DATA STRUCTURES FOR STATIC AND DYNAMIC ROUTER TABLES

By

KUN SUK KIM

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2003

To my mother,

Kyung Hwan Lee

my wife,

Hye Ryung

and my sons,

Jin Sung and **Daniel**

ACKNOWLEDGMENTS

I appreciate God for my exciting and wonderful time at the University of Florida. Interactions with the many multicultural students and great professors have enriched me in both academic and social aspects.

I would like to deeply thank Distinguished Professor Sartaj Sahni, my advisor for guiding me through the doctoral study. I thank God for having given me such an ideal mentor. I have learned from him how an advisor should treat his students. His enthusiastic and devoted attitude toward teaching and research has strongly affected me. He has inspired me to solve difficult problems and given me invaluable advices. I consider him as a role model for my life and hope to lift my abilities up to his high standards in the future. I also thank Drs. Randy Chow, Richard Newman, Shigang Chen, and Janise McNair for serving on my supervisory committee.

I am thankful to Venkatachary Srinivasan and Jon Sharp for giving their programming codes and comments for the multibit trie and BSL structures, respectively. With their help I could start my research smoothly.

Special thanks go to my former advisor, Dr. Yann-Hang Lee for supporting me for the first two semesters at University of Florida and encouraging me even after his moving to Arizona State University. I would like to thank former and current members of the Korean Student Association of the CISE department at the University of Florida for their assistance and friendship. Thanks go to Daeyoung Kim, Yoonmee Doh, Young Joon Byun, Myung Chul Song, and many others.

I am eternally grateful to Pastor Hee Young Sohn, who is my spiritual mentor; and to the ministers at Korean Baptist Church of Gainesville (KBCG), for continuously caring about me in Jesus love. Thanks go to Jin Kun Song, Dong Yul Sung,

and other former and current members of my cell church of KBCG for sharing their lives with me and praying for me.

I cannot adequately express my gratitude to my mother, Kyung Hwan Lee. She has worked hard to provide me with better educational opportunities that made it possible for me to get this degree. I would like to thank Moon Suk, my brother; and Mi Ok, my sister for their constant support and encouragement. I am also grateful to my parents-in-law, Dae Hun Song and Ok Jo Choi; my five elder sisters-in-law (and their husbands), Mi Hye, Mi Ah, Hye Young, and Hye Youn; and my younger sister-in-law, Hyo Jae for supporting me both materially and morally.

At the end I have to mention my nuclear family that has put up with my absence for many evenings while I finished this work. Thanks go to my wife, Hye Ryung; and my sons, Jin Sung and Daniel for their love and tolerance.

I am grateful to all for their help and guidance and hope to remember their love forever.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

DATA STRUCTURES FOR STATIC AND DYNAMIC ROUTER TABLES

By

Kun Suk Kim

August 2003

Chair: Sartaj K. Sahni
Major Department: Computer and Information Science and Engineering

The Internet has been growing exponentially since many users make use of new applications and want to connect their hosts to Internet. Because of increased processing power and high-speed communication links, packet header processing has become a major bottleneck in Internet routing. To improve packet forwarding, our study developed fast and efficient algorithms.

We improved on dynamic programming algorithms to determine the strides of optimal multibit tries by providing alternative dynamic programming formulations for both fixed- and variable-stride tries. While the asymptotic complexities of our algorithms are the same as those for the corresponding algorithms of [82], experiments using real IPv4 routing table data indicate that our algorithms run considerably faster. An added feature of our variable-stride trie algorithm is the ability to insert and delete prefixes taking a fraction of the time needed to construct an optimal variable-stride trie from scratch.

IP lookup in a collection of hash tables (CHT) organization can be done with $O(\log l_{dist})$ hash-table searches, where l_{dist} is the number of distinct prefix-lengths (also equal to the number of hash tables in the CHT). We developed an algorithm

that minimizes the storage required by the prefixes and markers for the resulting set of prefixes to reduce the value of l_{dist} by using the controlled prefix-expansion technique. Also, we proposed improvements to the heuristic of [80].

We developed a data structure called collection of red-black trees (CRBT) in which prefix matching, insertion, and deletion can each be done in $O(\log n)$ time, where n is the number of prefixes in the router table. Experiments using real IPv4 routing databases indicate that although the proposed data structure is slower than optimized variable-stride tries for longest prefix matching, the proposed data structure is considerably faster for the insert and delete operations.

We formulate a variant, alternative collection of red-black trees (ACRBT) from the CRBT data structure to develop data structures for bursty access patterns. By replacing the red-black trees used in the ACRBT with splay trees (or biased skip lists), we obtained the collection of splay trees (CST) structure (or the biased skip lists with prefix trees (BSLPT) structure) in which search, insert, and delete take $O(\log n)$ amortized time (or $O(\log n)$ expected time) per operation, where n is the number of prefixes in the router table. Experimental results using real IPv4 routing databases and synthetically generated search sequences as well as trace sequences indicate that the CST structure is best for extremely bursty access patterns. Otherwise, the ACRBT is recommended. Our experiments also indicate that a supernode implementation of the ACRBT usually has better search performance than does the traditional one-element-per-node implementation.

CHAPTER 1
INTRODUCTION

The influence of Internet evolution reaches not only to the technical fields of computer and communications but throughout society as we move toward increasing use of online services (e.g., electronic commerce and information acquisition). The Internet is a world-wide communication infrastructure in which individuals and their computers interact and collaborate without regard for geographic location. Beginning with the early research on *packet switching*[1] and the ARPANET,[2] government, industry, and academia have been cooperating to evolve and deploy this exciting new technology [12, 45].

The Internet is an internetwork that ties many groups of networks with a common Internet Protocol (IP) [6, 19]. Figure 1–1 shows routers as the switch points of an internetwork. A packet may pass through many different deployment classes of routers from source to destination. The enterprise router, located at the lowest level in the router hierarchy, must support tens to thousands of network routes and medium bandwidth interfaces of 100 Mbps to 1 Gbps. Access routers are aggregation points for corporations and residential services. Access routers must support thousands or tens of thousands of routes. Residential terminals are connected to modem pools of the telephone central office through plain old telephone service (POTS), cable service,

[1] This technology is fundamentally different from the *circuit switching* that was used by the telephone system. In a packet-switching system, data to be delivered is broken into small chunks called packet that are labeled to show where they come from and where they are to go.

[2] This project was sponsored by U.S. Department of Defense to develop a whole new scheme for postnuclear communication.

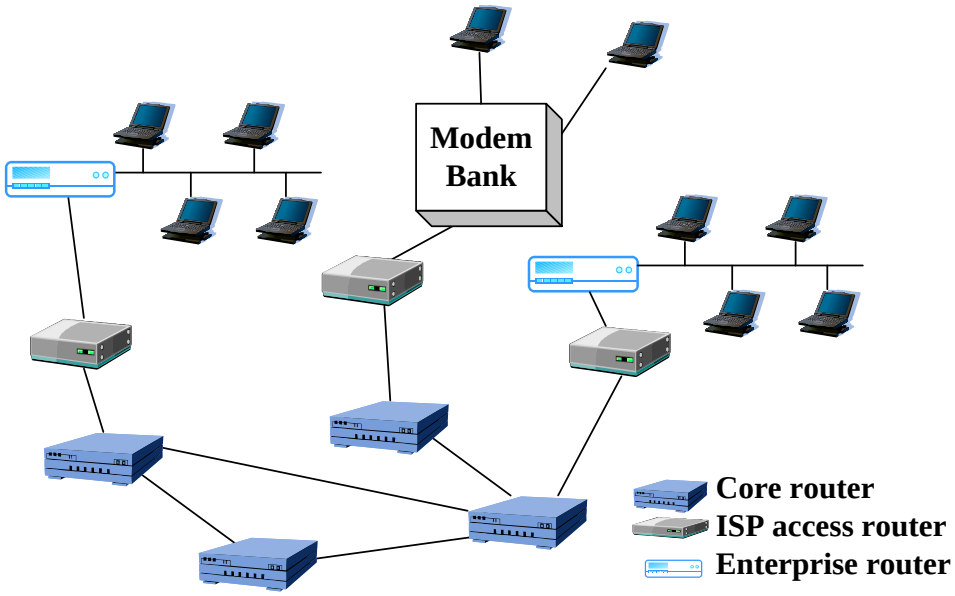


Figure 1–1: Internet structure

or one of the digital subscriber lines (DSLs). Backbone routers require multiples of high bandwidth ports such as OC-48 at 2.5 Gbps and OC-192 at 9.6 Gbps. Backbone routers cover national and international areas.

With the doubling of Internet traffic every 3 months [85] and the tripling of Internet hosts every 2 years [31], the importance of high speed scalable network routers cannot be overemphasized. Fast networking "will play a key role in enabling future progress" [54]. Fast networking requires fast routers; and fast routers require fast router table lookup.

The rest of this chapter is structured as follows. Section 1.1 introduces the basics of Internet routers. We describe the IP lookup and packet classification problems in Section 1.2. In Section 1.3 discusses related work in these fields. Finally, Section 1.4 presents an outline of the dissertation.

1.1 Internet Router

Figure 1–2 shows the generic architecture of an IP router. Generally, a router consists of the following basic components: the controller card, the router backplane, and line cards [3, 51]. The CPU in the controller card performs path computations

and router table maintenance. The line cards perform inbound and outbound packet forwarding. The router backplane transfers packets between the cards.

The basic functions in a router can be classified as routing, packet forwarding, switching, and queueing [3, 6, 51, 58]. We discuss the each function in more detail below.

- **Routing:** Routing is the process of communicating to other routers and exchanging route information to construct and maintain the router tables that are used by the packet-forwarding function. Routing protocols that are used to learn about and create a view of the network's topology include the routing information protocol (RIP) [37], open shortest path first (OSPF) [55], border gateway protocol (BGP) [47, 65], distance vector multicast routing protocol (DVMRP) [86], and protocol independent multicast (PIM) [27].
- **Packet forwarding:** The router looks at each incoming packet and performs a table lookup to decide which output port to use. This is based on the destination IP address in the incoming packet. The result of this lookup may imply a local, unicast, or multicast delivery. A local delivery is the case that the destination address is one of the router's local addresses and the packet is locally delivered. A unicast delivery sends the packet to a single output port. A multicast delivery is done through a set of output ports, depending on the multicast group membership of the router. In addition to table lookup, routers must perform other functions.
  - **Packet validation:** This function checks to see that the recieved IPv4 packet is properly formed for the protocol before it proceeds with protocol processing. However, because the checksum calculation is considered too expensive, current routers hardly verify the checksum, instead assuming that packets are transmitted through reliable media like fiber optics and assuming that end hosts will recognize any possible corruption.

  - **Packet lifetime control:** The router adjusts the time-to-live (TTL) field in the packet to prevent packets looping endlessly in the network. A host sending a packet initializes the TTL with 64 (recommended by [67]) or 255 (the maximum). A packet being routed to output ports has its TTL value decremented by 1. A packet whose TTL is zero before reaching the destination is discarded by the router.
  - **Checksum update:** The IP header checksum must be recalculated since the TTL field was changed. RFC 1071 [8] contains implementation techniques for computing the IP checksum. If only the TTL was decremented by 1, a router can efficiently update the checksum incrementally instead of calculating the checksum over the entire IP header [48].

- **Packet switching:** Packet switching is the process of moving packets from one interface to other port interface based on the forwarding decision. Packet switching can be done at very high speed [14, 52].
- **Queueing:** Queueing is the action of buffering each packet in a small memory for a short time (on the order of a few microseconds) during processing of the packet. Queueing can be done at the input, in the switch fabric, and/or at the output.

1.1.1 Internet Protocols

The headers of the IPv4 and IPv6 protocols [21, 59] are shown in Figure 1–3. Unicast packets are forwarded based on the destination address field. Each router between source and destination must look at this field. Multicast packets are forwarded based on the source network and destination group address. The protocol field defines the transport protocol (e.g., TCP [60] and UDP [61]) that is encapsulated within this IP packet. The type-of-service (ToS) field notices a packet's priority, its queueing, and its dropping behavior (to the routers). Some applications such as telnet and FTP set these flags.

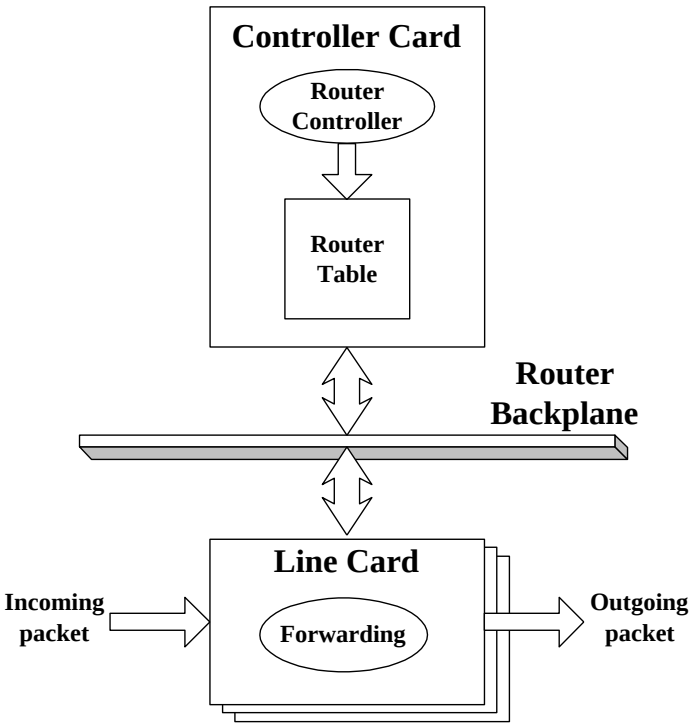
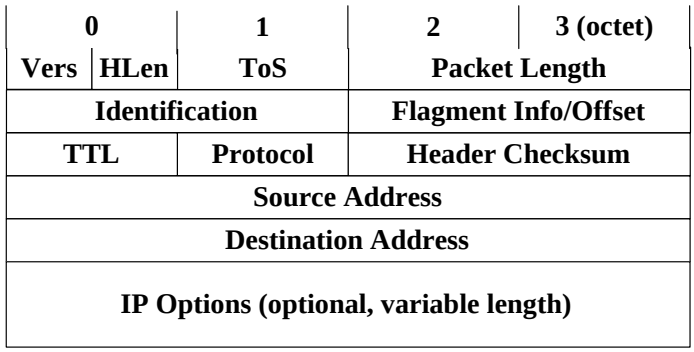


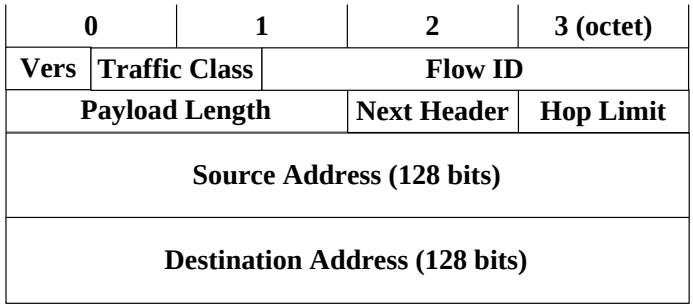
Figure 1–2: Generic router architecture

The most notable change from the IPv4 to the IPv6 header is the address length of 128 bits. Payload length is the length of the IPv6 payload in octets. Next header uses the same value as the IPv4 protocol field. Hop limit is decremented by 1 by each node that forwards the packet. The packet is discarded if the hop limit is decremented to zero. The flow ID field is added to simplify packet classification. The tuple (source address, flow ID) uniquely identifies a flow for any nonzero flow ID.

The headers of two transport protocols, as shown in Figure 1–4, provide more information (such as source and destination port numbers and flags that are used to further classify packets). In TCP and UDP networks, a port is an endpoint to a logical connection and the way a client program specifies a specific server program on a computer in a network. These two port numbers are used to distribute packets to the application and represent the fine-grained variety of flows. They can be used to identify a flow within the network. Thus, applications can reserve the resources to



(a) Format for an IPv4 header



(b) Format for an IPv6 header

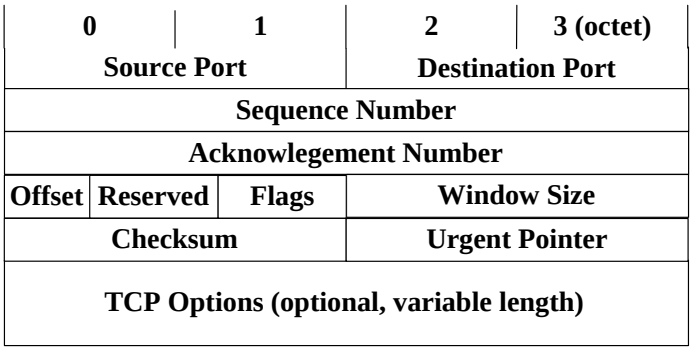
Figure 1–3: Formats for IP packet header

guarantee their service requirement using appropriate signalling protocol like RSVP [9].

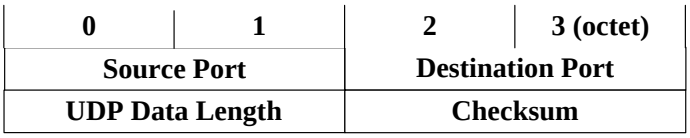
Some ports have numbers that are preassigned to them, and these are known as *well-known ports*. Port numbers range from 0 to 65536, but only ports numbers 0 to 1024 are reserved for privileged services and designated as well-known ports [67]. Each of these well-known port numbers specifies the port used by the server process as its contact port. For example, port numbers 20, 25, and 80 are assigned for FTP, simple mail transfer protocol (SMTP) [42], and hypertext transfer protocol (HTTP) [29] servers, respectively.

1.1.2 Classless Inter-Domain Routing (CIDR)

As the Internet has evolved and grown, it faced two serious scaling problems [38].



(a) TCP header



(b) UDP header

Figure 1–4: Transport protocol header formats

- **Exhaustion of IP address space:** In the old Class A, B, and C address scheme, a fundamental cause of this problem was the lack of a network class of a size that is appropriate for a mid-sized organization. Class C with a maximum of 254 host addresses is too small; while class B, which allows up to 65534 addresses, is too large to be densely populated. The result is inefficient use of class B network numbers. For example, if you needed 500 addresses to configure a network, you would be assigned Class B. However, that means 65034 unused addresses.
- **Routing information overload:** As the number of networks on the Internet increased, so did the number of routes. The size and rate of growth of the router tables in Internet routers is beyond the ability to effectively manage it.

CIDR is a mechanism to slow the growth of router tables and allow for more efficient allocation of IP addresses than the old Class A, B, and C address scheme. Two solutions to these problems were developed and adopted by the Internet community [66, 30].

