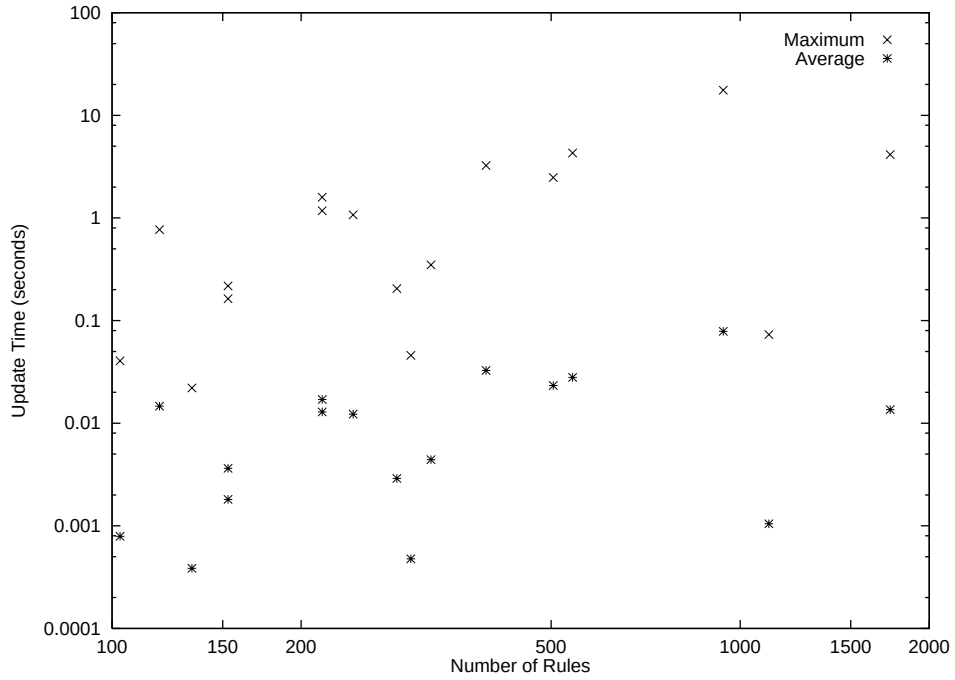

Figure 5.9 The average and maximum update times (averaged over 10,000 inserts and deletes of randomly chosen rules for a classifier). The measurements were taken using the *time()* linux system call in user level 'C' code on a 333 MHz Pentium-II PC with 96 Mbytes of memory and 512 Kbytes of L2 cache.

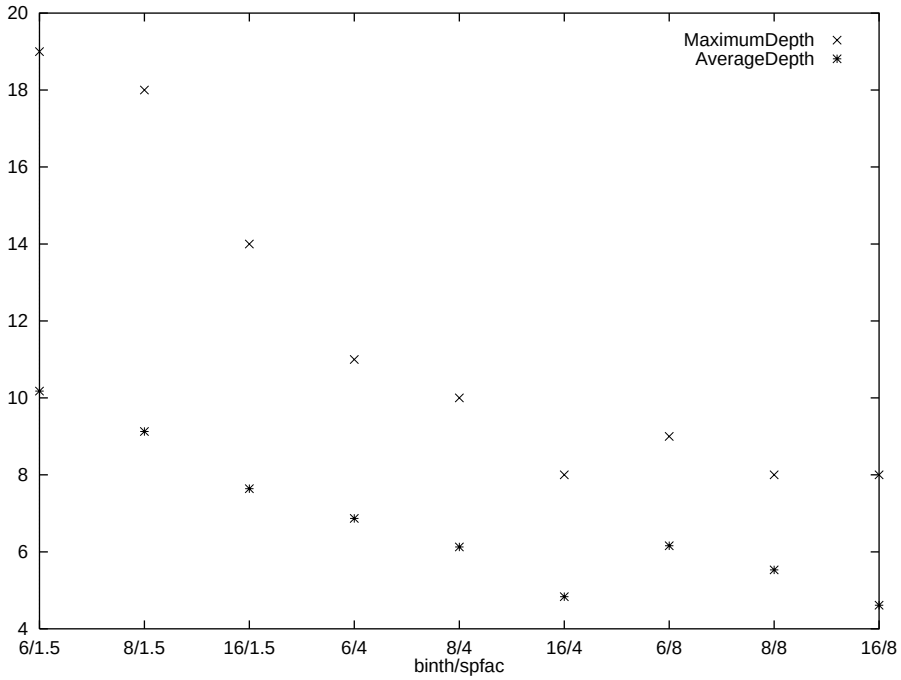

Figure 5.10 Variation of tree depth with parameters binth and spfac for a classifier with 1733 rules.

3.1 Variation with parameters *binth* and *spfac*

Next, we show the effect of varying the configuration parameters *binth* and *spfac* on the data structure for the largest 4-dimensional classifier available to us containing 1733 rules. We carried out a series of experiments where parameter *binth* took the values 6, 8 and 16; and parameter *spfac* took the values 1.5, 4 and 8. We make the following, somewhat expected, observations from our experiments:

1. The HiCuts tree depth is inversely proportional to both *binth* and *spfac. This is* shown in Figure 5.10.
2. As shown in Figure 5.11, the data structure storage requirements are directly proportional to *spfac* but inversely proportional to *binth*.

