CHAPTER

Packet classification 12

A classification is a definition comprising a system of definitions.
—Friedrich von Schlegel

Traditionally, the post office forwards messages based on the destination address in each letter. Thus all letters to Timbuctoo were forwarded in exactly the same way at each post office. However, to gain additional revenue, the post office introduced *service differentiation* between ordinary mail, priority mail, and express mail. Thus forwarding at the post office is now a function of the destination address and the traffic class. Further, with the specter of terrorist threats and criminal activity, forwarding could even be based on the source address, with special screening for suspicious sources.

In exactly the same way, routers have evolved from traditional destination-based forwarding devices to what are called *packet classification routers*. In modern routers, the route and resources allocated to a packet are determined by the destination address as well as other header fields of the packet, such as the source address and TCP/UDP port numbers.

Packet classification unifies the forwarding functions required by firewalls, resource reservations, QoS routing, unicast routing, and multicast routing. In classification, the forwarding database of a router consists of a potentially large number of rules on key header fields. A given packet header can match multiple rules. So each rule is given a cost, and the packet is forwarded using the *least-cost matching rule.*

The world has changed significantly since the first edition, but most of the changes relevant to packet classification are a subset of the changes described at the start of Chapter 11 on IP lookups. These are the significant use of IPv6 (which complicates packet classification), the increasing use of Software Defined Networks (SDN) and hypervisor switches (Pfaff et al., 2015) to do flexible forwarding using packet classification instead of simpler IP lookups, and the emergence of Network Function Virtualization (NFV) which requires software solutions to packet classification. We are grateful to Balajee Vamanan for helping us with more recent work in packet classification. Despite these technological changes, the essential ideas have remained.

This chapter is organized as follows. The packet classification problem is motivated in Section 12.1. The classification problem is formulated precisely in Section 12.2, and the metrics used to evaluate rule schemes are described in Section 12.3. Section 12.4 presents simple schemes such as linear search, tuple space search and TCAMs. Section 12.5 begins the discussion of more efficient schemes by describing an efficient scheme called *grid of tries* that works only for rules specifying values of only two fields. Section 12.6 transitions to general rule sets by describing a set of insights into the classification problem, including the use of a geometric viewpoint.

Section 12.7 begins the transition to algorithms for the general case with a simple idea to extend 2D schemes. A general approach based on divide-and-conquer is described in Section 12.8. This is followed

Table 12.1 Summary of the principles used in the classification algorithms described in this chapter.

Number	Principle	Lookup technique
P12 **P2a**	Add marker state Precompute filter info	Rectangle and tuple search
P15 **P2a**	Use Dest and SRC tries Precompute switch pointers	Grid of tries
P15 **P12, 2a**	Divide-and-conquer by first doing field lookups	Bit vector, pruned tuple, cross-producting
P11	Exploit lack of general ranges	Multiple 2D planes
P4a	Exploit bitmap memory locality	Bit vector scheme
P11	Exploit small number of prefixes that match any field	Pruned tuple
P11a, 4a	Exploit cross product locality	On-demand cross product
P1	Avoid redundant cross products	Equivalent cross-producting

by three very different examples of algorithms based on divide-and-conquer: simple and aggregated bit vector linear search (Section 12.9), cross-producting (Section 12.10), and RFC, or equivalenced cross-producting (Section 12.11). Section 12.12 presents the most promising of the current algorithmic approaches, an approach based on decision trees.

This chapter will continue to exhibit the set of principles introduced in Chapter 3, as summarized in Table 12.1. The chapter will also illustrate three general problem-solving strategies: solving simpler problems first before solving a complex problem, collecting different viewpoints, and exploiting the structure of input data sets.

Quick reference guide

The most important lookup algorithms for an implementor today are as follows. If memory is not an issue, the fastest scheme is one called recursive flow classification (RFC), described in Section 12.11. If memory is an issue, a simple scheme that works well for classifiers up to around 5000 rules is the Lucent bit vector scheme (Section 12.9). For larger classifiers, the best trade-off between speed and memory is provided by decision tree schemes, such as HyperCuts and EffiCuts (Section 12.12). For software settings which require fast updates as in Hypervisor switches, then a good solution is Tuple Space Search and the improvements implemented in Open Vswitch (Pfaff et al., 2015). Unfortunately, all these algorithms are based on heuristics and cannot guarantee performance on all databases. If guaranteed performance is required for more than two field classifiers, there is no alternative but to consider hardware schemes such as ternary CAMs.

12.1 Why packet classification?

Packet forwarding based on a longest-matching-prefix lookup of destination IP addresses is fairly well understood, with both algorithmic and CAM-based solutions in the market. Using basic variants of tries and some pipelining (see Chapter 11), it is fairly easy to perform one packet lookup every memory access time.

Unfortunately, the Internet is becoming more complex because of its use for mission-critical functions executed by organizations. Organizations desire that their critical activities not be subverted either by high traffic sent by other organizations (they require QoS guarantees) or by malicious intruders (they require security guarantees). Both QoS and security guarantees require a finer discrimination of packets, based on fields other than the destination. This is called *packet classification*. To quote John McQuillan (1997):

> Routing has traditionally been based solely on destination host numbers. In the future it will also be based on source host or even source users, as well as destination URLs (universal resource locators) and specific business policies. . . . Thus, in the future, you may be sent on one path when you casually browse the Web for CNN headlines. And you may be routed an entirely different way when you go to your corporate Web site to enter monthly sales figures, even though the two sites might be hosted by the same facility at the same location. . . . An order entry form may get very low latency, while other sections get normal service. And then there are Web sites comprised of different servers in different locations. Future routers and switches will have to use class of service and QoS to determine the paths to particular Web pages for particular end users. All this requires the use of layers 4, 5, and above.

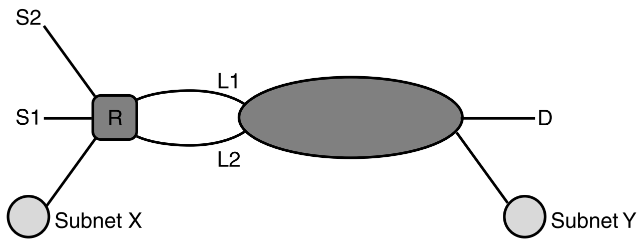
This (now standard) vision of forwarding is called *packet classification*. It is also sometimes called *layer 4 switching*, because routing decisions can be based on headers available at layer 4 or higher in the OSI architecture. Examples of other fields a router may need to examine include source addresses (to forbid or provide different service to some source networks), port fields (to discriminate between traffic types, such as Napster and E-mail), and even TCP flags (to distinguish between externally and internally initiated connections). Besides security and QoS, other functions that require classification include network address translation (NAT), metering, traffic shaping, policing, and monitoring.

Several variants of packet classification have already established themselves on the Internet. First, many routers implement *firewalls* (Cheswick and Bellovin, 1995) at trust boundaries, such as the entry and exit points of a corporate network. A firewall database consists of a series of packet rules that implement security policies. A typical policy may be to allow remote login from within the corporation but to disallow it from outside the corporation.

Second, the need for predictable and guaranteed service has led to proposals for reservation protocols, such as DiffServ (Blake et al., 1998), that reserve bandwidth between a source and a destination. Third, the cries for routing based on traffic type have become more strident recently—for instance, the need to route Web traffic between Site 1 and Site 2 on, say, Route A and other traffic on, say, Route B. Fig. 12.1 illustrates some of these examples.

Classifiers historically evolved from firewalls, which were placed at the edges of networks to filter out unwanted packets. Such databases are generally small, containing 10–500 rules, and can be handled by ad hoc methods. However, with the DiffServ movement, there is potential for classifiers that could support 100,000 rules for DiffServ and policing applications at edge routers.

While large classifiers are anticipated for edge routers to enforce QoS via DiffServ, it is perhaps surprising that even within the core, fairly large (e.g., 2000-rule) classifiers are commonly used for security. While these core router classifiers are nowhere near the anticipated size of edge router classifiers, there seems no reason why they should not continue to grow beyond the sizes reported in this book. For example, many of the rules appear to be denying traffic from a specified subnetwork outside



DATABASE AT ROUTER R

To	From	Traffic type	Forwarding directive
D	S1	Video	Forward via L1
*	S2	*	Drop all traffic
Y	X	*	Reserve 50 Mbps

FIGURE 12.1

Example of rules that provide traffic-sensitive routing, a firewall rule, and resource reservation. The first rule routes video traffic from S1 to D via L1; not shown is the default routing to D, which is via L2. The second rule blocks traffic from an experimental site, S2, from accidentally leaving the site. The third rule reserves 50 Mbps of traffic from an internal network X to an external network Y, implemented perhaps by forwarding such traffic to a special outbound queue that receives special scheduling guarantees; here X and Y are prefixes.

the ISP to a server (or subnetwork) within the ISP. Thus, new offending sources could be discovered and new servers could be added that need protection. In fact, we speculate that one reason why core router classifiers are not even bigger is that most core router implementations slow down (and do not guarantee true wire speed forwarding) as classifier sizes increase.

Third, after the emergence of SDN and virtualization, packet classification has also become a key component of software switches especially in hypervisors (Pfaff et al., 2015). The use of server virtualization has resulted in most data center and many enterprise networks becoming virtual networks that connect virtual ports corresponding to virtual machines. Further, these virtual networks may be reconfigured rapidly as virtual machines migrate. For example, Open VSwitch allows the switch to be reprogrammed at rapid rates using an Open Flow controller (Pfaff et al., 2015).

12.2 Packet-classification problem

Traditionally, the rules for classifying a message are called *rules* and the packet-classification problem is to determine the lowest-cost matching rule for each incoming message at a router.

Assume that the information relevant to a lookup is contained in K distinct *header fields* in each message. These header fields are denoted $H[1], H[2], \ldots, H[K]$, where each field is a string of bits.

For instance, the relevant fields for an IPv4 packet could be the destination address (32 bits), the source address (32 bits), the protocol field (8 bits), the destination port (16 bits), the source port (16 bits), and TCP flags (8 bits). The number of relevant TCP flags is limited, and so the protocol and TCP flags are combined into one field—for example, TCP-ACK can be used to mean a TCP packet with the ACK bit set.[1] Other relevant TCP flags can be represented similarly; UDP packets are represented by $H[3] = UDP$.

Thus, the combination $(D, S,$ TCP-ACK, 63, 125) denotes the header of an IP packet with destination D, source S, protocol TCP, destination port 63, source port 125, and the ACK bit set.

The *classifier*, or *rule database*, router consists of a finite set of rules, $R_1, R_2, \ldots, R_N$. Each rule is a combination of K values, one for each header field. Each field in a rule is allowed three kinds of matches: exact match, prefix match, and range match. In an *exact match*, the header field of the packet should exactly match the rule field—for instance, this is useful for protocol and flag fields. In a *prefix match*, the rule field should be a prefix of the header field—this could be useful for blocking access from a certain subnetwork. In a *range match*, the header values should lie in the range specified by the rule—this can be useful for specifying port number ranges.

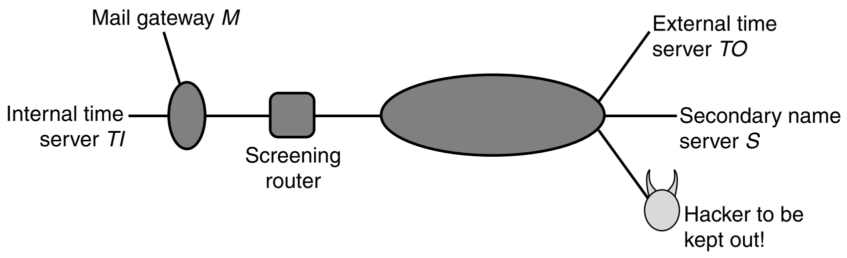
Each rule R_i has an associated directive $disp_i$, which specifies how to forward the packet matching this rule. The directive specifies if the packet should be blocked. If the packet is to be forwarded, the directive specifies the outgoing link to which the packet is sent and, perhaps, also a queue within that link if the message belongs to a flow with bandwidth guarantees.