- **Restructuring IP address assignments:** Instead of being limited to a network identifier (or *prefix*) of 8, 16, or 24 bits, CIDR uses generalized prefixes anywhere from 13 to 27 bits. Thus, blocks of addresses can be assigned to networks with 32 hosts or to those with over 500,000 hosts. This allows for address assignments that much more closely fit an organization's specific needs.
- **Hierarchical routing aggregation:** The CIDR addressing scheme also enables *route aggregation* in which a single high-level route entry can represent many lower-level routes in the global router tables. Big blocks of addresses are assigned to the large Internet Service Providers (ISPs) who then re-allocate portions of their address blocks to their customers. For example, a tier-1 ISP (e.g., Sprint and Pacific Bell) was assigned a CIDR address block with a prefix of 14 bits and typically assigns its customers, who may be smaller tier-2 ISPs, CIDR addresses with prefixes ranging from 27 bits to 18 bits. These customers in turn re-allocate portions of their address block to their users and/or customers (tier-3 or local ISPs). In the backbone router tables all these different networks and hosts can be represented by the single tier-1 ISP route entry. In this way, the growth in the number of router table entries at each level in the network hierarchy has been significantly reduced.

1.1.3 Packet Forwarding

Consider a part of the Internet in Figure 1–5(a) to get an intuitive idea of packet delivery. If a user in Chicago wishes to send a packet to Orlando, the packet is sent to a router R4. The router R4 may send this packet on the communication link L3 to a router R1. The router R1 may then send the packet on link L4 to a router R5 in Orlando. R4 then sends the packet to the final destination.

An Internet router table is a set of tuples of the form (p, a), where p is a binary string whose length is at most W ($W = 32$ for IPv4 destination addresses and $W = 128$ for IPv6), and a is an output link (or next hop). When a packet with destination

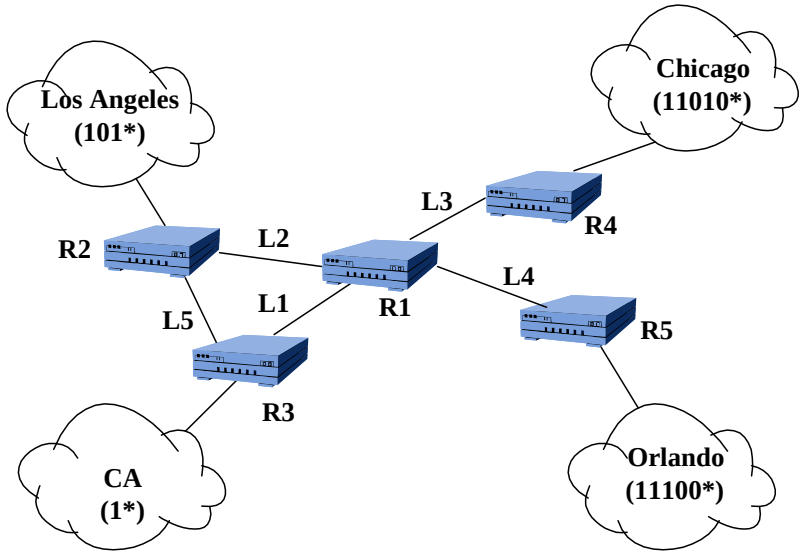
address A arrives at a router, we are to find the pair (p, a) in the router table for which p is a longest matching prefix of A (i.e., p is a prefix of A and there is no longer prefix q of A such that (q, b) is in the table). Once this pair is determined, the packet is sent to output link a. The speed at which the router can route packets is limited by the time it takes to perform this table lookup for each packet.

For example, consider a router table at the router R1 in Figure 1–5(a), shown in Figure 1–5(b). Assume that when a packet arrives on router R1, the packet carries the destination address 101110 in its header. In this example we assume that the longest prefix length is 6. To forward the packet to its final destination, router R1 consults a router table, which lists each possible prefix and the corresponding output link. The address 101110 matches both 1* and 101* in the router table, but 101* is the longest matching prefix. Since the table indicates output link L2, the router then switches the packet to L2.

Longest prefix routing is used because this results in smaller and more manageable router tables. It is impractical for a router table to contain an entry for each of the possible destination addresses. Two of the reasons this is so are

- The number of such entries would be almost one hundred million and would triple every 3 years.
- Every time a new host comes online, all router tables must incorporate the new host's address.

By using longest prefix routing, the size of router tables is contained to a reasonable quantity; and information about host/router changes made in one part of the Internet need not be propagated throughout the Internet.



(a) Backbone routers

Prefix	Output Link
1*	L1
101*	L2
11010*	L3
11100*	L4

(b) Router table for router R1

Figure 1–5: Router table example

1.2 Packet Classification

An Internet router classifies incoming packets into flows,[3] using information contained in packet headers and a table of (classification) rules. This table is called the *rule table* (equivalently, *router table*). The packet-header information that is used to perform the classification is some subset of the source and destination addresses, the source and destination ports, the protocol, protocol flags, type of service, and so

[3] A **flow** is a set of packets that are to be treated similarly for routing purposes.

on. The specific header information used for packet classification is governed by the rules in the rule table. Each rule-table rule is a pair of the form (F, A), where F is a filter and A is an action. The action component of a rule specifies what is to be done when a packet that satisfies the rule filter is received. Sample actions are drop the packet, forward the packet along a certain output link, and reserve a specified amount of bandwidth. A rule filter F is a tuple that is comprised of one or more fields. In the simplest case of destination-based packet forwarding, F has a single field, which is a destination (address) prefix; and A is the next hop for packets whose destination address has the specified prefix. For example, the rule $(01*, a)$ states that the next hop for packets whose destination address (in binary) begins with 01 is a. IP multicasting uses rules in which F comprises the two fields source prefix and destination prefix; QoS routers may use five-field rule filters (source-address prefix, destination-address prefix, source-port range, destination-port range, and protocol); and firewall filters may have one or more fields.

In the d-dimensional packet classification problem, each rule has a d-field filter. Our study was concerned solely with 1-dimensional packet classification. It should be noted, that data structures for multidimensional packet classification are usually built on top of data structures for 1-dimensional packet classification. Therefore, the study of data structures for 1-dimensional packet classification is fundamental to the design and development of data structures for d-dimensional, $d > 1$, packet classification.

For the 1-dimensional packet classification problem, we assume that the single field in the filter is the destination field; and that the action is the next hop for the packet. With these assumptions, 1-dimensional packet classification is equivalent to the destination-based packet forwarding problem. Henceforth, we use the terms rule table and router table to mean tables in which the filters have a single field, which is the destination address. This single field of a filter may be specified in one of two ways:

- **As a range:** For example, the range [35, 2096] matches all destination addresses d such that $35 \leq d \leq 2096$.
- **As an address/mask pair:** Let x_i denote the ith bit of x. The address/mask pair a/m matches all destination addresses d for which $d_i = a_i$ for all i for which $m_i = 1$. That is, a 1 in the mask specifies a bit position in which d and a must agree; while a 0 in the mask specifies a don't-care bit position. For example, the address/mask pair 101100/011101 matches the destination addresses 101100, 101110, 001100, and 001110. When all the 1-bits of a mask are to the left of all 0-bits, the address/mask pair specifies an address prefix. For example, 101100/110000 matches all destination addresses that have the prefix 10 (i.e., all destination addresses that begin with 10). In this case, the address/mask pair is simply represented as the prefix 10*, where the * denotes a sequence of don't-care bits. If W is the length, in bits, of a destination address, then the * in 10* represents all sequences of $(W - 2)$ bits. In IPv4 the address and mask are both 32 bits; while in IPv6 both of these are 128 bits.

Notice that every prefix may be represented as a range. For example, when $W = 6$, the prefix 10* is equivalent to the range [32, 47]. A range that may be specified as a prefix for some W is called a *prefix range*. The specification 101100/011101 may be abbreviated to ?011?0, where ? denotes a don't-care bit. This specification is not equivalent to any single range. Also, the range specification [3,6] is not equivalent to any single address/mask specification.

When more than one rule matches an incoming packet, a *tie* occurs. To select one of the many rules that may match an incoming packet, we use a *tie breaker*.

Let RS be the set of rules in a rule table and let FS be the set of filters associated with these rules. $rules(d, RS)$ (or simply $rules(d)$ when RS is implicit) is the subset of rules of RS that match/cover the destination address d. $filters(d, FS)$ and

$filters(d)$ are defined similarly. A tie occurs whenever $|rules(d)| > 1$ (equivalently, $|filters(d)| > 1$).

Three popular tie breakers are

- **First matching rule in table:** The rule table is assumed to be a linear list ([39]) of rules with the rules indexed 1 through n for an n-rule table. The action corresponding to the first rule in the table that matches the incoming packet is used. In other words, for packets with destination address d, the rule of $rules(d)$ that has least index is selected.

  For our example router table corresponding to the five prefixes of Figure 4–1, rule R1 is selected for every incoming packet, because P1 matches every destination address. When using the first-matching-rule criteria, we must index the rules carefully. In our example, P1 should correspond to the last rule so that every other rule has a chance to be selected for at least one destination address.

- **Highest-priority rule:** Each rule in the rule table is assigned a priority. From among the rules that match an incoming packet, the rule that has highest priority wins is selected. To avoid the possibility of a further tie, rules are assigned different priorities (it is actually sufficient to ensure that for every destination address d, $rules(d)$ does not have two or more highest-priority rules). Notice that the first-matching-rule criteria is a special case of the highest-priority criteria (simply assign each rule a priortiy equal to the negative of its index in the linear list).

- **Most-specific-rule matching:** The filter $F1$ is **more specific** than the filter $F2$ iff $F2$ matches all packets matched by $F1$ plus at least one additional packet. So, for example, the range $[2, 4]$ is more specific than $[1, 6]$, and $[5, 9]$ is more specific than $[5, 12]$. Since $[2, 4]$ and $[8, 14]$ are disjoint (i.e., they have no address in common), neither is more specific than the other. Also, since $[4, 14]$ and

$[6, 20]$ intersect[4] , neither is more specific than the other. The prefix 110* is more specific than the prefix 11*.

In most-specific-rule matching, ties are broken by selecting the matching rule that has the most specific filter. When the filters are destination prefixes, the most-specific-rule that matches a given destination d is the longest[5] prefix in $filters(d)$. Hence, for prefix filters, the most-specific-rule tie breaker is equivalent to the longest-matching-prefix criteria used in router tables. For our example rule set, when the destination address is 18, the longest matching-prefix is P4.

When the filters are ranges, the most-specific-rule tie breaker requires us to select the most specific range in $filters(d)$. Notice also that most-specific-range matching is a special case of the the highest-priority rule. For example, when the filters are prefixes, set the prefix priority equal to the prefix length. For the case of ranges, the range priority equals the negtive of the range size.

In a *static* rule table, the rule set does not vary in time. For these tables, we are concerned primarily with the following metrics:

- **Time required to process an incoming packet:** This is the time required to search the rule table for the rule to use.
- **Preprocessing time:** This is the time to create the rule-table data structure.
- **Storage requirement:** That is, how much memory is required by the rule-table data structure?

---

[4] Two ranges $[u, v]$ and $[x, y]$ intersect iff $u < x \leq v < y \vee x < u \leq y < v$.

[5] The length of a prefix is the number of bits in that prefix (note that the * is not used in determining prefix length). The length of P1 is 0 and that of P2 is 4.

In practice, rule tables are seldom truly static. At best, rules may be added to or deleted from the rule table infrequently. Typically, in a "static" rule table, inserts/deletes are batched and the rule-table data structure reconstructed as needed.

In a *dynamic* rule table, rules are added/deleted with some frequency. For such tables, inserts/deletes are not batched. Rather, they are performed in real time. For such tables, we are concerned additionally with the time required to insert/delete a rule. For a dynamic rule table, the initial rule-table data structure is constructed by starting with an empty data structures and then inserting the initial set of rules into the data structure one by one. So, typically, in the case of dynamic tables, the preprocessing metric, mentioned above, is very closely related to the insert time.

In this dissertation, we focus on data structures for static and dynamic router tables (1-dimensional packet classification) in which the filters are either prefixes or ranges. Although some of our data structures apply equally well to all three of the commonly used tie breakers, our focus, in this dissertation, is on longest-prefix matching.

1.3 Prior Work

Several solutions for the IP lookup problem (i.e., finding the longest matching prefix) have been proposed. Let $LMP(d)$ be the longest matching-prefix for address d.

1.3.1 Linear List

In this data structure, the rules of the rule table are stored as a linear list ([39]) L. The $LMP(d)$ is determined by examining the prefixes in L from left to right; for each prefix, we determine whether or not that prefix matches d; and from the set of matching prefixes, the one with longest length is selected. To insert a rule q, we first search the list L from left to right to ensure that L doesn't already have a rule with the same filter as does q. Having verified this, the new rule q is added to the end of the list. Deletion is similar. The time for each of the operations to determine

$LMP(d)$, insert a rule, and delete a rule is $O(n)$, where n is the number of rules in L. The memory required is also $O(n)$.

Note that this data structure may be used regardless of the form of the filter (i.e., ranges, Boolean expressions, etc.) and regardless of the tie breaker in use. The time and memory complexities are unchanged.

1.3.2 End-Point Array

Lampson, Srinivasan, and Varghese [44] proposed a data structure in which the end points of the ranges defined by the prefixes are stored in ascending order in an array. The $LMP(d)$ is found by performaing a binary search on this ordered array of end points. Although Lampson et al. [44] provide ways to reduce the complexity of the search for the LMP by a constant factor, these methods do not result in schemes that permit prefix insertion and deletion in $O(\log n)$ time.

It should be noted that the end-point array may be used even when ties are broken by selecting the first matching rule or the highest-priority matching rule. Further, the method applies to the case when the filters are arbitrary ranges rather than simply prefixes. The complexity of the preprocessing step (i.e., creation of the array of ordered end-points) and the search for the rule to use is unchanged. Further, the memory requirements are the same, $O(n)$ for an n-rule table, regardless of the tie breaker and whether the filters are prefixes or general ranges.

1.3.3 Sets of Equal-Length Prefixes

Waldvogel et al. [87] proposed a data structure to determine $LMP(d)$ by performing a binary search on prefix length. In this data structure, the prefixes in the router table T are partitioned into the sets S_0, S_1, ... such that S_i contains all prefixes of T whose length is i. For simplicity, we assume that T contains the default prefix *. So, $S_0 = \{*\}$. Next, each S_i is augmented with markers that represent prefixes in S_j such that $j > i$ and i is on the binary search path to S_j. For example, suppose that the length of the longest prefix of T is 32 and that the length of $LMP(d)$ is 22.

To find $LMP(d)$ by a binary search on length, we will first search S_{16} for an entry that matches the first 16 bits of d. This search[6] will need to be successful for us to proceed to a larger length. The next search will be in S_{24}. This search will need to fail. Then, we will search S_{20} followed by S_{22}. So, the path followed by a binary search on length to get to S_{22} is S_{16}, S_{24}, S_{20}, and S_{22}. For this to be followed, the searches in S_{16}, S_{20}, and S_{22} must succeed while that in S_{24} must fail. Since the length of $LMP(d)$ is 22, T has no matching prefix whose length is more than 22. So, the search in S_{24} is guaranteed to fail. Similarly, the search in S_{22} is guaranteed to succeed. However, the searches in S_{16} and S_{20} will succeed iff T has matching prefixes of length 16 and 20. To ensure success, every length 22 prefix P places a *marker* in S_{16} and S_{20}, the marker in S_{16} is the first 16 bits of P and that in S_{20} is the first 20 bits in P. Note that a marker M is placed in S_i only if S_i doesn't contain a prefix equal to M. Notice also that for each i, the binary search path to S_i has $O(\log l_{max}) = O(\log W)$, where l_{max} is the length of the longest prefix in T, S_js on it. So, each prefix creates $O(\log W)$ markers. With each marker M in S_i, we record the longest prefix of T that matches M (the length of this longest matching-prefix is neccessarily smaller than i).

To determine $LMP(d)$, we begin by setting $leftEnd = 0$ and $rightEnd = l_{max}$. The repetitive step of the binary search requires us to search for an entry in S_m, where $m = \lfloor (leftEnd + rightEnd)/2 \rfloor$, that equals the first m bits of d. If S_m does not have such an entry, set $rightEnd = m - 1$. Otherwise, if the matching entry is the prefix P, P becomes the longest matching-prefix found so far. If the matching entry is the marker M, the prefix recorded with M is the longest matching-prefix

[6] When searching S_i, only the first i bits of d are used, because all prefixes in S_i have exactly i bits.

found so far. In either case, set $leftEnd = m + 1$. The binary search terminates when $leftEnd > rightEnd$.

One may easily establish the correctness of the described binary search. Since, each prefix creates $O(\log W)$ markers, the memory requirement of the scheme is $O(n \log W)$. When each set S_i is represented as a hash table, the data structure is called SELPH (sets of equal length prefixes using hash tables). The expected time to find $LMP(d)$ is $O(\log W)$ when the router table is represented as an SELPH. When inserting a prefix, $O(\log W)$ markers must also be inserted. With each marker, we must record a longest-matching prefix. The expected time to find these longest matching-prefixes is $O(\log^2 W)$. In addition, we may need to update the longest-matching prefix information stored with the $O(n \log W)$ markers at lengths greater than the length of the newly inserted prefix. This takes $O(n \log^2 W)$ time. So, the expected insert time is $O(n \log^2 W)$. When deleting a prefix P, we must search all hash tables for markers M that have P recorded with them and then update the recorded prefix for each of these markers. For hash tables with a bounded loading density, the expected time for a delete (including marker-prefix updates) is $O(n \log^2 W)$. Waldvogel et al. [87] have shown that by inserting the prefixes in ascending order of length, an n-prefix SELPH may be constructed in $O(n \log^2 W)$ time.

When each set is represented as a balanced search tree, the data structure is called SELPT. In an SELPT, the time to find $LMP(d)$ is $O(\log n \log W)$; the insert time is $O(n \log n \log^2 W)$; the delete time is $O(n \log n \log^2 W)$; and the time to construct the data structure for n prefixes is $O(W + n \log n \log^2 W)$.

In the full version of [87], Waldvogel et al. show that by using a technique called marker partitioning, the SELPH data structure may be modified to have a search time of $O(\alpha + \log W)$ and an insert/delete time of $O(\alpha \sqrt[\alpha]{n} W \log W)$, for any $\alpha > 1$.

Because of the excessive insert and delete times, the sets of equal-length prefixes data structure is suitable only for static router tables. By using the prefix expansion

method [22, 82], we can limit the number of distinct lengths in the prefix set and so reduce the run time by a constant factor [87].

1.3.4 Tries

IP lookup in the BSD kernel is done using the Patricia data structure [78], which is a variant of a compressed binary trie [39]. This scheme requires $O(W)$ memory accesses per lookup, insert, and delete. We note that the lookup complexity of longest prefix matching algorithms is generally measured by the number of accesses made to main memory (equivalently, the number of cache misses). Dynamic prefix tries, which are an extension of Patricia, and which also take $O(W)$ memory accesses for lookup, were proposed by Doeringer et al. [23].

For IPv4 prefix sets, Degermark et al. [22] proposed the use of a three-level trie in which the strides are 16, 8, and 8. They propose encoding the nodes in this trie using bit vectors to reduce memory requirements. The resulting data structure requires at most 12 memory accesses. However, inserts and deletes are quite expensive. For example, the insertion of the prefix 1* changes up to 2^{15} entries in the trie's root node. All of these changes may propogate into the compacted storage scheme of [22].

