* [Start new chat](https://claude.ai/new)

Starred

Star chats you use often

Recents

[(New chat)](https://claude.ai/chat/7b4d55f9-42ef-40d7-a244-f3389eceddab)

[Implementing Path Compressed Trie for IP Lookup](https://claude.ai/chat/f0a65e39-93fd-40c4-b320-39b5ce54db6e)

[IP Lookup Algorithm Implementations and Performance Benchmarking](https://claude.ai/chat/7ed394ff-769e-4d54-a9f9-124eb3f89086)

[Routing Algorithms Simulator Project](https://claude.ai/chat/a3518670-06e6-430d-baaf-4a8bbfffd293)

[(New chat)](https://claude.ai/chat/f7c656ce-4c26-4035-80d2-f07f3ea544ee)

[(New chat)](https://claude.ai/chat/10d9f3b5-7663-4b6f-ac7d-066ffefa7019)

[Compiling Allegro 5 C++ Program](https://claude.ai/chat/1789c343-630f-4a2f-92ba-02a8b3fbcda0)

[Real-Time Task Scheduling with Allegro](https://claude.ai/chat/93ed02b6-424b-4b37-9ba3-ca0fb5c6d311)

[View all](https://claude.ai/recents)

Free plan

**H**

hbnsl890@gmail.com

[Help & support](https://support.anthropic.com/en/)

**H**

Untitled

What can I help you with today?

Convert this latex code into a single column article, also refer to this text for more theory. Please make a report of 3-4 pages, which illustrates about every algorithms, their advanatges and pitfalls, also make a compartitive table. Talk about what are the other ways to route it, the need of routing at the first place, Refer to the IEEE template that I shared in Latex. Make sure that width of table do not overflow, if they do, make them written in a single page format. Make 2 bar chart to compare different algorithms wrt different parameters. Add more figures from the interent.

Haiku

Choose style

3 messages [remaining](https://support.anthropic.com/en/articles/8602283-does-claude-ai-have-any-message-limits) until 5:30 AM[Subscribe to Pro](https://claude.ai/settings/billing?action=subscribe)

Subscribe to Pro for 5x more usage with [Claude 3.5 Sonnet](https://claude.ai/settings/billing).

[Subscribe to Pro](https://claude.ai/settings/billing?action=subscribe)

**The addressing architecture is of fundamental importance to the routing architecture and tracing its evolution will make it clear how it impacts the complexity of the lookup mecha nism. As discusse**

**pasted**

**\documentclass[conference]{IEEEtran} \IEEEoverridecommandlockouts % The preceding line is only needed to identify funding in the first footnote. If that is unneeded, please comment it out. \usepackage**

**pasted**

Chat controls

Haiku

Fastest model for daily tasks [Learn more](https://www.anthropic.com/product)

Content

No content added yet

Add images, PDFs, docs, spreadsheets, and more to summarize, analyze, and query content with Claude.

