Packet classification

Context

Following weeks:
- Now: "Packet" classification at input ports
- Next:
  - Forwarding modes across switch
  - Discard strategies at points of contention, especially output ports

After:
- Scheduling at points of contention, especially output ports
- Traffic conditioning
- Traffic management

Outline

The classification problem
- Types of address (part 1: Classfull)
- Classification variations
- Exact match classification
- Lookup tables
- Hashing
- CAMs
- Types of address (part 2: Classless)
- Partial match classification
- Exact match techniques revisited
- Ternary CAMs
- Tries, PARTICIA trees

Packet classification: applications

When "packets"† arrive at "switch"‡ inputs, need to use packet fields to look up state information:
- Which output port?
  - Switch input to the correct output. "Correct":
    - port leads to destination
    - port provides appropriate service, e.g. voice traffic over an ATM network
  - Filtering (no output port): Prevent flow of traffic
    - From certain addresses, e.g. from users who haven’t paid
    - To certain addresses/services, e.g. firewall
- What class of service (within the switch)? "appropriate service" again
  - Record usage (billing, conditioning, etc.)

If packet can’t be classified, then switch may take some default action, e.g. discard it, send it to a default router, send it out on all ports, etc.

† or frames, or segments
‡ or bridge or router
Keys: inputs to classification (1)

Packet fields that may be used for classification:
- **Source and destination address fields**
- Destination address is the most important
  - But source address may be relevant:
    - for filtering
    - when recording usage
- for “load balancing” traffic from multiple sources to one destination
  - Prefer all traffic from one source follow the same path to preserve sequence
- for bridged networks† — learn station locations by observing source addresses
- **MAC: address fields are 48b**
- **Network layer: address fields are 32b (IPv4) or 128b (IPv6)**

† Link layer interconnection (c.f. network layer routers)

Keys: inputs to classification (2)

Packet fields that may be used for classification:
- **Source and destination address fields**
- **Virtual Circuit Identifiers**: 16b for ATM (8 or 12b VPI), 12b VLAN ID
- **Fields indicating required Class of Service**:
  - 8b: Type of Service/Class (IPv4), Traffic Class (IPv6)
  - IPv6: 20b Flow Label
- **Mac Addresses**: learn station locations by observing source addresses
- **Network layer**: address fields are 32b (IPv4) or 128b (IPv6)

⇒ key width K. Classification is simple for small K (e.g. ≤ 32b), harder for larger K (e.g. 48b+)

Characteristics of keys

[In addition to key sizes (previous slide) & database sizes (next slide)]

**Values may be correlated**
- e.g. LAN with many NICs bought in bulk from one vendor: MAC addresses have same 24 OUI bits (one vendor), and similar (perhaps sequential) less-significant bits
  ⇒ deal with hash collisions

**Temporal locality**: Keys that have been queried recently will likely be queried again soon ⇒ caching (tries, small CAMs)

Some keys will be queried more often than others, e.g. a server carrying a high load ⇒ optimize for the common case

Sizes of databases

Databases tend to be sparse (few entries, D, relative to keyspace 2^K)
- Bridge D=1000 (1000/2^48 = 1 in 300 billion)
- Backbone router: D=100k (41,578/2^32 = 1 in 100,000)

<table>
<thead>
<tr>
<th>Site</th>
<th>Entries (D)</th>
<th>Next hops (2^E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mae East</td>
<td>39,819</td>
<td>56</td>
</tr>
<tr>
<td>Mae West</td>
<td>14,616</td>
<td>55</td>
</tr>
<tr>
<td>AADS</td>
<td>20,299</td>
<td>19</td>
</tr>
<tr>
<td>Pac Bell</td>
<td>20,661</td>
<td>3</td>
</tr>
<tr>
<td>FUNET</td>
<td>41,578</td>
<td>20</td>
</tr>
</tbody>
</table>

e.g. 56 possible next hops (output ports) ⇒ store in E=6 bits

OUI = Organizationally Unique Identifier

[Nilsson and Karlsson, 98]
Aggregation of database entries

If multiple addresses are associated with the same data then we can aggregate them:
- Matching the key to either address is sufficient to access the data.
- Only need one database entry for the group of addresses.