The multibit trie data structures of Srinivasan and Varghese [82] are, perhaps, the most flexible and effective trie structure for IP lookup. Using a technique called controlled prefix expansion, which is very similar to the technique used in [22], tries of a predetermined height (and hence with a predetermined number of memory accesses per lookup) may be constructed for any prefix set. Srinivasan and Varghese [82] develop dynamic programming algorithms to obtain space optimal fixed-stride tries (FSTs) and variable-stride tries (VSTs) of a given height.

Lampson et al. [44] proposed the use of hybrid data structures comprised of a stride-16 root and an auxiliary data structure for each of the subtries of the stride-16 root. This auxiliary data structure could be the end-point array (since each subtrie is expected to contain only a small number of prefixes, the number of end points in

each end-point array is also expected to be quite small). An alternative auxiliary data structure suggested by Lampson et al. [44] is a 6-way search tree for IPv4 router tables. In the case of these 6-way trees, the keys are the remaining up to 16 bits of the prefix (recall that the stride-16 root consumes the first 16 bits of a prefix). For IPv6 prefixes, a multicolumn scheme is suggested [44]. None of these proposed structures is suitable for a dynamic table.

Nilsson and Karlsson [57] propose a greedy heuristic to construct optimal VSTs. They call the resulting VSTs LC-tries (level-compressed tries). An LC-tries obtained from a 1-bit trie by replacing full subtries of the 1-bit trie by single multibit nodes. This replacement is done by examining the 1-bit trie top to bottom (i.e., from root to leaves).

1.3.5 Binary Search Trees

Suri et al. [84] proposed a B-tree data structure for dynamic router tables. Using their structure, we may find the longest matching prefix in $O(\log n)$ time. However, inserts/deletes take $O(W \log n)$ time. The number of cache misses is $O(\log n)$ for each operation. When W bits fit in $O(1)$ words (as is the case for IPv4 and IPv6 prefixes) logical operations on W-bit vectors can be done in $O(1)$ time each. In this case, the scheme of [84] takes $O(\log W * \log n)$ time for an insert and $O(W + \log n) = O(W)$ time for a delete.

Several researchers ([16, 25, 26, 36, 74], for example), have investigated router table data structures that account for bias in access patterns. Gupta, Prabhakar, and Boyd [36], for example, propose the use of ranges. They assume that access frequencies for the ranges are known, and they construct a bounded-height binary search tree of ranges. This binary search tree accounts for the known range access frequencies to obtain near-optimal IP lookup. Although the scheme of [36] performs IP lookup in near-optimal time, changes in the access frequencies, or the insertion

or removal of a prefix require us to reconstruct the data structure, a task that takes $O(n \log n)$ time.

Ergun et al. [25, 26] use ranges to develop a biased skip list structure that performs longest prefix-matching in $O(\log n)$ expected time. Their scheme is designed to give good expected performance for bursty[7] access patterns". The biased skip list scheme of Ergun et al. [25, 26] permits inserts and deletes in $O(\log n)$ time only in the severely restricted and impractical situation when all prefixes in the router table are of the same length. For the more general, and practical, case when the router table comprises prefixes of different length, their scheme takes $O(n)$ expected time for each insert and delete.

1.3.6 Others

Cheung and McCanne [16] developed "a model for table-driven route lookup and cast the table design problem as an optimization problem within this model." Their model accounts for the memory hierarchy of modern computers and they optimize average performance rather than worst-case performance.

Gupta and McKeown [33] examine the asymptotic complexity of a related problem, packet classification. They develop two data structures, heap-on-trie (HoT) and binary-search-tree-on-trie (BoT), for the dynamic packet classification problem. The complexity of these data structures (for packet classification and the insertion and deletion of rules) also is dependent on W. For d-dimensional rules, a search in a HoT takes $O(W^d)$ and an update (insert or delete) takes $O(W^d \log n)$ time. The corresponding times for a BoT are $O(W^d \log n)$ and $O(W^{d-1} \log n)$, respectively.

---

[7] In a *bursty* access pattern the number of different destination addresses in any subsequence of q packets is $<< q$. That is, if the destination of the current packet is d, there is a high probability that d is also the destination for one or more of the next few packets. The fact that Internet packets tend to be bursty has been noted in [18, 46], for example.

Hardware solutions that involve the use of content addressable memory [50] as well as solutions that involve modifications to the Internet Protocol (i.e., the addition of information to each packet) have also been proposed [10, 13, 56].

1.4 Dissertation Outline

The remainder of the dissertation is organized as follows. Chapters 2 and 3 concentrate on two data structures for static router table, in which the rule set does not vary in time. In Chapter 2, we develop new dynamic programming formulations for the construction of space optimal tries of a predetermined height. In Chapter 3, we develop an algorithm that minimizes storage requirement for collection of hash table optimization problem. Also, we propose improvements to the heuristic of [80].

Chapters 4 and 5 provide data structures for dynamic router tables, in which rules are added/deleted with some frequency. In Chapter 4, we show how to use the range encoding idea of [44] so that longest prefix matching as well as prefix insertion and deletion can be done in $O(\log n)$ time. Chapter 5 presents the management of router tables for a dynamic environment (i.e., search, insert, and deletes are performed dynamically) in which the access pattern is bursty.

CHAPTER 2
MULTIBIT TRIES

In this chapter, we focus on the controlled expansion technique of Srinivasan and Varghese [82]. In particular, we develop new dynamic programming formulations for the construction of space optimal tries of a predetermined height. While the asymptotic complexities of the algorithms that result from our formulations are the same as those for the corresponding algorithms of [82], experiments using real IPv4 routing table data indicate that our algorithms run considerably faster. Our fixed-stride trie algorithm is 2 to 4 times as fast on a SUN workstation and 1.5 to 3 times as fast on a Pentium 4 PC. On a SUN workstation, our variable-stride trie algorithm is between 2 and 17 times as fast as the corresponding algorithm of [82]; on a Pentium 4 PC, our algorithm is between 3 and 47 times as fast.

In Section 2.1, we describe the data structure for 1-bit tries. We develop our new dynamic programming formulations for both fixed-stride and variable-stride tries in Section 2.2 and 2.3, respectively. In Section 2.4, we present our experimental results.

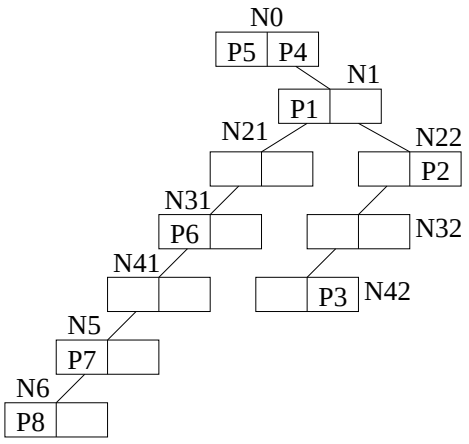
2.1 1-Bit Tries

A *1-bit trie* is a tree-like structure in which each node has a left child, left data, right child, and right data field. Nodes at level $l-1$ of the trie store prefixes whose length is l (the length of a prefix is the number of bits in that prefix; the terminating * (if present) does not count towards the prefix length). If the rightmost bit in a prefix whose length is l is 0, the prefix is stored in the left data field of a node that is at level $l-1$; otherwise, the prefix is stored in the right data field of a node that is at level $l-1$. At level i of a trie, branching is done by examining bit i (bits are numbered from left to right beginning with the number 0, and levels are numbered with the root being at level 0) of a prefix or destination address. When bit i is 0, we move into the

left subtree; when the bit is 1, we move into the right subtree. Figure 2–1(a) gives the prefixes in the 8-prefix example of [82], and Figure 2–1(b) shows the corresponding 1-bit trie. The prefixes in Figure 2–1(a) are numbered and ordered as in [82]. Since the trie of Figure 2–1(b) has a height of 6, a search into this trie may make up to 7 memory accesses. The total memory required for the 1-bit trie of Figure 2–1(b) is 20 units (each node requires 2 units, one for each pair of (child, data) fields). The 1-bit tries described here are an extension of the 1-bit tries described in [39]. The primary difference being that the 1-bit tries of [39] are for the case when all keys (prefixes) have the same length.

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

(a) 8-prefix example of [82]



(b) Corresponding 1-bit trie

Figure 2–1: Prefixes and corresponding 1-bit trie

When 1-bit tries are used to represent IPv4 router tables, the trie height may be as much as 31. A lookup in such a trie takes up to 32 memory accesses. Table 2–1 gives the characteristics of five IPv4 backbone router prefix sets, and Table 2–2 gives a more detailed characterization of the prefixes in the largest of these five databases, Paix [53]. For our five databases, the number of nodes in a 1-bit trie is between $2n$ and $3n$, where n is the number of prefixes in the database (Table 2–1).

Table 2–1: Prefix databases obtained from IPMA project on Sep 13, 2000

Database	Prefixes	16-bit prefixes	24-bit prefixes	Nodes*
Paix	85,682	6,606	49,756	173,012
Pb	35,151	2,684	19,444	91,718
MaeWest	30,599	2,500	16,260	81,104
Aads	26,970	2,236	14,468	74,290
MaeEast	22,630	1,810	11,386	65,862

* The last column shows the number of nodes in the 1-bit trie representation of the prefix database.
Note: the number of prefixes stored at level i of a 1-bit trie equals the number of prefixes whose length is $i + 1$.

Table 2–2: Distributions of the prefixes and nodes in the 1-bit trie for Paix

Level	Number of prefixes	Number of nodes	Level	Number of prefixes	Number of nodes
0	0	1	16	918	5,117
1	0	2	17	1,787	8,245
2	0	4	18	5,862	12,634
3	0	7	19	3,614	15,504
4	0	11	20	3,750	20,557
5	0	20	21	5,525	26,811
6	0	36	22	7,217	32,476
7	22	62	23	49,756	37,467
8	4	93	24	12	54
9	5	169	25	26	44
10	9	303	26	12	20
11	26	561	27	5	9
12	56	1,037	28	4	5
13	176	1,933	29	1	2
14	288	3,552	30	0	1
15	6,606	6,274	31	1	1

2.2 Fixed-Stride Tries

2.2.1 Definition

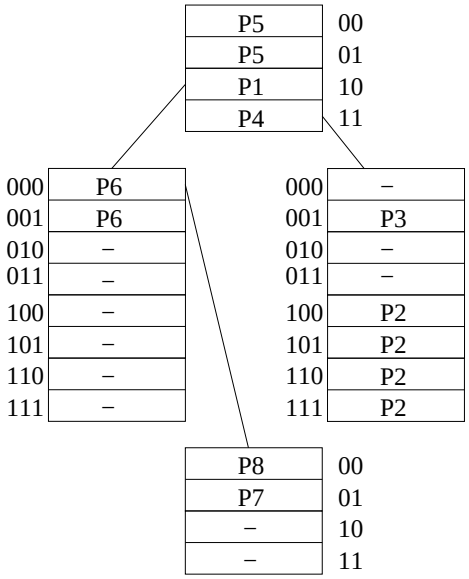
Srinivasan and Varghese [82] have proposed the use of fixed-stride tries to enable fast identification of the longest matching prefix in a router table. The *stride* of a node is defined to be the number of bits used at that node to determine which branch to take. A node whose stride is s has 2^s child fields (corresponding to the 2^s possible values for the s bits that are used) and 2^s data fields. Such a node requires 2^s memory units. In a *fixed-stride trie* (FST), all nodes at the same level have the same stride; nodes at different levels may have different strides.

Suppose we wish to represent the prefixes of Figure 2–1(a) using an FST that has three levels. Assume that the strides are 2, 3, and 2. The root of the trie stores prefixes whose length is 2; the level one nodes store prefixes whose length is 5 (2 + 3); and level three nodes store prefixes whose length is 7 (2 + 3 + 2). This poses a problem for the prefixes of our example, because the length of some of these prefixes is different from the storeable lengths. For instance, the length of P5 is 1. To get around this problem, a prefix with a nonpermissible length is expanded to the next permissible length. For example, P5 = 0* is expanded to P5a = 00* and P5b = 01*. If one of the newly created prefixes is a duplicate, natural dominance rules are used to eliminate all but one occurrence of the prefix. For instance, P4 = 1* is expanded to P4a = 10* and P4b = 11*. However, P1 = 10* is to be chosen over P4a = 10*, because P1 is a longer match than P4. So, P4a is eliminated. Because of the elimination of duplicate prefixes from the expanded prefix set, all prefixes are distinct. Figure 2–2(a) shows the prefixes that result when we expand the prefixes of Figure 2–1 to lengths 2, 5, and 7. Figure 2–2(b) shows the corresponding FST whose height is 2 and whose strides are 2, 3, and 2.

Since the trie of Figure 2–2(b) can be searched with at most 3 memory references, it represents a time performance improvement over the 1-bit trie of Figure 2–1(b),

Expanded prefixes (3 levels)
00* (P5a)
01* (P5b)
10* (P1)
11* (P4)
11100* (P2a)
11101* (P2b)
11110* (P2c)
11111* (P2d)
11001* (P3)
10000* (P6a)
10001* (P6b)
1000001* (P7)
1000000* (P8)

(a) Expanded prefixes



(b) Corresponding fixed-stride trie

Figure 2–2: Prefix expansion and fixed-stride trie

which requires up to 7 memory references to perform a search. However, the space requirements of the FST of Figure 2–2(b) are more than that of the corresponding 1-bit trie. For the root of the FST, we need 8 fields or 4 units; the two level 1 nodes require 8 units each; and the level 3 node requires 4 units. The total is 24 memory units.

We may represent the prefixes of Figure 2–1(a) using a one-level trie whose root has a stride of 7. Using such a trie, searches could be performed making a single memory access. However, the one-level trie would require $2^7 = 128$ memory units.

2.2.2 Construction of Optimal Fixed-Stride Tries

In the *fixed-stride trie optimization* (FSTO) problem, we are given a set P of prefixes and an integer k. We are to select the strides for a k-level FST in such a manner that the k-level FST for the given prefixes uses the smallest amount of memory.

For some P, a k-level FST may actually require more space than a $(k-1)$-level FST. For example, when $P = \{00^*, 01^*, 10^*, 11^*\}$, the unique 1-level FST for P requires 4 memory units while the unique 2-level FST (which is actually the 1-bit trie for P) requires 6 memory units. Since the search time for a $(k-1)$-level FST is less than that for a k-level FST, we would actually prefer $(k-1)$-level FSTs that take less (or even equal) memory over k-level FSTs. Therefore, in practice, we are really interested in determining the best FST that uses at most k levels (rather than exactly k levels). The *modified* FSTO problem (MFSTO) is to determine the best FST that uses at most k levels for the given prefix set P.

Let O be the 1-bit trie for the given set of prefixes, and let F be any k-level FST for this prefix set. Let s_0, ..., s_{k-1} be the strides for F. We shall say that level 0 of F covers levels $0, ..., s_0 - 1$ of O, and that level j, $0 < j < k$ of F covers levels $a, ..., b$ of O, where $a = \sum_{q=0}^{j-1} s_q$ and $b = \sum_{q=0}^{j} s_q - 1$. So, level 0 of the FST of Figure 2–2(b) covers levels 0 and 1 of the 1-bit trie of Figure 2–1(b). Level 1 of this FST covers levels 2, 3, and 4 of the 1-bit trie of Figure 2–1(b); and level 2 of this FST covers levels 5 and 6 of the 1-bit trie. We shall refer to levels $e_u = \sum_{q=0}^{u-1} s_q$, $0 \leq u < k$ as the *expansion levels* of O. The expansion levels defined by the FST of Figure 2–2(b) are 0, 2, and 5.

Let $nodes(i)$ be the number of nodes at level i of the 1-bit trie O. For the 1-bit trie of Figure 2–1(a), $nodes(0:6) = [1, 1, 2, 2, 2, 1, 1]$. The memory required by F is $\sum_{q=0}^{k-1} nodes(e_q) * 2^{s_q}$. For example, the memory required by the FST of Figure 2–2(b) is $nodes(0) * 2^2 + nodes(2) * 2^3 + nodes(5) * 2^2 = 24$.

Let $T(j, r)$, $r \leq j + 1$, be the cost (i.e., memory requirement) of the best way to cover levels 0 through j of O using exactly r expansion levels. When the maximum prefix length is W, $T(W-1, k)$ is the cost of the best k-level FST for the given set of prefixes. Srinivasan and Varghese [82] have obtained the following dynamic programming recurrence for T:

$$T(j,r) = \min_{m \in \{r-2..j-1\}} \{T(m,r-1) + nodes(m+1) * 2^{j-m}\}, r > 1 \qquad (2.1)$$

$$T(j,1) = 2^{j+1} \qquad (2.2)$$

The rationale for Equation 2.1 is that the best way to cover levels 0 through j of O using exactly r expansion levels, $r > 1$, must have its last expansion level at level $m+1$ of O, where m must be at least $r-2$ (as otherwise, we do not have enough levels between levels 0 and m of O to select the remaining $r-1$ expansion levels) and at most $j-1$ (because the last expansion level is $\leq j$). When the last expansion level is level $m+1$, the stride for this level is $j-m$, and the number of nodes at this expansion level is $nodes(m+1)$. For optimality, levels 0 through m of O must be covered in the best possible way using exactly $r-1$ expansion levels.

As noted by Srinivasan and Varghese [82], using the above recurrence, we may determine $T(W-1,k)$ in $O(kW^2)$ time (excluding the time needed to compute O from the given prefix set and determine $nodes()$). The strides for the optimal k-level FST can be obtained in an additional $O(k)$ time. Since, Equation 2.1 also may be used to compute $T(W-1,q)$ for all $q \leq k$ in $O(kW^2)$ time, we can actually solve the MFSTO problem in the same asymptotic complexity as required for the FSTO problem.

We can reduce the time needed to solve the MFSTO problem by modifying the definition of T. The modified function is C, where $C(j,r)$ is the cost of the best FST that uses *at most* r expansion levels. It is easy to see that $C(j,r) \leq C(j,r-1)$, $r > 1$. A simple dynamic programming recurrence for C is:

$$C(j,r) = \min_{m \in \{-1..j-1\}} \{C(m,r-1) + nodes(m+1) * 2^{j-m}\}, j \geq 0, r > 1 \qquad (2.3)$$

$$C(-1,r) = 0 \text{ and } C(j,1) = 2^{j+1}, j \geq 0 \tag{2.4}$$

To see the correctness of Equations 2.3 and 2.4, note that when $j \geq 0$, there must be at least one expansion level. If $r = 1$, then there is eactly one expansion level and the cost is 2^{j+1}. If $r > 1$, the last expansion level in the best FST could be at any of the levels 0 through j. Let $m + 1$ be this last expansion level. The cost of the covering is $C(m, r-1) + nodes(m+1) * 2^{j-m}$. When $j = -1$, no levels of the 1-bit trie remain to be covered. Therefore, $C(-1, r) = 0$.