Chat styles

Font

DefaultMatch systemDyslexic friendly

paste-2.txt

50.09 KB •690 lines•Formatting may be inconsistent from source

The addressing architecture is of fundamental importance to the routing architecture and tracing its evolution will make it clear how it impacts the complexity of the lookup mecha nism. As discussed earlier in Section 1.1, in the early days the Internet used a classful address ing scheme, known as Class A, Class B, and Class C addresses. With the classful addressing scheme, the forwarding of packets is straightforward. Routers need to examine only the network part of the destination address to forward it to the destination. Thus, the forwarding table needs to store a single entry (the network part) for routing the packets destined to all the hosts attached to a given network. Such a technique is called address aggregation and uses prefixes to represent a group of addresses. As described earlier in Section 14.1.4, a forwarding table entry consists of a prefix, the next-hop address, and the outgoing interface. Finding the forwarding information requires searching the pre f ixes in the forwarding table for the one that matches the same set of bits in the destination address. The lookup operation in a classful addressing scheme proceeds as shown in Figure 15.1. Theforwarding table is organized into three separate tables: one for each of the three allowed lengths: 7 bits, 14 bits, and 21 bits for classes A, B and C,respectively.AsshowninFigure15.1, f irst the addressclass is determinedfromthefirstfewbitsofthedestinationaddress.Basedon this information, one of the three tables is chosen to search. Meanwhile, the network part of the destination is extracted based on the class. Then the chosen table is searched for an exact match between the network part and the prefixes present in the table. The search for an exact match can be performed using well-known algorithms such as binary search or hashing. The class-based addressing scheme worked well in the early years of the Internet. How ever, as the Internet started growing, this scheme presented two problems: 490 15.1 Impact of Addressing on Lookup FIGURE15.1 Lookupoperationinaclassful IP addressing scheme (adapted from [273]). • Depletion of IP Address Space: With only three different network sizes to choose from, the IP address space was not used efficiently and it was being exhausted very rapidly, since only a fraction of the addresses allocated was actually in use (approximately 1%). For example, aclassB netid (good for 216 hosts) had to be allocated to any organization with more than 254 hosts. • Exponential Growth of Routing Tables: The route information stored in the forwarding tables of core IP routers grewinproportionto thenumberofnetworks.Asaresult,routingtables were growing exponentially. This led to higher lookup times on the processor and higher memory requirements in the routers. In an attempt to allow more efficient use of IP address space and to slow down the ex ponential growth of forwarding tables in routers, a new scheme called classless interdomain routing (CIDR) was introduced [239]; see Section 1.3.3 for an introduction about CIDR. 15.1.1 Address Aggregation Because of CIDR, address aggregation is possible so that a router can maintain one entry instead of its constituents before aggregation; however, sometimes it is not possible if an address block is missing. To understand aggregation and exception in aggregation, consider the following example. Example 15.1 Address aggregation and exception in address aggregation. First, we consider address aggregation. Assume that ISP1, a service provider, connects three customers—C1, C2, and C3—with the rest of the Internet; see Figure 15.2(a). ISP1 is, in turn, connected to some backbone provider through router R1. The backbone can also connect other service providers like ISP2. Assume that ISP1 owns IP prefix block 10.2.0.0/22 and partitions it among its customers. Let us say that prefix 10.2.1.0/24 has been allocated to C1, 10.2.2.0/24 to C2, and 10.2.3.0/24 to C3. Now the router in the backbone R1 needs to keep only a single forwarding table entry for IP prefix 10.2.0.0/22 that directs the traffic bound CHAPTER 15 IP Address Lookup Algorithms 491 (a) (b) FIGURE15.2 Examplesof(a)address aggregation, (b) exception in address aggregation. 492 15.2 LongestPrefixMatching toC1,C2,andC3throughrouterR3.Asyoucansee, thehierarchicalallocationofprefixes obviatestheneedforseparateroutingtableentriesforC1,C2,andC3atrouterR1.Inother words,thebackboneroutesthetrafficboundforISP1toR3anditistheresponsibilityofthe routerswithinISP1todistinguishthetrafficbetweenC1,C2,andC3. Next, assume that customerC2wouldlike tochange its serviceprovider fromISP1 to ISP2, butdoesnotwant torenumber itsnetwork. This isdepicted inFigure15.2(b). Nowall thetrafficinthebackboneboundtoprefix10.2.0.0/22wouldneedtobedirected toISP1except forprefix10.2.2.0/24.Wecannotperformaggregationasbeforeat router R1sincethere isa“hole” intheaddressspace.This iscalledexceptiontoaddressaggrega tion. ▲ Example15.1demonstrateshowrouteaggregationleads toareductioninthesizeof backboneroutingtables.Agreatdealofaggregationcanbeachievedifaddressesarecare fullyassigned.However,insomesituations,afewnetworkscaninterferewiththeprocessof aggregationasweillustratedabove.Thequestionis:howcanweaccommodatesuchahole? Astraightforwardsolutionistodeaggregateandincreasethenumberofentriesintheback bonerouterR1tothree,oneforeachofthecustomers.Thisisnotdesirablesincethenumber ofentriesinthebackbonewillincreasedramatically. Canwecomeupwithasolutionthatpreservesthebenefitsofaggregationandstillac commodatessuchexceptions?Yes,usingthefollowing.Keeptheexistingprefix10.2.0.0/22 andaddtheexceptionprefix10.2.2.0/24totheforwardingtableinrouterR1.Thenext-hop informationforexceptionprefixwilldirect thetraffictorouterR2.Thiswill result inonly twoentriesintheforwardingtable.Note,however,thatsomeaddresseswillmatchboththe entriesbecausetheprefixesoverlap.Tomakethecorrectforwardingdecision,routersmust domorethanlookforanexactprefixmatch.Sinceexceptionsintheaggregationsmayexist, itneedstofindthemostspecificmatchthatisthelongestmatchingprefix.Nowweareready todefinethelongestprefixmatchingproblemanddescribewhyitisdifficult. 15.2LongestPrefixMatching Theproblemofidentifyingtheforwardingentrycontainingthelongestprefixamongallthe prefixesmatchingthedestinationaddressofanincomingpacket isdefinedasthelongest matchingprefixproblem.This longestprefixiscalledthelongestmatchingprefix. It isalso referredtoasthebestmatchingprefix. Example15.2 Identifyingthelongestmatchingprefix. Considertheforwardingtableatarouter,asshowninTable15.1.Eachentrycontains theprefixandthenameoftheoutgoinginterface.Entry1indicatesthatthepacketmatching TABLE15.1 Aforwardingtable. EntryNumber Prefix Next-Hop 1 98.1.1.1/24 eth3 2 171.1.0.0/16 so6 3 171.1.1.0/24 fe5 CHAPTER 15 IP Address Lookup Algorithms 493 TABLE15.2 Prefixtableexample. Prefix label Prefix P1 0∗ P2 00001∗ P3 001∗ P4 1∗ P5 1000∗ P6 1001∗ P7 1010∗ P8 1011∗ P9 111∗ prefix 98.1.1.1/24 will go out on interface eth3. If the destination address of the incoming packet is 171.1.1.2, then it will match prefix 171.1.0.0/16 in entry 2 as well as 171.1.1.0/24 in entry 3.Sinceprefix171.1.1.0/24 is the longest matching prefix, the packet will be forwarded on the outgoing interface fe5. ▲ Thedifficulty of the longest prefix matching arises because of the following reasons. First, a destination IP address does not explicitly carry the netmask information when a packet tra verses through. Second, the prefixes in the forwarding table against which the destination ad dress needs to be matched can be of arbitrary lengths; this could be as a result of an arbitrary numberof network aggregations. Thus, the search for the longest prefix not only requires de termining the length of the longest matching prefix, but also involves finding the forwarding table entry containing the prefix of this length. To conclude, the adaptation of CIDR has made route lookups more complex than they were when the classful addressing scheme was used. Throughout the rest of the chapter, we will be using the following two parameters: N=Numberofprefixesin a forwarding table W=Maximumlengthoftheprefixes in bits Unless otherwise specified, we use the prefixes shown in Table 15.2 as a running example for various algorithms discussed in later sections. For ease of reading, this table is subsequently shown next to each figure that is associated with a specific lookup algorithm. 15.2.1 Trends, Observations, and Requirements It is imperative for any algorithm designer to understand the requirements of the problem and how these requirements are expected to evolve. The basic requirements for the longest prefix matching include the following: • Lookup Speed: Internet traffic measurements [696] show that roughly 50% of the pack ets that arrive at a router are TCP-acknowledgment packets, which are typically 40-byte packets. As a result, a router can be expected to receive a steady stream of such mini mumsize packets. Thus, the prefix lookup has to happen in the time it takes to forward a minimum-size packet (40 bytes), known as wire speed forwarding. At wire speed for 494 15.2 Longest Prefix Matching warding, the amount of time that it takes for a lookup should not exceed 320 nanosec at 1Gbps(=109 bps), which is computed as follows: 40 bytes×8 bits/byte 1×109 bps =320 nanosec. Similarly, the lookup cannot exceed the budget time of 32 nanosec at 10 Gbps and 8 nanosec at 40 Gbps. The main bottleneck in achieving such high lookup speed is the cost of memory access. Thus, the lookup speed is measured in terms of the number of memoryaccesses. • MemoryUsage:Theamountofmemoryconsumedbythedatastructuresofthealgorithmis also important. Ideally, it should occupy as little memory as possible. A memory-efficient algorithm can effectively use the fast but small cache memory if implemented in software. Ontheother hand, hardwareimplementations allow the use of fast but expensive on-chip SRAMneededtoachieve high speeds. • Scalability: The algorithms are expected to scale both in speed and memory as the size of the forwarding table increases. While core routers presently contain as many as 200,000 prefixes, it is expected to increase to 500,000 to 1 million prefixes with the possible use of host routes and multicast routes. When routers are deployed in the real network, the service providers expect them to provide consistent and predictable performance despite the increase in routing table size. This is expected since a router needs to have a useful lifetime of at least five years to recuperate the return on investment. • Updatability: It has been observed in practice that the route changes occur fairly frequently. Studies [387] show that core routers may receive bursts of these changes at rates varying from a few prefixes per second to a few hundred prefixes per second. Thus, the route changes require updating the forwarding table data structure, in the order of millisec onds or less. These requirements are still several orders of magnitude less than the lookup speed requirements. Nonetheless, it is important for an algorithm to support incremental updates. To summarize, the important requirements of a lookup algorithm are speed, storage, up date time, and scalability. We ideally require algorithms to perform well in the worst case. However, exploiting some of the following practical observations to improve the expected case is desirable. • Mostoftheprefixes are 24 bits or less in core routers, while more than half are 24 bits (see Table 9.2). • Therearenotverymanyprefixesthat are prefixes of other prefixes. Practical observations show that the number of prefixes of a given prefix is at most seven. These practical observations can be further exploited to come up with efficient schemes. CHAPTER 15 IP Address Lookup Algorithms 495 15.3 Naïve Algorithms The simplest algorithm for finding the best matching prefix is a linear search of prefixes. It uses an array in which the prefixes are stored in an unordered fashion. The search iterates through each prefix and compares it with the destination address. If a match occurs, it is remembered as the best match and the search continues. The best match is updated as the search walks through each prefix in the array. When the search terminates the last prefix re membered is the best matching prefix. The time complexity for such a search is O(N). Linear search might be useful if there are very few prefixes to search; however, the search time de grades as N becomes large. Some researchers proposed the idea of route caching in conjunction with linear search to speed up the lookup time. A cache is a fast buffer for storing recently accessed data. The main use of the cache is to speed up subsequent access to the same data if there is a sufficient amount of locality in data access requests. The average time to access data is significantly re duced since access to cache takes significantly less time than access to SRAM or other storage media [293]. For lookup the cache stores the recently seen 32-bit destination addresses and the asso ciated next-hop information. When a packet arrives at the router, the destination address is extracted and the route cache is consulted. If an entry exists in the cache, then the lookup op eration is completed. Otherwise, the linear search discussed above is invoked and the result is cached, possibly replacing an existing entry. Such a caching scheme is effective only when there is locality in a stream of packets, i.e., a packet arrival implies another packet with the same destination address will arrive with high probability in the near future. However, locality exhibited by flows in the backbone has been observed to be very poor [527]. This leads to a much lower cache hit ratio and degenerates to a linear search for every lookup. In summary, caching can be useful when used in conjunction with other algorithms, but that precludes the need for fast prefix lookups. 15.4 Binary Tries The binary trie is the simplest of a class of algorithms that is tree-like. The term trie comes from “retrieval” and is pronounced “tree.” However, most often to verbally distinguish a trie from ageneral tree, it is pronounced as “try.” The trie data structure was first proposed in the context of file searching [169], [238]. Abinary trie is a tree structure that allows a natural way to organize IP prefixes and uses the bits of prefixes to direct the branching. Each internal node in the tree can have zero, one, or two descendants. The left branch of a node is labeled 0 and the right branch is labeled 1. For instance, the binary trie for the prefixes in Table 15.2 is shown in Figure 15.3. Inabinarytrie,anodel represents a prefix formed by concatenating the labels of all the branches in the path from the root node to l. For example, the concatenated label along the path to node P2 is 00001, which is the same as prefix P2. Note that some of the nodes are shaded in gray while the remaining nodes are not. The gray-shaded nodes correspond to actual prefixes. These nodes contain the next-hop information or a pointer to it. As can be seen, prefixes can be either in the internal nodes or at the leaf nodes. Such a situation arises if there are exception prefixes in the prefix aggregation. For instance, in Figure 15.3, the prefixes P2 and P3 represent exceptions to prefix P1. 496 15.4 Binary Tries FIGURE15.3 Binarytriedata structure for the prefixes of Table 15.2. Let us try to understand this better. A complete binary trie represents all the 5-bit address space. Each leaf represents one possible address. We can see that the address space covered by P1 overlaps the addresses of P2 and P3. Thus, prefixes P2 and P3 represent exceptions to prefix P1 and refer to specific subintervals of the address space of prefix P1. Is it possible to identify such exception prefixes by simply looking at the trie in Figure 15.3? Indeed, yes. Exception prefixes are always descendants of an aggregation prefix. In Figure 15.3, P2 and P3 are exceptions of prefix P1 since they are its descendants. 