If these addresses are numerically adjacent, then they can be summarised by a broader range of addresses, e.g. phone exchange:
- (02) 9381 ..., (02) 9382 ..., (02) 9389 ... all go to Sydney
- Melbourne phone exchange can summarise this as all (02) 938 ... calls (indeed, all (02) calls) go to Sydney

Aggregation
- reduces number of entries, but
- complicates classification through variable-length labels

We’ll return to this in the context of Classless Interdomain Routing

Classification problem: Parameters

QoS router example

Use K bit key

\[ K = 48 \text{ (32b IP DA + 16b dest port #)} \]

to select one of D entries in database

\[ D = 1000 \text{ (known addresses)} \]

read out E bits of data from entry

\[ E = 4 \text{ (2^4=16 interfaces on router)} \]

Evaluating classifiers

Performance figures:
- **Query time**: Time to search the database.
  - Shorter is better
  - Want result in a predictable time – hardware processes
- **Update time**: Time to update the database.
  - Would prefer fast updates to the table (slow updates may be acceptable in a firewall)

Cost (e.g. RAM size) for given capacity \( (K, D, E) \)
Outline

The classification problem
Types of address (part 1: Classfull)

Addressing at various layers

Types of identifiers:
• Names: Textual, and hierarchical, e.g. subjects.ee.unsw.edu.au
  • Identify computers or services, e.g. yahoo.com
  • User friendly ⇒ Common at application layer
• Addresses: Compact binary, generally of fixed length.
  • Generally identify network interfaces.
  • A node may have multiple network interfaces.
  • Protocol friendly ⇒ Common at transport layer and below

We’ll concentrate on addresses, but the classification principles also apply to names (e.g. peer-to-peer file sharing: hash file names to point to the machine that stores that file, in a way that evenly distributes files amongst available machines)

Examples of “addresses”

• Internet Protocol addresses: (32b for IPv4, 128b IPv6)
• MAC addresses (48b)
• Protocol and port identifiers (16b for TCP/UDP): Distinguish different processes that may share a network interface.
  While they are not called “addresses”, their functionality is similar to that of other addresses.

Hierarchy in addresses

MAC addresses: Hierarchy in allocation
$ ifconfig
eth0 Link encap:Ethernet HWaddr 00:03:47:AA:BB:CC
Hierarchy only eases allocation: vendor/device
Once product is sold, it may be located anywhere ⇒ hierarchy doesn’t indicate location ⇒ ‘non-routable’

IP addresses: Hierarchy in allocation and in location.
Hierarchical: network/subnetwork/host
$ host subjects.ee.unsw.edu.au
subjects.ee.unsw.edu.au is an alias for alpha400.ee.unsw.edu.au
alpha400.ee.unsw.edu.au has address 149.171.36.48
Hierarchy suggests topological locality ⇒ IP addresses can be aggregated; addresses are “routable”.

Structure of IEEE MAC addresses

Roughly divided into two 24-bit halves:
• 24b Organizationally Unique Identifier (OUI)
  • Identifies which organization (manufacturer) a range of addresses has been assigned to.
  • Least significant 2 bits of first byte indicate:
    • least significant: 1=group/0=individual (multicast/unicast)
    • 2nd least significant: 0=global/1=local (OUI assigned by IEEE/assigned locally by network admin)
• 24b “extension identifier”
  • Assigned by the organization to differentiate individual devices that it manufactures.

Hierarchy only eases allocation: vendor/device
Once product is sold, it may be located anywhere⇒ hierarchy doesn’t indicate location⇒ ‘non-routable’

Hierarchy in IP address allocation

Internet Corporation for Assigned Names and Numbers (ICANN)

Asia Pacific Network Information Centre

American Registry for Internet Numbers

Réseaux IP Européens

U. New South Wales
149.171...., 129.94....

U. Sydney
149.171.92... 129.94.242...

School of EE
149.171.92... 129.94.162...

School of CSE
129.78....

School of Physics
129.94.242...

Individual computers

IP addresses

• IPv4 has 32b addresses
  • Originally each address belonged to a particular class (see next slide)
  • Later: Classless addressing
• IPv6 has 128b addresses
Outline

Exact match classification
Lookup tables
Linked lists
Hashing
CAMs

Student classification: problem

- How can a lecturer rapidly look up student data of E bits?
  - e.g. to record result after marking exam (E=3)
  - Each student is identified by a 7 digit student number (K=7)
  - 10^7 possible numbers
  - only 78 students in class (D=78)
  - numbers of students in class are correlated, since they are likely to have enrolled at similar times and numbers are allocated sequentially over time (e.g. many beginning 310...)