We may obtain an alternative recurrence for $C(j,r)$ in which the range of m on the right side is $r-2..j-1$ rather than $-1..j-1$. First, we obtain the following dynamic programming recurrence for C:

$$C(j,r) = \min\{C(j,r-1), T(j,r)\}, \quad r > 1 \tag{2.5}$$

$$C(j,1) = 2^{j+1} \tag{2.6}$$

The rationale for Equation 2.5 is that the best FST that uses at most r expansion levels either uses at most $r-1$ levels or uses exactly r levels. When at most $r-1$ levels are used, the cost is $C(j, r-1)$, and when exactly r levels are used, the cost is $T(j,r)$, which is defined by Equation 2.1.

Let $U(j,r)$ be as defined in Equation 2.7.

$$U(j,r) = \min_{m \in \{r-2..j-1\}} \{C(m, r-1) + nodes(m+1) * 2^{j-m}\} \tag{2.7}$$

From Equations 2.1 and 2.5 we obtain

$$C(j,r) = \min\{C(j,r-1), U(j,r)\} \tag{2.8}$$

To see the correctness of Equation 2.8, note that for all j and r such that $r \leq j+1$, $T(j,r) \geq C(j,r)$. Furthermore,

$$\begin{aligned} T(j,r) &= \min_{m \in \{r-2..j-1\}} \{T(m,r-1) + nodes(m+1) * 2^{j-m}\} \\ &\geq \min_{m \in \{r-2..j-1\}} \{C(m,r-1) + nodes(m+1) * 2^{j-m}\} \\ &= U(j,r) \end{aligned} \tag{2.9}$$

Therefore, when $C(j,r-1) \leq U(j,r)$, Equations 2.5 and 2.8 compute the same value for $C(j,r)$ (i.e., $C(j,r-1)$). When $C(j,r-1) > U(j,r)$, it appears from Equation 2.9 that Equation 2.8 may compute a smaller $C(j,r)$ than is computed by Equation 2.5. However, from Equation 2.3, which is equivalent to Equation 2.5, the $C(j,r)$ computed by Equations 2.3 and 2.5 satisifes

$$\begin{aligned} C(j,r) &= \min_{m \in \{-1..j-1\}} \{C(m,r-1) + nodes(m+1) * 2^{j-m}\} \\ &\leq \min_{m \in \{r-2..j-1\}} \{C(m,r-1) + nodes(m+1) * 2^{j-m}\} \\ &= U(j,r) \end{aligned}$$

where $C(-1,r) = 0$. However, when $C(j,r-1) > U(j,r)$, the $C(j,r)$ computed by Equation 2.8 is $U(j,r)$. Therefore, when $C(j,r-1) > U(j,r)$, the $C(j,r)$ computed by Equation 2.8 cannot be smaller than that computed by Equation 2.5. Therefore, the $C(j,r)$s computed by Equations 2.5 and 2.8 are equal.

In the remainder of this section, we use Equations 2.3 and 2.4 for C. The range for m (in Equation 2.3) may be restricted to a range that is (often) considerably smaller than $r-2..j-1$. To obtain this narrower search range, we first establish a few properties of 1-bit tries and their corresponding optimal FSTs.

Lemma 1 *For every 1-bit trie O, (a) $nodes(i) \leq 2^i$, $i \geq 0$ and (b) $nodes(i+j) \leq 2^j * nodes(i)$, $j \geq 0$, $i \geq 0$.*

Proof Follows from the fact that a 1-bit trie is a binary tree. ■

Let $M(j,r)$, $r > 1$, be the smallest m that minimizes

$$C(m, r-1) + nodes(m+1) * 2^{j-m},$$

in Equation 2.3.

Lemma 2 $\forall(j \geq 0, r > 1)[M(j+1,r) \geq M(j,r)]$.

Proof Let $M(j,r) = a$ and $M(j+1,r) = b$.

Suppose $b < a$. Then,

$$\begin{aligned} C(j,r) &= C(a, r-1) + nodes(a+1) * 2^{j-a} \\ &< C(b, r-1) + nodes(b+1) * 2^{j-b} \end{aligned}$$

since, otherwise, $M(j,r) = b$. Furthermore,

$$\begin{aligned} C(j+1,r) &= C(b, r-1) + nodes(b+1) * 2^{j+1-b} \\ &\leq C(a, r-1) + nodes(a+1) * 2^{j+1-a}. \end{aligned}$$

Therefore,

$$\begin{aligned} & nodes(a+1) * 2^{j-a} + nodes(b+1) * 2^{j+1-b} \\ < \;& nodes(b+1) * 2^{j-b} + nodes(a+1) * 2^{j+1-a} \end{aligned}$$

So,

$$nodes(b+1) * 2^{j-b} < nodes(a+1) * 2^{j-a}$$

Hence,

$$2^{a-b} * nodes(b+1) < nodes(a+1)$$

This contradicts Lemma 1(b). So, $b \geq a$. ■

Lemma 3 $\forall(j \geq 0, r > 0)[C(j,r) < C(j+1,r)]$.

Proof The case $r = 1$ follows from $C(j,1) = 2^{j+1}$. So, assume $r > 1$. From the definition of M, it follows that

$$C(j+1,r) = C(b,r-1) + nodes(b+1) * 2^{j+1-b},$$

where $-1 \leq b = M(j+1,r) \leq j$. When $b < j$, we get

$$\begin{aligned} C(j,r) &\leq C(b,r-1) + nodes(b+1) * 2^{j-b} \\ &< C(b,r-1) + nodes(b+1) * 2^{j+1-b} \\ &= C(j+1,r) \end{aligned}$$

When $b = j$,

$$C(j+1,r) = C(j,r-1) + nodes(j+1) * 2 > C(j,r-1),$$

since $nodes(j+1) > 0$. ■

The next few lemmas use the function Δ, which is defined as $\Delta(j,r) = C(j,r-1) - C(j,r)$. Since, $C(j,r) \leq C(j,r-1)$, $\Delta(j,r) \geq 0$ for all $j \geq 0$ and all $r \geq 2$.

Lemma 4 $\forall(j \geq 0)[\Delta(j,2) \leq \Delta(j+1,2)]$.

Proof If $C(j,2) = C(j,1)$, there is nothing to prove as $\Delta(j+1,2) \geq 0$. The only other possibility is $C(j,2) < C(j,1)$ (i.e., $\Delta(j,2) > 0$). In this case, the best cover for levels 0 through j uses exactly 2 expansion levels. From the recurrence for C (Equations 2.3 and 2.4), it follows that $C(j,1) = 2^{j+1}$, and

$$\begin{aligned} C(j,2) &= C(a,1) + nodes(a+1) * 2^{j-a} \\ &= 2^{a+1} - nodes(a+1) * 2^{j-a}, \end{aligned}$$

for some a, $0 \leq a < j$. Therefore,

$$\begin{aligned} \Delta(j,2) &= C(j,1) - C(j,2) \\ &= 2^{j+1} - 2^{a+1} - nodes(a+1) * 2^{j-a}. \end{aligned}$$

From Equations 2.3 and 2.4, it follows that

$$\begin{aligned} C(j+1,2) &\leq C(a,1) + nodes(a+1) * 2^{j+1-a} \\ &= 2^{a+1} + nodes(a+1) * 2^{j+1-a}. \end{aligned}$$

Hence,

$$\Delta(j+1,2) \geq 2^{j+2} - 2^{a+1} - nodes(a+1) * 2^{j+1-a}.$$

Therefore,

$$\begin{aligned} \Delta(j+1,2) - \Delta(j,2) &\geq 2^{j+2} - 2^{a+1} - nodes(a+1) * 2^{j+1-a} \\ &\quad -2^{j+1} + 2^{a+1} + nodes(a+1) * 2^{j-a} \\ &= 2^{j+1} - nodes(a+1) * 2^{j-a} \\ &\geq 2^{j+1} - 2^{a+1} * 2^{j-a} \text{ (Lemma 1(a))} \\ &= 0 \end{aligned}$$

■

Lemma 5 $\forall(j \geq 0, k > 2)[\Delta(j,k-1) \leq \Delta(j+1,k-1)] \Longrightarrow \forall(j \geq 0, k > 2)[\Delta(j,k) \leq \Delta(j+1,k)]$.

Proof Assume that $\forall(j \geq 0, k > 2)[\Delta(j,k-1) \leq \Delta(j+1,k-1)]$. We shall show that $\forall(j \geq 0, k > 2)[\Delta(j,k) \leq \Delta(j+1,k)]$. Let $M(j,k) = b$ and $M(j+1,k-1) = c$.

Case 1: $c \geq b$.

$$\begin{aligned}
\Delta(j,k) &= C(j,k-1) - C(j,k) \\
&= C(j,k-1) - C(b,k-1) - nodes(b+1) * 2^{j-b} \\
&\leq C(b,k-2) + nodes(b+1) * 2^{j-b} \\
&\quad -C(b,k-1) - nodes(b+1) * 2^{j-b} \\
&= \Delta(b,k-1).
\end{aligned}$$

Also,

$$\begin{aligned}
\Delta(j+1,k) &= C(j+1,k-1) - C(j+1,k) \\
&\geq C(c,k-2) + nodes(c+1) * 2^{j+1-c} \\
&\quad -C(c,k-1) - nodes(c+1) * 2^{j+1-c} \\
&= \Delta(c,k-1).
\end{aligned}$$

Since $c \geq b$, $\Delta(b,k-1) \leq \Delta(c,k-1)$. Therefore,

$$\Delta(j+1,k) \geq \Delta(c,k-1) \geq \Delta(b,k-1) \geq \Delta(j,k).$$

Case 2: $c < b$.

Let $M(j+1,k) = a$, $M(j,k) = b$, $M(j+1,k-1) = c$, and $M(j,k-1) = d$. From Lemma 2, $a \geq b$ and $c \geq d$. Since $c < b$, $a \geq b > c \geq d$. Also,

$$\begin{aligned}
\Delta(j,k) &= C(j,k-1) - C(j,k) \\
&= [C(d,k-2) + nodes(d+1) * 2^{j-d}] \\
&\quad -[C(b,k-1) + nodes(b+1) * 2^{j-b}]
\end{aligned}$$

and

$$\begin{aligned}\Delta(j+1,k) &= C(j+1,k-1) - C(j+1,k) \\ &= [C(c,k-2) + nodes(c+1) * 2^{j+1-c}] \\ &\quad -[C(a,k-1) + nodes(a+1) * 2^{j+1-a}].\end{aligned}$$

Therefore,

$$\begin{aligned}\Delta(j+1,k) - \Delta(j,k) &= [C(c,k-2) + nodes(c+1) * 2^{j+1-c}] \\ &\quad -[C(d,k-2) + nodes(d+1) * 2^{j-d}] \\ &\quad +[C(b,k-1) + nodes(b+1) * 2^{j-b}] \\ &\quad -[C(a,k-1) + nodes(a+1) * 2^{j+1-a}]. \quad (2.10)\end{aligned}$$

Since $j > b > c \geq d = M(j,k-1)$,

$$\begin{aligned}C(c,k-2) &+ nodes(c+1) * 2^{j-c} \\ &\geq C(d,k-2) + nodes(d+1) * 2^{j-d} \quad (2.11)\end{aligned}$$

Furthermore, since $M(j+1,k) = a \geq b$,

$$\begin{aligned}C(b,k-1) &+ nodes(b+1) * 2^{j+1-b} \\ &\geq C(a,k-1) + nodes(a+1) * 2^{j+1-a} \quad (2.12)\end{aligned}$$

Substituting Equations 2.11 and 2.12 into Equation 2.10, we get

$$\Delta(j+1,k) - \Delta(j,k) \geq nodes(c+1) * 2^{j-c} - nodes(b+1) * 2^{j-b}.$$

Lemma 1 and $c < b$ imply $nodes(c+1) * 2^{b-c} \geq nodes(b+1)$. Therefore,

$$nodes(c+1) * 2^{j-c} \geq nodes(b+1) * 2^{j-b}.$$

So, $\Delta(j+1,k) - \Delta(j,k) \geq 0$. ■

Lemma 6 $\forall(j \geq 0, k \geq 2)[\Delta(j,k) \leq \Delta(j+1,k)]$.

Proof Follows from Lemmas 4 and 5. ■

Lemma 7 *Let* $k > 2$. $\forall(j \geq 0)[\Delta(j,k-1) \leq \Delta(j+1,k-1)] \Longrightarrow \forall(j \geq 0)[M(j,k) \geq M(j,k-1)]$.

Proof Assume that $\forall(j \geq 0)[\Delta(j,k-1) \leq \Delta(j+1,k-1)]$. Suppose that $M(j,k-1) = a$, $M(j,k) = b$, and $b < a$ for some j, $j \geq 0$. From Equation 2.3, we get

$$\begin{aligned} C(j,k) &= C(b,k-1) + nodes(b+1) * 2^{j-b} \\ &\leq C(a,k-1) + nodes(a+1) * 2^{j-a} \end{aligned}$$

and

$$\begin{aligned} C(j,k-1) &= C(a,k-2) + nodes(a+1) * 2^{j-a} \\ &< C(b,k-2) + nodes(b+1) * 2^{j-b}. \end{aligned}$$

Hence,

$$C(b,k-1) + C(a,k-2) < C(a,k-1) + C(b,k-2).$$

Therefore,

$$\Delta(a,k-1) < \Delta(b,k-1).$$

However, $b < a$ and $\forall(j \geq 0)[\Delta(j,k-1) \leq \Delta(j+1,k-1)]$ imply that $\Delta(b,k-1) \leq \Delta(a,k-1)$. Since our assumption that $b < a$ leads to a contradiction, it must be that there is no $j \geq 0$ for which $M(j,k-1) = a$, $M(j,k) = b$, and $b < a$. ■

Lemma 8 $\forall(j \geq 0, k > 2)[M(j,k) \geq M(j,k-1)]$.

Proof Follows from Lemmas 6 and 7. ■

Theorem 1 $\forall(j \geq 0, k > 2)[M(j,k) \geq max\{M(j-1,k), M(j,k-1)\}]$.

Proof Follows from Lemmas 2 and 8. ■

```
Algorithm FixedStrides(W, k)
// W is length of longest prefix.
// k is maximum number of expansion levels desired.
// Return C(W − 1, k) and compute M(∗, ∗).
{
    for (j = 0; j < W; j++){
        C(j, 1) := 2^(j+1);
        M(j, 1) := −1;}
    for (r = 1; r < k; r++)
        C(−1, r) := 0;
    for (r = 2; r ≤ k; r++)
        for (j = r − 1; j < W; j++){
        // Compute C(j, r).
            minJ := max(M(j − 1, r), M(j, r − 1));
            minCost := C(j, r − 1);
            minL := M(j, r − 1);
            for (m = minJ; m < j; m++){
                cost := C(m, j − 1) + nodes(m + 1) ∗ 2^(j−m);
                if (cost < minCost) then
                    {minCost := cost; minL := m;}}
            C(j, r) := minCost; M(j, r) := minL;}
    return C(W − 1, k);
}
```

Figure 2–3: Algorithm for fixed-stride tries.

Note 1 *From Lemma 6, it follows that whenever* $\Delta(j,k) > 0$, $\Delta(q,k) > 0$, $\forall q > j$.

Theorem 1 leads to Algorithm *FixedStrides* (Figure 2–3), which computes $C(W-1,k)$. The complexity of this algorithm is $O(kW^2)$. Using the computed M values, the strides for the OFST that uses at most k expansion levels may be determined in an additional $O(k)$ time. Although our algorithm has the same asymptotic complexity as does the algorithm of Srinivasan and Varghese [82], experiments conducted by us using real prefix sets indicate that our algorithm runs faster.

2.3 Variable-Stride Tries

2.3.1 Definition and Construction

In a *variable-stride trie* (VST) [82], nodes at the same level may have different strides. Figure 2–4 shows a two-level VST for the 1-bit trie of Figure 2–1. The stride

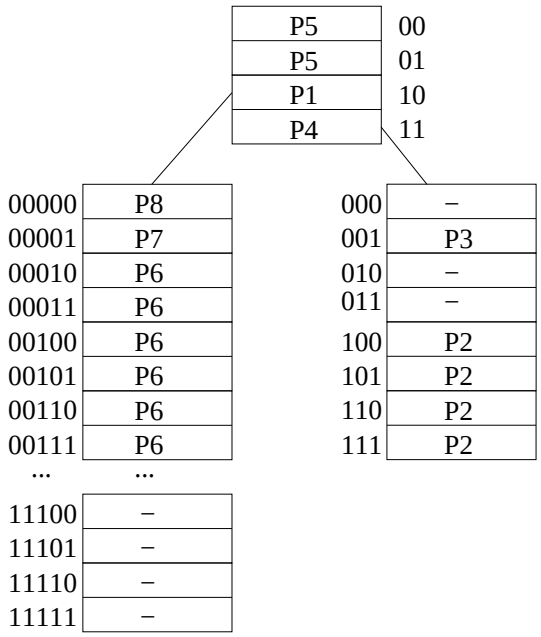


Figure 2–4: Two-level VST for prefixes of Figure 2–1(a)

for the root is 2; that for the left child of the root is 5; and that for the root's right child is 3. The memory requirement of this VBT is 4 (root) + 32 (left child of root) + 8 (right child of root) = 44.

Since FSTs are a special case of VSTs, the memory required by the best VST for a given prefix set P and number of expansion levels k is less than or equal to that required by the best FST for P and k. Despite this, FSTs may be preferred in certain router applications "because of their simplicity and slightly faster search time" [82].

Let r-VST be a VST that has at most r levels. Let $Opt(N, r)$ be the cost (i.e., memory requirement) of the best r-VST for a 1-bit trie whose root is N. Srinivasan and Varghese [82] have obtained the following dynamic programming recurrence for $Opt(N, r)$.

$$Opt(N, r) = \min_{s \in \{1..1+height(N)\}} \{2^s + \sum_{Q \in D_s(N)} Opt(Q, r-1)\}, \quad r > 1 \tag{2.13}$$

where $D_s(N)$ is the set of all descendents of N that are at level s of N. For example, $D_1(N)$ is the set of children of N and $D_2(N)$ is the set of grandchildren of N. $height(N)$ is the maximum level at which the trie rooted at N has a node. For example, in Figure 2–1(b), the height of the trie rooted at N1 is 5. When $r = 1$,

$$Opt(N, 1) = 2^{1+height(N)} \tag{2.14}$$

Equation 2.14 is equivalent to Equation 2.2; the cost of covering all levels of N using at most one expansion level is $2^{1+height(N)}$. When more than one expansion level is permissible, the stride of the first expansion level may be any number s that is between 1 and $1 + height(N)$. For any such selection of s, the next expansion level is level s of the 1-bit trie whose root is N. The sum in Equation 2.13 gives the cost of the best way to cover all subtrees whose roots are at this next expansion level. Each such subtree is covered using at most $r - 1$ expansion levels. It is easy to see that $Opt(R, k)$, where R is the root of the overall 1-bit trie for the given prefix set P, is the cost of the best k-VST for P. Srinivasan and Varghese [82] describe a way to determine $Opt(R, k)$ using Equations 2.13 and 2.14. Although Srinivasan and Varghese state that the complexity of their algorithm is $O(nW^2k)$, where n is the number of prefixes in P and W is the length of the longest prefix, a close examination reveals that the complexity is $O(pWk)$, where p is the number of nodes in the 1-bit trie. Since $p = O(n)$ for realistic router prefix sets, the complexity of their algorithm is $O(nWk)$ on realistic router prefix sets.