15.4.1 Search and Update Operations We have seen so far how the prefixes are organized in a trie. The next question is given a destination address, how do the search/insert/delete operations work? The search in the trie is guided by the bits of the destination address starting from the root node of the trie. At each node, the left or right branch is taken depending upon the inspection of the appropriate bit. While traversing the trie, we may come across gray-shaded nodes that contain the prefix information. The search needs to remember this prefix information since it is the longest match found so far. Finally, the search terminates when there are no more branches to be taken and the best matching prefix will be the last prefix remembered. Example 15.3 Searching for the best matching prefix in a binary trie. Consider searching the binary trie shown in Figure 15.3 for an address that begins with 001. The search starts at the root node of the trie and the first bit is examined. Since it is a 0, the search proceeds toward the left branch and encounters the node with the prefix P1. We remember P1 as the best matching prefix found so far. Then, we move left as the second address bit is another 0; the node encountered this time does not have a prefix, so P1 remains the best matchingprefixsofar. Next, weexaminethethirdbit,whichisa 1,andleadstoprefix P3.SinceP3 is a better matching prefix than P1, it is remembered. Now, P3 is a leaf node and, thus, the search terminates and the prefix P3 is declared the best matching prefix. ▲ CHAPTER 15 IP Address Lookup Algorithms 497 The insert and delete operations are also straightforward to implement in binary tries. The insert operation proceeds by using the same bit-by-bit search. Once the search arrives at a node with no further branch to proceed, the necessary nodes are created to represent the prefix. Example 15.4 Inserting prefixes in a binary trie. Consider inserting prefixes 110 and 0110, referred to as P10 and P11, respectively, in the binary trie shown in Figure 15.3. Figure 15.4 illustrates the insertion of both the prefixes. Since the first bit of P10 is 1, the search moves to the right and reaches the gray node P4. Now the second bit is examined, which again guides the search right. As the third bit is 0, there is no left branch to take and, thus, a new node P10 is created and attached. The next-hop information for this prefix is stored in the node itself. Now consider inserting prefix P11.After inspecting the bits, we find that there is no right branch to take on the node with prefix P1. Thus, new nodes are added that create the path to prefix node P11. ▲ Similar to the insert operation, the deletion of a prefix starts by a search to locate the prefix to be deleted. Once the node containing the prefix is found, different operations are performed depending on the node type. If it is an internal node (gray node), then the node is unmarked, indicating there is no more prefix information on it. For example, to delete prefix P1, simply unmarkit, whichis equivalent to removing the next-hop information or nullifying it. If the node to bedeletedisaleafnode,alltheone-childnodesleadingtotheleafnodemight have to be deleted as well. Abinary trie is implemented using two entries per node: one entry for bit 0 and the other for bit 1. Each entry contains two fields, nhop that stores the next-hop information and ptr that stores the pointer to the subtrie. If next-hop information is not present, the field is set to null and, similarly, if the subtrie is not present the ptr field is set to null. Note that the prefix information itself is not stored in each node. This is because it can be derived based on the current bit position being examined in the address that is being looked up. The implemen FIGURE15.4 Inserting newprefixes in a binary trie. 498 15.4 Binary Tries FIGURE15.5 Implementationof the binary trie shown in Figure 15.3. tation of the binary trie in Figure 15.3 is shown in Figure 15.5, in which prefixes indicate the presence of next-hop information and the arrowsindicate the presence of pointers to subtries. Binary tries, in the worst case, during a search must traverse a number of trie nodes equal to the length of addresses; thus, the search complexity is O(W). Inserting a prefix to a binary trie might require adding a sequence of W nodes, and the worst case update complexity is O(W). Similarly, for deletion the worst-case time complexity is O(W). In terms of memory consumption, the complexity is O(NW) since each prefix at most can have W nodes. Note that some of the nodes are shared along the prefix paths and, thus, the upper bound might not be tight. 15.4.2 Path Compression A binary trie can represent arbitrary-length prefixes but it has the characteristic that long sequences of one-child nodes may exist. Such long sequences are undesirable since the bits corresponding to those nodes would need to be examined even though no actual branching decision is made. This increases the search time more than necessary in some cases. Also, one-child nodes consume additional memory. Now, assume the objective is to reduce the search time and reduce the memory space; whatcanwedoaboutit?Onepossibilityisnottoinvolveanyofthebitscorrespondingtoone child nodesduringinspection.Iftheydonotneedtobeinspected,thenwecaneliminatethem as well. This is exactly the idea behind path compression. By collapsing the one-way branch nodes, path compression improves search time and consumes less memory space. However, additional information needs to be maintained in other nodes so that a search operation can be performed correctly. Path compression is derived from a scheme called PATRICIA [502]; PATRICIAwasmeantprimarilyforstoringstringsofcharactersanditdidnotsupportlongest prefix matching. It was later adapted for longest prefix matching [645]. CHAPTER 15 IP Address Lookup Algorithms 499 FIGURE15.6 Pathcompressedtrie data structure for the prefixes of Table 15.2. Path compression applied to the binary trie in Figure 15.3 is shown in Figure 15.6. Ob serve that the nodes with prefixes P2 and P3 have moved up as immediate descendants of node P1. The two nodes preceding P2 in the binary trie have been removed since they are one-child nodes and redundant. Note that the node P2, which was originally in the right branch of its parent, has moved as the left branch of P1. This is because it is the only prefix in the entire left subtrie of node P1.NotethatprefixP3, which was in the left branch of P1 in the binary trie, has shifted to the right. The immediate descendant of P1 in the binary trie can be eliminated and the decision of branching can be made at node P1 itself. However, it requires extra information to be stored at node P1—the bit number to be examined next in the prefix. Because one-child nodes are now removed, it is possible to jump directly to the bit where a significant decision needs to be made, thereby bypassing the inspection of some intermediate bits. In Figure 15.6, the bit numbers (or positions) that need to be examined are shown adjacent to the node enclosed in squares. Since one–child nodes are being removed, what will happen to the prefixes originally present in those nodes? They are simply moved to their nearest descendants. Hence, a list of prefixes needs to be maintained in some of the nodes. Asearch in a path compressed trie proceeds in a manner similar to that in a binary trie by descending the tree under the guidance of the address bits. However, the search inspects only the bit positions indicated by the bit number in the nodes traversed. As a gray node (node with a prefix) is encountered, comparisons are performed with the actual prefix. These comparisons are necessary since the search can potentially skip some bits. If a match is found, the best matching prefix is remembered and the search proceeds further. The search ends when aleaf node has been reached or a mismatch is found. Example 15.5 Search for the best matching prefix in a path compressed trie. Consider the search for an address beginning with 001010 in the path compressed trie shown in Figure 15.6. The search starts with the root node. The node specifies that the bit number 1 needs to be examined. Since the first bit is 0, the search goes left and reaches the 500 15.5 Multibit Tries prefix P1 node. Now, we compare the prefix P1 with the corresponding part of the address 0. Since they match, prefix P1 is saved as the best matching prefix encountered so far. Now the bit number in node P1 indicates that the third bit needs to be inspected, which guides the search to the right. Again, we check whether the prefix P3 (001∗) matches the corresponding part of the address (001010). Since they match and a leaf node is encountered, the search terminates and P3 is the best matching prefix. ▲ ApathcompressedtriehasasearchcomplexityofO(W) in the worst case. Remember path compression is effective on a sparse binary trie. In the case of prefix distribution that results in a dense binary trie, height reduction is less effective. Using similar arguments, we can infer that the update complexity in the worst case is O(W). Since path compressed tries are full binary tries, the total amount of memory required will be at most 2N −1,withN for the leaf nodes and N −1 for the internal nodes. Hence the space complexity is O(N).Thus, path compressed tries reduce the space requirements, but not the search complexity. 15.5 Multibit Tries While binary tries can handle prefixes of arbitrary length easily, the search can be very slow since we examine one bit at a time. In the worst case, it requires 32 memory accesses for the 32-bit IPv4 address. If the cost of a memory access is 10 nanosec, the lookup will consume 320 nanosec. This translates to a maximum forwarding speed of 3.125 million packets per second (1/320 nanosec). At 40 bytes per packet, this can support an aggregate link speed of at most 1 Gbps. However, the increase in Internet traffic requires supporting aggregate link speeds as high as 40 Gbps. Clearly, sustaining such a high rate is not feasible with binary trie–based structures. After closely examining the binary trie, we can ask: why restrict ourselves to only one bit at a time? Instead, examine multiple bits so that we can speed up the search by reducing the number of memoryaccess. For instance, if we inspect 4 bits at a time, the search will finish in 8 memoryaccesses as compared to 32 memory accesses in a binary trie. This is the basic idea behind the multibit trie [661]. The number of bits to be inspected per step is called a stride. Strides can be either fixed-size or variable-size. A multibit trie is a trie structure that allows the inspection of bits in strides of several bits. Each node in the multibit trie has 2k children where k is the stride. If all the nodes at the same level have the same stride size, we call it a f ixed stride; otherwise, it is a variable stride. As one can see, since multibit tries allow the data structure to be traversed in strides of several bits at a time, they cannot supportprefixesofarbitrarylengths. To useagivenmultibit trie, a prefix must be transformed into an equivalent prefix of longer length to conform with the prefix lengths allowed by the structure. In the next section, we discuss some useful prefix transformation techniques that expand a prefix into an equivalent set of prefixes of longer lengths followed by a detailed discussion of various types of multibit tries. 15.5.1 Prefix Transformations AnIPprefixassociatedwiththenext-hopinformationcanbeexpressedasanequivalentsetof prefixes with the same next-hop information after a series of transformations. Various types CHAPTER 15 IP Address Lookup Algorithms 501 TABLE15.3 Expansionofprefixes. Prefix Value ExpandedPrefixes P1 0∗ 000∗,010∗,011∗ P2 00001∗ 00001∗ P3 001∗ 001∗ P4 1∗ 100∗,101∗,110∗ P5 1000∗ 10000∗,10001∗ P6 1001∗ 10010∗,10011∗ P7 1010∗ 10100∗,10101∗ P8 1011∗ 10110∗,10111∗ P9 111∗ 111∗ of transformation are possible butwerestrict the discussionto the commonlyusedones:prefix expansion and disjoint prefixes. PREFIX EXPANSION Oneof the most commonprefix transformation techniques is prefix expansion.Aprefixissaid to be expanded if it is converted into several longer and more specific prefixes that cover the same range of addresses. For instance, consider the prefix 0∗. The range of addresses covered by 0∗ can be also specified with the two prefixes 00∗ and 01∗, or with the four prefixes 000∗, 001∗, 010∗,and011∗. Now the basic question is, how is this useful? If we do prefix expansion appropriately, a given set of prefixes of different lengths can be transformed into a set of prefixes that has fewer different lengths. Consider the set of prefixes in Table 15.3, which is the same set of prefixes shown in Table 15.2. These sets of prefixes have lengths ranging from 1 to 5 and have 4 distinct lengths. Now we want to transform these prefixes into an equivalent set with prefixes of lengths 3 and 2—two distinct lengths. Prefix P1 of length 1 cannot remain unchanged since the closest choice of length is 3. Hence we need to expand it into four equivalent prefixes of length 3. For the prefix 0∗,some of the addresses will start with 000, 001, 010, and the rest will start with 011.Thus,theprefix 0∗ is equivalent to the union of four prefixes 000∗, 001∗, 010∗,and011∗. Both of these prefixes will inherit the same next-hop information as the original prefix P1. Similarly, we can expand prefix P5 into twoprefixesof length 5: 10000∗ and 10001∗. Thus wecaneasily expanda prefix into multiple prefixes that are greater in length. Now, by the same principle, if prefix P4 is expanded into four prefixes 100∗, 101∗, 110∗, and 111∗, we find that prefix 111∗ already exists as prefix P9. Since multiple copies of the same prefix are not desirable, we must break the tie somehow. In such cases, according to the longest matching rule, prefix P9 is the correct choice during a search. In general, when a smaller length prefix is expanded in length and one of its expansions “collides” with an existing prefix, then we say the existing prefix captures the expansion prefix. In such cases, we simply get rid of the expansion prefix. In the example, we remove the expansion prefix 111∗ corresponding to P4, since it has already been captured by the existing prefix P9.The complete expanded prefixes are shown in the last column of Table 15.3. 502 15.5 Multibit Tries FIGURE15.7 Disjointprefix (leaf pushed) binary trie. DISJOINT PREFIXES Aswehaveseenearlier, prefixes can overlap with each other. Furthermore, prefixes represent intervals of contiguous addresses. When two prefixes overlap, it means that one interval of addresses contains another interval of addresses. That is why an address lookup can match several prefixes. The longest prefix matching rule breaks the tie by choosing the prefix that matches as many bits as possible. Is it possible to avoid the longest prefix matching rule and still find the longest matching prefix? Indeed, it is. The trick is to transform a given set of prefixes into a set of disjoint prefixes. Inadisjoint set of prefixes, one prefix does not overlap with another. A trie used to represent disjoint prefixes will have prefixes at the leaf nodes and not at the internal nodes. Now, given a set of prefixes, how can you transform them into a set of disjoint prefixes? Construct a binary trie with the given set of prefixes. Now add leaf nodes to nodes that have only one child. These new leaf nodes represent new prefixes and they inherit forwarding information from the closest ancestor marked as a prefix. Then unmarktheinternal nodescontaining the prefix. The disjoint-prefix binary trie for the binary trie in Figure 15.3 is shown in Figure 15.7. Observe that new prefixes P1a, P1b, andP1c have been added. They inherit the next-hop information fromtheoriginalprefix P1.Similarly,prefix P4a inherits the next-hop information from prefix P4. If an address lookup in the original binary trie ends up with prefix P1 being the best match, then in the disjoint-prefix binary trie it will match P1a, P1b,orP1c.Consider an exampleoflooking uptheprefix 01∗. In the original binary trie, the best matching prefix is P1. In the disjoint-prefix binary trie, the search will end with P1c.SinceP1c has the same next hop information as prefix P1, the result will be equivalent. Since this transformation pushes all the prefixes in the internal nodes to the leaves, it is also known as leaf pushing. 15.5.2 Fixed Stride Multibit Trie If all the nodes at the samelevelhavethesamestridesize,thenthemultibittrieiscalleda fixed stride multibit trie. The fixed stride multibit trie, corresponding to the prefixes in Table 15.2, is CHAPTER 15 IP Address Lookup Algorithms 503 FIGURE15.8 Fixedstridemultibit trie data structure for the prefixes of Table 15.2. shown in Figure 15.8. The example multibit trie uses a stride of 3 bits and 2 bits for all nodes in level 1 and level 2, respectively. Asnotedearlier, someoftheprefixesmighthavetobeexpandedtothelengthssupported by the trie structure. Here prefixes of lengths other than 3 and 5 should be transformed into prefixes of lengths 3 and 5. Applying prefix expansion, prefix P1 is expanded into four pre f ixes 000∗, 001∗, 010∗,and011∗ of length 3. One of the expanded prefixes 001\* is the same as prefix P3. What do we do about it? According to the prefix capture in Section 15.5.1, prefix P3 captures the expanded prefix 001∗ of P1 since it is more specific. In such cases, we have to retain the forwarding information of the existing prefix according to the longest matching rule. Now prefix P5 of length 4 is expanded to two prefixes 10000∗ and 10001∗ of length 5. Similarly, prefixes P6, P7,andP8 are expanded. 