A packet P is said to *match* a rule R if each field of P matches the corresponding field of R—the match type is implicit in the specification of the field. For instance, if the destination field is specified as 1010$*$, then it requires a prefix match; if the protocol field is UDP, then it requires an exact match; if the port field is a range, such as 1024–1100, then it requires a range match. For instance, let $R = (1010*, *, TCP, 1024\text{–}1080, *)$ be a rule, with $disp = block$. Then, a packet with header (10101...111, 11110...000, TCP, 1050, 3) matches R and is therefore blocked. The packet (10110...000, 11110...000, TCP, 80, 3), on the other hand, doesn't match R.

Since a packet may match multiple rules in the database, each rule R in the database is associated with a nonnegative number, $cost(R)$. Ambiguity is avoided by returning the least-cost rule matching the packet's header. The cost function generalizes the implicit precedence rules that are used in practice to choose between multiple matching rules. In firewall applications or Cisco ACLs, for instance, rules are placed in the database in a specific linear order, where each rule takes precedence over a subsequent rule. Thus, the goal there is to find the *first* matching rule. Of course, the same effect can be achieved by making $cost(R)$ equal to the position of rule R in the database.

As an example of a rule database, consider the topology and firewall database (Cheswick and Bellovin, 1995) shown in Fig. 12.2, where a screened subnet configuration interposes between a company subnetwork (shown on top left) and the rest of the Internet (including hackers). There is a so-called bastion host M within the company that mediates all access to and from the external world. M serves as the mail gateway and also provides external name server access. TI, TO are network time protocol (NTP) sources, where TI is internal to the company and TO is external. S is the address of the secondary name server, which is external to the company.

[1] TCP flags are important for packet classification because the first packet in a connection does not have the ACK bit set, while the others do. This allows a simple rule to block TCP connections initiated from the outside while allowing responses to internally initiated connections.


Mail gateway *M*
Internal time server *TI*
Screening router
External time server *TO*
Secondary name server *S*
Hacker to be kept out!

Destination	Source	Destination port	Source port	Flags	Comments
M	*	25	*	*	Allow inbound mail
M	*	53	*	UDP	Allow DNS access
M	S	53	*	*	Secondary access
M	*	23	*	*	Incoming telnet
TI	*TO*	123	123	UDP	NTP time info
*	Net	*	*	*	Outgoing packets
Net	*	*	*	TCP ack	Return ACKs OK
*	*	*	*	*	Block everything!

FIGURE 12.2

The top half of the figure shows the topology of a small company; the bottom half shows a sample firewall database for this company as described in the book by Cheswick and Bellovin (1995). The *block* flags are not shown in the figure; the first seven rules have *block* $= false$ (i.e., allow) and the last rule has *block* $= true$ (i.e., block). We assume that all the addresses within the company subnetwork (shown on top left) start with the prefix Net, including M and TI.

Clearly, the site manager wishes to allow communication from within the network to TO and S and yet wishes to block hackers. The database of rules shown at the bottom of Fig. 12.2 implements this intention. Terse explanations of each rule are shown on the right of each rule. Assume that all addresses of machines within the company's network start with the CIDR prefix Net. Thus M and TI both match the prefix Net. All packets matching any of the first seven rules are allowed; the remaining (last rule) are dropped by the screening router. A more general firewall could arbitrarily interleave rules that allow packets with rules that drop packets.

As an example, consider a packet sent to M from S with UDP destination port equal to 53. This packet matches Rules 2, 3, and 8 but must be allowed through because the first matching rule is Rule 2.

Note that this description uses N for the number of rules and K for the number of packet fields. K is sometimes called the *number of dimensions*, for reasons that will become clearer in Section 12.6.

12.3 Requirements and metrics

The requirements for rule matching are similar to those for IP lookups (Chapter 11). We wish to do packet classification at wire speed for minimum-size packets, and thus speed is the dominant metric. To allow the database to fit in high-speed memory, it is useful to reduce the amount of memory needed. For most firewall databases, insertion speed is not an issue because rules are rarely changed.

However, this is not true for *dynamic* or *stateful* packet rules. This capability is useful, for example, for handling UDP traffic. Because UDP headers do not contain an ACK bit that can be used to determine whether a packet is the bellwether packet of a connection, the screening router cannot tell the difference between the first packet sent from the outside to an internal server (which it may want to block) and a response sent to a UDP request to an internal client (which it may want to pass). The solution used in some products is to have the outgoing request packet dynamically trigger the insertion of a rule (which has addresses and ports that match the request) that allows the inbound response to be passed. This requires very fast update times, a third metric.

Besides stateful firewalls, Software-defined Networking (SDN) may also require frequent rule insertions. For instance, reactive SDN controllers add a new rule wherever new traffic starts. Rule updates could become more frequent as network management becomes more agile and programmatic.

12.4 Simple solutions

There are six simple solutions that are often used or considered: linear search, tuple space search, caching, demultiplexing algorithms, MPLS, and content-addressable memories (CAMs). While CAMs have difficult hardware design issues, they effectively represent a parallelization of the simplest algorithmic approach: linear search.

12.4.1 Linear search

Some older firewall implementations do a linear search of the database and keep track of the best-matching rule. Linear search is reasonable for small rule sizes but is extremely slow for large rule sets. For example, a core router that does linear search among a rule set of 2000 rules (used at the time of writing by some ISPs) will considerably degrade its forwarding performance below wire speed.

12.4.2 Tuple space search

A simple improvement of linear search is a simple generalization of IP lookups using hash tables where prefixes were partitioned by length, and all prefixes of the same length are placed in a common hash table. While this requires 32 memory accesses in the worst case for IPv4, recall that the previous chapter described a major improvement, binary search on prefix lengths that could perform IPv4 lookups in $\log_2 32 = 5$ memory accesses.

While the use of binary search does not generalize from IP lookups to packet classification as far as we know, the idea of using linear search on hash tables does and was first called *tuple space search (TSS)* by Srinivasan et al. (1998). Consider a simple packet classifier with four rules on IPv4 Destination (D) and Source (S) fields. Ignoring the directive fields, assume that R_1 must match $D = 01*$, $S = 10*$;

second, R_2 must match $D = 10*$, $S = 00*$; third, R_3 must match $D = 0*$, $S = 1*$; finally, R_4 must match $D = 1*$, $S = 0*$.

While there are four rules, there are only 2 combinations of specified lengths or tuples in the table: the tuple $(2, 2)$ (length two specified in source and destination fields as in R_1 and R_2), and the tuple $(1, 1)$ (length 1 specified in source and destination fields as in R_3 and R_4). Thus the key idea is to place all rules with the same tuple in a common hash table and do a linear search across all hash tables corresponding to valid tuples. Each hash table is indexed by a key formed by concatenating the number of bits in each field specified by the tuple. For example, the hash table corresponding to tuple $(2, 2)$ is indexed by a key formed by concatenating the first two bits of the Destination IP address and the first two bits of the Source IP address of the packet to be classified. Note that one cannot stop after a successful match because later tuples could have a lower cost match.

In this simple example, tuple search requires only two memory accesses (assuming each hash table takes one memory access to search) while linear search takes 4 memory access. However, if each tuple has 1000's of rules in the corresponding hash table, tuple space search can be 1000 times faster. However, the worst case number of tuples can still be very large. For example, even for IPv4 and considering only source and destination fields, the number of tuples (length combinations) can be as bad as 32 * 32 = 1024.

While there are more efficient schemes such as decision trees that are described later, tuple space search has some important advantages. First, it has very fast update times because the time taken to add (or delete) a new rule is $O(1)$: we simply insert or delete the rule from the hash table corresponding to its length tuple. Second, the memory required is $O(n)$ where n is the number of rules. Third, tuple space search easily generalizes to adding new fields as may be required in Software Defined Networks. Perhaps for this reason, tuple space search is used in Open vSwitch (Pfaff et al., 2015) because in a virtualized network new rules can be added several times every second (Pfaff et al., 2015).

The actual use of Tuple Space search in Open vSwitch is much cleverer with several clever heuristics to reduce the average lookup time, an important metric for software switches unlike hardware switches. Two important ideas that speed up the average case in Open vSwitch (Pfaff et al., 2015) are the use of caching and tuple priority sorting. We will discuss caching below, but priority sorting is a way of stopping tuple search early by remembering the highest priority flow associated with each tuple hash table. The details can be found in (Pfaff et al., 2015).

12.4.3 Caching

Some implementations even cache the result of the search keyed against the whole header. There are two problems with this scheme. First, the cache hit rate of caching full IP addresses in the backbones is often small. Early studies show a hit rate of at most 80%–90% (Partridge, 1996; Newman et al., 1997). Part of the problem is Web accesses and other flows that send only a small number of packets; if a Web session sends just five packets to the same address, then the cache hit rate is 80%. Since caching full headers takes a lot more memory, this should have an even worse hit rate (for the same amount of cache memory).

Second, even with a 90% hit rate cache, a slow linear search of the rule space will result in poor performance.[2] For example, suppose that a search of the cache costs 100 nanoseconds (one memory

[2] This is an application of a famous principle in computer architecture called *Amdahl's law*.

access) and that a linear search of 10,000 rules costs 1,000,000 nanoseconds = 1 millisecond (one memory access per rule). Then the average search time with a cache hit rate of 90% is still 0.1 millisecond, which is very slow.

However, caching could be combined with some of the fast algorithms in this chapter to improve the expected search time even further. An investigation of the use of caching for classification can be found in Xu et al. (2000). As a more recent example, Open vSwitch (Pfaff et al., 2015) combines Tuple Space search with caching in clever ways. In particular, the Open vSwitch implementation does caching not of full IP and TCP headers of a TCP connection (which they call microflows) but instead for larger aggregates (that they call macroflows). Macroflow caching requires some more complexity to handle correctly (Pfaff et al., 2015).

12.4.4 Demultiplexing algorithms

Chapter 8 describes the use of packet rules for demultiplexing and algorithms such as Pathfinder, Berkeley packet filter, and dynamic path finder. Can't these existing solutions simply be reused? It is important to realize that the two problems are similar but subtly different.

The first packet-classification scheme that avoids a linear search through the set of rules is Pathfinder (Bailey et al., 1994). However, Pathfinder allows wildcards to occur only at the end of a rule. For instance, $(D, S, *, *, *)$ is allowed, but not $(D, *, Prot, *, SourcePort)$. With this restriction, all rules can be merged into a generalized trie—with hash tables replacing array nodes—and rule lookup can be done in time proportional to the number of packet fields. DPF (Engler and Kaashoek, 1996) uses the Pathfinder idea of merging rules into a trie but adds the idea of using dynamic code generation for extra performance. However, it is unclear how to handle intermixed wildcards and specified fields, such as $(D, *, Prot, *, SourcePort)$, using these schemes.

Because packet classification allows more general rules, the Pathfinder idea of using a trie does not work well. There does exist a simple trie scheme (set-pruning tries; see Section 12.5.1) to perform a lookup in time $O(M)$, where M is the number of packet fields. Such schemes are described in Decasper et al. (1998) and Malan and Jahanian (1998). Unfortunately, such schemes require $\Theta(N^K)$ storage, where K is the number of packet fields and N is the number of rules. Thus such schemes are not scalable for large databases. By contrast, some of the schemes we will describe require only $O(NM)$ storage.

12.4.5 Passing labels

Recall from Chapter 11 that one way to finesse lookups is to pass a label from a previous-hop router to a next-hop router. One of the most prominent examples of such a technology is multiprotocol label switching (MPLS) (IETF MPLS Charter, 1997). While IP lookups have been able to keep pace with wire speeds, the difficulties of algorithmic approaches to packet classification have ensured an important niche for MPLS. Refer to Chapter 11 for a description of tag switching and MPLS.

Today MPLS is useful mostly for traffic engineering. For example, if Web traffic between two sites A and B is to be routed along a special path, a label-switched path is set up between the two sites. Before traffic leaves site A, a router does packet classification and maps the Web traffic into an MPLS header. Core routers examine only the label in the header until the traffic reaches B, at which point the MPLS header is removed.