Classification variations

- Do all stored rules apply to the same set of key digits?
  - Yes is the simplest case, e.g. MAC bridges: All rules apply to all 48b of MAC address.
  - No can be more complicated, e.g.
    - IPv4 classfull addresses: limited set of key digits:
      - Some rules apply to 8b network number for Class A,
      - Some to 16b network number for Class B...
    - IP classless addresses: Longest prefix matching:
      - Address doesn’t indicate which bits indicate network number.
      - Firewalls: Different rules may apply to different fields

- Can a key match multiple rules?
  - No is the simplest case.
  - Yes with Longest prefix matching and exceptions

Manually classifying students

- How might a lecturer do it?
  1. Sort list of student IDs into numerical order
  2. Search through list to find match
     - Sequential search is slow
     - Binary search is possible
     - In practice, initial searches may be binary, until lecturer gets a “feel” for the space (vague memory of experience), then initial stride can be more accurate (e.g. 222* ⇒ look at top of list)
Automatically classifying students

A1: Manual approach: List them in order &:
  • search sequentially
  • binary search

A2: Use student number to index a table in RAM.
  Wastes RAM since $78 \ll 10^6 = 10^9$

A3: Convert student number into smaller number to
  index a table, e.g. choose least significant 5 digits. More efficient
  with RAM but may have collisions (Hashing)

A4: Special associative memory that can simultaneously
  match all stored words with supplied value. (CAMs)

A5: Other data structures (linked lists, tries, etc)

Exact-match classification

e.g. for MAC addresses; IP addresses without aggregation
exceptions

Techniques:
  • Lookup tables
  • Lists
  • Hashing
  • Content Addressable Memories
  + techniques that can classify by partial match

Naïve solution: Single lookup table

Use the key to index a RAM of $2^E$ words of $E$ bits each

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>3104005</td>
<td>1</td>
</tr>
<tr>
<td>3104004</td>
<td>0</td>
</tr>
<tr>
<td>3104883</td>
<td>1</td>
</tr>
<tr>
<td>3104884</td>
<td>0</td>
</tr>
<tr>
<td>3104885</td>
<td>1</td>
</tr>
<tr>
<td>3104886</td>
<td>0</td>
</tr>
<tr>
<td>3105090</td>
<td>1</td>
</tr>
</tbody>
</table>

• Advantage: Simple and fast (one lookup of simple
  memory).
• Disadvantage: sparsely populated large RAM ($D \ll 2^E$)
  OK when key is small ⇒ label swapping for virtual circuits

Feasibility of a single lookup table

IPv4: 32b destination address lookup ⇒ feasible RAM size:
  Size of RAM is feasible with classfull addresses:
  Class A: 7b network number = 128 word memory
  Class B: 14b network number = 16K word memory
  Class C: 21b network number = 2M word memory

With larger keys (e.g. 48b MAC DAs, IPv4 with fields other
than DA, e.g. SA and class of service, 128b IPv6 DAs,
etc), size becomes infeasible
List of data in RAM

- Saves space: \( D \) words, each storing \( K + E \) bits
- Takes time

  - If data is unsorted:
    - Search: To perform a match: average time \( D/2 \)
    - Updating1: After removing an item from the list either:
      - invalidate entry (matching process must traverse invalid entries)
    - Updating2: Easy to add an item to the list

  - If data is sorted
    - Update: To insert/remove an item: average time \( D/2 \)
    - Search: To perform a match using a binary search: average time \( \log_2(D) \)

Linked lists

- Efficient use of space: \( D \) words, each storing \( K + E + \log_2(D) \) bits
- Timing:
  - Easy to update the list
  - To search: average time \( D/2 \)
  - But: Easy to relocate item to head of list (remove+add), and expedite future matching if the item is likely to be matched again in the near future, e.g. due to
    - bursty traffic – many packets in succession going to same dest
    - traffic is focussed on certain destinations, e.g. servers

Hashing

Definition: Mathematical mapping from large value to smaller one (ideally \( K \rightarrow \log D \)).
Smaller value can then be used to directly index a table.

If table contains an entry, check if it matches large value (since multiple large values may map into same smaller value).

Issues: Mapping function should minimise collisions ("hash bash") in which multiple large values being used map to the same smaller value.

Maintaining hash tables

To add an entry to the hash table:
1. Check whether the hash of the new entry collides with an existing value.
2a. If not, add new entry directly to table.
2b. If there is a clash, update the existing entry s.t. it points to classifier used to resolve collisions (e.g. linked list or CAM). Add new entry to that classifier.
Example hash functions

- Choosing a subset of the key bits
e.g. for student numbers or MAC addresses:
  - Least significant bits: good (differentiate individuals)
  - Most significant bits: bad (students: similar for same year; MAC: LAN may have many devices from one manufacturer)

Usually apply a more sophisticated function s.t. hashing process is insensitive to format and it is difficult to find a set of addresses that cause collisions
- Arithmetic operations
e.g. add digits in student number
- More complex operations, e.g. CRC

Content Addressable Memories

Each word of memory contains:
- $X$ bits of label storage
- Comparison logic

Figure shows the essence of a CAM with:
- 4 words with 3b labels
- $D \leq 4$, $K=3$

Matching process:
1. Key is distributed to all words simultaneously.
2. Comparison operations are done in parallel.
3. Words with matching labels generate match signal.
CAM example

Note that data doesn’t have to be stored in any particular order.