15.5.3 Search Algorithm The search proceeds by breaking up the destination address into chunks that correspond to the strides at each level. Then these chunks are used to follow a path through the trie until there are no more branches to take. Each time a prefix is found at a node, it is remembered as the new best matching prefix seen so far. At the end, the last best matching prefix found is the correct one for the given address. Example 15.6 Search for the best matching prefix in a fixed stride trie. Consider searching for the best matching prefix for the address 100111 in the fixed stride trie shown in Figure 15.8. First, the address is divided into multiple chunks: a chunk made of the first 3 bits, 100, corresponds to level 1 of the trie; another chunk made of the next two bits, 11, corresponds to level 2 of the trie, and the last incomplete chunk consists of the remaining bits. Using the first chunk 100 at the root node leads to the prefix P4 that is noted as the best matching prefix. Next, using the second chunk of 11 leads to the prefix P6, which is updated to be the best matching prefix so far. Since the search cannot proceed further, the final answer is P6. It can be seen that the number of memory accesses required is 2 as compared to 5 when using a binary trie for the same search. ▲ 504 15.5 Multibit Tries Since a multibit trie is traversed in strides of k bits, the search time is bounded by O(W/k). 15.5.4 Update Algorithm Before examining how updates work, let us examine the concept of a local best matching pre f ix in multibit tries. A multibit trie can be viewed as a tree of one-level subtries. For instance, in Figure 15.8, there is one subtrie at the first level and three subtries at the second level. The prefix expansion in a subtrie is nothing but actually computing the local best matching pre f ix for each node. The best matching prefix is local because it is computed from a subset of prefixes of the entire prefix set. ConsideragaintheexampleinFigure15.8.Inthesubtrieatthefirstlevelweareinterested in finding the best matching prefix amongprefixes P1, P3, P4,andP9.Fortheleftmost subtrie at the second level the best matching prefix will be selected from only prefix P2. Similarly, in the second subtrie at the second level, the best matching prefix is selected from the prefix set P5 andP6 whilefortherightmostsubtrie it is selected from prefixes P7 and P8. Thus, multibit tries divide the problem of finding a best matching prefix into smaller subproblems in which the local best matching prefixes are selected from among a subset of prefixes. This works out to the advantage of prefix updates as illustrated by the following example. Example 15.7 Inserting prefixes to a fixed stride trie. Assumethatweneedtoinserttwoprefixes,1100∗and10111∗,tothefixedstrideshownin Figure 15.8. These prefixes are referred to as P10 and P11, respectively. Figure 15.9 illustrates the insertion of both prefixes P10 and P11. Let us start with the insertion of prefix P10 by dividing it into chunks 110 and 0∗. We look up the root node using chunk 110, which leads to node P4. Nowwehavetheincomplete chunk 0∗ that needs to be expanded to 00∗ and 01∗ as the prefix length is 2 in the second level. Since P4 does not have any children, we create four nodes as required for the second level and all of them are linked to P4. The two nodes 00 and 01 are augmented with the prefix information P10 while the other two are not used. Note that only the subtrie rooted at P4 has been affected, in this case creating the subtrie itself. FIGURE15.9 Insertingnewprefixes in fixed stride multibit trie. CHAPTER 15 IP Address Lookup Algorithms 505 The insertion of prefix P11 proceeds by dividing P11 into chunks 101 and 11∗. The search using these chunks leads to node P8. Recall that prefix P8 has been expanded to two prefixes, 10110∗ and 10111∗. Now we compare the expanded prefix of P8 and the new prefix P11.We f indthat the best match is the newprefix P11, whichisoflonger length. In other words, prefix P11 captures prefix P8. Hence we update the node with the new prefixandnewprefixlength. To distinguish such cases, the length of the original prefixes needs to be stored in every node. Again note that the update is restricted to only a single subtrie rooted at P4. ▲ Since inserting new prefixes might require prefix expansion, deletion becomes more com plex. Deletion involves removing expanded prefixes and, more importantly, updating the en tries with the next best matching prefix. The problem is that original prefixes are not actually stored in the trie. Suppose we insert prefixes 100∗, 101∗,and110∗ into the trie in Figure 15.8. This will overwrite the expanded prefixes for P4 and the original prefix P4 will not exist. Later if the prefix 110∗ gets deleted, the old best matching prefix of P4 needs to be restored. Hence these operations require maintaining an additional structure for the original prefixes. Typically, these structures are maintained in the route control processor. These examples show that the operation of inserting or deleting a prefix involves only an update of one of the subtries, since the best matching prefixes computed for each subtrie are independent of the other subtries. In other words, prefix update is completely local. The stride of a subtrie determines an upper bound on the time in which the actual updates will occur. If the stride of the subtrie has l bits, then at most 2l−1 nodes need to be modified. To illustrate this, consider the case of a prefix that has the last incomplete chunk of either 1∗ or 0∗. If the stride on that subtrie is l bits, then half of the nodes will start with 0 and the other half will start with 1. Hence the prefix and the next-hop information corresponding to the incomplete chunk have to be inserted in half of the nodes. The complexity of insertion and deletion includes the time for search, which is O(W/k) where k is the size of the stride and the time to modify a maximum of 2k−1 entries. Thus, the update complexity is O(W/k + 2k). From the perspective of storage, a prefix might require the entire path of length W/k, and paths consists of one-level subtries of size 2k. Hence, the memorycomplexity is O(2kNW/k) and increases exponentially with k. 15.5.5 Implementation Afixed stride trie is typically implemented using arrays for each trie node and linking them using pointers. The trie nodes at different levels will have different array sizes as determined by the stride at that level. If the stride at a level is k, then the size of the array required will be 2k. Each entry in the array consists of two fields: the field nhop contains the next-hop information and the field ptr contains the pointer to the subtrie, if any. The implementation of a fixed stride trie in Figure 15.8 is shown in Figure 15.10. Since the stride at the first level is 3, we use an array containing 23 = 8 elements for the f irst level. For the second level subtries we use arrays of size 22 = 4 elements as the stride is 2. The prefix used to index into the array is shown adjacent to each element and note that this information is not stored. The presence of prefix information in an element indicates that the field nhop is not empty and stores the next-hop information associated with that prefix. The arrows indicate that the field ptr is not empty and point to the subtrie. Note the waste 506 15.5 Multibit Tries FIGURE 15.10 Implementation of the fixed stride multibit trie shown in Figure 15.8. of space in the leftmost array in the second level that contains only prefix P2; therestofthe three elements do not contain any information. 15.5.6 Choice of Strides As wehaveseenearlier in the search algorithm, the number of memory accesses required, in the worst case, is bounded by the number of levels (alternatively referred to as the height) of the trie. The number of levels, in turn, is dependent on the size of the strides for each level. With large strides, the number of levels will be smaller as more bits are examined at each level. As a result, the number of memory accesses will be smaller. But at the same time, the amount of memoryconsumedwillbelarger. Hence the choice of strides represents a tradeoff between search speed and memory consumption. In the extreme case, using a single stride of size 32 bits, the search can be accomplished in onememoryaccess,buttheamountofmemory consumed is rather large. Generally, the number of levels of the trie is chosen depending on the desired lookup speed bythedesigner. For example, if the allowed budget time for a lookup is 30 nanosec and the speed of memory access is 10 nanosec, then the number of levels can be at most 3. It is clearly desirable that this constraint be satisfied with the least amount of memory possible. In other words, it is necessary to choose an optimaltrie T thathasatmostthreelevelsforagiven prefix set but still occupies minimum storage. A space-optimized trie is heavily dependent on the size of the strides and, thus, choosing an optimal set of strides is important; see [658] for a dynamic programming-based approach. 15.5.7 Variable Stride Multibit Trie If the nodes atthesamelevelhavedifferent stride size, then the multibit trie is called a variable stride multibit trie. An example of a variable stride multibit trie is shown in Figure 15.11. We can see that the subtrie at level 1 has a stride of 2 bits. Some subtries at level 2 have strides of 2 bits and the rest 1 bit. As in a fixed stride multibit trie, each node will have the same information in addition to the stride length. This is needed since the search algorithms need to know at every subtrie how many bits need to be examined. Algorithms for search and incremental updates are very similar to a fixed stride multibit trie. CHAPTER 15 IP Address Lookup Algorithms 507 FIGURE15.11 Variablestride multibit trie. 15.6 Compressing Multibit Tries Theaggressiveuseofprefixexpansioninmultibittriesintroduces severalnewprefixes.These newprefixesinheritthesamenext-hopinformationasthatoftheoriginalprefix.Furthermore, the use of large strides creates a greater number of contiguous nodes that have the same best matching prefix. For instance, take a look at the implementation of the fixed stride trie in Figure 15.10. It shows, for example, that the prefixes P1 and P4 in the first level are repli cated, which means their next-hop information is repeated. Such redundant information can be compressed, saving memory and at the same time making the search faster because of the smaller height of the trie. After compression, the entire data structure can even fit into an L1 cache, which further speeds up the search as the access times are an order of magnitude faster than SRAM. In the next few sections, we discuss various compression schemes for multibit tries using bitmaps and compressed arrays. 15.6.1 Level Compressed Tries Multibit tries, as we saw earlier, use prefix expansion to reduce the number of levels in a trie; however, this is at the expense of increased storage space. This can be viewed alternatively as compressing the levels of the trie, which sometimes is referred to as level compression. A closer examination of multibit tries shows that space is especially wasted in the sparsely populated regions of the trie. For instance, consider the binary trie in Figure 15.3. The trie region containing the prefix P2 is sparse as no other prefixes are nearby. Now examine the f ixed stride multibit trie variant of the binary trie in Figure 15.8. The leftmost subtrie contains only one prefix (P2) and the rest of the the three locations are not used. Such sparse regions of a binary trie that contain long sequences of one-child nodes can be compressed by the tech nique called path compression discussed in Section 15.4.2. There is another trie-based scheme called level-compressed tries (LC-tries) that combines both path and level compression [529]. The main motivation behind this scheme is to reduce storage by ensuring that the resulting trie nodes do not contain empty locations. TheschemestartswithabinarytrieandtransformsitintoanLC-trieinmultiplesteps.We illustrate this transformation using the binary trie shown in Figure 15.3. First, path compres 508 15.6 Compressing Multibit Tries FIGURE 15.12 Identifying the paths to be compressed in the binary trie shown in Figure 15.3. FIGURE 15.13 Identifying the levels to be compressed in the trie shown in Figure 15.12. sion is applied as described in Section 15.4.2. This removes the sequences of internal nodes having one child. However, we need to keep track of the missing nodes somehow. A simple solution is to store a number called the skip value (s) in each node that indicates how many bits need to be skipped on the path. In Figure 15.12, we show that the sequences of nodes leading to P2 and P9 haveonlyonechild andhencearecandidates for path compression. The path compressed trie is shown in Figure 15.13. After path compression,level compressionis used forcompressingthepartsofthebinary trie that are densely populated. Instead of a node having two children, as in a binary trie, each internal node in a multibit trie is allowed to have 2k children, where k is called the CHAPTER 15 IP Address Lookup Algorithms 509 FIGURE 15.14 Levelcompressedtrie resulting from the trie shown in Figure 15.13. branching factor. The level compression is applied as follows. Every node n that is the root of a complete subtrie of maximum depth d is replaced by a corresponding one-level multibit subtrie. This process is repeated recursively on the children of the multibit trie thus obtained. Again, referring to Figure 15.13, we find that the trie rooted at the left child of node P4 is a complete subtrie of depth 2. This trie can be replaced by a single-level multibit trie with four nodes as shown in Figure 15.14. The branching factor for the left child of node P4 is 2, indicated by the number enclosed in a circle adjacent to the node. The rest of the internal nodes have a default branching factor of 1 and are not shown in Figure 15.13(a). The leaf nodes have a branching factor of 0. Since at each step we replace several levels with a single-level multibit trie, this process can be viewed as the compression of levels of the original binary trie. Hence, the resultant trie is termed an LC-trie or simply an LC-trie. The main drawback of the scheme is that the structure of the binary trie determines the choice of strides at any given level without any regard for the height of the resulting multibit trie. This is because a subtrie of depth d can be substituted only if it contains all the 2d leaves (a full binary subtrie). Hence, a few missing nodes might have a considerable negative influence on the efficacy of the level compression. A simple optimization is to use a relaxed criterion where nearly complete binary tries are replaced with a multibit subtrie. In other words, if the nearly complete binary subtrie has a sufficient fraction of the 2d tries at level d, then it is replaced with a multibit subtrie of stride d pointing to 2d nodes. The parameter that controls the fraction is referred to as the fill factor x,0 < x ≤ 1. Such a relaxed criterion will decrease the depth of the trie but will introduce empty nodes into the trie. However, in practice, this scheme results in substantial time improvements with only a moderate increase in space. For optimizing storage, LC-tries do not the use the standard implementation technique that usesasetofchildpointersateachinternalnode.Instead,anLC-trieisrepresentedusinga single array, and each trie node is an element in the array. An interested reader is encouraged to refer [529] for further details. 510 15.6 Compressing Multibit Tries The LC-trie can be considered a special case of a variable stride trie. The algorithm for a variable stride trie using dynamic programming would indeed result in an LC-trie if it were the optimal solution for a given set of prefixes [658]. The worst-case time complexity for an LC-trie is O(W/k) andthespacecomplexityis O(2kNW/k),whichareverysimilartomultibit tries