The gain from the MPLS header is that the intermediate routers do not have to repeat the packet-classification effort expended at the edge router; simple table lookup suffices. The DiffServ (Blake et al., 1998) proposal for QoS is actually similar in this sense. Classification is done at the edges to mark packets that deserve special quality of service. The only difference is that the classification information is used to mark the Type of Service (TOS) bits in the IP header, as opposed to an MPLS label. Both are examples of Principle **P10**, passing hints in protocol headers.

Despite MPLS and DiffServ, core routers still do classification at the very highest speeds. This is largely motivated by security concerns, for which it may be infeasible to rely on label switching. For example, Singh et al. (2004a) describe a number of core router classifiers, the largest of which contained 2000 rules.

12.4.6 Content-addressable memories

Recall from Chapter 11 that a CAM is a content-addressable memory, where the first cell that matches a data item will be returned using a parallel lookup in hardware. A ternary CAM allows each bit of data to be either a 0, a 1, or a wildcard. Clearly, ternary CAMs can be used for rule matching as well as for prefix matching. However, the CAMs must provide wide lengths—for example, the combination of the IPv4 destination, source, and two port fields is 96 bits.

Because of problems with algorithmic solutions described in the remainder of this chapter, there is a general belief that hardware solutions such as ternary CAMs are needed for core routers, despite the problems (Gupta and McKeown, 2001) of ternary CAMs. There are, however, several reasons to consider algorithmic alternatives to ternary CAMs, which were presented in Chapter 11.

Recall that these reasons include the smaller density and larger power of CAMs versus SRAMs and the difficulty of integrating forwarding logic with the CAM. These problems remain valid when considering CAMs for classification. An additional issue that arises is the *rule multiplication* caused by ranges. In CAM solutions, each range has to be replaced by a potentially large number of prefixes, thus causing extra entries. Some algorithmic solutions can handle ranges in rules without converting ranges to rules.

These arguments are strengthened by the fact that several CAM vendors have also considered algorithmic solutions, motivated by some of the difficulties with CAMs. While better CAM cell designs that reduce density and power requirements may emerge, it is still important to understand the corresponding advantages and disadvantages of algorithmic solutions. The remainder of the chapter is devoted to this topic. We will return to combinations of TCAMs and Algorithmic methods to get the best of both worlds at the end of the chapter.

12.5 Two-dimensional schemes

A useful problem-solving technique is first to solve a simpler version of a complex problem such as packet classification and to use the insight gained to solve the more complex problem. Since packet classification with just *one* field has been solved in Chapter 11, the next simplest problem is *two*-dimensional packet classification.

Two-dimensional rules may be useful in their own right. This is because large backbone routers may have a large number of destination–source rules to handle virtual private networks and multicast

Rule	Destination	Source
R_1	0*	10*
R_2	0*	01*
R_3	0*	1*
R_4	00*	1*
R_5	00*	11*
R_6	10*	1*
R_7	*	00*

FIGURE 12.3

An example with seven destination–source rules.

forwarding and to keep track of traffic between subnets. Further, as we will see, there is a heuristic observation that reduces the general case to the two-dimensional case.

Since there are only three distinct approaches to one-dimensional prefix matching—using tries, binary search on prefix lengths, and binary search on ranges—it is worth looking for generalizations of each of these distinct approaches. All three generalizations exist. However, this chapter will describe only the most efficient of these (the generalization of tries) in this section.

The appropriate generalization of standard prefix tries to two dimensions is called the *grid of tries*. The main idea will be explained using an example database of seven destination–source rules, shown in Fig. 12.3. We arrive at the final solution by first considering two naive variants.

12.5.1 Fast searching using set-pruning tries

Consider the two-dimensional rule set in Fig. 12.3. The simplest idea is first to build a trie on the destination prefixes in the database and then to hang a number of source tries off the leaves of the destination trie. Fig. 12.4 illustrates the construction for the rules in Fig. 12.3. Each valid prefix in the destination trie points to a trie containing some source prefixes. The question is: Which source prefixes should be stored in the source trie corresponding to each destination prefix?

For instance, consider $D = 00*$. Both rules R_4 and R_5 have this destination prefix, and so the trie at D clearly needs to store the corresponding source prefixes $1*$ and $11*$. But storing only these source prefixes is insufficient. This is because the destination prefix $0*$ in rules R_1, R_2, and R_3 also matches any destination that D matches. In fact, the wildcard destination prefix $*$ of R_7 also matches whatever D matches. This suggests that the source trie at $D = 00$ must contain the source prefixes for $\{R_1, R_2, R_3, R_4, R_5, R_7\}$, because these are the set of rules *whose destination is a prefix of D*.

Fig. 12.4 shows a schematic representation of this data structure for the database of Fig. 12.3. Note that $S1$ denotes the source prefix of rule R_1, $S2$ of rule R_2, and so on. Thus each prefix D in the destination trie *prunes* the set of rules from the entire set of rules down to the set of rules compatible

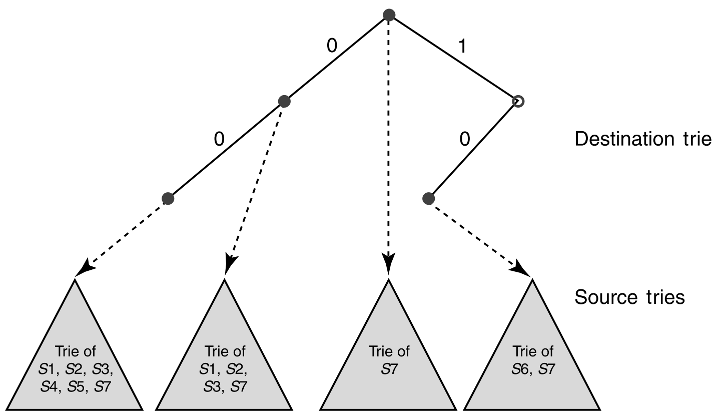


FIGURE 12.4

The set-pruning trie data structure in two dimensions corresponding to the database of Fig. 12.3. Destination trie is a trie for the destination prefixes. The nodes corresponding to a valid destination prefix in the database are shown as filled circles; others are shown as empty circle. Each valid destination prefix D has a pointer to a trie containing the source prefixes that belong to rules whose destination field is a prefix of D.

with D. The same idea can be extended to more than two fields, with each field value in the path pruning the set of rules further.

In this trie of tries, the search algorithm first matches the destination of the header in the destination trie. This yields the longest match on the destination prefix. The search algorithm then traverses the associated source trie to find the longest source match. While searching the source trie, the algorithms keep track of the lowest-cost matching rule. Since all rules that have a matching destination prefix are stored in the source trie being searched, the algorithm finds the correct least-cost rule. This is the basic idea behind set-pruning trees (Decasper et al., 1998).

Unfortunately, this simple extension of tries from one to two dimensions has a memory-explosion problem. The problem arises because a source prefix can occur in multiple tries. In Fig. 12.4 for instance, the source prefixes $S1$, $S2$, $S3$ appear in the source trie associated with $D = 00*$ as well as the trie associated with $D = 0*$.

How bad can this replication get? A worst-case example forcing roughly N^2 memory is created using the set of rules shown in Fig. 12.5. The problem is that since the destination prefix $*$ matches any destination header, each of the $N/2$ source prefixes is replicated $N/2$ times, one for each destination prefix. The example (see exercises) can be extended to show a $O(N^K)$ bound for general set-pruning tries in K dimensions.

While set-pruning tries do not scale to large classifiers, the natural extension to more than two fields has been used in Decasper et al. (1998) as part of a router toolkit, and in Malan and Jahanian (1998) as part of a flexible monitoring system. The performance of set-pruning tries is also studied in Qiu et al. (2001). One interesting optimization introduced in Decasper et al. (1998) and Malan and Jahanian (1998) is to avoid obvious waste (**P1**) when two subtries $S1$ and $S2$ have exactly the same contents. In this case, one can replace the pointers to $S1$ and $S2$ with a pointer to a common subtrie, S. This changes the structure from a tree to a directed acyclic graph (DAG). The DAG optimization can greatly reduce

Rule	Destination	Source
R_1	$D1$	*
R_2	$D2$	*
	⋮	
$R_{N/2}$	$D_{N/2}$	*
$R_{N/2+1}$	*	S_1
$R_{N/2+2}$	*	S_2
	⋮	
R_N	*	S_N

FIGURE 12.5

An example forcing $N^2/2$ memory for two-dimensional set-pruning trees. Similar examples, which apply to a number of other simple schemes, can be used to show $O(N^K)$ storage for K-dimensional rules.

storage for set-pruning tries (see Qiu et al., 2001 for other, related optimizations) and can be used to implement small classifiers, say, up to 100 rules, in software.

12.5.2 Reducing memory using backtracking

The previous scheme pays in memory in order to reduce search time. The dual idea is to pay with time in order to reduce memory. In order to avoid the memory blowup of the simple trie scheme, observe that rules associated with a destination prefix D are copied into the source trie of D' whenever D' is a prefix of D. For instance, in Fig. 12.4, the prefix $D = 00*$ has two rules associated with it: R_4 and R_5. The other rules, R_1, R_2, R_3, are copied into D's trie because their destination field $0*$ is a prefix of D.

The copying can be avoided by having each destination prefix D point to a source trie that stores the rules whose destination field is *exactly* D. This requires modifying the search strategy as follows: Instead of just searching the source trie for the best-matching destination prefix D, the search algorithm must now search the source tries associated with all *ancestors* of D.

In order to search for the least-cost rule, the algorithm first traverses the destination trie and finds the longest destination prefix D' matching the header. The algorithm then searches the source trie of D' and updates the least-cost-matching rule. Unlike set-pruning tries, however, the search algorithm is not finished at this point.

Instead, the search algorithm must now work its way back up the destination trie and search the source trie associated with every prefix of D' that points to a nonempty source trie.[3]

[3] Note that backtracking search can actually search the source tries corresponding to destination prefixes in any order; this particular order was used only to motivate the grid-of-tries scheme. Another search order that minimizes the state required for backtracking is described in Qiu et al. (2001).

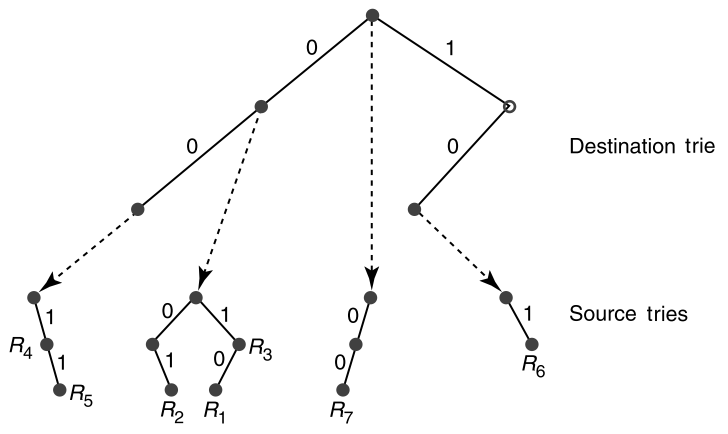


FIGURE 12.6

Avoiding the memory blowup by storing each rule in exactly one trie.

Since each rule now is stored exactly once, the memory requirement for the new structure is $O(NW)$, which is a significant improvement over the previous scheme. Unfortunately, the lookup cost for backtracking is worse than for set-pruning tries: In the worst case, the lookup costs $\Theta(W^2)$, where W is the maximum number of bits specified in the destination or source fields.

The $\Theta(W^2)$ bound on the search cost follows from the observation that, in the worst case, the algorithm may end up searching W source tries, each at the cost of $O(W)$, for a total of $O(W^2)$ time. For $W = 32$ and using 1-bit tries, this is 1024 memory accesses. Even using 4-bit tries, this scheme requires 64 memory accesses.

While backtracking can be very slow in the worst case, it turns out that all classification algorithms exhibit pathological worst-case behavior. For databases encountered in practice, backtracking can work very well. Qiu et al. (2001) describe experimental results using backtracking and also describe potential hardware implementations on pipelined processors.

12.5.3 The best of both worlds: grid of tries

The two naive variants of two-dimensional tries pay either a large price in memory (set-pruning tries) or a large price in time (backtracking search). However, a careful examination of backtracking search reveals obvious waste (**P1**), which can be avoided using precomputation (**P2a**).