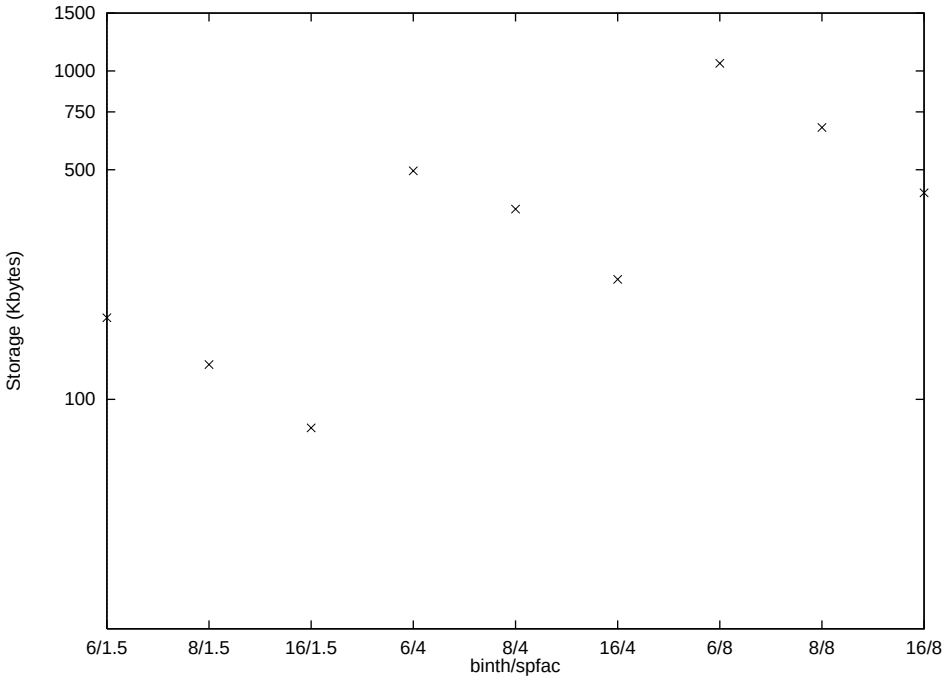

Figure 5.11 Variation of storage requirements with parameters *binth* and *spfac* for a classifier with 1733 rules.

3. The preprocessing time is proportional to the storage requirements, as shown in Figure 5.12.

3.2 Discussion of implementation of HiCuts

Compared with the RFC algorithm described in Chapter 4, the HiCuts algorithm is slower but consumes a smaller amount of storage. As with RFC, it seems difficult to characterize the storage requirements of HiCuts as a function of the number of rules in the classifier. However, given certain design constraints in terms of the maximum available storage space or the maximum available classification time, HiCuts seems to provide greater flexibility in satisfying these constraints by allowing variation of the two parameters, *binth* and *spfac*.

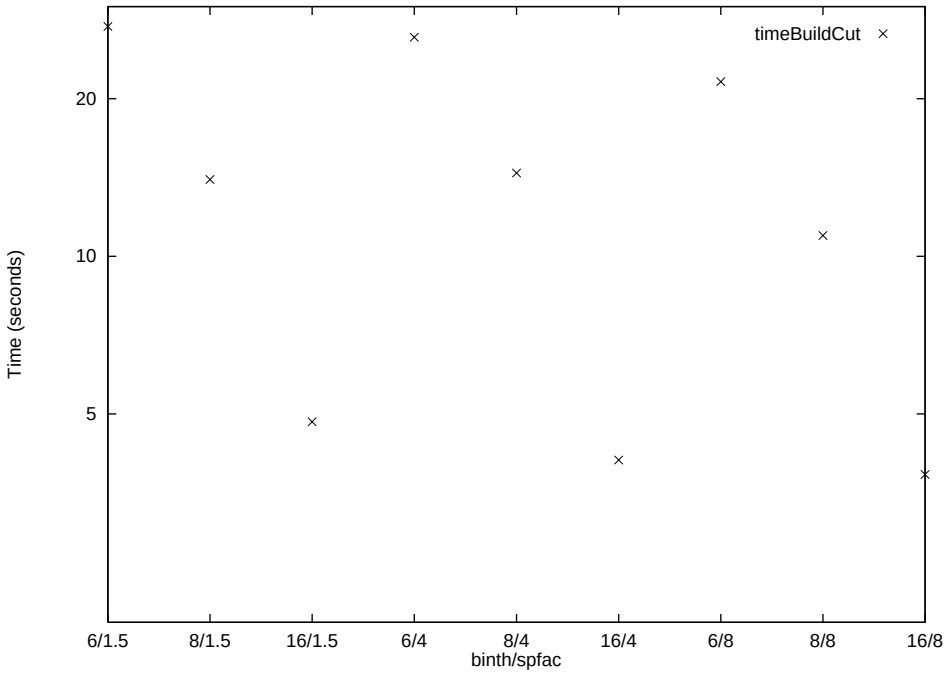

Figure 5.12 Variation of preprocessing times with *binth* and *spfac* for a classifier with 1733 rules. The measurements were taken using the time() linux system call in user level 'C' code on a 333 MHz Pentium-II PC with 96 Mbytes of memory and 512 Kbytes of L2 cache.

4 Conclusions and summary of contributions

The design of multi-field classification algorithms is hampered by worst-case bounds on query time and storage requirements that are so onerous as to make generic algorithms unusable. So instead we must search for characteristics of real-life classifiers that can be exploited in pursuit of fast algorithms that are also space-efficient. Similar to Chapter 4, this chapter resorts to heuristics that, while hopefully well-founded in a solid understanding of today's classifiers, exploit the structure of classifiers to reduce query time and storage requirements.

While the data structure of Chapter 4 remains the same for all classifiers, HiCuts goes a step further in that it attempts to compute a data structure that varies depending on the structure of the classifier — the structure is itself discovered and utilized while prepro-

cessing the classifier. The HiCuts algorithm combines two data structures for better performance — a tree and a linear search data structure — each of which would not be as useful separately. The resulting HiCuts data structure is the only data structure we know of that simultaneously supports quick updates along with small deterministic classification time and reasonable data structure storage requirements.