* [Start new chat](https://claude.ai/new)

Starred

Star chats you use often

Recents

[(New chat)](https://claude.ai/chat/7b4d55f9-42ef-40d7-a244-f3389eceddab)

[Implementing Path Compressed Trie for IP Lookup](https://claude.ai/chat/f0a65e39-93fd-40c4-b320-39b5ce54db6e)

[IP Lookup Algorithm Implementations and Performance Benchmarking](https://claude.ai/chat/7ed394ff-769e-4d54-a9f9-124eb3f89086)

[Routing Algorithms Simulator Project](https://claude.ai/chat/a3518670-06e6-430d-baaf-4a8bbfffd293)

[(New chat)](https://claude.ai/chat/f7c656ce-4c26-4035-80d2-f07f3ea544ee)

[(New chat)](https://claude.ai/chat/10d9f3b5-7663-4b6f-ac7d-066ffefa7019)

[Compiling Allegro 5 C++ Program](https://claude.ai/chat/1789c343-630f-4a2f-92ba-02a8b3fbcda0)

[Real-Time Task Scheduling with Allegro](https://claude.ai/chat/93ed02b6-424b-4b37-9ba3-ca0fb5c6d311)

[View all](https://claude.ai/recents)

Free plan

**H**

hbnsl890@gmail.com

[Help & support](https://support.anthropic.com/en/)

**H**

Untitled

What can I help you with today?