To see the wasted time in backtracking search, consider matching the packet with destination address 001 and source address 001 in Fig. 12.6. The search in the destination trie gives $D = 00$ as the best match. So the backtracking algorithm starts its search for the matching source prefix in the associated source trie, which contains rules R_4 and R_5. However, the search immediately fails, since the first bit of the source is 0. Next, backtracking search backs up along the destination trie and *restarts* the search in the source trie of $D = 0*$, the parent of $00*$.

But backing up the trie is a waste because if the search fails after searching destination bits 00 and source bit 0, then any matching rule must be shorter in the destination (e.g., 0) and must contain all the

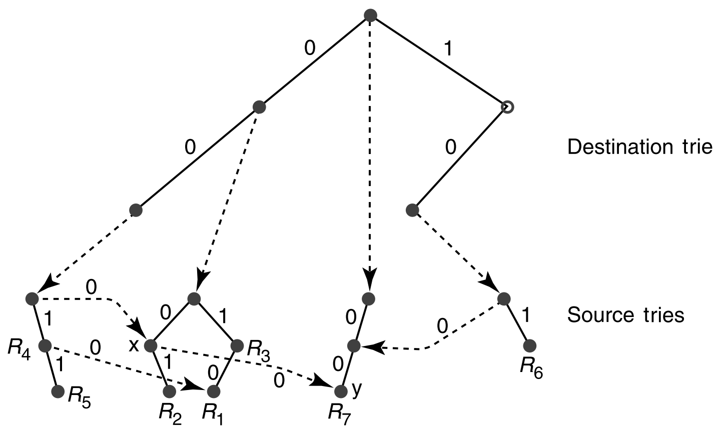


FIGURE 12.7

Improving the search cost with the use of switch pointers.

source bits searched so far, *including* the failed bit. Thus backing up to the source trie of $D = 0*$ and then traversing the source bit 0 to the parent of R_2 in Fig. 12.6 (as done in backtracking search) is a waste.

The algorithm could predict that this sequence of bits would be traversed when it first failed in the source trie of $D = 00$. This motivates a simple idea: Why not jump directly to the parent of R_2 from the failure point in the source trie of $D = 00*$?

Thus in the new scheme (Fig. 12.7), for each failure point in a source trie, the trie-building algorithm *precomputes* what we call a *switch pointer*. Switch pointers allow search to jump directly to the next possible source trie that can contain a matching rule. Thus in Fig. 12.7, notice that the source trie containing R_4 and R_5 has a dashed line labeled with 0 that points to a node x in the source trie containing $\{R_1, R_2, R_3\}$. All the dashed lines between source tries in Fig. 12.7 are switch pointers. Please distinguish the dashed switch pointers from the dotted lines that connect the destination and source tries.

Now consider again the same search for the packet with destination address 001 and source address 001 in Fig. 12.7. As before, the search in the destination trie gives $D = 00$ as the best match. Search fails in the corresponding source trie (containing R_4 and R_5) because the source trie contains a path only if the first source bit is a 1. However, in Fig. 12.7, instead of failing and backtracking, the algorithm follows the switch pointer labeled 0 directly to node x. It then continues matching from node x, without skipping a beat, using the remaining bits of the source.

Since the next bit of the source is a 0, the search in Fig. 12.7 fails again. The search algorithm once again follows the switch pointer labeled 0 and jumps to node y of the third source trie (associated with the destination prefix $*$). Effectively, the switch pointers allow skipping over all rules in the next ancestor source trie whose source fields are shorter than the current source match. This in turn improves the search complexity from $O(W^2)$ to $O(W)$.

It may help to define *switch pointers* more precisely. Call a destination string D' an *ancestor* of D if D' is a *prefix* of D. Call D' the *lowest ancestor* of D if D' is the *longest* prefix of D in the destination trie. Let $T(D)$ denote the source trie pointed to by D. Recall that $T(D)$ contains the source fields of exactly those rules whose destination field is D.

Let u be a node in $T(D)$ that *fails* on bit 0; that is, if u corresponds to the source prefix s, then the trie $T(D)$ has no string starting with $s0$. Let D'' be the *lowest* ancestor of D whose source trie contains a source string *starting with prefix* $s0$, say, at node v. Then we place a switch pointer at node u pointing to node v. If no such node v exists, the switch pointer is nil. The switch pointer for failure on bit 1 is defined similarly. For instance, in Fig. 12.7, the node labeled x fails on bit 0 and has a switch pointer to the node labeled y.

As a second example, consider the packet header (00∗, 10∗). Search starts with the first source trie, pointed to by the destination trie node 00∗. After matching the first source bit, 1, search encounters rule R_4. But then search fails on the second bit. Search therefore follows the switch pointer, which leads to the node in the second trie labeled with R_1. The switch pointers at the node containing R_1 are both nil, and so search terminates. Note, however, that search has missed the rule $R_3 = (0*, 1*)$, which also matches the packet header. While in this case R_3 has a higher cost than R_1, in general, the overlooked rule could have a lower cost.

Such problems can be avoided by having each node in a source trie maintain a variable *storedRule*. Specifically, a node v with destination prefix D and source prefix S stores in *storedRule*(v) the least-cost rule whose destination field is a prefix of D and whose source field is a prefix of S. With this precomputation, the node labeled with R_1 in Fig. 12.7 would store information about R_3 instead of R_1 if R_3 had a lower cost than R_1.

Finally, here is an argument that the search cost in the final scheme is at most $2W$. The time to find the best destination prefix is at most W. The remainder of the time is spent traversing the source tries. However, in each step, the length of the match on the source field increases by 1—either by traversing further down in the same trie or by following a switch pointer to an ancestral trie. Since the maximum length of the source prefixes is W, the total time spent in searching the source tries is also W. The memory requirement is $O(NW)$, since each of the N rules is stored only once, and each rule requires $O(W)$ space.

Note that k-bit tries (Chapter 11) can be used in place of 1-bit tries by expanding each destination or source prefix to the next multiple of k. For instance, suppose $k = 2$. Then, in the example of Fig. 12.7, the destination prefix 0∗ of rules R_1, R_2, R_3 is expanded to 00 and 01. The source prefixes of R_3, R_4, R_6 are expanded to 10 and 11. Using k-bit expansion, a single prefix can expand to 2^{k-1} prefixes. The total memory requirement grows from $2NW$ to $NW2^k/k$, and so the memory increases by the factor $2^{k-1}/k$. On the other hand, the depth of the trie reduces to W/k, and so the total lookup time becomes $O(W/k)$.

The bottom line is that by using multibit tries, the time to search for the best matching rule in an arbitrarily large two-dimensional database is effectively the time for two IP lookups.

Just as the grid of tries represents a generalization of familiar trie search for prefix matching, there is a corresponding generalization of binary search on prefix lengths (Chapter 11) that searches a database of two field rules in $2W$ hashes, where W is the length of the larger of the two fields. This is a big gap from the $\log W$ time required for prefix matching using binary search on prefix lengths. In the special case where the rules do not overlap, the search time reduces even further to $\log_2 W$, as shown in Suri et al. (2001). While these results are interesting theoretically, they seem to have less relevance to real routers, mostly because of the difficulties of implementing hashing in hardware.

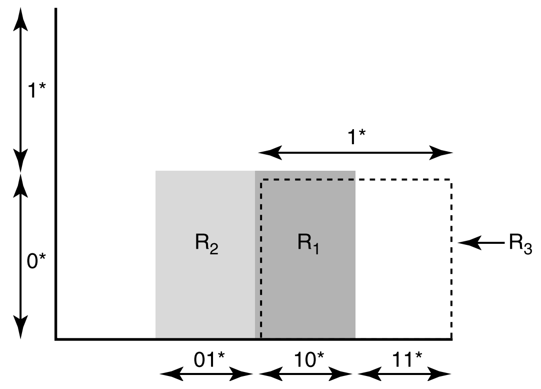


FIGURE 12.8

Geometric view of the first three rules, R_1, R_2, R_3, in the rule database of Fig. 12.3. For example, the rule $R_1 = 0*$, $10*$ is the box whose projection on the destination axis is the range corresponding to $0*$ and whose projection on the source axis is the range corresponding to $10*$. Note that because $R_3 = 0*$, $1*$ has the same destination range as R_1 and a source range that strictly includes the range of R_1, the dashed box, R_3, contains the box R_1.

12.6 Approaches to general rule sets

So far this chapter has concentrated on the special case of rules on just two header fields. Before moving to algorithms for rules with more than two fields, this section brings together some insights that inform the algorithms in later sections. Section 12.6.1 describes a geometric view of classification that provides visual insight into the problem. Section 12.6.2 utilizes the geometric viewpoint to obtain bounds on the fundamental difficulty of packet classification in the general case. Section 12.6.3 describes several observations about real rule sets that can be exploited to provide efficient algorithms that will be described in subsequent sections.

12.6.1 Geometric view of classification

A second problem-solving technique that is useful is to collect different viewpoints for the same problem. This section describes a *geometric* view of classification that was introduced by Lakshman and Stidialis (1998) and independently by Adisheshu (1998).

Recall from Chapter 11 that we can view a 32-bit prefix like $00*$ as a range of addresses from $000\ldots00$ to $001\ldots11$ on the number line from 0 to 2^{32}. If prefixes correspond to *line segments* geometrically, two-dimensional rules correspond to rectangles (Fig. 12.8), three-dimensional rules to cubes, and so on. A given packet header is a point. The problem of packet classification reduces to finding the lowest-cost box that contains the given point.

Fig. 12.8 shows the geometric view of the first three two-dimensional rules in Fig. 12.3. Destination addresses are represented on the y-axis and source addresses on the x-axis. In the figure, some sample prefix ranges are marked off on each axis. For example, the two halves of the y-axis are the prefix ranges $0*$ and $1*$. Similarly, the x-axis is divided into the four prefix ranges $00*$, $01*$, $10*$, and $11*$. To draw the box for a rule like $R_1 = 0*$, $10*$, draw the $0*$ range on the y-axis and the $10*$ range on the

x-axis, and extend the range lines to meet, forming a box. Multiple-rule matches, such as R_1 and R_2, correspond to overlapping boxes.

The first advantage of the geometric view is that it enables the application of algorithms from computational geometry. For example, Lakshman and Stidialis (1998) adapt a technique from computational geometry known as *fractional cascading* to do binary search for two-field rule matching in $O(\log N)$ time, where N is the number of rules. In other words, two-dimensional rule matching is asymptotically as fast as one-dimensional rule matching using binary search. This is consistent with the results for the grid of tries. The result also generalizes binary search on values for prefix searching as described in Chapter 11.

Unfortunately, the constants for fractional cascading are quite high. Perhaps this suggests that adapting existing geometric algorithms may actually not result in the most efficient algorithms. However, the second and main advantage of the geometric *viewpoint* is that it is suggestive and useful.

For example, the geometric view provides a useful metric, the number of disjoint (i.e., nonintersecting) *classification regions*. Since rules can overlap, this is not the number of rules. In two dimensions, for example, with N rules, one can create N^2 classification regions by having $N/2$ rules that correspond geometrically to horizontal strips together with $N/2$ rules that correspond geometrically to vertical strips. The intersection of the $N/2$ horizontal strips with the $N/2$ vertical strips creates $O(N^2)$ disjoint classification regions. For example, the database in Fig. 12.5 has this property. Similar constructions can be used to generate $O(N^K)$ regions for K-dimensional rules.

As a second example, the database of Fig. 12.8 has four classification regions: the rule R_1, the rule R_2, the points in R_3 not contained in R_1, and all points not contained in R_1, R_2, or R_3. We will use the number of classification regions later to characterize the complexity of a given classifier or rule database.

12.6.2 Beyond two dimensions: the bad news

The success of the grid of tries may make us optimistic about generalizing to larger dimensions. Unfortunately, this optimism is misplaced; either the search time or the storage blows up exponentially with the number of dimensions K for $K > 2$.