We develop an alternative dynamic programming formulation that also permits the computation of $Opt(R, k)$ in $O(pWk)$ time. However, the resulting algorithm is considerably faster. Let

$$Opt(N, s, r) = \sum_{Q \in D_s(N)} Opt(Q, r),\ s > 0,\ r > 1,$$

and let $Opt(N, 0, r) = Opt(N, r)$. From Equations 2.13 and 2.14, we obtain:

$$Opt(N,0,r) = min_{s\in\{1..1+height(N)\}}\{2^s + Opt(N,s,r-1)\},\ r>1 \tag{2.15}$$

and

$$Opt(N,0,1) = 2^{1+height(N)}. \tag{2.16}$$

For $s>0$ and $r>1$, we get

$$\begin{aligned} Opt(N,s,r) &= \sum_{Q\in D_s(N)} Opt(Q,r) \\ &= Opt(LeftChild(N), s-1, r) \\ &+ Opt(RightChild(N), s-1, r). \end{aligned} \tag{2.17}$$

For Equation 2.17, we need the following initial condition:

$$Opt(null,*,*) = 0 \tag{2.18}$$

The number of $Opt(*,*,*)$ values is $O(pWk)$. Each $Opt(*,*,*)$ value may be computed in $O(1)$ time using Equations 2.15 through 2.18 provided the Opt values are computed in postorder. Therefore, we may compute $Opt(R,k) = Opt(R,0,k)$ in $O(pWk)$ time. Although both our algorithm and that of [82] run in $O(pWk)$ time, our algorithm is expected to do less work. We arrive at this expectation by performing a somewhat crude operation count analysis. In the algorithm of [82], for each value of r (see Equation 2.13), $Opt(M,r-1)$ is used $level_M$ times, where $level_M$ is the level for node M. Adding in 1 unit for the initial storage of $Opt(M,r-1)$, we see that a $level_M$ node contributes roughly $level_M+1$ to the total cost of computing $Opt(*,r)$. Therefore, a rough operation count for the algorithm of [82] is

$$OpCountSrini = k * \sum_M (level_M + 1)$$

where the sum is taken over all nodes M of the 1-bit trie.

Let $height_M$ be the height of the subtree rooted at node M of the 1-bit trie (the height of a subtree that has only a root is 0). Our algorithm computes $(height_M+1)k$ $Opt(M,*,*)$ values at node M. Each of these values is computed using a single addition. So, the operation count for our algorithm is crudely estimated to be

$$OpCountOur = k * \sum_{M} (height_M + 1)$$

For our five databases Paix, Pb, MaeWest, Aads, and MaeEast, the ratios $OpCountSrini/OpCountOur$ are 6.7, 5.9, 5.7, 5.6, and 5.4. We can determine the possible range for this ratio by computing the ratio for skewed as well as full binary trees.

For a totally skewed 1-bit trie (e.g., a left or right skewed trie), the two operation count estimates are the same. For a 1-bit trie that is a full binary tree of height $W-1$,

$$\begin{aligned} OpCountSrini/k &= \sum_{0}^{W-1} (p+1)2^p \\ &= (W-1)2^W + 1 \end{aligned}$$

and

$$\begin{aligned} OpCountOur/k &= \sum_{0}^{W-1} (W-p)2^p \\ &= 2^{W+1} - W - 2 \end{aligned}$$

So, $OpCountSrini/OpCountOur \approx (W-1)/2$. Since skewed and full binary trees represent two extremes for the operation count ratio, the operation count ratio is expected to be between 1 and $(W-1)/2$. For IPv4, $W = 32$ and this ratio lies between 1 and 15.5. For IPv6, $W = 128$ and this ratio is between 1 and 63.5.

Although the number of operations being performed is an important contributing factor to the observed run time of an algorithm, the number of cache misses often has significant impact. For the algorithm of [82], we estimate the number of cache

misses to be of the same order as the number of operations (i.e., $OpCountSrini$). Because our algorithm is a simple postorder traversal that visits each node of the 1-bit trie exactly once, the number of cache misses for our algorithm is estimated to be $OpCountOur/L$, where L is the smaller of k and the number of $Opt(M,*,*)$ values that fit in a cache line. The cache miss count gives our algorithm another factor of L advantage over the algorithm of [82].

When the cost of operations dominates the run time, our crude analysis indicates that our algorithm will be about 6 times as fast as that of [82] (for our test databases). When cache miss time dominates the run time, our algorithm could be 12 times as fast when $k = 2$ and 42 times as fast when $k = 7$. Of course, since our analysis doesn't include all of the overheads associated with the two algorithms, actual speedups may be quite different.

Our algorithm requires $O(W^2k)$ memory for the $Opt(*,*,*)$ values. To see this, notice that there can be at most $W + 1$ nodes N whose $Opt(N,*,*)$ values must be retained at any given time, and for each of these at most $W + 1$ nodes, $O(Wk)$ $Opt(N,*,*)$ values must be retained. To determine the optimal strides, each node of the 1-bit trie must store the stride s that minimizes the right side of Equation 2.15 for each value of r. For this purpose, each 1-bit trie node needs $O(k)$ space. Therefore, the memory requirements of the 1-bit trie are $O(pk)$. The total memory required is, therefore, $O(pk + W^2k)$.

In practice, we may prefer an implementation that uses considerably more memory. If we associate a cost array with each of the p nodes of the 1-bit trie, the memory requirement increases to $O(pWk)$. The advantage of this increased memory implementation is that the optimal strides can be recomputed in $O(W^2k)$ time (rather than $O(pWk)$) following each insert or delete of a prefix. This is so because, the $Opt(N,*,*)$ values need be recomputed only for nodes along the insert/delete path of the 1-bit trie. There are $O(W)$ such nodes.

$P1 = 0*$	$0 * (P1)$
$P2 = 1*$	$1 * (P2)$
$P3 = 11*$	$101 * (P4)$
$P4 = 101*$	$110 * (P3)$
$P5 = 10001*$	$111 * (P3)$
$P6 = 1100*$	$10001 * (P5)$
$P7 = 110000*$	$11000 * (P6)$
$P8 = 1100000*$	$11001 * (P6)$
	$1100000 * (P8)$
	$1100001 * (P7)$
(a) Original prefixes	(b) Expanded prefixes

Figure 2–5: A prefix set and its expansion to four lengths

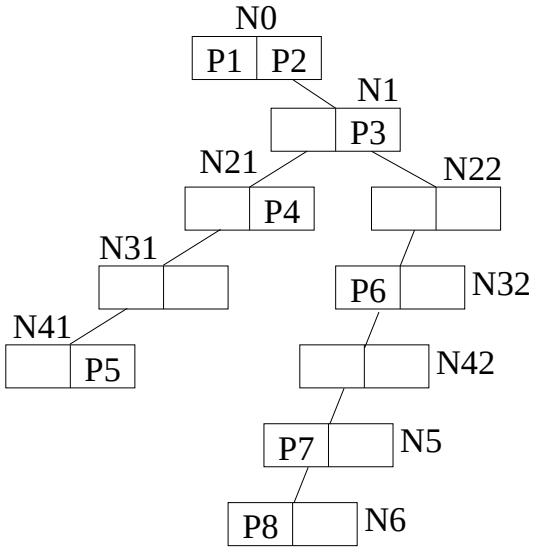


Figure 2–6: 1-bit trie for prefixes of Figure 2–5(a)

2.3.2 An Example

Figure 2–5(a) gives a prefix set P that contains 8 prefixes. The length of the longest prefix (P8) is 7. Figure 2–5(b) gives the prefixes that remain when the prefixes of P are expanded into the lengths 1, 3, 5, and 7. As we shall see, these expanded prefixes correspond to an optimal 4-VST for P. Figure 2–6 gives the 1-bit trie for the prefixes of Figure 2–5.

To determine the cost, $Opt(N0, 0, 4)$, of the best 4-VST for the prefix set of Figure 2–5(a), we must compute all the Opt values shown in Figure 2–7. In this figure Opt_1, for example, refers to $Opt(N1, *, *)$ and Opt_{42} refers to $Opt(N42, *, *)$.

Opt_0	$r=1$	2	3	4
$s=0$	128	26	20	18
1	64	18	16	16
2	40	18	16	16
3	20	12	12	12
4	10	8	8	8
5	4	4	4	4
6	2	2	2	2

Opt_1	$r=1$	2	3	4
$s=0$	64	18	16	16
1	40	18	16	16
2	20	12	12	12
3	10	8	8	8
4	4	4	4	4
5	2	2	2	2

Opt_{21}	$r=1$	2	3	4
$s=0$	8	6	6	6
1	4	4	4	4
2	2	2	2	2

Opt_{22}	$r=1$	2	3	4
$s=0$	32	12	10	10
1	16	8	8	8
2	8	6	6	6
3	4	4	4	4
4	2	2	2	2

Opt_{31}	$r=1$	2	3	4
$s=0$	4	4	4	4
1	2	2	2	2

Opt_{32}	$r=1$	2	3	4
$s=0$	16	8	8	8
1	8	6	6	6
2	4	4	4	4
3	2	2	2	2

Opt_{41}	$r=1$	2	3	4
$s=0$	2	2	2	2

Opt_5	$r=1$	2	3	4
$s=0$	4	4	4	4
1	2	2	2	2

Opt_{42}	$r=1$	2	3	4
$s=0$	8	6	6	6
1	4	4	4	4
2	2	2	2	2

Opt_6	$r=1$	2	3	4
$s=0$	2	2	2	2

Figure 2–7: *Opt* values in the computation of $Opt(N0, 0, 4)$

The *Opt* arrays shown in Figure 2–7 are computed in postorder; that is, in the order N41, N31, N21, N6, N5, N42, N32, N22, N1, N0. The *Opt* values shown in Figure 2–7 were computed using Equations 2.15 through 2.18.

From Figure 2–7, we determine that the cost of the best 4-VST for the given prefix set is $Opt(N0, 0, 4) = 18$. To construct this best 4-VST, we must determine the strides for all nodes in the best 4-VST. These strides are easily determined if, with each $Opt(*, 0, *)$, we store the s value that minimizes the right side of Equation 2.15. For $Opt(N0, 0, 4)$, this minimizing s value is 1. This means that the stride for the root of the best 4-VST is 1, its left subtree is empty (because N0 has an empty

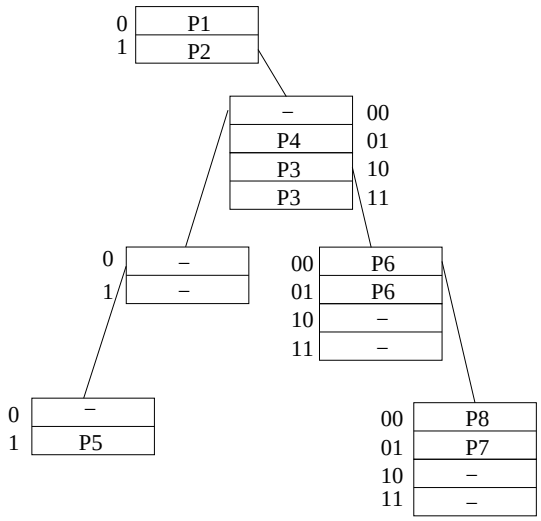


Figure 2–8: Optimal 4-VST for prefixes of Figure 2–5(a)

left subtree), its right subtree is the best 3-VST for the subtree rooted at N1. The minimizing s value for $Opt(N1, 0, 3)$ is 2 (actually, there is a tie between $s = 2$ and $s = 3$; ties may be broken arbitrarily). Therefore, the right child of the root of the best 4-VST has a stride of 2. Its first subtree is the best 2-VST for N31; its second subtree is empty; its third subtree is the best 2-VST for N32; and its fourth subtree is empty. Continuing in this manner, we obtain the 4-VST of Figure 2–8. The cost of this 4-VST is 18.

2.3.3 Faster $k = 2$ Algorithm

The algorithm of Section 2.3.1 may be used to determine the optimal 2-VST for a set of n prefixes in $O(pW)$ (equal to $O(nW)$ for practical prefix sets) time, where p is the number of nodes in the 1-bit trie and W is the length of the longest prefix. In this section, we develop an $O(p)$-time algorithm for this task.

From Equation 2.13, we see that the cost, $Opt(root, 2)$ of the best 2-VST is

```
Algorithm ComputeC(t)
// Initial invocation is ComputeC(root).
// The C array and level are initialized to 0 prior to initial invocation.
// Return height of tree rooted at node t.
{
    if (t != null) {
        level++;
        leftHeight = ComputeC(t.leftChild);
        rightHeight = ComputeC(t.rightChild);
        level--;
        height = max{leftHeight, rightHeight} + 1;
        C[level] += 2^(height+1);
        return height;
        }
    else return -1;
}
```

Figure 2–9: Algorithm to compute C using Equation 2.20.

$$\begin{aligned} Opt(root, 2) &= min_{s\in\{1..1+height(root)\}}\{2^s + \sum_{Q\in D_s(root)} Opt(Q, 1)\} \\ &= min_{s\in\{1..1+height(root)\}}\{2^s + \sum_{Q\in D_s(root)} 2^{1+height(Q)}\} \\ &= min_{s\in\{1..1+height(root)\}}\{2^s + C(s)\} \end{aligned} \tag{2.19}$$

where

$$C(s) = \sum_{Q\in D_s(root)} 2^{1+height(Q)} \tag{2.20}$$

We may compute $C(s)$, $1 \le s \le 1 + height(root)$, in $O(p)$ time by performing a postorder traversal (see Figure 2–9) of the 1-bit trie rooted at $root$. (Recall that p is the number of nodes in the 1-bit trie.)

Once we have determined the C values using Algorithm $ComputeC$ (Figure 2–9), we may determine $Opt(root, 2)$ and the optimal stride for the root in an additional $O(height(root))$ time using Equation 2.19. If the optimal stride for the root is s, then

the second expansion level is level s (unless, $s = 1 + height(root)$, in which case there isn't a second expansion level). The stride for each node at level s is one plus the height of the subtree rooted at that node. The height of the subtree rooted at each node was computed by Algorithm *ComputeC*, and so the strides for the nodes at the second expansion level are easily determined.

2.3.4 Faster $k = 3$ Algorithm

Using the algorithm of Section 2.3.1 we may determine the optimal 3-VST in $O(pW)$ time. In this section, we develop a simpler and faster $O(pW)$ algorithm for this task. On practical prefix sets, the algorithm of this section runs in $O(p)$ time.

From Equation 2.13, we see that the cost, $Opt(root, 3)$ of the best 3-VST is

$$\begin{aligned} Opt(root, 3) &= min_{s \in \{1..1+height(root)\}}\{2^s + \sum_{Q \in D_s(root)} Opt(Q, 2)\} \\ &= min_{s \in \{1..1+height(root)\}}\{2^s + T(s)\} \end{aligned} \tag{2.21}$$

where

$$T(s) = \sum_{Q \in D_s(root)} Opt(Q, 2) \tag{2.22}$$

Figure 2–10 gives our algorithm to compute $T(s)$, $1 \leq s \leq 1 + height(root)$. The computation of $Opt(M, 2)$ is done using Equations 2.19 and 2.20. In Algorithm *ComputeT* (Figure 2–10), the method *allocate* allocates a one-dimensional array that is to be used to compute the C values for a subtree. The allocated array is initialized to zeroes; it has positions 0 through W, where W is the length of the longest prefix (W also is $1 + height(root)$); and when computing the C values for a subtree whose root is at level j, only positions j through W of the allocated array may be modified. The method *deallocate* frees a C array previously allocated.

```
Algorithm ComputeT(t)
// Initial invocation is ComputeT(root).
// The T array and level are initialized to 0 prior to initial invocation.
// Return cost of best 2-VST for subtree rooted at node t and height
// of this subtree.
{
    if (t! = null) {
        level++;
        // compute C values and heights for left and right subtrees of t
        (leftC, leftHeight) = ComputeT(t.leftChild);
        (rightC, rightHeight) = ComputeT(t.rightChild);
        level--;
        // compute C values and height for t as well as
        // bestT = Opt(t, 2) and t.stride = stride of node t
        // in this best 2-VST rooted at t.
        height = max{leftHeight, rightHeight} + 1;
        bestT = leftC[level] = 2^(height+1);
        t.stride = height + 1;
        for (int i = 1; i <= height; i++) {
            leftC[level + i] += rightC[level + i];
            if (2^i + leftC[level + i] < t.bestT) {
                bestT = 2^i + leftC[level + i];
                t.stride = i;
            }}
        T[level]+ = bestT;
        deallocate(rightC);
        return (leftC, height);
        }
    else {//t is null
        allocate(C);
        return (C, -1);
        }
}
```

Figure 2–10: Algorithm to compute T using Equation 2.22.

The complexity of Algorithm $ComputeT$ is readily seen to be $O(pW)$. Once the T values have been computed using Algorithm $ComputeT$, we may determine $Opt(root, 3)$ and the stride of the root of the optimal 3-VST in an additional $O(W)$ time. The strides of the nodes at the remaining expansion levels of the optimal 3-VST may be determined from the $t.stride$ and subtree height values computed by Algorithm $ComputeT$ in $O(p)$ time. So the total time needed to determine the best 3-VST is $O(pW)$.

When the difference between the heights of the left and right subtrees of nodes in the 1-bit trie is bounded by some constant d, the complexity of Algorithm $ComputeT$ is $O(p)$. We use an amortization scheme to prove this. First, note that, exclusive of the recursive calls, the work done by Algorithm $ComputeT$ for each invocation is $O(height(t))$. For simplicity, assume that this work is exactly $height(t) + 1$ (the 1 is for the work done outside the **for** loop of $ComputeT$). Each active C array will maintain a credit that is at least equal to the height of the subtree it is associated with. When a C array is allocated, it has no credit associated with it. Each node in the 1-bit trie begins with a credit of 2. When $t = N$, 1 unit of the credits on N is used to pay for the work done outside of the **for** loop. The remaining unit is given to the C array $leftC$. The cost of the **for** loop is paid for by the credits associated with $rightC$. These credits may fall short by at most $d+1$, because the height of the left subtree of N may be up to d more than the height of N's right subtree. Adding together the initial credits on the nodes and the maximum total shortfall, we see that $p(2 + d + 1)$ credits are enough to pay for all of the work. So, the complexity of $ComputeT$ is $O(pd) = O(p)$ (because d is assumed to be a constant). In practice, we expect that the 1-bit tries for router prefixes will not be too skewed and that the difference between the heights of the left and right subtrees will, in fact, be quite small. Therefore, in practice, we expect $ComputeT$ to run in $O(p)$ time.