Convert this latex code into a single column article, also refer to this text for more theory. Please make a report of 3-4 pages, which illustrates about every algorithms, their advanatges and pitfalls, also make a compartitive table. Talk about what are the other ways to route it, the need of routing at the first place, Refer to the IEEE template that I shared in Latex. Make sure that width of table do not overflow, if they do, make them written in a single page format. Make 2 bar chart to compare different algorithms wrt different parameters. Add more figures from the interent.

Haiku

Choose style

3 messages [remaining](https://support.anthropic.com/en/articles/8602283-does-claude-ai-have-any-message-limits) until 5:30 AM[Subscribe to Pro](https://claude.ai/settings/billing?action=subscribe)

Subscribe to Pro for 5x more usage with [Claude 3.5 Sonnet](https://claude.ai/settings/billing).

[Subscribe to Pro](https://claude.ai/settings/billing?action=subscribe)

**The addressing architecture is of fundamental importance to the routing architecture and tracing its evolution will make it clear how it impacts the complexity of the lookup mecha nism. As discusse**

**pasted**

**\documentclass[conference]{IEEEtran} \IEEEoverridecommandlockouts % The preceding line is only needed to identify funding in the first footnote. If that is unneeded, please comment it out. \usepackage**

**pasted**

Chat controls

Haiku

Fastest model for daily tasks [Learn more](https://www.anthropic.com/product)

Content

No content added yet

Add images, PDFs, docs, spreadsheets, and more to summarize, analyze, and query content with Claude.