Using the geometric viewpoint just described, it is easy to adapt a lower bound from computational geometry. Thus, it is known that general multidimensional range searching over N ranges in K dimensions requires $\Omega((\log N)^{K-1})$ worst-case time if the memory is limited to about linear size (Chazelle, 1990a,b) or requires $O(N^K)$ size memory. While $\log N$ could be reasonable (say, 10 memory accesses), $\log^4 N$ ($K = 5$ in the case of packet classification over source IP, destination IP, source port number, destination port number, and protocol) will be very large (say, 10,000 memory accesses). Notice that this lower bound is consistent with solutions for the two-dimensional cases that take linear storage but are as fast as $O(\log N)$.

The lower bound implies that for perfectly general rule sets, *algorithmic approaches to classification require either a large amount of memory or a large amount of time.* Unfortunately, classification at high speeds, especially for core routers, requires the use of limited and expensive SRAM. Thus the lower bound seems to imply that content address memories are required for reasonably sized classifiers (say, 10,000 rules) that must be searched at high speeds (e.g., OC-768 speeds).

12.6.3 Beyond two dimensions: the good news

The previous subsection may have left the reader wondering whether there is any hope left for algorithmic approaches to packet classification in the general case. Fortunately, real databases have more *structure*, which can be exploited to efficiently solve multidimensional packet classification using algorithmic techniques.

The *good* news about packet classification can be articulated using four observations. Subsequent sections describe a series of heuristic algorithms, all of which do very badly in the worst case but quite well on databases that satisfy one or more of the assumptions.

The expected case can be characterized using four observations drawn from a set of firewall databases studied in Srinivasan et al. (1998) and Gupta and McKeown (1999a) (and not from publically available lookup tables as in the previous chapter). The first is identical to an observation made in Chapter 11 and repeated here. The observations are numbered starting from *O2* to be consistent with observation *O1* made in the lookup chapter.

O2: *Prefix containment is rare*. It is somewhat rare to have prefixes that are prefixes of other prefixes, as, for example, the prefixes 00* and 0001*. In fact, the maximum number of prefixes of a given prefix in lookup tables and classifiers is seven.

O3: *Many fields are not general ranges*. For the destination and source port fields, most rules contain either specific port numbers (e.g., port 80 for Web traffic), the wildcard range (i.e., $*$), or the port ranges that separate server ports from client ports (1024 or greater and less than 1024). The protocol field is limited to either the wildcard or (more commonly) TCP, UDP. This field also rarely contains protocols such as IGMP and ICMP. While other TCP fields are sometimes referred to, the most common reference is to the ACK bit.

O4: *The number of disjoint classification regions is small*. This is perhaps the most interesting observation. Harking back to the geometric view, the lower bounds in Chazelle (1990a) depend partly on the worst-case possibility of creating N^K classification regions using N rules. Such rules require either N^K space or a large search time. However, Gupta and McKeown (1999a), after an extensive survey of 8000 rule databases, show that the number of classification regions is much smaller than the worst case. Instead of being exponential in the number of dimensions, the number of classification regions is linear in N, with a small constant.

O5: *Source–Destination matching*: In Singh et al. (2004b), several core router classifiers used by real ISPs are analyzed and the following interesting observation is made. Almost all packets match at most five distinct source–destination values found in the classifier. No packet matched more than 20 distinct source–destination pairs. This is a somewhat more refined observation than *O4* because it says that the number of classification regions is small, even when projected only to the source and destination fields. By "small," we mean that the number of regions grows much more slowly than N, the size of the classifier.

12.7 Extending two-dimensional schemes

The simplest general scheme uses observation *O5* to trivially extend any efficient 2D scheme to multiple dimensions. A number of algorithms simply use linear search to search through all possible rules. These scales well in storage but poorly in time. The source–destination matching observation leads to a very

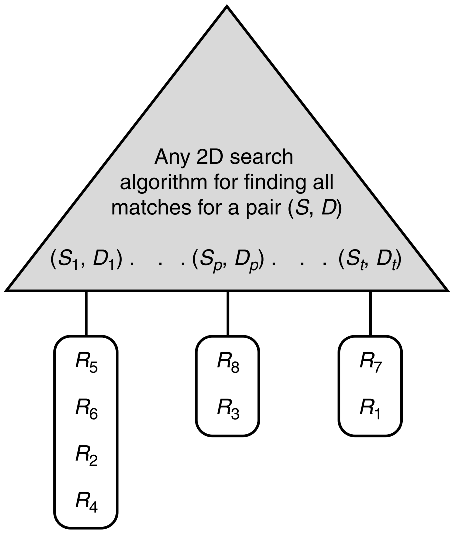

Any 2D search algorithm for finding all matches for a pair (S, D)
(S1, D1) . . . (Sp, Dp) . . . (St, Dt)
R5
R6
R2
R4
R8
R3
R7
R1

FIGURE 12.9

Extending two-dimensional schemes.

simple idea depicted in Fig. 12.9. Use source–destination address matching to reduce linear searching to just the rules corresponding to source–destination prefix pairs in the database that match the given packet header. Alternately, one can use source-destination matching to reduce power in TCAMs as in SmartPC (Ma and Banerjee, 2012) as we will see later.

By observation *O5*, at most 20 rules match any packet when considering only the source and destination fields. Thus pruning based on source–destination fields will reduce the number of rules to be searched to less than 20, compared to searching the entire database. For example, Singh et al. (2004a) describe a database with 2800 rules used by a large ISP.

Thus in Fig. 12.9 the general idea is to use any efficient two-dimensional matching scheme to find *all* distinct source–destination prefix pairs $(S_1, D_1) \dots (S_t, D_t)$ that match a header. For each distinct pair (S_i, D_i), there is a linear array or list with all rules that contain (S_i, D_i) in the source and destination fields. Thus in the figure, the algorithm has to traverse the list at (S_1, D_1), searching through all the rules for R_5, R_6, R_2, and R_4. Then the algorithm moves on to consider the lists at (S_2, D_2), and so on.

This structure has two important advantages:

- Each rule is represented only once without replication. However, one may wish to replicate rules to reduce search times even further.
- The port range specifications stay as ranges in the individual lists without the associated blowup associated with range translation in, say, CAMs.

Since the grid-of-tries implementation described earlier is one of the most efficient two-dimensional schemes in the literature, it is natural to instantiate this general schema by using a grid of tries as the two-dimensional algorithm in Fig. 12.9.

Unfortunately, it turns out that there is a delicacy about extending the grid of tries. In the grid of tries, whenever one rule, R, is at least as specific in all fields as a second rule, R', rule R' precomputes its matching directive to be that of R if R is the lower cost of the two rules. This allows the traversal through the grid of tries to safely skip rule R when encountering rule R'. While this works correctly with two-field rules, it requires some further modifications to handle the general case.

One solution, equivalent to precomputing rule costs, is to precompute the list for R' to include all the list elements for R. Unfortunately, this approach can increase storage because each rule is no longer represented exactly once. A more sophisticated solution, called the *extended grid of tries* (EGT) and described in Singh et al. (2004b), is based on extra traversals beyond the standard grid of tries.

The performance of EGT can be described as follows.

Assumption: The extension of two-dimensional schemes depends critically on observation *O5*.

Performance: The scheme takes at least one grid-of-tries traversal plus the time to linearly search c rules, where c is the constant embodied in observation *O5*. Assuming linear storage, the search performance can increase (Singh et al., 2004b) by an additive factor representing the time to search for less specific rules. The addition of a new rule R requires only rebuilding of the individual two-dimensional structure of which R is a part. Thus rule update should be fairly fast.

12.8 Using divide-and-conquer

The next three schemes (bit vector linear search, on-demand cross-producting, and equivalenced cross-producting) all exploit the simple algorithmic idea (**P15**) of divide-and-conquer. Divide-and-conquer refers to dividing a problem into simpler pieces and then efficiently combining the answers to the pieces. We briefly motivate a skeletal framework of this approach in this section. The next three sections will flesh out specific instantiations of this framework.

Chapter 11 has already outlined techniques to do lookups on individual fields. Given this background, the common idea in all three divide-and-conquer algorithms is the following. Start by slicing the rule database into columns, with the ith column storing all distinct prefixes (or ranges) in field i. Then, given a packet P, determine the best-matching prefix (or narrowest-enclosing range) for each of its fields separately. Finally, combine the results of the best-matching-prefix lookups on individual fields. The main problem, of course, lies in finding an efficient method for combining the lookup of individual fields into a single compound lookup.

All the divide-and-conquer algorithms conceptually start by slicing the database of Fig. 12.2 into individual prefix fields. In the sliced columns, from now on, we will sometimes refer to the wildcard character $*$ by the string *default*. Recall that the mail gateway M and internal NTP agent TI are full IP addresses that lie within the prefix range of Net. The sliced database corresponding to Fig. 12.2 is shown in Fig. 12.10.

Clearly, any divide-and-conquer algorithm starts by doing an individual lookup in each column and then combines the results. The next three sections show that each of the three schemes returns different results with lookup and follows different strategies to combine the individual field results, despite using the same sliced database shown in Fig. 12.10.

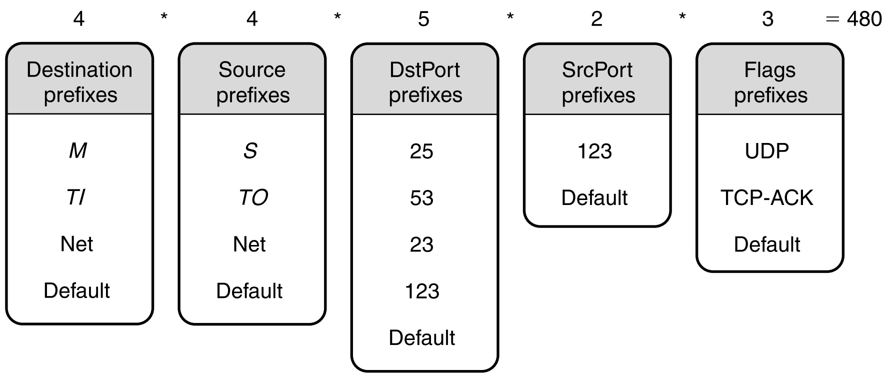


FIGURE 12.10

The database of Fig. 12.2 “sliced” into columns where each column contains the set of prefixes corresponding to a particular field.

12.9 Bit vector linear search

Consider doing a match in one of the individual columns in Fig. 12.10, say, the destination address field, and finding a bit string S as the longest match. Clearly, this lookup result eliminates any rules that *do not* match S in this field. Then the search algorithm can do a linear search in the set of all remaining rules that match S. The logical extension is to perform individual matches in each field; each field match will prune away a number of rules, leaving a remaining set. The search algorithm needs to search only the *intersection* of the remaining sets obtained by each field lookup.

This would clearly be a good heuristic for optimizing the average case if the remaining sets are typically small. However, one can guarantee performance even in the worst case (to some extent) by representing the remaining sets as bitmaps and by using wide memories to retrieve a large number of set members in a single memory access (**P4a**, exploit locality).

In more detail, as in Section 12.8, divide-and-conquer is used to slice the database, as in Fig. 12.10. However, in addition with each possible value M of field i, the algorithm stores the set of rules $S(M)$ that match M in field i as a bit vector. This is easy to do when building the sliced table. The algorithm that builds the data structure scans through the rules linearly to obtain the rules that match M using the match rule (e.g., exact, prefix, or range) specified for the field.

For example, Fig. 12.11 shows the sliced database of Fig. 12.10 together with bit vectors for each sliced field value. The bit vector has 8 bits, one corresponding to each of the eight possible rules in Fig. 12.2. Bit j is set for value M in field i if value M matches Rule j in field i.

Consider the destination prefix field and the first value M in Fig. 12.11. If we compare it to Fig. 12.2, we see that the first four rules specify M in this field. The fifth rule specifies TI (which does not match M), and the sixth and eighth rules specify a wildcard (which matches M). Finally, the seventh rule specifies the prefix Net (which matches M, because Net is assumed to be the prefix of the company network in which M is the mail gateway). Thus the bitmap for M is 11110111, where the only bit not set is the fifth bit. This is because the fifth rule has TI, which does not match M.