Recall also, the RTL 8308 chip, used in the D-link DES-1008D shared memory switch, has a 128-entry CAM to accommodate hash hashing.

Associating data with keys in a CAM

1. Store in CAM words
   • Useful when key bits are programmable with a mask – remaining bits can store data.

2. Encode matching CAM word and use to index a RAM

Updating CAMs

Don’t care where the key is stored†. Usually mark free words with some key value that is distinct from valid keys. Distinction may take the form of:
• 1 b per word indicating free/used. Must specify whether matching on that bit or remaining bits – e.g. by using a mask‡.
• Invalid key value indicating free word, e.g. all 0s or broadcast address.

Deletion:
• Search for key to delete.
• Latch match line & use to enable writing of distinct key to that word.

Addition:
1. Search for free word.
2. Choose one of the free words (e.g. priority encoder‡).
3. Latch match line & use to enable writing of distinct key to that word.

† unless multiple matches are possible and rules need to be prioritised – see later.
‡ masks and priority encoders are also useful for ternary searching – again, see later.

Example CAM: MUAA8K80

50MHz clock speed.
32b I/O => multiple cycles
80-bit width with programmable CAM/RAM partition
For bridges: Auto-learn Internal aging with 9-bit time stamp
Synchronous port for high speed packet processing
Asynchronous port for table maintenance
Package: 160 PQFP

Figure from Music Semiconductors datasheet.
Are CAMs the ideal classifier?

Sample specs:

<table>
<thead>
<tr>
<th>Device</th>
<th>Capacity</th>
<th>Clock Speed (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Music Semiconductors</td>
<td>8K x 60b</td>
<td>50</td>
</tr>
<tr>
<td>IDT 75K62100</td>
<td>128K x 72b</td>
<td>133</td>
</tr>
</tbody>
</table>

✓ High speed: 64B Ethernet frames @ 1Gb/s = 2M frames/s << 50MHz
  (Although multiple, e.g. 2x32b bus, clock cycles required to feed data in/out)

✓ Low storage capacity/density

✓ Comparison logic for each cell

✓ Relatively low density (transistors per unit area)
  e.g. 4-5 transistors for SRAM cell, 11-15 for CAM ⇒ more space/bit for CAMs

✓ High power consumption
  e.g. 2W for 6Mb SRAM @ 200MHz, 7W for 2Mb CAM @ 50MHz

✓ Niche memory ⇒ not improved through competitive pressures (ala DRAM)

Hybrid classifiers

Try using one fast selective classifier & if that fails, revert to another slower but comprehensive classifier, e.g.

• Hash then CAM (e.g. as in RTL8308 chip used in D-link DES-1008D):
  1. Try hashing first.
  2. Resolve hash bashes by storing colliding labels in a CAM.
  Small CAM resolves rare hash bashes.

• CAM then trie (described shortly):
  Store most recently used labels in a CAM.
  1. Try the CAM first (fast)
  2. If CAM is inconclusive, then use a trie that stores all labels.

Common cases are processed fast. Trie (large but slow) deals with exceptional cases.

Outline

Types of address (part 2: Classless)

Problems with classfull addresses

Context: 1990s: Internet becomes popular ⇒ larger addresses needed (e.g. 128b IPv6) ⇒ larger key size (K).

Organisation with 2^1 hosts can choose between:

• Class B space and use only 1/256
  ⇒ poor utilisation of address space
  ⇒ further demand for larger key size

• 2 separate Class C networks
  × 1-1 mapping between organisation and address space would be neater
  × Increases number of entries in router databases (D)
  × Router tables becoming “too” large (2M Class C nets)

Solution: Classless InterDomain Routing (CIDR)
Classless Inter Domain Routing

CIDR allows network prefixes of arbitrary length

Different way of indicating which bits identify network, and which bits identify host.

- No longer indicated by address class
- Instead, indicated by a separate network mask (=1 in network ID bits)
- Routers exchange masks with other reachability info.