Table 2–3: Memory required (in Kbytes) by best $k-$level FST

Levels(k)	2	3	4	5	6	7
Paix	49,192	3,030	1,340	1,093	960	922
Pb	47,925	2,328	896	699	563	527
MaeWest	44,338	2,168	819	636	499	468
Aads	42,204	2,070	782	594	467	436
MaeEast	38,890	1,991	741	549	433	398

2.4 Experimental Results

We programmed our dynamic programming algorithms in C and compared their performance against that of the C codes for the algorithms of Srinivasan and Varghese [82]. All codes that were run on a SUN workstation were compiled using the `gcc` compiler and optimization level -O2; codes run on a PC were compiled using Microsoft Visual C++ 6.0 and optimization level -O2. The codes were run on a SUN Ultra Enterprise 4000/5000 computer as well as on a 2.26 GHz Pentium 4 PC. For test data, we used the five IPv4 prefix databases of Table 2–1.

2.4.1 Performance of Fixed-Stride Algorithm

Table 2–3 and Figure 2–11 shows the memory required by the best k-level FST for each of the five databases of Table 2–1. Note that the y-axis of Figure 2–11 uses a semilog scale. The k values used by us range from a low of 2 to a high of 7 (corresponding to a lookup performance of at most 2 memory accesses per lookup to at most 7 memory accesses per lookup). As was the case with the data sets used in [82], using a larger number of levels does not increase the required memory. We note that for $k = 11$ and 12, [82] reports no decrease in memory required for three of their data sets. We did not try such large k values for our data sets.

Table 2–4 and Figure 2–12 show the time taken by both our algorithm and that of [82] (we are grateful to Dr. Srinivasan for making his fixed- and variable-stride codes available to us) to determine the optimal strides of the best FST that has at most k levels. These times are for the Pentium 4 PC. Times in Table 2–5 and Figure 2–13 are for the SUN workstation. As expected, the run time of the algorithm of [82] is

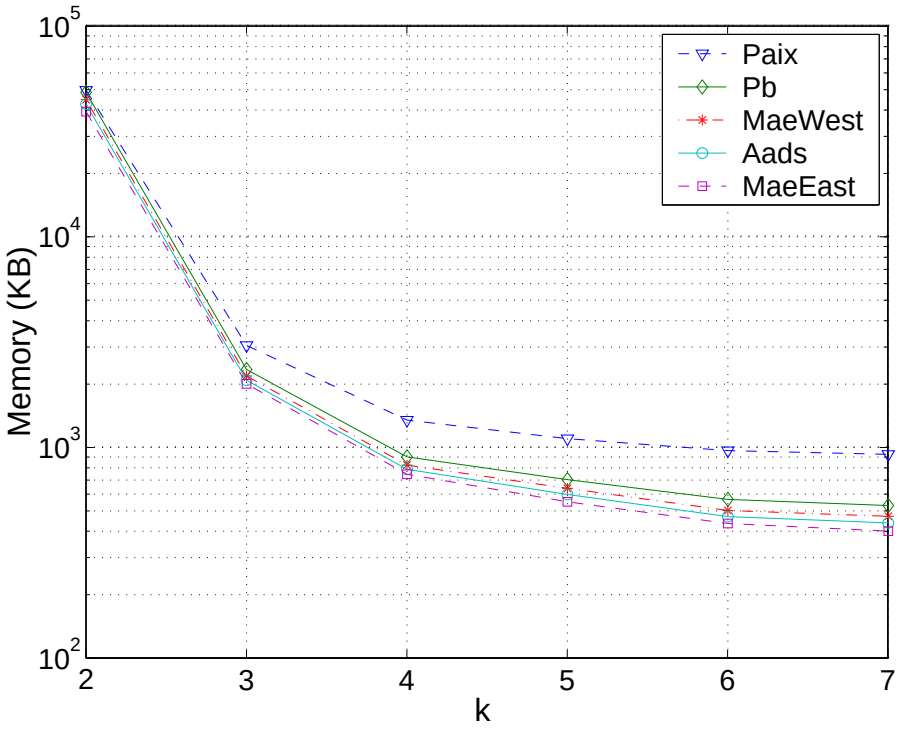


Figure 2–11: Memory required (in KBytes) by best k-level FST

quite insensitive to the number of prefixes in the database. Although the run time of our algorithm is independent of the number of prefixes, the run time does depend on the values of $nodes(*)$ as these values determine $M(*,*)$ and hence determine $minJ$ in Figure 2–3. As indicated by the graph of Figure 2–12, the run time for our algorithm varies only slightly with the database. As can be seen, our algorithm provides a speedup of between $\approx$1.5 and $\approx$3 compared to that of [82]. When the codes were run on our SUN workstation, the speedup was between $\approx$2 and $\approx$4.

2.4.2 Performance of Variable-Stride Algorithm

Table 2–6 shows the memory required by the best k-level VST for each of the five databases of Table 2–1. The columns labeled "Yes" give the memory required when the VST is permitted to have Butler nodes [44]. This capability refers to the replacing of subtries with three or fewer prefixes by a single node that contains these prefixes [44]. The columns labeled "No" refer to the case when Butler nodes are not permitted (i.e., the case discussed in this chapter). The data of Table 2–6 as well

Table 2–4: Execution time (in μsec) for FST algorithms, Pentium 4 PC

	Paix		Pb		MaeWest		Aads		MaeEast	
k	[82]	Our	[82]	Our	[82]	Our	[82]	Our	[82]	Our
2	5.23	3.20	5.19	3.15	5.13	3.15	5.15	3.17	5.17	3.09
3	9.99	4.87	9.73	4.73	9.98	4.81	9.96	4.90	10.00	4.73
4	14.68	6.23	14.53	6.15	14.62	6.29	14.59	6.31	14.64	6.10
5	19.54	7.36	19.42	7.31	19.42	7.40	19.15	7.42	19.45	7.28
6	24.32	9.39	24.08	8.37	24.07	8.47	24.03	8.46	24.23	8.29
7	28.99	9.48	28.72	9.42	28.68	9.45	28.68	9.38	28.76	9.34

Table 2–5: Execution time (in μsec) for FST algorithms, SUN Ultra Enterprise 4000/5000

	Paix		Pb		MaeWest		Aads		MaeEast	
k	[82]	Our	[82]	Our	[82]	Our	[82]	Our	[82]	Our
2	39	21	41	21	39	21	37	20	37	21
3	85	30	81	30	84	31	74	31	96	31
4	123	39	124	40	128	38	122	40	130	40
5	174	46	174	48	147	46	161	45	164	46
6	194	53	201	54	190	55	194	54	190	53
7	246	62	241	63	221	63	264	62	220	62

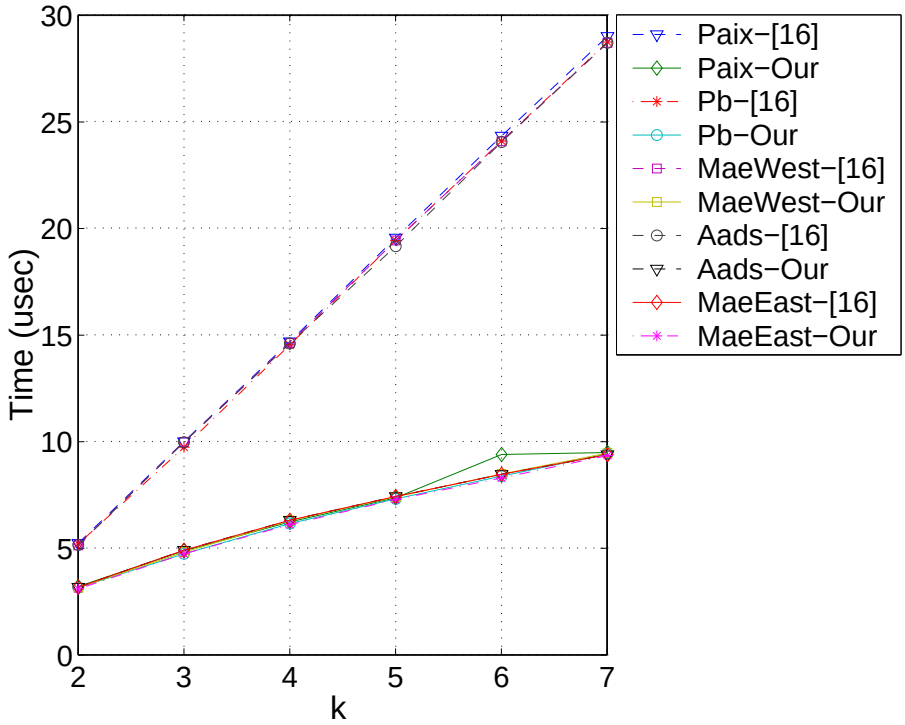


Figure 2–12: Execution time (in μsec) for FST algorithms, Pentium 4 PC

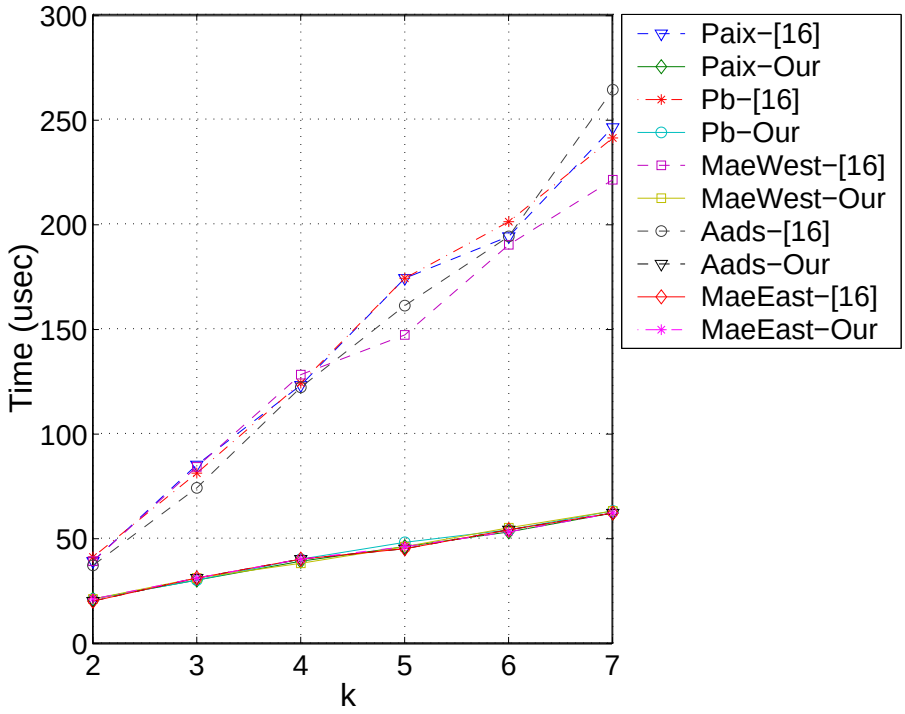


Figure 2–13: Execution time (in μsec) for FST algorithms, SUN Ultra Enterprise 4000/5000

as the memory requirements of the best FST are plotted in Figure 2–14. As can be seen, the Butler node provision has far more impact when k is small than when k is large. In fact, when $k = 2$ the Butler node provision reduces the memory required by the best VST by almost 50%. However, when $k = 7$, the reduction in memory resulting from the use of Butler nodes versus not using them results in less than a 20% reduction in memory requirement.

Table 2–6: Memory required (in Kbytes) by best $k-$VST

	Paix		Pb		MaeWest		Aads		MaeEast	
k	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes
2	2,528	1,722	1,806	1,041	1,754	949	1,631	891	1,621	837
3	1,080	907	677	496	619	443	582	405	537	367
4	845	749	489	397	441	351	410	320	371	286
5	780	706	440	370	393	327	363	297	326	264
6	763	695	426	361	379	319	350	290	313	257
7	759	692	422	358	376	316	346	287	310	254

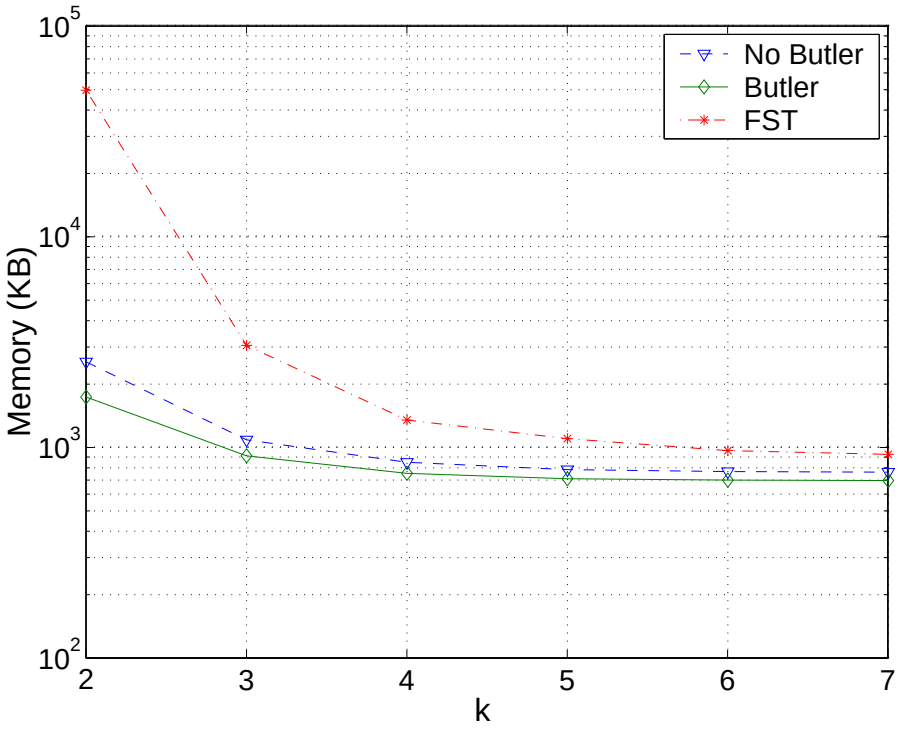


Figure 2–14: Memory required (in KBytes) for Paix by best k-VST and best FST

For the run time comparison of the VST algorithms, we implemented three versions of our VST algorithm of Section 2.3.1. None of these versions permitted the use of Butler nodes. The first version, called the $O(pk+W^2k)$ Static Memory Implementation, is the $O(pk+W^2k)$ memory implementation described in Section 2.3.1. The $O(W^2k)$ memory required by this implementation for the cost arrays is allocated at compile time. During execution, memory segments from this preallocated $O(W^2k)$ memory are allocated to nodes, as needed, for their cost arrays. The second version, called the $O(pWk)$ Dynamic Memory Implementation, dynamically allocates a cost array to each node of the 1-bit trie nodes using C's `malloc` method. Neither the first nor second implementations employ the fast algorithms of Sections 2.3.3 and 2.3.4. Tables 2–7 and 2–8 give the run time for these two implementations.

The third implementation of our VST algorithm uses the faster $k = 2$ and $k = 3$ algorithms of Section 2.3.3 and 2.3.4 and also uses $O(pWk)$ memory. The $O(pWk)$ memory is allocated in one large block making a single call to `malloc`. Following

Table 2–7: Execution times (in msec) for first two implementations of our VST algorithm, Pentium 4 PC

	Paix		Pb		MaeWest		Aads		MaeEast	
k	S	D	S	D	S	D	S	D	S	D
2	34.3	107.5	17.6	56.2	31.0	50.1	15.9	46.9	12.2	40.2
3	39.1	115.4	22.4	65.2	15.2	58.2	19.0	53.1	15.1	46.5
4	47.0	131.4	28.2	74.8	20.0	66.2	23.3	57.6	16.6	54.2
5	51.5	140.7	29.6	78.0	20.3	66.2	23.2	62.0	19.9	56.0
6	59.0	146.7	32.9	82.7	27.9	69.4	26.3	71.6	21.4	62.5
7	63.7	159.3	31.0	88.6	32.8	79.0	32.7	73.3	29.4	67.1

S = $O(pk + W^2k)$ Static Memory Implementation
D = $O(pWk)$ Dynamic Memory Implementation

Table 2–8: Execution times (in msec) for first two implementations of our VST algorithm, SUN Ultra Enterprise 4000/5000

	Paix		Pb		MaeWest		Aads		MaeEast	
k	S	D	S	D	S	D	S	D	S	D
2	290	500	150	280	150	260	120	200	120	230
3	360	790	190	460	180	430	150	340	150	340
4	430	900	210	520	220	430	180	430	160	390
5	490	1140	260	610	240	570	200	520	190	470
6	530	1170	290	670	270	570	270	550	220	510
7	590	1390	330	780	300	690	300	630	260	560

S = $O(pk + W^2k)$ Static Memory Implementation
D = $O(pWk)$ Dynamic Memory Implementation

Table 2–9: Execution times (in msec) for third implementation of our VST algorithm, Pentium 4 PC

k	Paix	Pb	MaeWest	Aads	MaeEast
2	21.0	10.6	9.0	8.2	7.3
3	27.8	15.0	13.2	12.1	10.7
4	48.5	27.6	24.6	22.9	20.6
5	56.2	32.3	28.7	26.7	24.0
6	62.1	36.4	32.5	30.4	27.1
7	69.3	40.3	36.1	33.7	30.3

Table 2–10: Execution times (in msec) for third implementation of our VST algorithm, SUN Ultra Enterprise 4000/5000

k	Paix	Pb	MaeWest	Aads	MaeEast
2	70	30	30	20	20
3	210	100	90	80	70
4	550	290	270	270	240
5	640	350	370	330	260
6	740	430	390	410	350
7	920	530	450	400	350

this, the large allocated block of memory is partitioned into cost arrays for the 1-bit trie nodes by our program. The run time for the third implementation is given in Tables 2–9 and 2–10. The run times for all three of our implementations is plotted in Figures 2–15 and 2–16. Notice that this third implementation is significantly faster than our other $O(pWk)$ memory implementation. Note also that this third implementation is also faster than the $O(pk + W^2k)$ memory implementation for the cases $k = 2$ and $k = 3$ (this is because, in our third implementation, these cases use the faster algorithms of Sections 2.3.3 and 2.3.4).