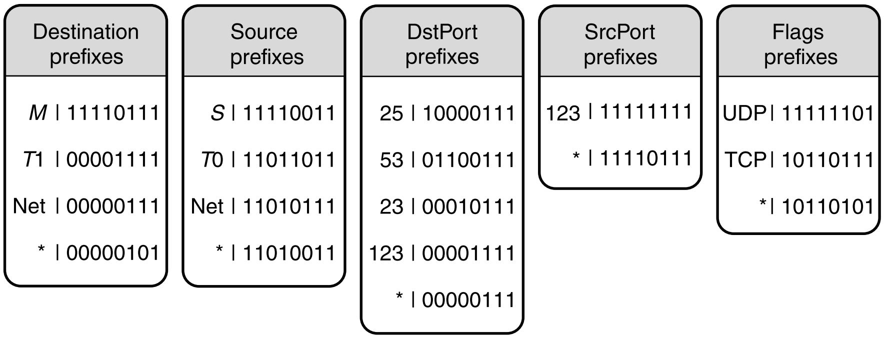


FIGURE 12.11

The sliced database of Fig. 12.10 together with bit vectors for every possible sliced value. The bit vector has 8 bits, one corresponding to each of the eight possible rules in Fig. 12.2. Bit j is set for value M in field i if value M matches Rule j in field i.

When a packet header arrives with fields $H[1] \dots H[K]$, the search algorithm first performs a longest-matching-prefix lookup in each field i to obtain matches M_i and the corresponding set $S(M_i)$ of matching rules. The search algorithm then proceeds to compute the intersection of all the sets $S(M_i)$ and returns the lowest-cost element in the intersection set.

But if rules are arranged in nondecreasing order of cost and all sets are bitmaps, then the intersection set is the AND of all K bitmaps. Finally, the lowest-cost element corresponds to the index of the first bit set in the intersection bitmap. But, the reader may object, since there are N rules, the intersected bitmaps are N bits long. Hence, computing the AND requires $O(N)$ operations. So the algorithm is effectively doing a linear search after slicing and doing individual field matches. Why not do simple linear search instead?

The reason is subtle and requires a good grasp of models and metrics. Basically, the preceding argument above is correct but ignores the large constant-factor improvement that is possible using bitmaps. Thus computing the AND of K bit vectors and searching the intersection bit vector is still an $O(K \cdot N)$ operation; however, the constants are much lower than doing naive linear search because we are dealing with bitmaps. Wide memories (**P4a**) can be used to make these operations quite cheap, even for a large number of rules.

This is because the cost in memory accesses for these bit operations is $N \cdot (K + 1)/\mathrm{W}$ memory accesses, where W is the width of a memory access. Even with $\mathrm{W} = 32$, this brings down the number of memory accesses by a factor of 32. A specialized hardware classification chip can do much better. Using wide memories and wide buses (the bus width is often the limiting factor), a chip can easily achieve $\mathrm{W} = 1000$ with today's technology. As technology scales, one can expect even larger memory widths.

For example, using $\mathrm{W} = 1000$ and $k = 5$ fields, the number of memory accesses for 5000 rules is $5000 * 6/1000 = 30$. Using 10-nanosecond SRAM, this allows a rule lookup in 300 nanoseconds, which is sufficient to process minimum-size (40-byte) packets at wire speed on a gigabit link. By using K-fold parallelism, the further factor of $K + 1$ can be removed, allowing 30,000 rules. Of course, even linear search can be parallelized, using N-way parallelism; what matters are the amount of parallelism that can be employed at a reasonable cost.

Using our old example, consider a lookup for a packet to M from S with UDP destination port equal to 53 and source port equal to 1029 in the database of Fig. 12.2, as represented by Fig. 12.11. This packet matches Rules 2, 3, and 8 but must be allowed through because the first matching rule is Rule 2.

Using the bit vector algorithm just described (see Fig. 12.11), the longest match in the destination field (i.e., M) yields the bitmap 11110111. The longest match in the source field (i.e., S) yields the bitmap 11110011. The longest match in the destination port field (i.e., 53) yields the bitmap 01100111. The longest match in the source port field (i.e., the wildcard) yields the bitmap 11110111; the longest match in the protocol field (i.e., UDP) yields the bitmap 11111101. The AND of the five bitmaps is 01100001. This bitmap corresponds to matching Rules 2, 3, and 8. The index of the first bit set is 2. This corresponds to the second rule, which is indeed the correct match.

The bit vector algorithm was described in detail in Lakshman and Stidialis (1998) and also in a few lines in a paper on network monitoring (Malan and Jahanian, 1998). The first paper (Lakshman and Stidialis, 1998) also describes some trade-offs between search time and memory. A later paper (Baboescu and Varghese, 2001) shows how to add more state for speed (**P12**) by using summary bits. For every W bits in a bitmap, the summary is the OR of the bits. The main intuition is that if, say, W^2 bits are zero, this can be ascertained by checking W summary bits.

The bit vector scheme is a good one for moderate-size databases. However, since the heart of the algorithm relies on linear search, it cannot scale to both very large databases and very high speeds.

The performance of this scheme can be described as follows.

Assumption: The number of rules will stay reasonably small or will grow only in proportion to increases in bus width and parallelism made possible by technology improvements.

Performance: The number of memory accesses is $N \cdot (K + 1)/\mathrm{W}$ plus the number of memory accesses for K longest-matching-prefix or narrowest-range operations. The memory required is that for the K individual field matches (see schemes in Chapter 11) plus potentially $N^2 K$ bits. Recall that N is the number of rules, K is the number of fields, and W is the width of a memory access. Updating rules is slow and generally requires rebuilding the entire database.

12.10 Cross-producting

This section describes a crude scheme called cross-producting (Srinivasan et al., 1998). In the next section, we describe a crucial refinement we call *equivalenced cross-producting* (but called RFC by the authors (Gupta and McKeown, 1999a)) that makes cross-producting more feasible. The top of each column in Fig. 12.10 indicates the number of elements in the column. Consider a 5-tuple, formed by taking one value from each column. Call this a *cross product*. Altogether, there are $4 * 4 * 5 * 2 * 3 = 480$ possible cross products. Some sample cross products are shown in Fig. 12.12. Considering the destination field to be most significant and the flags field to be least significant, and pretending that values increase down a column, cross products can be ordered from the smallest to the largest, as in any number system.

A key insight into the utility of cross products is as follows.

Number	Cross product	Matching rule
1	*M*, *S*, 25, 123, UDP	Rule 1
2	*M*, *S*, 25, 123, TCP-ACK	Rule 1
3	*M*, *S*, 25, 123, *default*	Rule 1
4	*M*, *S*, 25, *default*, UDP	Rule 1
5	*M*, *S*, 25, *default*, TCP-ACK	Rule 1
6	*M*, *S*, 25, *default*, *default*	Rule 1
⋮	⋮ ⋮ ⋮	⋮
479	*default*, *default*, *default*, *default*, TCP-ACK	Rule 8
480	*default*, *default*, *default*, *default*, *default*	Rule 8

FIGURE 12.12

A sample of the cross products obtained by cross-producting the individual prefix tables of Fig. 12.10.

Given a packet header H, if the longest-matching-prefix operation for each field $H[i]$ is concatenated to form a cross product C, then the least-cost rule matching H is identical to the least-cost rule matching C.

Suppose this were not true. Since each field in C is a prefix of the corresponding field in H, every rule that matches C also matches H. Thus the only case in which H has a different matching rule is if there is some rule R that matches H but not C. This implies that there is some field i such that $R[i]$ is a prefix of $H[i]$ but not of $C[i]$, where $C[i]$ is the contribution of field i to cross product C. But since $C[i]$ is a prefix of $H[i]$, this can happen only if $R[i]$ is longer than $C[i]$. But that contradicts the fact that $C[i]$ is the longest-matching prefix in column/field i.

Thus, the basic cross-producting algorithm (Srinivasan et al., 1998) builds a table of all possible cross products and *precomputes* the least-cost rule matching each cross product. This is shown in Fig. 12.12. Then, given a packet header, the search algorithm can determine the least-cost matching rule for the packet by performing K longest-matching-prefix operations, together with a single hash lookup of the cross-product table. In hardware, each of the K prefix lookups can be done in parallel.

Using our example, consider matching a packet with header $(M, S, \text{UDP}, 53, 57)$ in the database of Fig. 12.2. The cross product obtained by performing best-matching prefixes on individual fields is $(M, S, \text{UDP}, 53, \textit{default})$. It is easy to check that the precomputed rule for this cross product is Rule 2—although Rules 3 and 8 also match the cross product, Rule 2 has the least cost.

The naive cross-producting algorithm suffers from a memory explosion problem: In the worst case, the cross-product table can have N^K entries, where N is the number of rules and K is the number of fields. Thus, even for moderate values, say, $N = 100$ and $K = 5$, the table size can reach 10^{10}, which is prohibitively large.

One idea to reduce memory is to build the cross products on demand (**P2b**, lazy evaluation) (Srinivasan et al., 1998): Instead of building the complete cross-product table at the start, the algorithm incrementally adds entries to the table. The prefix tables for each field are built as before, but the cross-product table is initially empty. When a packet header H arrives, the search algorithm performs longest-matching prefixes on the individual fields to compute a cross-product term C.

If the cross-product table has an entry for C, then of course the associated rule is returned. However, if there is no entry for C in the cross-product table, the search algorithm finds the best-matching rule for C (possibly using a linear search of the database) and inserts that entry into the cross-product table. Of course, any subsequent packets with cross product C will yield fast lookups.

On-demand cross-producting can improve both the building time of the data structure and its storage cost. In fact, the algorithm can treat the cross-product table as a cache and remove all cross products that have not been used recently. Caching based on cross products can be more effective than full header caching because a single cross product can represent multiple headers (see Exercises). However, a more radical improvement of cross-producting comes from the next idea, which essentially aggregates cross products into a much smaller number of equivalence classes.

12.11 Equivalenced cross-producting

Gupta and McKeown (1999a) have invented a scheme called *recursive flow classification* (RFC), which is an improved form of cross-producting that significantly compresses the cross-product table, at a slight extra expense in search time. We prefer to call their scheme *equivalenced cross-producting,* for the following reason. The scheme works by building larger cross products from smaller cross products; the main idea is to place the smaller cross products into equivalence classes before combining them to form larger cross products. This equivalencing of partial cross products considerably reduces memory requirements, because several original cross-product terms map into the same equivalence class.

Recall that in simple cross-producting when a header H arrives, the individual field matches are immediately concatenated to form a cross product that is then looked up in a cross-product table. By contrast, equivalenced cross-producting builds the final cross product in several pairwise combining steps instead of in one fell swoop.

For example, one could form the destination–source cross product and separately form the destination port–source port cross product. Then, a third step can be used to combine these two cross products into a cross product on the first four fields, say, C'. A fourth step is then needed to combine C' with the protocol field to form the final cross product, C. The actual combining sequence is defined by a combining tree, which can be chosen to reduce overall memory.

Just forming the final cross product in several pairwise steps does not reduce memory below N^K. What does reduce memory is the observation that when two partial cross products are combined, many of these pairs are equivalent: Geometrically, they correspond to the same region of space; algebraically, they have the same set of compatible rules.

Thus the main trick is to give each class a class number and to form the larger cross products using the *class numbers* instead of the original matches. Since the algebraic view is easier for computation, we will describe an example of equivalencing using the first two columns of Fig. 12.10 under the algebraic view.

Destination–source prefix pairs	Rule bitmap	Class number
M, S	11110011	C1
M, T0	11010011	C2
M, Net	11010111	C3
*M, **	11010011	C2
T1, S	00000011	C4
T1, T0	00001011	C5
T1, Net	00000111	C6
*T1, **	00000011	C4
Net, S	00000011	C4
Net, T0	00000011	C4
Net, Net	00000111	C6
*Net, **	00000011	C4
**, S*	00000001	C7
**, T0*	00000001	C7
**, Net*	00000101	C8
*, *	00000001	C7

FIGURE 12.13

Forming the partial cross products of the first two columns in Fig. 12.10 and then assigning these cross products into the same equivalence class if they have the same rule set (rule bitmap). Notice that 16 partial cross products form only eight classes.