Chat styles

Font

DefaultMatch systemDyslexic friendly

paste-2.txt

50.09 KB •690 lines•Formatting may be inconsistent from source

The addressing architecture is of fundamental importance to the routing architecture and tracing its evolution will make it clear how it impacts the complexity of the lookup mecha nism. As discussed earlier in Section 1.1, in the early days the Internet used a classful address ing scheme, known as Class A, Class B, and Class C addresses. With the classful addressing scheme, the forwarding of packets is straightforward. Routers need to examine only the network part of the destination address to forward it to the destination. Thus, the forwarding table needs to store a single entry (the network part) for routing the packets destined to all the hosts attached to a given network. Such a technique is called address aggregation and uses prefixes to represent a group of addresses. As described earlier in Section 14.1.4, a forwarding table entry consists of a prefix, the next-hop address, and the outgoing interface. Finding the forwarding information requires searching the pre f ixes in the forwarding table for the one that matches the same set of bits in the destination address. The lookup operation in a classful addressing scheme proceeds as shown in Figure 15.1. Theforwarding table is organized into three separate tables: one for each of the three allowed lengths: 7 bits, 14 bits, and 21 bits for classes A, B and C,respectively.AsshowninFigure15.1, f irst the addressclass is determinedfromthefirstfewbitsofthedestinationaddress.Basedon this information, one of the three tables is chosen to search. Meanwhile, the network part of the destination is extracted based on the class. Then the chosen table is searched for an exact match between the network part and the prefixes present in the table. The search for an exact match can be performed using well-known algorithms such as binary search or hashing. The class-based addressing scheme worked well in the early years of the Internet. How ever, as the Internet started growing, this scheme presented two problems: 490 15.1 Impact of Addressing on Lookup FIGURE15.1 Lookupoperationinaclassful IP addressing scheme (adapted from [273]). • Depletion of IP Address Space: With only three different network sizes to choose from, the IP address space was not used efficiently and it was being exhausted very rapidly, since only a fraction of the addresses allocated was actually in use (approximately 1%). For example, aclassB netid (good for 216 hosts) had to be allocated to any organization with more than 254 hosts. • Exponential Growth of Routing Tables: The route information stored in the forwarding tables of core IP routers grewinproportionto thenumberofnetworks.Asaresult,routingtables were growing exponentially. This led to higher lookup times on the processor and higher memory requirements in the routers. In an attempt to allow more efficient use of IP address space and to slow down the ex ponential growth of forwarding tables in routers, a new scheme called classless interdomain routing (CIDR) was introduced [239]; see Section 1.3.3 for an introduction about CIDR. 15.1.1 Address Aggregation Because of CIDR, address aggregation is possible so that a router can maintain one entry instead of its constituents before aggregation; however, sometimes it is not possible if an address block is missing. To understand aggregation and exception in aggregation, consider the following example. Example 15.1 Address aggregation and exception in address aggregation. First, we consider address aggregation. Assume that ISP1, a service provider, connects three customers—C1, C2, and C3—with the rest of the Internet; see Figure 15.2(a). ISP1 is, in turn, connected to some backbone provider through router R1. The backbone can also connect other service providers like ISP2. Assume that ISP1 owns IP prefix block 10.2.0.0/22 and partitions it among its customers. Let us say that prefix 10.2.1.0/24 has been allocated to C1, 10.2.2.0/24 to C2, and 10.2.3.0/24 to C3. Now the router in the backbone R1 needs to keep only a single forwarding table entry for IP prefix 10.2.0.0/22 that directs the traffic bound CHAPTER 15 IP Address Lookup Algorithms 491 (a) (b) FIGURE15.2 Examplesof(a)address aggregation, (b) exception in address aggregation. 492 15.2 LongestPrefixMatching toC1,C2,andC3throughrouterR3.Asyoucansee, thehierarchicalallocationofprefixes obviatestheneedforseparateroutingtableentriesforC1,C2,andC3atrouterR1.Inother words,thebackboneroutesthetrafficboundforISP1toR3anditistheresponsibilityofthe routerswithinISP1todistinguishthetrafficbetweenC1,C2,andC3. Next, assume that customerC2wouldlike tochange its serviceprovider fromISP1 to ISP2, butdoesnotwant torenumber itsnetwork. This isdepicted inFigure15.2(b). Nowall thetrafficinthebackboneboundtoprefix10.2.0.0/22wouldneedtobedirected toISP1except forprefix10.2.2.0/24.Wecannotperformaggregationasbeforeat router R1sincethere isa“hole” intheaddressspace.This iscalledexceptiontoaddressaggrega tion. ▲ Example15.1demonstrateshowrouteaggregationleads toareductioninthesizeof backboneroutingtables.Agreatdealofaggregationcanbeachievedifaddressesarecare fullyassigned.However,insomesituations,afewnetworkscaninterferewiththeprocessof aggregationasweillustratedabove.Thequestionis:howcanweaccommodatesuchahole? Astraightforwardsolutionistodeaggregateandincreasethenumberofentriesintheback bonerouterR1tothree,oneforeachofthecustomers.Thisisnotdesirablesincethenumber ofentriesinthebackbonewillincreasedramatically. Canwecomeupwithasolutionthatpreservesthebenefitsofaggregationandstillac commodatessuchexceptions?Yes,usingthefollowing.Keeptheexistingprefix10.2.0.0/22 andaddtheexceptionprefix10.2.2.0/24totheforwardingtableinrouterR1.Thenext-hop informationforexceptionprefixwilldirect thetraffictorouterR2.Thiswill result inonly twoentriesintheforwardingtable.Note,however,thatsomeaddresseswillmatchboththe entriesbecausetheprefixesoverlap.Tomakethecorrectforwardingdecision,routersmust domorethanlookforanexactprefixmatch.Sinceexceptionsintheaggregationsmayexist, itneedstofindthemostspecificmatchthatisthelongestmatchingprefix.Nowweareready todefinethelongestprefixmatchingproblemanddescribewhyitisdifficult. 15.2LongestPrefixMatching Theproblemofidentifyingtheforwardingentrycontainingthelongestprefixamongallthe prefixesmatchingthedestinationaddressofanincomingpacket isdefinedasthelongest matchingprefixproblem.This longestprefixiscalledthelongestmatchingprefix. It isalso referredtoasthebestmatchingprefix. Example15.2 Identifyingthelongestmatchingprefix. Considertheforwardingtableatarouter,asshowninTable15.1.Eachentrycontains theprefixandthenameoftheoutgoinginterface.Entry1indicatesthatthepacketmatching TABLE15.1 Aforwardingtable. EntryNumber Prefix Next-Hop 1 98.1.1.1/24 eth3 2 171.1.0.0/16 so6 3 171.1.1.0/24 fe5 CHAPTER 15 IP Address Lookup Algorithms 493 TABLE15.2 Prefixtableexample. Prefix label Prefix P1 0∗ P2 00001∗ P3 001∗ P4 1∗ P5 1000∗ P6 1001∗ P7 1010∗ P8 1011∗ P9 111∗ prefix 98.1.1.1/24 will go out on interface eth3. If the destination address of the incoming packet is 171.1.1.2, then it will match prefix 171.1.0.0/16 in entry 2 as well as 171.1.1.0/24 in entry 3.Sinceprefix171.1.1.0/24 is the longest matching prefix, the packet will be forwarded on the outgoing interface fe5. ▲ Thedifficulty of the longest prefix matching arises because of the following reasons. First, a destination IP address does not explicitly carry the netmask information when a packet tra verses through. Second, the prefixes in the forwarding table against which the destination ad dress needs to be matched can be of arbitrary lengths; this could be as a result of an arbitrary numberof network aggregations. Thus, the search for the longest prefix not only requires de termining the length of the longest matching prefix, but also involves finding the forwarding table entry containing the prefix of this length. To conclude, the adaptation of CIDR has made route lookups more complex than they were when the classful addressing scheme was used. Throughout the rest of the chapter, we will be using the following two parameters: N=Numberofprefixesin a forwarding table W=Maximumlengthoftheprefixes in bits Unless otherwise specified, we use the prefixes shown in Table 15.2 as a running example for various algorithms discussed in later sections. For ease of reading, this table is subsequently shown next to each figure that is associated with a specific lookup algorithm. 15.2.1 Trends, Observations, and Requirements It is imperative for any algorithm designer to understand the requirements of the problem and how these requirements are expected to evolve. The basic requirements for the longest prefix matching include the following: • Lookup Speed: Internet traffic measurements [696] show that roughly 50% of the pack ets that arrive at a router are TCP-acknowledgment packets, which are typically 40-byte packets. As a result, a router can be expected to receive a steady stream of such mini mumsize packets. Thus, the prefix lookup has to happen in the time it takes to forward a minimum-size packet (40 bytes), known as wire speed forwarding. At wire speed for 494 15.2 Longest Prefix Matching warding, the amount of time that it takes for a lookup should not exceed 320 nanosec at 1Gbps(=109 bps), which is computed as follows: 40 bytes×8 bits/byte 1×109 bps =320 nanosec. Similarly, the lookup cannot exceed the budget time of 32 nanosec at 10 Gbps and 8 nanosec at 40 Gbps. The main bottleneck in achieving such high lookup speed is the cost of memory access. Thus, the lookup speed is measured in terms of the number of memoryaccesses. • MemoryUsage:Theamountofmemoryconsumedbythedatastructuresofthealgorithmis also important. Ideally, it should occupy as little memory as possible. A memory-efficient algorithm can effectively use the fast but small cache memory if implemented in software. Ontheother hand, hardwareimplementations allow the use of fast but expensive on-chip SRAMneededtoachieve high speeds. • Scalability: The algorithms are expected to scale both in speed and memory as the size of the forwarding table increases. While core routers presently contain as many as 200,000 prefixes, it is expected to increase to 500,000 to 1 million prefixes with the possible use of host routes and multicast routes. When routers are deployed in the real network, the service providers expect them to provide consistent and predictable performance despite the increase in routing table size. This is expected since a router needs to have a useful lifetime of at least five years to recuperate the return on investment. • Updatability: It has been observed in practice that the route changes occur fairly frequently. Studies [387] show that core routers may receive bursts of these changes at rates varying from a few prefixes per second to a few hundred prefixes per second. Thus, the route changes require updating the forwarding table data structure, in the order of millisec onds or less. These requirements are still several orders of magnitude less than the lookup speed requirements. Nonetheless, it is important for an algorithm to support incremental updates. To summarize, the important requirements of a lookup algorithm are speed, storage, up date time, and scalability. We ideally require algorithms to perform well in the worst case. However, exploiting some of the following practical observations to improve the expected case is desirable. • Mostoftheprefixes are 24 bits or less in core routers, while more than half are 24 bits (see Table 9.2). • Therearenotverymanyprefixesthat are prefixes of other prefixes. Practical observations show that the number of prefixes of a given prefix is at most seven. These practical observations can be further exploited to come up with efficient schemes. CHAPTER 15 IP Address Lookup Algorithms 495 15.3 Naïve Algorithms The simplest algorithm for finding the best matching prefix is a linear search of prefixes. It uses an array in which the prefixes are stored in an unordered fashion. The search iterates through each prefix and compares it with the destination address. If a match occurs, it is remembered as the best match and the search continues. The best match is updated as the search walks through each prefix in the array. When the search terminates the last prefix re membered is the best matching prefix. The time complexity for such a search is O(N). Linear search might be useful if there are very few prefixes to search; however, the search time de grades as N becomes large. Some researchers proposed the idea of route caching in conjunction with linear search to speed up the lookup time. A cache is a fast buffer for storing recently accessed data. The main use of the cache is to speed up subsequent access to the same data if there is a sufficient amount of locality in data access requests. The average time to access data is significantly re duced since access to cache takes significantly less time than access to SRAM or other storage media [293]. For lookup the cache stores the recently seen 32-bit destination addresses and the asso ciated next-hop information. When a packet arrives at the router, the destination address is extracted and the route cache is consulted. If an entry exists in the cache, then the lookup op eration is completed. Otherwise, the linear search discussed above is invoked and the result is cached, possibly replacing an existing entry. Such a caching scheme is effective only when there is locality in a stream of packets, i.e., a packet arrival implies another packet with the same destination address will arrive with high probability in the near future. However, locality exhibited by flows in the backbone has been observed to be very poor [527]. This leads to a much lower cache hit ratio and degenerates to a linear search for every lookup. In summary, caching can be useful when used in conjunction with other algorithms, but that precludes the need for fast prefix lookups. 15.4 Binary Tries The binary trie is the simplest of a class of algorithms that is tree-like. The term trie comes from “retrieval” and is pronounced “tree.” However, most often to verbally distinguish a trie from ageneral tree, it is pronounced as “try.” The trie data structure was first proposed in the context of file searching [169], [238]. Abinary trie is a tree structure that allows a natural way to organize IP prefixes and uses the bits of prefixes to direct the branching. Each internal node in the tree can have zero, one, or two descendants. The left branch of a node is labeled 0 and the right branch is labeled 1. For instance, the binary trie for the prefixes in Table 15.2 is shown in Figure 15.3. Inabinarytrie,anodel represents a prefix formed by concatenating the labels of all the branches in the path from the root node to l. For example, the concatenated label along the path to node P2 is 00001, which is the same as prefix P2. Note that some of the nodes are shaded in gray while the remaining nodes are not. The gray-shaded nodes correspond to actual prefixes. These nodes contain the next-hop information or a pointer to it. As can be seen, prefixes can be either in the internal nodes or at the leaf nodes. Such a situation arises if there are exception prefixes in the prefix aggregation. For instance, in Figure 15.3, the prefixes P2 and P3 represent exceptions to prefix P1. 496 15.4 Binary Tries FIGURE15.3 Binarytriedata structure for the prefixes of Table 15.2. Let us try to understand this better. A complete binary trie represents all the 5-bit address space. Each leaf represents one possible address. We can see that the address space covered by P1 overlaps the addresses of P2 and P3. Thus, prefixes P2 and P3 represent exceptions to prefix P1 and refer to specific subintervals of the address space of prefix P1. Is it possible to identify such exception prefixes by simply looking at the trie in Figure 15.3? Indeed, yes. Exception prefixes are always descendants of an aggregation prefix. In Figure 15.3, P2 and P3 are exceptions of prefix P1 since they are its descendants. 15.4.1 Search and Update Operations We have seen so far how the prefixes are organized in a trie. The next question is given a destination address, how do the search/insert/delete operations work? The search in the trie is guided by the bits of the destination address starting from the root node of the trie. At each node, the left or right branch is taken depending upon the inspection of the appropriate bit. While traversing the trie, we may come across gray-shaded nodes that contain the prefix information. The search needs to remember this prefix information since it is the longest match found so far. Finally, the search terminates when there are no more branches to be taken and the best matching prefix will be the last prefix remembered. Example 15.3 Searching for the best matching prefix in a binary trie. Consider searching the binary trie shown in Figure 15.3 for an address that begins with 001. The search starts at the root node of the trie and the first bit is examined. Since it is a 0, the search proceeds toward the left branch and encounters the node with the prefix P1. We remember P1 as the best matching prefix found so far. Then, we move left as the second address bit is another 0; the node encountered this time does not have a prefix, so P1 remains the best matchingprefixsofar. Next, weexaminethethirdbit,whichisa 1,andleadstoprefix P3.SinceP3 is a better matching prefix than P1, it is remembered. Now, P3 is a leaf node and, thus, the search terminates and the prefix P3 is declared the best matching prefix. ▲ CHAPTER 15 IP Address Lookup Algorithms 497 The insert and delete operations are also straightforward to implement in binary tries. The insert operation proceeds by using the same bit-by-bit search. Once the search arrives at a node with no further branch to proceed, the necessary nodes are created to represent the prefix. Example 15.4 Inserting prefixes in a binary trie. Consider inserting prefixes 110 and 0110, referred to as P10 and P11, respectively, in the binary trie shown in Figure 15.3. Figure 15.4 illustrates the insertion of both the prefixes. Since the first bit of P10 is 1, the search moves to the right and reaches the gray node P4. Now the second bit is examined, which again guides the search right. As the third bit is 0, there is no left branch to take and, thus, a new node P10 is created and attached. The next-hop information for this prefix is stored in the node itself. Now consider inserting prefix P11.After inspecting the bits, we find that there is no right branch to take on the node with prefix P1. Thus, new nodes are added that create the path to prefix node P11. ▲ Similar to the insert operation, the deletion of a prefix starts by a search to locate the prefix to be deleted. Once the node containing the prefix is found, different operations are performed depending on the node type. If it is an internal node (gray node), then the node is unmarked, indicating there is no more prefix information on it. For example, to delete prefix P1, simply unmarkit, whichis equivalent to removing the next-hop information or nullifying it. If the node to bedeletedisaleafnode,alltheone-childnodesleadingtotheleafnodemight have to be deleted as well. Abinary trie is implemented using two entries per node: one entry for bit 0 and the other for bit 1. Each entry contains two fields, nhop that stores the next-hop information and ptr that stores the pointer to the subtrie. If next-hop information is not present, the field is set to null and, similarly, if the subtrie is not present the ptr field is set to null. Note that the prefix information itself is not stored in each node. This is because it can be derived based on the current bit position being examined in the address that is being looked up. The implemen FIGURE15.4 Inserting newprefixes in a binary trie. 498 15.4 Binary Tries FIGURE15.5 Implementationof the binary trie shown in Figure 15.3. tation of the binary trie in Figure 15.3 is shown in Figure 15.5, in which prefixes indicate the presence of next-hop information and the arrowsindicate the presence of pointers to subtries. Binary tries, in the worst case, during a search must traverse a number of trie nodes equal to the length of addresses; thus, the search complexity is O(W). Inserting a prefix to a binary trie might require adding a sequence of W nodes, and the worst case update complexity is O(W). Similarly, for deletion the worst-case time complexity is O(W). In terms of memory consumption, the complexity is O(NW) since each prefix at most can have W nodes. Note that some of the nodes are shared along the prefix paths and, thus, the upper bound might not be tight. 15.4.2 Path Compression A binary trie can represent arbitrary-length prefixes but it has the characteristic that long sequences of one-child nodes may exist. Such long sequences are undesirable since the bits corresponding to those nodes would need to be examined even though no actual branching decision is made. This increases the search time more than necessary in some cases. Also, one-child nodes consume additional memory. Now, assume the objective is to reduce the search time and reduce the memory space; whatcanwedoaboutit?Onepossibilityisnottoinvolveanyofthebitscorrespondingtoone child nodesduringinspection.Iftheydonotneedtobeinspected,thenwecaneliminatethem as well. This is exactly the idea behind path compression. By collapsing the one-way branch nodes, path compression improves search time and consumes less memory space. However, additional information needs to be maintained in other nodes so that a search operation can be performed correctly. Path compression is derived from a scheme called PATRICIA [502]; PATRICIAwasmeantprimarilyforstoringstringsofcharactersanditdidnotsupportlongest prefix matching. It was later adapted for longest prefix matching [645]. CHAPTER 15 IP Address Lookup Algorithms 499 FIGURE15.6 Pathcompressedtrie data structure for the prefixes of Table 15.2. Path compression applied to the binary trie in Figure 15.3 is shown in Figure 15.6. Ob serve that the nodes with prefixes P2 and P3 have moved up as immediate descendants of node P1. The two nodes preceding P2 in the binary trie have been removed since they are one-child nodes and redundant. Note that the node P2, which was originally in the right branch of its parent, has moved as the left branch of P1. This is because it is the only prefix in the entire left subtrie of node P1.NotethatprefixP3, which was in the left branch of P1 in the binary trie, has shifted to the right. The immediate descendant of P1 in the binary trie can be eliminated and the decision of branching can be made at node P1 itself. However, it requires extra information to be stored at node P1—the bit number to be examined next in the prefix. Because one-child nodes are now removed, it is possible to jump directly to the bit where a significant decision needs to be made, thereby bypassing the inspection of some intermediate bits. In Figure 15.6, the bit numbers (or positions) that need to be examined are shown adjacent to the node enclosed in squares. Since one–child nodes are being removed, what will happen to the prefixes originally present in those nodes? They are simply moved to their nearest descendants. Hence, a list of prefixes needs to be maintained in some of the nodes. Asearch in a path compressed trie proceeds in a manner similar to that in a binary trie by descending the tree under the guidance of the address bits. However, the search inspects only the bit positions indicated by the bit number in the nodes traversed. As a gray node (node with a prefix) is encountered, comparisons are performed with the actual prefix. These comparisons are necessary since the search can potentially skip some bits. If a match is found, the best matching prefix is remembered and the search proceeds further. The search ends when aleaf node has been reached or a mismatch is found. Example 15.5 Search for the best matching prefix in a path compressed trie. Consider the search for an address beginning with 001010 in the path compressed trie shown in Figure 15.6. The search starts with the root node. The node specifies that the bit number 1 needs to be examined. Since the first bit is 0, the search goes left and reaches the 500 15.5 Multibit Tries prefix P1 node. Now, we compare the prefix P1 with the corresponding part of the address 0. Since they match, prefix P1 is saved as the best matching prefix encountered so far. Now the bit number in node P1 indicates that the third bit needs to be inspected, which guides the search to the right. Again, we check whether the prefix P3 (001∗) matches the corresponding part of the address (001010). Since they match and a leaf node is encountered, the search terminates and P3 is the best matching prefix. ▲ ApathcompressedtriehasasearchcomplexityofO(W) in the worst case. Remember path compression is effective on a sparse binary trie. In the case of prefix distribution that results in a dense binary trie, height reduction is less effective. Using similar arguments, we can infer that the update complexity in the worst case is O(W). Since path compressed tries are full binary tries, the total amount of memory required will be at most 2N −1,withN for the leaf nodes and N −1 for the internal nodes. Hence the space complexity is O(N).Thus, path compressed tries reduce the space requirements, but not the search complexity. 