Address aggregation: Scenario

Router has moderate number of ports (e.g. 4-64) & different networks may be reachable through same port

- e.g. two organisations
  - O with network 192.16.32/23 through port P
  - Q with network 192.16.34/23 through port P
- forward aggregate 192.16.32/23 through port P

Address aggregation: Implementation

Aggregation reduces the number of router table entries.

Ideally assign topologically-adjacent organisations numerically-adjacent addresses; e.g. APNIC

But organizations may move (e.g. Q shifts headquarters overseas or changes network service provider) ⇒ aggregation exceptions

e.g. 192.16.32/19 through port P, except 192.16.34/23 through port R

Search for longest prefix match (aka “best prefix match”): longer prefix is more specific and so prefer to route there (give it priority)
Prioritisation of rules

Classification rules may intersect ⇒ need to prioritize, e.g.:
- Router: Longest-prefix matching
- Firewall: Filtering traffic may have priority over sending certain traffic on matching path, e.g.:
  1. Block traffic from network \( N \)
  2. Forward traffic to network \( M \) through port \( P \)
Packet from \( N \) to \( M \) should be blocked.

Outline

Partial matching with previously described classifiers

Tables
- Hard to update – e.g. to install rule with mask that doesn’t care about \( b \) bits may have to update \( 2^b \) table entries.
- No benefit from aggregation

Lists
- OK if place longest prefix first; can’t sort to expedite search
- Search time still doesn’t scale well to many entries.

Hashing
- Only suitable for exact matches – not longest prefix

CAMs – add ternary functionality …

Ternary CAMs

Ternary comparison: 0, 1, X (“Don’t care” = 0 or 1)
Don’t care bits generally specified as a mask (e.g. 1 = “Do care”, \( 0 \) = “Don’t”)

Word mask
- Word-specific, stored with label e.g. mask = 23 ones, then 22, … then 19
- Apply to all words; iterative search
- e.g. 192.16.001xxxxx.X port \( P \) (192.16.32/19)
- 192.16.0010001x.X port \( R \) (192.16.34/23)
- and prioritize match lines s.t. lowest rule has priority

Key mask
- Disable comparison
- e.g. 0 1 X

Key in
- Data out

CAMs are good when excluded bits needn’t form a suffix, e.g. firewall
Tries

"a tree where each node corresponds to a string that is defined by the path to that node from the root."

Rule Prefix

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>01000*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>011*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>100*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>1100*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>1101*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>1110*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>1111*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit 12345</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Application to partial match: Locate “don’t care” at end of key. OK for CIDR and address aggregation, but not for firewalls

Implementing tries in RAM

Each node stored in one word containing at most 3 ptrs:

- left: next rule if bit is 0
- centre: next rule if bit is 1
- right: rule if this is last matched bit

Start search at node 1, continue until either:

- match leaf node: e.g. 011 matches c
- can’t progress past a node: then use previous best match: e.g. 0101 matches a en route to blockage at node 7.

Optimising tries

Even if trie uses lots of RAM, most recently used nodes will be stored in (processor’s) cache ⇒ Faster for recently used queries.

- Lengthening the “stride” with multi-bit tries
- Collapsing nonbranching branches

Multi-bit strides

Rather than binary movement at each node (at most 2 child nodes),
we could make bigger strides at each node, e.g.:
- decimal: up to 10 children
- more likely: an integer-sized group of bits (e.g. 2 bits = up to 4 children, 3b = up to 8 children…)

Terminology: “stride” size = number of bits used to branch out from each node, e.g. stride of 1 = binary

Larger stride:
- reduces number of nodes traversed, increasing speed
- requires larger tables that may be sparsely occupied (with memories increasing in size faster than increasing in speed, wasting some space for number of accesses makes sense)

A multi-bit trie is similar to hashing that directly uses a subset of the bits of the key, and then resolves hash clashes using further bits.
Path-compressed tries

Observation: When keys are sparse, many trie nodes have only one descendant, taking many steps to reach a single result.

Solution: Rather than use every bit of key to determine subtree, record in node which bits should be used. At leaves, compare (in parallel) all bits of leaf with key. If mismatch, then use previous best match.

Provides “path compression” by recording only genuine branches.

PATRICIA (a similar path-compression scheme) is the classification technique used with BSD Unix (so claims [Ruiz-Sanchez] – bonus mark if you can identify the Linux algorithm).

To search further, look at bit 3; skip bit 2 until checking leaves.

Prefixes
a. 0*
b. 01000*
c. 011*
d. 1*

To start search, look at bit 1

Resources

Relevant sections of Keshav:
- Chapter 10: Addressing
- pp. 176-8: classification (“port mappers”)