Fig. 12.13 shows the partial cross products formed by only the destination and source columns in Fig. 12.10. For each pair (e.g., M, S), we compute the set of rules that are compatible with such a pair of matches exactly, as in the bit vector linear search scheme. In fact, we can find the bit vector of any pair, such as M, S, by taking the intersection of the rule bitmaps for M and S in Fig. 12.11. Thus from Fig. 12.11, since the rule bitmap for M is 11110111 and the bitmap for S is 11110011, the intersection bitmap for M, S is 11110011, as shown in Fig. 12.13.

Doing this for each possible pair, we soon see that several bitmaps repeat themselves. For example, M, TO, and $M, *$ (second and fourth entries in Fig. 12.13) have the same bitmap. Two rules that have the same bitmap are assigned to the same equivalence class, and each class is given a class number. Thus in Fig. 12.13, the classes are numbered starting with 1; the table-building algorithm increments

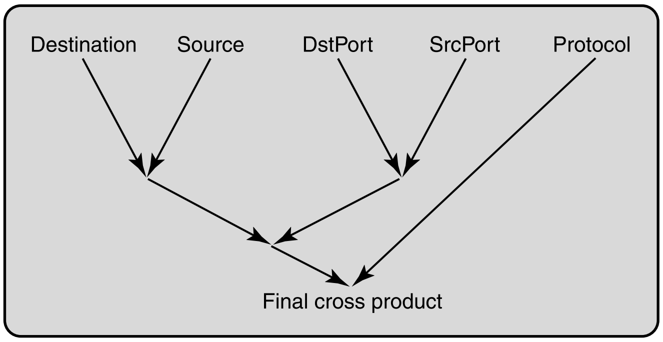


FIGURE 12.14

The combining tree used in the example.

the class number whenever it encounters a new bitmap. Thus, there are only eight distinct class numbers, compared to 16 possible cross products, because there are only eight distinct bitmaps.

Now assume we combine the two port columns to form six classes from 10 possible cross products. When we combine the port pairs with the destination–source pairs, we combine all possible combinations of the destination–source and port pair class numbers and not the original field matches. Thus after combining all four columns, we get $6 * 8 = 48$ cross products. Note that in Fig. 12.10, naive cross-producting will form $4 * 4 * 5 * 2 = 160$ cross products from the first four columns. Thus we have saved a factor of nearly 3 in memory.

Of course, we do not stop here. After combining the destination–source and port pair class numbers, we equivalence them again using the same technique. When combining class number C with class number C', the bitmap for C, C' is the intersection of the bitmaps for C and C'. Once again, pairs with identical bitmaps are equivalenced into groups. After this is done, the final cross product is formed by combining the classes corresponding to the first four columns with the matches in the fifth column.

Our example combined fields 1 and 2, then fields 3 and 4, and then the first four and finally combined in the fifth (Fig. 12.14). Clearly, other pairings are possible, as defined by a binary tree with the fields as nodes and edges representing pairwise combining steps. One could choose the optimal combining tree to reduce memory.

The search process is similar to cross-producting, except the cross products are calculated pairwise (just as they are built) using the same tree. Each pairwise combining uses the two class numbers as input into a table that outputs the class number of the combination. Finally, the class number of the root of the tree is looked up in a table to yield the best-matching rule. Since each class has the same set of matching rules, it is easy to precompute the lowest-cost matching rule for the final classes. Note that the search process does not need to access the rule bitmaps, as is needed for the bit vector linear search scheme. The bitmaps are used only to *build* the structure.

Clearly, each pairwise combining step can take $O(N^2)$ memory because there can be N distinct field values in each field. However, the total memory falls very short of the N^K worst-case memory for real rule databases. To see why this might be the case, we return to the geometric view.

Using a survey of 8000 rule databases, Gupta and McKeown (1999a) observe that all databases studied have only $O(N)$ classification regions, instead of the N^K worst-case number of classification regions. It is not hard to see that when the number of classification regions is N^K, then the number of cross products in the equivalenced scheme and in the naive scheme is also N^K.

But when the number of classification regions is linear, equivalenced cross-producting can do better. However, it is possible to construct counterexamples where the number of classification regions is linear, but equivalenced cross-producting takes exponential memory. Despite such potentially pathological cases, the performance of RFC can be summarized as follows.

Assumption: There is a series of subspaces of the complete rule space (as embodied by nodes in the combining tree) that all have a linear number of classification regions. Note that this is stronger than *O4* and even *O5*. For example, if we combine two fields i and j first, we require that this intermediate two-dimensional subspace have a linear number of regions.

Performance: The memory required is $O(N^2) * T$, where T is the number of nodes in the combining tree. The sequential performance (in terms of time) is $O(T)$ memory accesses, but the time required in a parallel implementation can be $O(1)$ because the tree can be pipelined. Note that the $O(N^2)$ memory is still very large in practice and would preclude the use of SRAM-based solutions.

12.12 Decision tree approaches

This chapter ends with a description of a very simple scheme that performs well in practice, better even than RFC and comparable to or better than the extended grid of tries. This scheme was introduced by Woo (2000). A similar idea, with range tests replacing bit tests, was independently described by Gupta and McKeown (1999b).

The basic idea is extremely close to the simple set-pruning tries described in Section 12.5.1, with the addition of some important degrees of freedom. Recall that set-pruning tries work one field at a time; thus in Fig. 12.7, the algorithm tests *all* the bits for the destination address before testing *all* the bits for the source address. The extension to multiple fields in Decasper et al. (1998) similarly tests all the bits of one field before moving on to another field. The set-pruning trie can be seen as an instance of a general decision tree.

Clearly, an obvious degree of freedom (**P13**) not considered in set-pruning tries is to arbitrarily interleave the bit tests for all fields. Thus the root of the trie could test for (say) bit 15 of the source field; if the bit is 0, this could lead to a node that tests for, say, bit 22 of the port number field. Clearly, there is an exponential number of such decision trees. The schemes in Woo (2000) and Gupta and McKeown (1999b) build the final decision tree using *local optimization* decisions at each node to choose the next bit to test. A simple criterion used in Gupta and McKeown (1999b) is to balance storage and time.

A second important degree of freedom considered in Woo (2000) is to use multiple decision trees. For example, for examples such as Fig. 12.5, it may help to place all the rules with wildcards in the source field in one tree and the remainder in a second tree. While this can increase overall search time, it can greatly reduce storage.

A third degree of freedom exploited in both Woo (2000) and Gupta and McKeown (1999b) is to allow a small amount of linear searching after traversing the decision tree. This is similar to the common strategy of using an insert. Consider a decision tree with 10,000 leaves where each leaf is associated with one of four rules. While it may be possible to distinguish these four rules by lengthening the decision tree in height, this lengthened decision tree could add 40,000 extra nodes of storage.

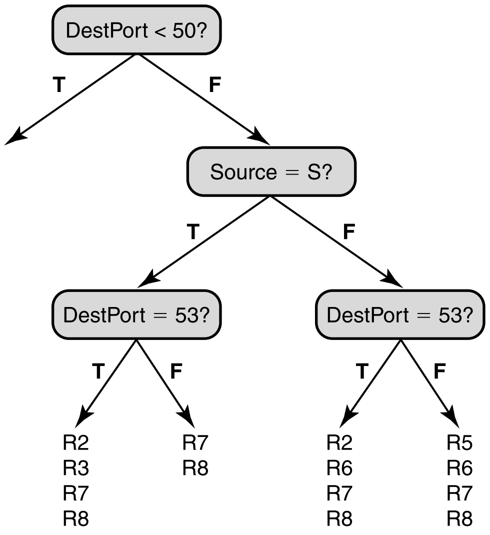


FIGURE 12.15

The HiCuts data structure is essentially a range tree that has pointers corresponding to some ranges of some dimension variable with linear search at the end.

Thus, in balancing storage with time, it may be better to settle for a small amount of linear searching (e.g., among one of four possible rules) at the end of tree search. Intuitively, this can help because the storage of a tree can increase exponentially with its height. Reducing the height by employing some linear search can greatly reduce storage.

The hierarchical cuttings (HiCuts) scheme described in Gupta and McKeown (1999b) is similar in spirit to that in Woo (2000) but uses range checks instead of bit tests at each node of the decision tree. Range checks are slightly more general than bit tests because a range check such as $10 < D < 35$ for a destination address D cannot be emulated by a bit test. A range test (cut) can be viewed geometrically in two dimensions as a line in either dimension that splits the space into half; in general, each range cut is a hyperplane.

In what follows, we describe HiCuts in more detail using an example. The HiCuts local optimization criterion works well when tested on real core router classifiers.

Fig. 12.15 shows a fragment of a HiCuts decision tree on the database of Fig. 12.2. The nodes contain range comparisons on values of any specified fields, and the edges are labeled **T**rue or **F**alse. Thus the root node tests whether the destination port field is less than 50. The fragment follows the case only when this test is false. Notice in Fig. 12.2 that this branch eliminates $R1$ (i.e., Rule 1) and $R4$, because these rules contain port numbers 25 and 23, respectively.

The next test checks whether the source address is equal to that of the secondary name server S in Fig. 12.2. If this test evaluates to true, then $R5$ is eliminated (because it contains TO), and so is $R6$ (because it contains Net and because S does not belong to the internal prefix Net). This leads to a second test on the destination port field. If the value is not 53, the only possible rules that can match are $R7$ and $R8$.

Thus on a packet header in which the destination port is 123 and the source is S, the search algorithm takes the right branch at the root, the left branch at the next node, and a right branch at the final node. At this point, the packet header is compared to rules $R7$ and $R8$ using linear search. Note that, unlike set pruning trees, the HiCuts decision tree of Fig. 12.15 uses ranges, interleaves the range checks between the destination port and source fields, and uses linear searching.

Of course, the real trick is to find a way to build an efficient decision tree that minimizes the worst-case height and yet has reasonable storage. Rather than consider the general optimization problem, which is NP-complete, HiCuts (Gupta and McKeown, 1999b) uses a more restricted heuristic based on the repeated application of the following greedy strategy.

- *Pick a field:* The HiCuts paper suggests first picking a field to cut on at each stage based on the number of distinct field values in that field. For example, in Fig. 12.15, this heuristic would pick the destination port field.
- *Pick the number of cuts:* For each field, rather than just pick one range check as in Fig. 12.15, one can pick k ranges or cuts. Of course, these can be implemented as separate range checks, as in Fig. 12.15. To choose k, the algorithm suggested in Gupta and McKeown (1999a) is to keep doubling k and to stop when the storage caused by the k cuts exceeds a prespecified threshold.

Several details are needed to actually implement this somewhat general framework. Assuming the cuts or ranges are equally spaced, the storage cost of k cuts on a field is estimated by counting the sum of the rules assigned to each of the k cuts. Clearly, cuts that cause rule replication will have a large storage estimate. The threshold that defines acceptable storage is a constant (called *spfac*, for space factor) times the number of rules at the node. The intent is to keep the storage linear in the number of rules up to a tunable constant factor.

Finally, the process stops when all decision tree leaves have no more than *binth* (bin threshold) rules. *binth* controls the amount of linear searching at the end of tree search.

The HiCuts paper (Gupta and McKeown, 1999b) mentions the use of the DAG optimization. A more novel optimization, described in Woo (2000) and Gupta and McKeown (1999b), is to eliminate a rule, R, that completely overlaps another rule, R', at a node but has a higher cost. There are also several further degrees of freedom (**P13**) left unexplored in Gupta and McKeown (1999b) and Woo (2000): unequal-size cuts at each node, more sophisticated strategies that pick more than field at a time, and linear searching at nodes other than the leaves.

HiCuts has inspired several follow-up papers that make improvements to the basic idea. First, in HyperCuts (Singh et al., 2004a) the decision tree approach is taken a step further by allowing the use of several cuts in a single step. If the cuts in each dimension are a power of two as well, lookup can be done in a single step via multidimensional array indexing. Once again this is an extra degree of freedom (**P13**) being exploited. Because each cut is now a general hypercube, the scheme is called *HyperCuts*. HyperCuts works significantly faster than HiCuts on many real databases (Singh et al., 2004a).