To compare the run time performance of our algorithm with that of [82], we use the times for implementation 3 when $k = 2$ or 3 and the times for the faster of implementations 1 and 3 when $k > 3$. That is, we compare our best times with the times for the algorithm of [82]. The times for the algorithm of [82] were obtained using their code and running it with the Butler node option off. Since the code of [82] does no dynamic memory allocation, our use of the times for the static memory allocation

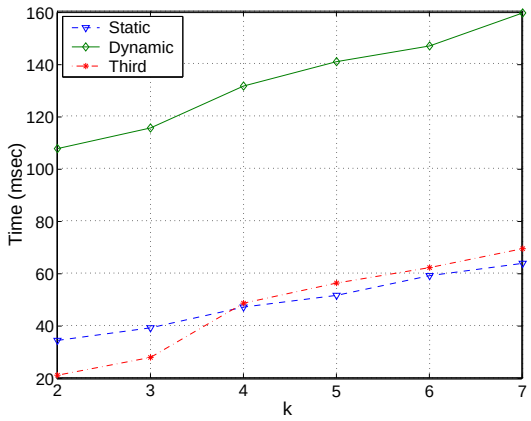


Figure 2–15: Execution times (in msec) for Paix for our three VST implementations, Pentium 4 PC

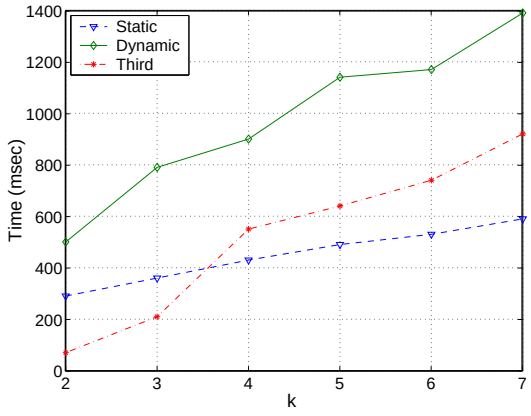


Figure 2–16: Execution times (in msec) for Paix for our three VST implementations, SUN Ultra Enterprise 4000/5000

Table 2–11: Execution times (in msec) for our best VST implementation and the VST algorithm of Srinivasan and Varghese, Pentium 4 PC

	Paix		Pb		MaeWest		Aads		MaeEast	
k	[82]	Our	[82]	Our	[82]	Our	[82]	Our	[82]	Our
2	64.6	21.0	37.4	10.6	31.1	9.0	27.9	8.2	26.6	7.3
3	665.6	27.8	339.2	15.0	297.0	13.2	269.8	12.1	244.1	10.7
4	1262.7	47.0	629.8	27.6	559.4	20.0	503.2	22.9	448.5	16.6
5	1858.0	51.5	928.4	29.6	817.1	20.3	737.2	23.2	659.8	19.9
6	2441.0	59.0	1215.8	32.9	1073.2	27.9	971.4	26.3	868.9	21.4
7	3034.7	63.7	1512.7	31.0	1328.0	32.8	1209.3	32.7	1072.0	29.4

Table 2–12: Execution times (in msec) for our best VST implementation and the VST algorithm of Srinivasan and Varghese, SUN Ultra Enterprise 4000/5000

	Paix		Pb		MaeWest		Aads		MaeEast	
k	[82]	Our	[82]	Our	[82]	Our	[82]	Our	[82]	Our
2	190	70	130	30	50	30	40	20	40	20
3	1960	210	1230	100	360	90	320	80	280	70
4	3630	430	2330	210	700	220	590	180	530	160
5	5340	490	3440	260	1030	240	860	200	780	190
6	7510	530	4550	290	1340	270	1150	270	1020	220
7	9280	590	5650	330	1650	300	1420	300	1270	260

does not, in any way, disadvantage the algorithm of [82]. The run times, on our 2.26 GHz PC, are shown in Table 2–11 and these times are plotted in Figure 2–17. For our largest database, Paix, our new algorithm takes less than one-third the time taken by the algorithm of [82] when $k = 2$ and about 1/47 the time when $k = 7$. On our SUN workstation, as shown in Table 2–12 and Figure 2–18, the observed speedups for Paix ranges from a low of 2.7 to a high of 15.7. The observed speedups aren't as high as predicted by our crude analysis because actual speedup is governed by both the operation cost and the cache-miss cost; further, our crude analysis doesn't account for all operations. The higher speedup observed on a PC suggests a higher relative cache-miss cost on the PC (relative to the cost of an operation) versus on a SUN workstation.

The times reported in Tables 2–7–2–12 are only the times needed to determine the optimal strides for a given 1-bit trie. Once these strides have been determined,

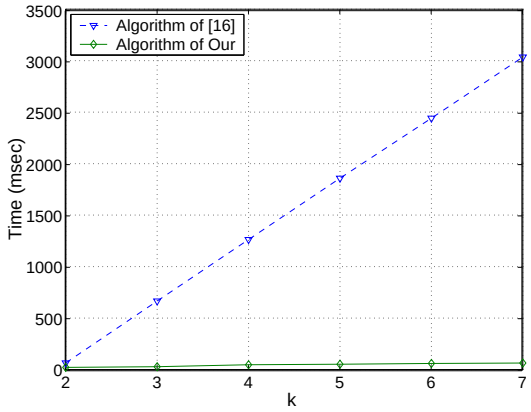


Figure 2–17: Execution times (in msec) for Paix for our best VST implementation and the VST algorithm of Srinivasan and Varghese, Pentium 4 PC

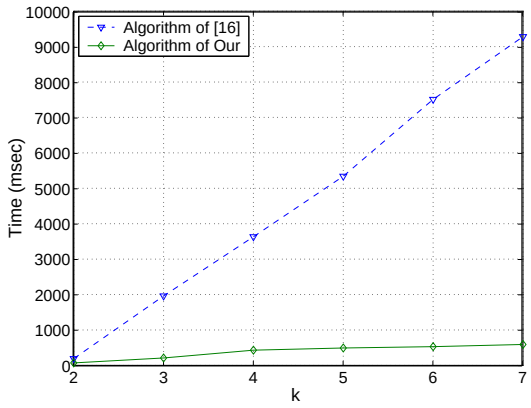


Figure 2–18: Execution times (in msec) for Paix for our best VST implementation and the VST algorithm of Srinivasan and Varghese, SUN Ultra Enterprise 4000/5000

Table 2–13: Time (in msec) to construct optimal VST from optimal stride data, Pentium 4 PC

k	Paix	Pb	MaeWest	Aads	MaeEast
2	117.1	78.0	68.5	67.4	64.3
3	107.8	62.6	55.7	47.0	47.0
4	115.5	66.2	61.3	50.9	47.0
5	126.6	78.0	63.6	62.1	56.5
6	131.4	82.6	64.5	68.9	59.5
7	139.3	78.0	75.6	71.6	62.0

Table 2–14: Search time (in μsec) in optimal VST, Pentium 4 PC

k	Paix	Pb	MaeWest	Aads	MaeEast
2	0.55	0.46	0.44	0.43	0.42
3	0.71	0.64	0.62	0.61	0.59
4	0.79	0.74	0.73	0.72	0.72
5	0.92	0.89	0.89	0.88	0.90
6	1.01	1.00	0.99	0.99	0.98
7	1.10	1.10	1.08	1.09	1.10

it is necessary to actually construct the optimal VST. Table 2–13 shows the time required to construct the optimal VST once the optimal strides are known. For our databases, the VST construction time is more than the time required to compute the optimal strides using our best optimal stride computation implementation.

The primary operation performed on an optimal VST is a lookup or search in which we begin with a destination address and find the longest prefix that matches this destination address. To determine the average lookup/search time, we searched for as many addresses as there are prefixes in a database. The search addresses were obtained by using the 32-bit expansion available in the database for all prefixes in the database. Table 2–14 and Figure 2–19 show the average time to perform a lookup/search. As expected, the average search time increases monotonically with k. For our databases, the search time for a 2-VST is less than or equal to half that for a 7-VST.

Inserts and deletes are performed less frequently than searches in a VST. We experimented with three strategies for these two operations:

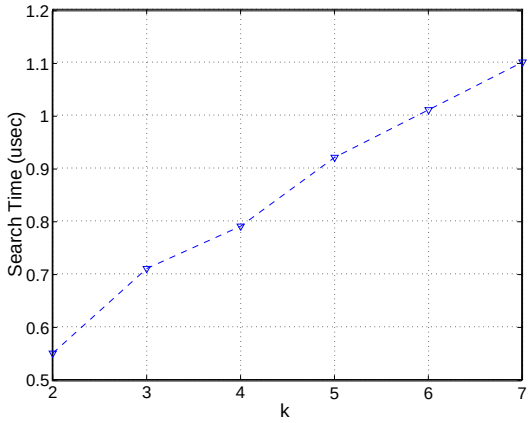


Figure 2–19: Search time (in nsec) in optimal VST for Paix, Pentium 4 PC

- **OptVST:** In this strategy, the VST was always the best possible k-VST for the current set of prefixes. To insert a new prefix, we first insert the prefix into the 1-bit trie of all prefixes. Then, the cost arrays on the insert path are recomputed. This is done efficiently using implementation 2 (i.e., the $O(pWk)$ dynamic memory implementation) of our VST stride computation algorithm. Following this, the optimal strides for vertices on the insert path are computed. Since, the optimal VST for the new prefix set differs from the optimal VST for the original prefix set only along the insert path, we modify the original optimal VST only along this insert path using the newly computed strides for the vertices on this path. Deletion works in a similar fashion.
- **Batch1:** In this strategy, the optimal VST is computed periodically (say, after a sufficient number of inserts/deletes have taken place) rather than following each insert/delete. Inserts and deletes are done directly into the current VST without regard to maintaining optimality. If the insertion results in the creation of a new node, the stride of this new node is such that the sum of the strides of the nodes on the path from the root to this new node equals the length of the newly inserted prefix. The deletion of a prefix may require us to search a node

for a replacement prefix of the next (lower) length that matches the deleted prefix.

- **Batch2:** This differs from strategy Batch1 in that inserts and deletes are done in both the current VST and in the 1-bit trie. This increases the time for an insert as well as for a delete. In the case of deletion, by first deleting from the 1-bit trie, we determine the next (lower) length matching prefix from the delete path taken in the 1-bit trie. This eliminates the need to search a node for this next (lower) length matching prefix when deleting from the VST. The result is a net reduction in time for the delete operation.

The batch modes described above may also be useful when the insert/delete rate is sufficiently small that following each insert or delete done as above, the optimal VST is computed in the background using another processor. While this computation is being done, routes are made using the suboptimal VST resulting from the insert or delete that was done as described for the batch modes. When the new optimal VST has been computed, the new optimal VST is swapped with the suboptimal one.

Tables 2–15-2–20 give the measured run times for the insert and delete operations using each of the three strategies described above. Figures 5–8 and 5–9 plot these times for the Paix database. For the insert time experiments, we started with an optimal VST for 75% of the prefixes in the given database and then measured the time to insert the remaining 25%. The reported times are the average time for one insert. For Paix and $k = 2$, it takes $21 + 78 = 99$ milli seconds to construct the optimal VST (time to compute optimal strides plus time to construct the VST for these strides). However, the cost of an incremental insert that maintains the optimality of the VST is only 50.75 micro seconds; the cost of an incremental delete is 51.85 micro seconds; a speedup of about 2000 over the from scratch optimal VST construction!

Table 2–15: Insertion time (in μsec) for OptVST, Pentium 4 PC

k	Paix	Pb	MaeWest	Aads	MaeEast
2	50.75	49.72	49.53	49.17	208.44
3	325.95	71.25	67.28	66.23	68.19
4	146.74	165.60	126.74	122.92	99.87
5	186.22	191.17	187.68	169.06	186.36
6	2247.96	333.87	252.99	746.92	192.36
7	912.03	446.03	2453.73	445.81	375.32

Table 2–16: Deletion time (in μsec) for OptVST, Pentium 4 PC

k	Paix	Pb	MaeWest	Aads	MaeEast
2	51.85	51.39	51.79	50.95	52.15
3	61.29	60.77	60.94	59.80	61.71
4	74.86	72.90	73.72	71.99	74.58
5	87.47	85.71	86.31	84.97	87.80
6	99.74	97.70	98.50	96.97	99.90
7	111.92	109.89	110.49	108.82	113.15

Although batch insertion is considerably faster than insertion using strategy OptVST, batch insertion increases the number of levels in the VST, and so results in slower searches. For example, in the experiments with Paix, the batch inserts increased the number of levels in the initial k-VST from k to 5 for $k = 2$, to 6 for $k = 3$ and 4, and to 8 for $k = 5$, 6, and 7. The delete times were measured by starting with an optimal VST for 100% of the prefixes in the given database and then measuring the time to delete 25% of these prefixes. Once again, the average time for a single delete is reported.

Table 2–17: Insertion time (in μsec) for Batch1, Pentium 4 PC

k	Paix	Pb	MaeWest	Aads	MaeEast
2	1.51	1.73	1.69	1.81	1.89
3	2.37	2.97	3.40	2.58	2.79
4	2.86	3.50	4.04	3.31	3.09
5	3.63	4.33	4.52	3.54	3.93
6	4.18	5.05	6.53	4.35	5.19
7	5.00	4.98	9.03	4.58	5.19

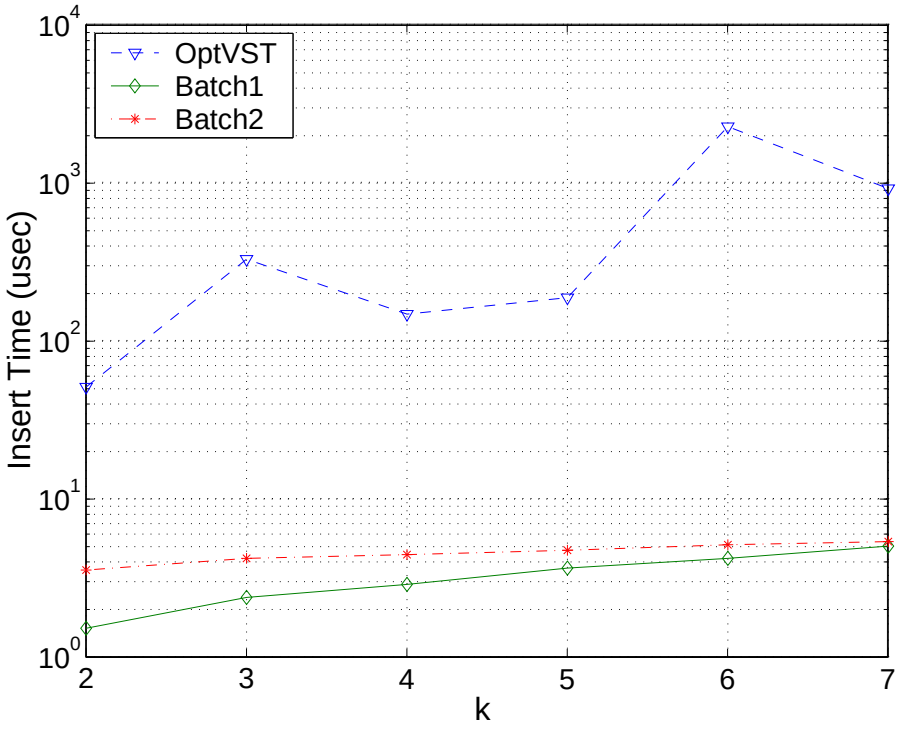


Figure 2–20: Insertion time (in μsec) for Paix, Pentium 4 PC

Table 2–18: Deletion time (in μsec) for Batch1, Pentium 4 PC

k	Paix	Pb	MaeWest	Aads	MaeEast
2	4.72	6.13	6.53	6.24	6.44
3	2.54	2.14	2.18	2.34	2.48
4	2.22	2.69	2.59	2.39	2.65
5	2.58	3.25	2.79	2.92	2.94
6	2.73	3.05	3.15	3.86	3.06
7	2.80	3.43	3.10	3.51	3.62

Table 2–19: Insertion time (in μsec) for Batch2, Pentium 4 PC

k	Paix	Pb	MaeWest	Aads	MaeEast
2	3.53	3.68	4.01	4.07	4.00
3	4.18	4.62	4.97	4.72	4.79
4	4.42	4.96	4.93	4.94	5.03
5	4.70	5.32	5.48	4.94	5.51
6	5.10	5.96	6.31	5.72	6.15
7	5.34	6.13	7.26	5.76	5.47

Table 2–20: Deletion time (in μsec) for Batch2, Pentium 4 PC

k	Paix	Pb	MaeWest	Aads	MaeEast
2	5.70	7.34	7.37	7.16	7.45
3	3.59	3.59	3.67	3.63	3.49
4	3.48	3.78	3.93	3.89	3.98
5	3.67	3.77	4.05	3.79	4.05
6	3.81	4.07	4.14	3.98	4.15
7	3.87	4.19	4.23	3.99	3.90

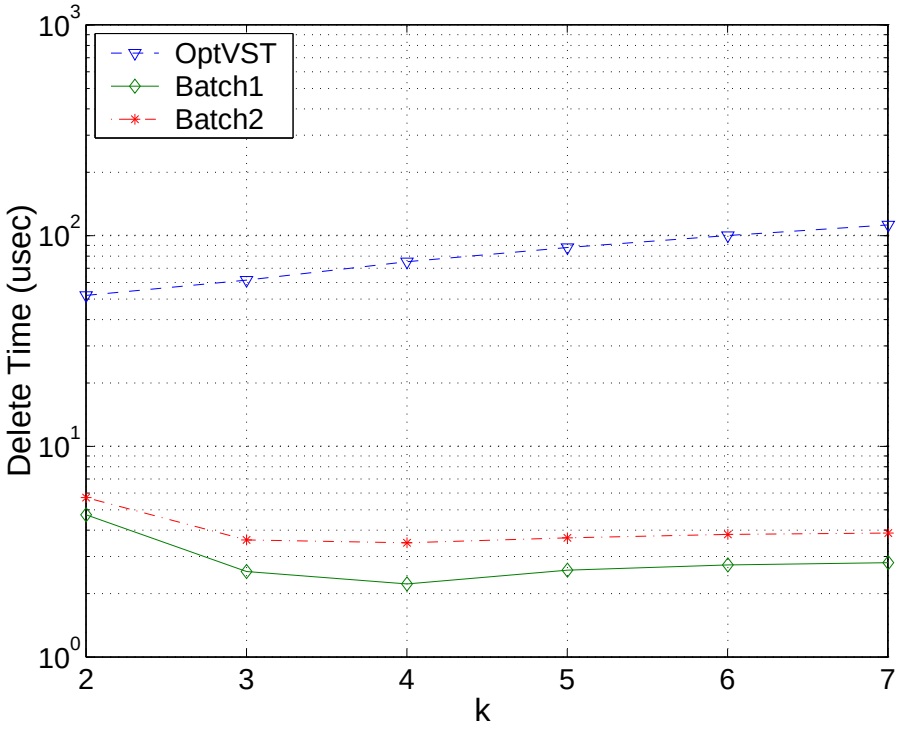


Figure 2–21: Deletion time (in μsec) for Paix, Pentium 4 PC

2.5 Summary

We have developed faster algorithms to compute the optimal strides for fixed- and variable-stride tries than those proposed in [82]. On IPv4 prefix databases and a 2.26 GHz Pentium 4 PC, our algorithm for fixed-stride tries is faster than the corresponding algorithm of [82] by a factor of between 1.5 and 3; on a SUN Ultra Enterprise 4000/5000, the speedup is between 2 and 4. This speedup results from narrowing the search range in the dynamic-programming formulation. Since the search range is at most 32 for IPv4 databases and at most 128 for IPv6 databases, the potential to narrow the range (and hence speed up the computation) is greater for IPv6 data. Hence, we expect that our narrowed-range FST algorithm will exhibit greater speedup on IPv6 databases. We are unable to verify this expectation, because of the non-availability of IPv6 prefix databases.