15.5 Multibit Tries While binary tries can handle prefixes of arbitrary length easily, the search can be very slow since we examine one bit at a time. In the worst case, it requires 32 memory accesses for the 32-bit IPv4 address. If the cost of a memory access is 10 nanosec, the lookup will consume 320 nanosec. This translates to a maximum forwarding speed of 3.125 million packets per second (1/320 nanosec). At 40 bytes per packet, this can support an aggregate link speed of at most 1 Gbps. However, the increase in Internet traffic requires supporting aggregate link speeds as high as 40 Gbps. Clearly, sustaining such a high rate is not feasible with binary trie–based structures. After closely examining the binary trie, we can ask: why restrict ourselves to only one bit at a time? Instead, examine multiple bits so that we can speed up the search by reducing the number of memoryaccess. For instance, if we inspect 4 bits at a time, the search will finish in 8 memoryaccesses as compared to 32 memory accesses in a binary trie. This is the basic idea behind the multibit trie [661]. The number of bits to be inspected per step is called a stride. Strides can be either fixed-size or variable-size. A multibit trie is a trie structure that allows the inspection of bits in strides of several bits. Each node in the multibit trie has 2k children where k is the stride. If all the nodes at the same level have the same stride size, we call it a f ixed stride; otherwise, it is a variable stride. As one can see, since multibit tries allow the data structure to be traversed in strides of several bits at a time, they cannot supportprefixesofarbitrarylengths. To useagivenmultibit trie, a prefix must be transformed into an equivalent prefix of longer length to conform with the prefix lengths allowed by the structure. In the next section, we discuss some useful prefix transformation techniques that expand a prefix into an equivalent set of prefixes of longer lengths followed by a detailed discussion of various types of multibit tries. 15.5.1 Prefix Transformations AnIPprefixassociatedwiththenext-hopinformationcanbeexpressedasanequivalentsetof prefixes with the same next-hop information after a series of transformations. Various types CHAPTER 15 IP Address Lookup Algorithms 501 TABLE15.3 Expansionofprefixes. Prefix Value ExpandedPrefixes P1 0∗ 000∗,010∗,011∗ P2 00001∗ 00001∗ P3 001∗ 001∗ P4 1∗ 100∗,101∗,110∗ P5 1000∗ 10000∗,10001∗ P6 1001∗ 10010∗,10011∗ P7 1010∗ 10100∗,10101∗ P8 1011∗ 10110∗,10111∗ P9 111∗ 111∗ of transformation are possible butwerestrict the discussionto the commonlyusedones:prefix expansion and disjoint prefixes. PREFIX EXPANSION Oneof the most commonprefix transformation techniques is prefix expansion.Aprefixissaid to be expanded if it is converted into several longer and more specific prefixes that cover the same range of addresses. For instance, consider the prefix 0∗. The range of addresses covered by 0∗ can be also specified with the two prefixes 00∗ and 01∗, or with the four prefixes 000∗, 001∗, 010∗,and011∗. Now the basic question is, how is this useful? If we do prefix expansion appropriately, a given set of prefixes of different lengths can be transformed into a set of prefixes that has fewer different lengths. Consider the set of prefixes in Table 15.3, which is the same set of prefixes shown in Table 15.2. These sets of prefixes have lengths ranging from 1 to 5 and have 4 distinct lengths. Now we want to transform these prefixes into an equivalent set with prefixes of lengths 3 and 2—two distinct lengths. Prefix P1 of length 1 cannot remain unchanged since the closest choice of length is 3. Hence we need to expand it into four equivalent prefixes of length 3. For the prefix 0∗,some of the addresses will start with 000, 001, 010, and the rest will start with 011.Thus,theprefix 0∗ is equivalent to the union of four prefixes 000∗, 001∗, 010∗,and011∗. Both of these prefixes will inherit the same next-hop information as the original prefix P1. Similarly, we can expand prefix P5 into twoprefixesof length 5: 10000∗ and 10001∗. Thus wecaneasily expanda prefix into multiple prefixes that are greater in length. Now, by the same principle, if prefix P4 is expanded into four prefixes 100∗, 101∗, 110∗, and 111∗, we find that prefix 111∗ already exists as prefix P9. Since multiple copies of the same prefix are not desirable, we must break the tie somehow. In such cases, according to the longest matching rule, prefix P9 is the correct choice during a search. In general, when a smaller length prefix is expanded in length and one of its expansions “collides” with an existing prefix, then we say the existing prefix captures the expansion prefix. In such cases, we simply get rid of the expansion prefix. In the example, we remove the expansion prefix 111∗ corresponding to P4, since it has already been captured by the existing prefix P9.The complete expanded prefixes are shown in the last column of Table 15.3. 502 15.5 Multibit Tries FIGURE15.7 Disjointprefix (leaf pushed) binary trie. DISJOINT PREFIXES Aswehaveseenearlier, prefixes can overlap with each other. Furthermore, prefixes represent intervals of contiguous addresses. When two prefixes overlap, it means that one interval of addresses contains another interval of addresses. That is why an address lookup can match several prefixes. The longest prefix matching rule breaks the tie by choosing the prefix that matches as many bits as possible. Is it possible to avoid the longest prefix matching rule and still find the longest matching prefix? Indeed, it is. The trick is to transform a given set of prefixes into a set of disjoint prefixes. Inadisjoint set of prefixes, one prefix does not overlap with another. A trie used to represent disjoint prefixes will have prefixes at the leaf nodes and not at the internal nodes. Now, given a set of prefixes, how can you transform them into a set of disjoint prefixes? Construct a binary trie with the given set of prefixes. Now add leaf nodes to nodes that have only one child. These new leaf nodes represent new prefixes and they inherit forwarding information from the closest ancestor marked as a prefix. Then unmarktheinternal nodescontaining the prefix. The disjoint-prefix binary trie for the binary trie in Figure 15.3 is shown in Figure 15.7. Observe that new prefixes P1a, P1b, andP1c have been added. They inherit the next-hop information fromtheoriginalprefix P1.Similarly,prefix P4a inherits the next-hop information from prefix P4. If an address lookup in the original binary trie ends up with prefix P1 being the best match, then in the disjoint-prefix binary trie it will match P1a, P1b,orP1c.Consider an exampleoflooking uptheprefix 01∗. In the original binary trie, the best matching prefix is P1. In the disjoint-prefix binary trie, the search will end with P1c.SinceP1c has the same next hop information as prefix P1, the result will be equivalent. Since this transformation pushes all the prefixes in the internal nodes to the leaves, it is also known as leaf pushing. 15.5.2 Fixed Stride Multibit Trie If all the nodes at the samelevelhavethesamestridesize,thenthemultibittrieiscalleda fixed stride multibit trie. The fixed stride multibit trie, corresponding to the prefixes in Table 15.2, is CHAPTER 15 IP Address Lookup Algorithms 503 FIGURE15.8 Fixedstridemultibit trie data structure for the prefixes of Table 15.2. shown in Figure 15.8. The example multibit trie uses a stride of 3 bits and 2 bits for all nodes in level 1 and level 2, respectively. Asnotedearlier, someoftheprefixesmighthavetobeexpandedtothelengthssupported by the trie structure. Here prefixes of lengths other than 3 and 5 should be transformed into prefixes of lengths 3 and 5. Applying prefix expansion, prefix P1 is expanded into four pre f ixes 000∗, 001∗, 010∗,and011∗ of length 3. One of the expanded prefixes 001\* is the same as prefix P3. What do we do about it? According to the prefix capture in Section 15.5.1, prefix P3 captures the expanded prefix 001∗ of P1 since it is more specific. In such cases, we have to retain the forwarding information of the existing prefix according to the longest matching rule. Now prefix P5 of length 4 is expanded to two prefixes 10000∗ and 10001∗ of length 5. Similarly, prefixes P6, P7,andP8 are expanded. 15.5.3 Search Algorithm The search proceeds by breaking up the destination address into chunks that correspond to the strides at each level. Then these chunks are used to follow a path through the trie until there are no more branches to take. Each time a prefix is found at a node, it is remembered as the new best matching prefix seen so far. At the end, the last best matching prefix found is the correct one for the given address. Example 15.6 Search for the best matching prefix in a fixed stride trie. Consider searching for the best matching prefix for the address 100111 in the fixed stride trie shown in Figure 15.8. First, the address is divided into multiple chunks: a chunk made of the first 3 bits, 100, corresponds to level 1 of the trie; another chunk made of the next two bits, 11, corresponds to level 2 of the trie, and the last incomplete chunk consists of the remaining bits. Using the first chunk 100 at the root node leads to the prefix P4 that is noted as the best matching prefix. Next, using the second chunk of 11 leads to the prefix P6, which is updated to be the best matching prefix so far. Since the search cannot proceed further, the final answer is P6. It can be seen that the number of memory accesses required is 2 as compared to 5 when using a binary trie for the same search. ▲ 504 15.5 Multibit Tries Since a multibit trie is traversed in strides of k bits, the search time is bounded by O(W/k). 15.5.4 Update Algorithm Before examining how updates work, let us examine the concept of a local best matching pre f ix in multibit tries. A multibit trie can be viewed as a tree of one-level subtries. For instance, in Figure 15.8, there is one subtrie at the first level and three subtries at the second level. The prefix expansion in a subtrie is nothing but actually computing the local best matching pre f ix for each node. The best matching prefix is local because it is computed from a subset of prefixes of the entire prefix set. ConsideragaintheexampleinFigure15.8.Inthesubtrieatthefirstlevelweareinterested in finding the best matching prefix amongprefixes P1, P3, P4,andP9.Fortheleftmost subtrie at the second level the best matching prefix will be selected from only prefix P2. Similarly, in the second subtrie at the second level, the best matching prefix is selected from the prefix set P5 andP6 whilefortherightmostsubtrie it is selected from prefixes P7 and P8. Thus, multibit tries divide the problem of finding a best matching prefix into smaller subproblems in which the local best matching prefixes are selected from among a subset of prefixes. This works out to the advantage of prefix updates as illustrated by the following example. Example 15.7 Inserting prefixes to a fixed stride trie. Assumethatweneedtoinserttwoprefixes,1100∗and10111∗,tothefixedstrideshownin Figure 15.8. These prefixes are referred to as P10 and P11, respectively. Figure 15.9 illustrates the insertion of both prefixes P10 and P11. Let us start with the insertion of prefix P10 by dividing it into chunks 110 and 0∗. We look up the root node using chunk 110, which leads to node P4. Nowwehavetheincomplete chunk 0∗ that needs to be expanded to 00∗ and 01∗ as the prefix length is 2 in the second level. Since P4 does not have any children, we create four nodes as required for the second level and all of them are linked to P4. The two nodes 00 and 01 are augmented with the prefix information P10 while the other two are not used. Note that only the subtrie rooted at P4 has been affected, in this case creating the subtrie itself. FIGURE15.9 Insertingnewprefixes in fixed stride multibit trie. CHAPTER 15 IP Address Lookup Algorithms 505 The insertion of prefix P11 proceeds by dividing P11 into chunks 101 and 11∗. The search using these chunks leads to node P8. Recall that prefix P8 has been expanded to two prefixes, 10110∗ and 10111∗. Now we compare the expanded prefix of P8 and the new prefix P11.We f indthat the best match is the newprefix P11, whichisoflonger length. In other words, prefix P11 captures prefix P8. Hence we update the node with the new prefixandnewprefixlength. To distinguish such cases, the length of the original prefixes needs to be stored in every node. Again note that the update is restricted to only a single subtrie rooted at P4. ▲ Since inserting new prefixes might require prefix expansion, deletion becomes more com plex. Deletion involves removing expanded prefixes and, more importantly, updating the en tries with the next best matching prefix. The problem is that original prefixes are not actually stored in the trie. Suppose we insert prefixes 100∗, 101∗,and110∗ into the trie in Figure 15.8. This will overwrite the expanded prefixes for P4 and the original prefix P4 will not exist. Later if the prefix 110∗ gets deleted, the old best matching prefix of P4 needs to be restored. Hence these operations require maintaining an additional structure for the original prefixes. Typically, these structures are maintained in the route control processor. These examples show that the operation of inserting or deleting a prefix involves only an update of one of the subtries, since the best matching prefixes computed for each subtrie are independent of the other subtries. In other words, prefix update is completely local. The stride of a subtrie determines an upper bound on the time in which the actual updates will occur. If the stride of the subtrie has l bits, then at most 2l−1 nodes need to be modified. To illustrate this, consider the case of a prefix that has the last incomplete chunk of either 1∗ or 0∗. If the stride on that subtrie is l bits, then half of the nodes will start with 0 and the other half will start with 1. Hence the prefix and the next-hop information corresponding to the incomplete chunk have to be inserted in half of the nodes. The complexity of insertion and deletion includes the time for search, which is O(W/k) where k is the size of the stride and the time to modify a maximum of 2k−1 entries. Thus, the update complexity is O(W/k + 2k). From the perspective of storage, a prefix might require the entire path of length W/k, and paths consists of one-level subtries of size 2k. Hence, the memorycomplexity is O(2kNW/k) and increases exponentially with k. 15.5.5 Implementation Afixed stride trie is typically implemented using arrays for each trie node and linking them using pointers. The trie nodes at different levels will have different array sizes as determined by the stride at that level. If the stride at a level is k, then the size of the array required will be 2k. Each entry in the array consists of two fields: the field nhop contains the next-hop information and the field ptr contains the pointer to the subtrie, if any. The implementation of a fixed stride trie in Figure 15.8 is shown in Figure 15.10. Since the stride at the first level is 3, we use an array containing 23 = 8 elements for the f irst level. For the second level subtries we use arrays of size 22 = 4 elements as the stride is 2. The prefix used to index into the array is shown adjacent to each element and note that this information is not stored. The presence of prefix information in an element indicates that the field nhop is not empty and stores the next-hop information associated with that prefix. The arrows indicate that the field ptr is not empty and point to the subtrie. Note the waste 506 15.5 Multibit Tries FIGURE 15.10 Implementation of the fixed stride multibit trie shown in Figure 15.8. of space in the leftmost array in the second level that contains only prefix P2; therestofthe three elements do not contain any information. 15.5.6 Choice of Strides As wehaveseenearlier in the search algorithm, the number of memory accesses required, in the worst case, is bounded by the number of levels (alternatively referred to as the height) of the trie. The number of levels, in turn, is dependent on the size of the strides for each level. With large strides, the number of levels will be smaller as more bits are examined at each level. As a result, the number of memory accesses will be smaller. But at the same time, the amount of memoryconsumedwillbelarger. Hence the choice of strides represents a tradeoff between search speed and memory consumption. In the extreme case, using a single stride of size 32 bits, the search can be accomplished in onememoryaccess,buttheamountofmemory consumed is rather large. Generally, the number of levels of the trie is chosen depending on the desired lookup speed bythedesigner. For example, if the allowed budget time for a lookup is 30 nanosec and the speed of memory access is 10 nanosec, then the number of levels can be at most 3. It is clearly desirable that this constraint be satisfied with the least amount of memory possible. In other words, it is necessary to choose an optimaltrie T thathasatmostthreelevelsforagiven prefix set but still occupies minimum storage. A space-optimized trie is heavily dependent on the size of the strides and, thus, choosing an optimal set of strides is important; see [658] for a dynamic programming-based approach. 15.5.7 Variable Stride Multibit Trie If the nodes atthesamelevelhavedifferent stride size, then the multibit trie is called a variable stride multibit trie. An example of a variable stride multibit trie is shown in Figure 15.11. We can see that the subtrie at level 1 has a stride of 2 bits. Some subtries at level 2 have strides of 2 bits and the rest 1 bit. As in a fixed stride multibit trie, each node will have the same information in addition to the stride length. This is needed since the search algorithms need to know at every subtrie how many bits need to be examined. Algorithms for search and incremental updates are very similar to a fixed stride multibit trie. CHAPTER 15 IP Address Lookup Algorithms 507 FIGURE15.11 Variablestride multibit trie. 15.6 Compressing Multibit Tries Theaggressiveuseofprefixexpansioninmultibittriesintroduces severalnewprefixes.These newprefixesinheritthesamenext-hopinformationasthatoftheoriginalprefix.Furthermore, the use of large strides creates a greater number of contiguous nodes that have the same best matching prefix. For instance, take a look at the implementation of the fixed stride trie in Figure 15.10. It shows, for example, that the prefixes P1 and P4 in the first level are repli cated, which means their next-hop information is repeated. Such redundant information can be compressed, saving memory and at the same time making the search faster because of the smaller height of the trie. After compression, the entire data structure can even fit into an L1 cache, which further speeds up the search as the access times are an order of magnitude faster than SRAM. In the next few sections, we discuss various compression schemes for multibit tries using bitmaps and compressed arrays. 15.6.1 Level Compressed Tries Multibit tries, as we saw earlier, use prefix expansion to reduce the number of levels in a trie; however, this is at the expense of increased storage space. This can be viewed alternatively as compressing the levels of the trie, which sometimes is referred to as level compression. A closer examination of multibit tries shows that space is especially wasted in the sparsely populated regions of the trie. For instance, consider the binary trie in Figure 15.3. The trie region containing the prefix P2 is sparse as no other prefixes are nearby. Now examine the f ixed stride multibit trie variant of the binary trie in Figure 15.8. The leftmost subtrie contains only one prefix (P2) and the rest of the the three locations are not used. Such sparse regions of a binary trie that contain long sequences of one-child nodes can be compressed by the tech nique called path compression discussed in Section 15.4.2. There is another trie-based scheme called level-compressed tries (LC-tries) that combines both path and level compression [529]. The main motivation behind this scheme is to reduce storage by ensuring that the resulting trie nodes do not contain empty locations. TheschemestartswithabinarytrieandtransformsitintoanLC-trieinmultiplesteps.We illustrate this transformation using the binary trie shown in Figure 15.3. First, path compres 508 15.6 Compressing Multibit Tries FIGURE 15.12 Identifying the paths to be compressed in the binary trie shown in Figure 15.3. FIGURE 15.13 Identifying the levels to be compressed in the trie shown in Figure 15.12. sion is applied as described in Section 15.4.2. This removes the sequences of internal nodes having one child. However, we need to keep track of the missing nodes somehow. A simple solution is to store a number called the skip value (s) in each node that indicates how many bits need to be skipped on the path. In Figure 15.12, we show that the sequences of nodes leading to P2 and P9 haveonlyonechild andhencearecandidates for path compression. The path compressed trie is shown in Figure 15.13. After path compression,level compressionis used forcompressingthepartsofthebinary trie that are densely populated. Instead of a node having two children, as in a binary trie, each internal node in a multibit trie is allowed to have 2k children, where k is called the CHAPTER 15 IP Address Lookup Algorithms 509 FIGURE 15.14 Levelcompressedtrie resulting from the trie shown in Figure 15.13. branching factor. The level compression is applied as follows. Every node n that is the root of a complete subtrie of maximum depth d is replaced by a corresponding one-level multibit subtrie. This process is repeated recursively on the children of the multibit trie thus obtained. Again, referring to Figure 15.13, we find that the trie rooted at the left child of node P4 is a complete subtrie of depth 2. This trie can be replaced by a single-level multibit trie with four nodes as shown in Figure 15.14. The branching factor for the left child of node P4 is 2, indicated by the number enclosed in a circle adjacent to the node. The rest of the internal nodes have a default branching factor of 1 and are not shown in Figure 15.13(a). The leaf nodes have a branching factor of 0. Since at each step we replace several levels with a single-level multibit trie, this process can be viewed as the compression of levels of the original binary trie. Hence, the resultant trie is termed an LC-trie or simply an LC-trie. The main drawback of the scheme is that the structure of the binary trie determines the choice of strides at any given level without any regard for the height of the resulting multibit trie. This is because a subtrie of depth d can be substituted only if it contains all the 2d leaves (a full binary subtrie). Hence, a few missing nodes might have a considerable negative influence on the efficacy of the level compression. A simple optimization is to use a relaxed criterion where nearly complete binary tries are replaced with a multibit subtrie. In other words, if the nearly complete binary subtrie has a sufficient fraction of the 2d tries at level d, then it is replaced with a multibit subtrie of stride d pointing to 2d nodes. The parameter that controls the fraction is referred to as the fill factor x,0 < x ≤ 1. Such a relaxed criterion will decrease the depth of the trie but will introduce empty nodes into the trie. However, in practice, this scheme results in substantial time improvements with only a moderate increase in space. For optimizing storage, LC-tries do not the use the standard implementation technique that usesasetofchildpointersateachinternalnode.Instead,anLC-trieisrepresentedusinga single array, and each trie node is an element in the array. An interested reader is encouraged to refer [529] for further details. 510 15.6 Compressing Multibit Tries The LC-trie can be considered a special case of a variable stride trie. The algorithm for a variable stride trie using dynamic programming would indeed result in an LC-trie if it were the optimal solution for a given set of prefixes [658]. The worst-case time complexity for an LC-trie is O(W/k) andthespacecomplexityis O(2kNW/k),whichareverysimilartomultibit tries