Finally, Efficuts (Vamanan et al., 2010) adds another degree of freedom by using what they call “equidense cuts”: the cuts are not of equal size but instead distribute the child pointers evenly among cuts. They do this by selectively merging equal size cuts in HyperCuts to save memory. Of course, the tradeoff is that lookup is slightly slower because simple array indexing can no longer work. To further reduce redundancy in the data structure, EffiCuts exploits the degree of freedom first considered in Woo (2000) to create mutually exclusive sets of rules and creates multiple decision trees for each such set, with different heuristics to achieve a good trade-off between number of trees (lookup time) and storage (redundancy).

Using a publicly available benchmark of synthetic and other classifiers called ClassBench (Taylor and Turner, 2007), the authors show that for comparable performance EffiCuts needs 57 times less memory than HyperCuts and 4-8 times less power than a TCAM. The last experiment is notable because unlike earlier papers that informally claimed that algorithmic schemes used less power than TCAMs, this is one of the few papers to *quantify* the comparison using a hardware model called CACTI (Wilton and Jouppi, 1996) to model CAM and RAM. However, the benchmarks used (Taylor and Turner, 2007) are mostly synthetic ones; thus a more modern benchmark of real classifiers would be ideal to compare all these schemes.

HiCuts, HyperCuts, and Efficuts all use manually created heuristics to build decision trees – in other words, to decide how to partition rules among multiple trees and how to perform cuts at each node in a tree. A later paper, NeuroCuts (Liang et al., 2019), further advances the state-of-the-art in decision trees by using Reinforcement Learning. First, note that using a neural network for classification, is problematic because a neural classification network cannot guarantee correct results (the answers are correct only with high probability). Further, neural networks are resource intensive, making it hard to guarantee results in time to forward a packet. Instead, NeuroCuts uses deep Reinforcement Learning to *build* efficient decision tree, pushing the cost of the neural network to the time when new rules are added to the classifier.

Thus, while previous approaches attempt to heuristically meet performance objective (e.g., reduce storage), Reinforcement Learning explicitly maximizes the given performance objective. Fortunately, the learning time to evaluate a large number of models, which is one of main drawbacks of RL, is not very high for packet classification. Using synthetic workloads generated using ClassBench, the authors (Liang et al., 2019) show that NeuroCuts outperforms existing hand-tuned decisions in both classification time and memory footprint. More specifically, NeuroCuts improves median classification time by 18%, and reduces both time and memory usage by up to a factor of 3.

In conclusion, the decision tree approach described by (Woo, 2000), (Gupta and McKeown, 1999b), (Singh et al., 2004a), (Vamanan et al., 2010) and (Liang et al., 2019) is best viewed as a framework that encompasses a number of potential algorithms. However, experimental evidence (Singh et al., 2004a; Vamanan et al., 2010) shows that this approach works well in practice. The performance of this scheme can be summarized as follows.

Assumption: The scheme assumes there is a sufficient number of distinct fields to make reasonable cuts without much storage replication. This rather general observation needs to be sharpened.

Performance: The memory required can be kept to roughly linear in the number of rules using various heuristics. The tree can be of relatively small height if it is reasonably balanced. Search can easily be pipelined to allow $O(1)$ lookup times. Finally, the updates are likely to be slow if sophisticated heuristics are used to build the decision tree.

12.13 Hybrid algorithms

Based on the earlier descriptions of the various algorithms it should be clear that different algorithms (and even technlogies like CAMs) can do well for different databases. Thus it is natural to consider

Hybrid Schemes. For example, CAMs are fast but use a great deal of power and updates can be slow for rules with ranges. Can TCAMs be combined with algorithmic schemes to get some of the advantages of both schemes? We now sample a few of these ideas:

Combining TCAMs and Algorithmic Schemes: SmartPC (Ma and Banerjee, 2012) reduces the power consumption of TCAMs by pre-classifying a packet on two header fields, source and destination IP addresses. The packet is only sent to TCAM if there is a match, thus only a small portion of TCAM needs to be activated. The authors show that SmartPC reduces average power by about 90% on real and synthetic classifiers. SAX-PAC (Kogan et al., 2014) exploits order independence to reduce the cost (lookup time or memory) of adding additional range or prefix fields using a hybrid software and TCAM-based approach. The order independent rules are implemented in software using linear memory and logarithmic worst case lookup time, whereas the rest of the rules are in TCAM. The paper shows that on real-life classifiers from Cisco Systems, about 90% of rules are handled in software and only the remaining 10% of rules need to be stored in TCAM.

TreeCAM (Vamanan and Vijaykumar, 2011) is a hybrid decision tree and TCAM-based approach for improving the update complexity (not power) of decision trees without sacrificing lookup performance. TreeCAM employs two versions of decision trees: a coarse version with a few thousand rules per leaf achieves efficient lookups and a fine version with a few tens of rules per leaf reduces update effort. Note that many of the papers that combine TCAM and algorithmic schemes are quite feasible to implement with the advent of pipelined programmable hardware architectures like the Tofino-3 (Intel Corporation, 2022) that have stages that contain both RAM and CAM.

Other combinations: HybridCuts (Li and Li, 2013) uses a combination of decomposition and decision-tree techniques to improve both storage and performance. HybridCuts uses memory comparable to EffiCuts, but outperforms EffiCuts in terms of memory accesses without the added lookup complexity in each node. Finally, CutTSS (Li et al., 2020) combines the good update performance of tuple space search with the quick lookup time of Decision Trees. The CutTSS paper (Li et al., 2020) also has pointers to other more recent papers in packet classification.

12.14 Conclusions

This chapter describes several algorithms for packet classification at gigabit speeds. The grid of tries provides a two-dimensional classification algorithm that is fast and scalable. All the remaining schemes require exploiting some assumptions about real rule databases to avoid the geometric lower bound. While much progress has been made, it is important to reduce the number of such assumptions required for classification and to validate these assumptions extensively.

At the time of writing, decision tree approaches (Woo, 2000; Gupta and McKeown, 1999b; Singh et al., 2004a; Vamanan et al., 2010) and the extended grid of tries method (Singh et al., 2004b) appear to be the most attractive algorithmic schemes for hardware. While the latter depends on each packet's matching only a small number of source–destination prefixes, it is still difficult to characterize what assumptions or parameters influence the performance of decision tree approaches. For software settings, Tuple Space Search (Srinivasan et al., 1998) is attractive, especially when fast updates are required. Caching and early stopping can be used to improve average lookup times (Pfaff et al., 2015).

Of the other general schemes, the bit vector scheme is suitable for hardware implementation for a modest number of rules (say, up to 10,000). Equivalenced cross-producting seems to scale to roughly the same number of rules as the Lucent scheme but perhaps can be improved to lower its memory consumption.

The author and his students have placed code for many of the algorithms described in this chapter on a publicly available Web site (Singh et al., 2004a). Packet classification has stagnated because of the lack of standard comparisons and freely available code. Readers are encouraged to experiment with and contribute to this code base.

Although the schemes described in this chapter require some algorithmic thinking, they make heavy use of the other principles we have stressed. The two-dimensional scheme makes heavy use of precomputation; the Lucent scheme uses memory locality to turn what is essentially linear search into a fast scheme for moderate rule sizes; all the other schemes rely on some expected-case assumption about the structure of rules, such as the lack of general ranges and the small number of classification regions. Table 12.1 summarizes the schemes and the principles used in them.

Because the best-matching prefix is a special case of the lowest-cost matching rule, it is not surprising that rule search schemes are generalizations of prefix search schemes. Thus, the grid of tries and set-pruning tries generalize trie schemes for prefix matching. Multidimensional range-matching schemes generalize prefix-matching schemes based on range matching. Tuple search generalizes binary search on hash tables. While cross-producting is not a generalization of an existing prefix-matching scheme, it can also be specialized for prefix lookups.

The high-level message of this chapter is as follows. Applications such as QoS routing, firewalls, virtual private networks, and DiffServ require a more flexible form of forwarding based on multiple header fields. The techniques in this chapter indicate that such forwarding flexibility can go together with high performance using algorithmic solutions without relying on ternary CAMs.

Returning to the quote at the start of this chapter, it should be easy to see how packet classification gets its name if the word *definition* is replaced with *rule*. Notice that classification in the sciences also encompasses overlapping definitions: Men belong to both the mammal and *Homo sapiens* categories. However, it is hard to imagine a biological analog of the concept of a lowest-cost matching rule, or the requirement to classify species several million times a second!

12.15 Exercises

1. **Range to Prefix Mappings:** CAMs require the use of prefix ranges, but many rules use general ranges. Describe an algorithm that converts an arbitrary range on, say, 16-bit port number fields to a logarithmic number of prefix ranges. Describe the prefix ranges produced by the arbitrary but common range of greater than 1024. Given a rule R with arbitrary range specifications on port numbers, what is the worst-case number of CAM entries required to represent R? Solutions to this problem are discussed in Srinivasan et al. (1998, 1999).
2. **Worst-Case Storage for Set-Pruning Tries:** Generalize the example of Fig. 12.5 to K fields to show that storage in set-pruning-trie approaches can be as bad as $O(N^k/k)$.
3. **Improvements to the Grid of Tries:** In the grid of tries, the only role played by the destination trie is in determining the longest-matching destination prefix. Show how to use other lookup techniques to obtain a total search time of ($\log W + W$) for the destination–source rules instead of $2W$.
4. **Reasoning about the Correctness of the Grid of Tries:** Given any source and destination IP address pair (o, d), let S be the set of nodes (destination-source rules) that (o, d) matches with. A

node α in S is called a *skyline point* if there is no other node (rule) in S whose source and destination prefixes are both no shorter than those of node α. Now solve the following two subproblems. First, prove that the grid of tries algorithm guarantees to traverse all skyline points in S. Second, explain why this guarantee, in combination with the pre-computation of the stored rule for each node, is sufficient to guarantee the correctness of the grid of tries algorithm.

5. **Aggregate Bit Vector Search:** Use 3-bit summaries in Fig. 12.11 and determine the improvement in the worst-case time by adding summaries, and compare it to the increase in storage for using summaries. Details of the algorithm, if needed, can be found in Baboescu and Varghese (2001).
6. **Aggregate Bit Vector Storage:** The use of summary bits appears to increase storage. Show, however, a simple modification in which the use of aggregates can reduce storage if the bit vectors contain large strings of zeroes. Describe the modifications to the search process to achieve this compression. Does it slow down search?
7. **On-Demand Cross-Producting:** Consider the database of Fig. 12.2, and imagine a series of Web accesses from an internal site to the external network. Suppose the external destinations accessed are $D_1, \ldots, D_M$. How many cache terms will these headers produce in the case of full header caching versus on-demand cross-producting?
8. **Equivalenced Cross-Producting:** Why do the fifth and eighth entries in Fig. 12.13 have the same bitmaps? Check your answer two ways, first by intersecting the corresponding bitmaps for the two fields from Fig. 12.11 and then by arguing directly that they match the same set of rules.
9. **Combining Trees for RFC:** The equivalenced cross-producting idea in RFC leaves unspecified how to choose a combining tree. One technique is to compute all possible combining trees and then to pick the tree with the smallest storage. Describe an algorithm based on dynamic programming to find the optimal tree. Compare the running times of the two algorithms.
10. **Reducing Rule Databases Using Redundancy:** If a smaller prefix has the same next hop as a longer prefix, the longer prefix can be removed from an IP lookup table. Find similar techniques to spot redundancies in classifiers. Compare your ideas with the techniques described in Gupta and McKeown (1999a). Note that as in the case of IP lookups, such techniques to remove redundancy are orthogonal to the classification scheme chosen and can be implemented in a separate preprocessing step.
11. **Generalizing Linear Searching in HiCuts:** In HiCuts, all the linear lists are at the leaves. However, a rule with all wildcarded entries will be replicated at all leaves. This suggests that such rules be placed once in a linear list at the *root* of the HiCuts tree. Generalizing, one could place linear lists at any node to reduce storage. Describe a bottom-up algorithm that starts with the base HiCuts decision tree and then hoists rules to nodes higher up in the tree to reduce storage. Try to do so with minimal impact on the search time.