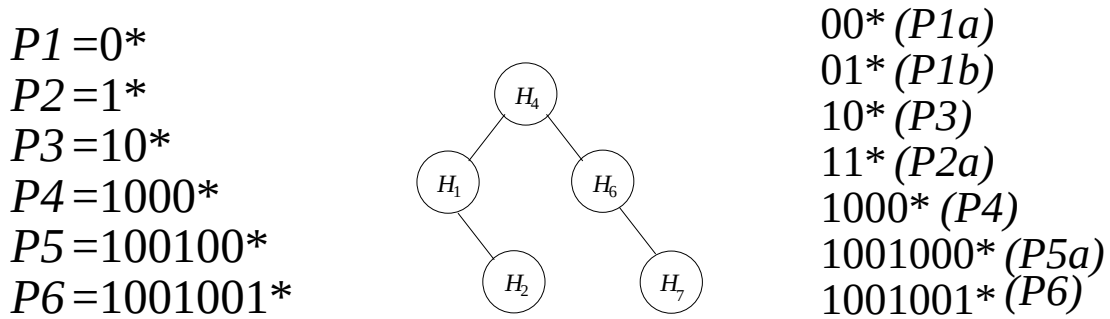
On our PC, our algorithm to compute the strides for an optimal variable-stride trie is faster than the corresponding algorithm of [82] by a factor of between 3 and 47; on our SUN workstation, the speedup is between 2 and 17. Our VST stride computation method permits the insertion and removal of prefixes without having to recompute the optimal strides from scratch. The incremental insert and delete algorithms are about 3 orders of magnitude faster than the "from scratch" algorithm. We also have proposed two batch strategies for the insertion and removal of prefixes. Although, these strategies permit faster insertion and deletion, they increase the height of the VST, which results in slowing down the search operation. These batch strategies are, nonetheless, useful in applications where it is practical to rebuild the optimal VST whenever the search performance has become unacceptible.

CHAPTER 3
BINARY SEARCH ON PREFIX LENGTH

In this chapter, we focus on the collection of hash tables (CHT) scheme of Waldvogel et al. [87]. Let P be the set of prefixes in a router table, and let P_i be the subset of P comprised of prefixes whose length is i. In the scheme of Waldvogel et al. [87], we maintain a hash table H_i for every P_i that is not empty. H_i includes the prefixes of P_i as well as markers for prefixes in $\cup_{i<j\leq W} P_j$. Each marker m in H_i is i-bits long and $m.lmp$ is the longest matching-prefix for m. Consider the prefix set $P = \{P1, ..., P6\}$ of Figure 3–1(a). The prefixes of P have 5 distinct lengths 1, 2, 4, 6, and 7. So, the CHT of [87] will comprise H_1, H_2, H_4, H_6, and H_7. Given a destination address d, the longest matching-prefix, $lmp(d)$ is found by searching the H_is using a binary search. Suppose that the binary search follows a path as determined by the binary tree of Figure 3–1(b). That is, if the first four bits of d correspond to a prefix in H_4, this prefix becomes the longest matching-prefix found so far and the search continues to H_6; if the first four bits of d correspond to a marker m in H_4, then $m.lmp$ becomes the longest-matching prefix found so far and the search continues to H_6; otherwise, the search continues to H_1. The quest for $lmp(d)$ examines at most 3 hash tables in our example. When the number of distinct lengths is l_{dist}, the number of hash tables examined is $O(\log l_{dist})$.

For the described search to work correctly, H_4 must have markers for $P5$ and $P6$; H_1 for $P3$; and H_6 for $P6$. H_1, for example, will include $P1$ and $P2$ plus the marker $1*$ for $P3$ (actually, since $P2 = 1*$, the marker isn't needed); while H_4 will include $P4$ plus the marker 1001* for $P5$ and $P6$. The lmp value for the marker 1001* is $P3$.

Srinivasan and Varghese [82] have proposed the use of controlled prefix-expansion to reduce the number of distinct lengths and hence the number of hash tables in the



(a) Prefixes

(b) Tree for binary search

(c) Expanded prefixes

Figure 3–1: Controlled prefix expansion

CHT. By reducing the number of hash tables in the CHT, the worst-case number of hash tables searched in the quest for $lmp(d)$ may be reduced. Prefix expansion [82] replaces a prefix of length u with 2^{v-u} prefixes of length v, $v > u$. The new prefixes are obtained by appending all possible bit sequences of length $v - u$ to the prefix being expanded. So, for example, the prefix 1* may be expanded to the length 2 prefixes 10* and 11* or to the length 3 prefixes 100*, 101*, 110*, and 111*. In case an expanded prefix is already present in the prefix set of the router table, it is dominated by the existing prefix (the expanded prefix 10* represents a shorter original prefix 1* that cannot be used to match destination addresses that begin with 10 when longest-prefix matching is used) and so is discarded. So, if we expand $P2 = 1*$ in our collection of Figure 3–1(a) to length 2, the expaned prefix $P2b = 10*$ is dominated by $P3 = 10*$. Figure 3–1(c) shows the prefixes that result when the length 1 prefixes of Figure 3–1(a) are expanded to length 2 and the length 6 prefix is expanded to length 7. You may verify that $lmp(d)$ is the same for all d regardless of whether we use the prefix set of Figure 3–1(a) or (c) (when the latter set is used, we need to map back to the original prefix from which an expanded prefix came). Since, the prefixes of Figure 3–1(c) have only 3 distinct lengths, the corresponding CHT has only 3 hash tables and may be searched for $lmp(d)$ with at most 2 hash-table searches. Hence,

the CHT scheme results in faster lookup when the prefixes of Figure 3–1(c) are used than when those of Figure 3–1(a) are used. [71, 72, 73, 82] use prefix expansion to improve the lookup performance of trie-representations of router tables.

When reducing the number of distinct lengths from u to v, the choice of the target v lengths affects the number of markers and prefixes that have to be stored in the resulting CHT but not the number of hash tables, which is always v. Although the number of target lengths may be determined from the expected number of packets to be processed per second and the performance characteristics of the computer to be used for this purpose, the target lengths are determined so as to minimize the storage requirements of the CHT. Consequently, Srinivasan and Varghese [82] formulated the following optimization problem.

Exact Collection of Hash Tables Optimization Problem (ECHT)
Given a set P of n prefixes and a target number of distinct lengths k, determine target lengths $l_1, ..., l_k$ such that the storage required by the prefixes and markers for the prefix set expansion(P) obtained from P by prefix expansion to the determined target lengths is minimum.

When P and k are not implicit, we use the notation $ECHT(P, k)$. For simplicity, Srinivasan [80] assumes that the strorage required by the prefixes and markers for the prefix set *expansion*(P) equals the number of prefixes and markers. We make the same assumption in this chapter. Srinivasan [80] provides an $O(nW^2)$-time heuristic for ECHT. We first show, in Section 3.1, that the heuristic of Srinivasan [80] may be implemented so that its complexity is $O(nW + kW^2)$ on practical prefix-sets. Then, in Section 3.2, we provide an $O(nW^3 + kW^4)$-time algorithm for ECHT. In Section 3.3, we formulate an alternative version ACHT of the ECHT problem. In this alternative version, we are to find at most k distinct target lengths to minimize storage rather than exactly k target lengths. The ACHT problem also may be solved

in $O(nW^3 + kW^4)$ time. In Section 3.4, we propose a reduction in the search range used by the heuristic of [80]. The proposed range reduction reduces the run time by more than 50% exclusive of the preprocessing time. The reduced-range heuristic generates the same results on our benchmark prefix data-sets as are generated by the full-range heuristic of [80]. A more accurate cost estimator than is used in the heuristic of [80] is proposed in Section 3.5. Experimental results highlighting the relative performance of the various algorithms and heuristics for ECHT and ACHT are presented in Section 3.6.

3.1 Heuristic of Srinivasan

The ECHT heuristic of Srinivasan [80] uses the following definitions:

- $ExpansionCost(i, j)$

  This is the number of distinct prefixes that result when all prefixes in $P_q \in P$, $i \leq q < j$ are expanded to length j. For example, when $P = \{0*, 1*, 01*, 100*\}$, $ExpansionCost(1, 3) = 8$ (note that 0* and 1* contribute 4 prefixes each; 01* contributes none because its expanded prefixes are included in the expanded prefixes of 0*).

- $Entries(j)$

  This is the maximum number of markers in H_j (should j be a target length) plus the number of prefixes in P whose length is j. Srinivasan [80] uses "maximum number of markers" in the definition of $Entries(j)$ rather than the exact number of markers because of the reported difficulty in computing this latter quantity.

- $T(j, r)$

  This is an upper bound on the storage required by the optimal solution to $ECHT(Q, r)$, where $Q \subseteq P$ comprises all prefixes of P whose length is at most j; the optimal solution to $ECHT(Q, r)$ is required to contain markers, as necessary, for prefixes of P whose length exceeds j.

Srinivasan [80] provides the following dynamic programming recurrence for $T(j, r)$.

$$T(j,r) = Entries(j) + \min_{m \in \{r-1 \ldots j-1\}} \{T(m, r-1) + ExpansionCost(m+1, j)\} \quad (3.1)$$

$$T(j,1) = Entries(j) + ExpansionCost(1, j) \quad (3.2)$$

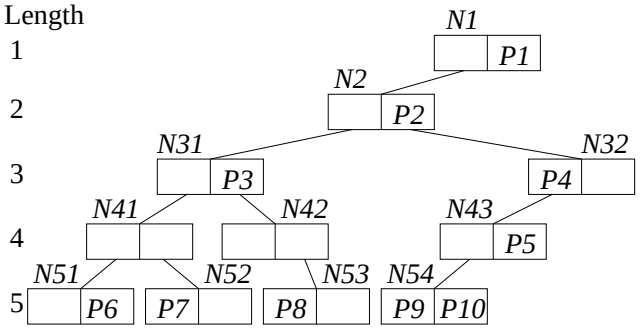
We may verify the correctness of Equations 3.1 and 3.2. When $r = 1$, there is only 1 target length and this length is no more than j. When Q has a prefix whose length is j, then j must be the target length. In this case, the number of expanded prefixes is at most $ExpansionCost(1, j)$ plus the number of prefixes whose length is j. So, the number of prefixes and markers is at most $Entries(j) + ExpansionCost(1, j)$. When Q has no prefix whose length is j, the optimal target length is the largest l, $l < j$ such that Q has a prefix whose length is l. In this case, $Entries(l) + ExpansionCost(1, l) \leq Entries(j) + ExpansionCost(1, j)$ is an upper bound on the number of prefixes and markers.

To compute $ExpansionCost$ and $Entries$, a 1-bit trie [39] is used. Figure 3–2 shows a prefix set and its corresponding 1-bit trie. Notice that nodes at level i (the root is at level 0) of the 1-bit trie store prefixes whose length is $i + 1$. Srinivasan [80] states how $ExpansionCost(i, j), 1 \leq i < j \leq W$ may be computed in $O(nW^2)$ time using a 1-bit trie for P. Sahni and Kim [71] have observed that, for practical prefix sets, the 1-bit trie has $O(n)$ nodes. So, by performing a postorder traversal of the 1-bit trie, $ExpansionCost(i, j), 1 \leq i < j \leq W$ may be computed in $O(nW)$ time (note that $n > W$). Details of this process are provided in Section 3.2.1 where we show how a closely related function may be computed.

For $Entries(j)$, Srinivasan [80] proposes counting the number of prefixes stored in level $j - 1$ of the 1-bit trie and the number of (non-null) pointers (in the 1-bit trie) to nodes at level j (the number of pointers actually equals the number of nodes).

Prefixes
P1 =1*
P2 =01*
P3 =001*
P4 =010*
P5 =0101*
P6 =00001*
P7 =00010*
P8 =00110*
P9 =01000*
P10=01001*

(a) A prefix set



(b) Corresponding 1-bit trie

Figure 3–2: Prefixes and corresponding 1-bit trie

The former gives the number of prefixes whose length is j and the latter gives the maximum number of markers needed for the longer-length prefixes.

Suppose that m and j are target lengths and that no l, $m < l < j$ is a target length. The actual number of prefixes and markers in H_j may be considerably less than $Entries(j) + ExpansionCost(m+1, j)$ for the following reasons.

- An expanded prefix counted in $ExpansionCost(m+1, j)$ may be identical to a prefix in P whose length is j.
- Some of the prefixes in P whose length is more than j may not need to leave a marker in H_j because their length is not on any binary search (sub)path that is preceded by the length j. For example, for the binary search described by Figure 3–1(b), H_1 needs markers only for prefixes in H_2; not for those in H_4, H_6, and H_7. However, $Entries(1)$ accounts for markers needed by prefixes in H_2 as well as those in H_4, H_6, and H_7.
- $Entries(j)$ doesn't account for the fact that a marker may be identical to a prefix in which case the storage count for the marker and the prefix together should be 1 and not 2. For example, in Figure 3–2(b), the marker corresponding to the non-null pointer to node $N42$ is identical to the prefix P3 and that for the non-null pointer to $N43$ is identical to $P4$. So, we can safely reduce the value

of $Entries(3)$ from 5 to 3. Note also that if the target lengths for the example of Figure 3–2(a) are 1, 3, and 5, then the number of prefixes and markers in H_3 is 4. However, $ExpansionCost(2,3) + Entries(3) = 2 + 5 = 7$.

Exclusive of the time needed to compute $ExpansionCost$ and $Entries$, the complexity of computing $T(W,k)$ and the target k lengths using Equations 3.1 and 3.2 is $O(kW^2)$ [80]. So, the overall complexity is $O(nW^2)$ (note that $n \geq k$). As noted above, we may reduce the time required to compute $ExpansionCost$ on practical prefix-sets by performing a postorder traversal of the 1-bit trie. Hence, for practical prefix-sets, the overall run time is $O(nW + kW^2)$.

3.2 Optimal-Storage Algorithm

As noted in Section 3.1, the algorithm of Srinivasan [80] is only a heuristic for ECHT. Since $T(W,k)$ is only an upper bound on the cost of an optimal solution for $ECHT(P,k)$, there is no assurance that the determined target lengths actually result in an optimal or close-to-optimal solution to $ECHT(P,k)$. In this section, we develop an algorithm to determine the storage cost of an optimal solution to $ECHT(P,k)$. The algorithm is easily extended to determine the target lengths that yield this optimal storage cost. Like the heuristic of Srinivasan [80], our algorithm uses dynamic programming. However, we modify the definition of expansion cost and introduce an accurate way to count the number of markers.

Although the heuristic of Srinivasan [80] is insensitive to the shape of the binary tree that describes the binary search, the optimal-storage algorithm cannot be insensitive to this shape. To see this, notice that the binary tree of Figure 3–1(b) corresponds to the traditional way to program a binary search. In this, if low and up define the current search range, then the next comparison is made at $mid = \lfloor (low + up)/2 \rfloor$. If instead, we were to make the next comparison at $mid = \lceil (low + up)/2 \rceil$, the search is described by the binary tree of Figure 3–3. When a binary search is performed according to this tree, only H_4 need have markers. The markers in H_4 are the same regardless

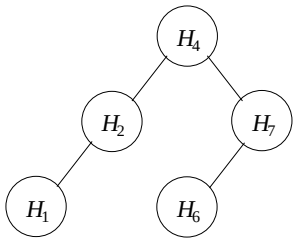


Figure 3–3: Alternative binary tree for binary search

of whether we use $mid = \lfloor (low + up)/2 \rfloor$ or $mid = \lceil (low + up)/2 \rceil$. By using the latter definition of mid, we eliminate markers from all remaining hash tables. In our development of the optimal-storage algorithm, we assume that $mid = \lceil (low + up)/2 \rceil$ is used. Our development is easily altered for the case when $mid = \lfloor (low + up)/2 \rfloor$ is used.

3.2.1 Expansion Cost

Define $EC(i, j)$, $1 \leq i \leq j \leq W$, to be the number of distinct prefixes that result when all prefixes in $P_q \in P$, $i \leq q \leq j$ are expanded to length j. Note that $EC(i, i)$ is the number of prefixes in P whose length is i.

We may compute EC by traversing the 1-bit trie for P in a postorder fashion. Each node x at level $i-1$ of the trie maintains a local set of $EC(i, j)$ values, $LEC(i, j)$, which is the expansion cost measured relative to the prefixes in the subtree of which x is root. Some of the cases for the computation of $x.LEC(i, j)$ are given below.

- $x.LEC(i, i)$ equals the number of prefixes stored in node x. For example, for node $N1$ of Figure 3–2(a), $LEC(1, 1) = 1$ and for node $N54$, $LEC(5, 5) = 2$. For the remaining cases, assume $i < j$.
- If x has a prefix in its left data field (e.g., the prefix in the left data field of node $N32$ is $P4$) and also has one in its right data field, then $x.LEC(i, j) = 2^{j-i+1}$.
- If x has no prefixes (e.g., nodes $N41$ and $N42$) and x has non-null left and right subtrees, then $x.LEC(i, j) = x.leftChild.LEC(i+1, j) + x.rightChild(i+1, j)$.

- If x has a right prefix and a non-null left subtree, then $x.LEC(i,j) = x.leftChild.LEC(i+1,j) + 2^{j-i}$.

The remaining cases are similar to those given above. One may verify that $EC(i,j)$ is just the sum of the $LEC(i,j)$ values taken over all nodes at level $i-1$ of the trie. Figure 3–4 gives the LEC and EC values for the example of Figure 3–2. In this figure, $LEC51$, for example, refers to the LEC values for node $N51$.

LEC51[5,j]	j = 5
	1

LEC52[5,j]	j = 5
	1

LEC53[5,j]	j = 5
	1

LEC54[5,j]	j = 5
	2

LEC41[4,j]	j = 4	5
	0	2

LEC42[4,j]	j = 4	5
	0	1

LEC43[4,j]	j = 4	5
	1	4

LEC31[3,j]	j = 3	4	5
	1	2	6

LEC32[3,j]	j = 3	4	5
	1	2	4

LEC2[2,j]	j = 2	3	4	5
	1	3	6	14

LEC1[1,j]	j = 1	2	3	4	5
	1	3	7	14	30

(a) LEC values

EC[i,j]	j = 1	2	3	4	5
i = 1	1	3	7	14	30
2		1	3	6	14
3			2	4	10
4				1	7
5					5

(b) EC values

Figure 3–4: LEC and EC values for Figure 3–2

Since a 1-bit trie for n prefixes may have $O(nW)$ nodes, we may compute all EC values in $O(nW^2)$ time by computing the LEC values as above and summing up the computed LEC values. A postorder traversal suffices for this. As noted in [71], the 1-bit tries for practical prefix sets have $O(n)$ nodes. Hence, in practice, the EC values take only $O(nW)$ time to compute.

3.2.2 Number of Markers

Define $MC(i,j,m)$, $1 \le i \le j \le m \le W$, to be the number of markers in H_j under the following assumptions.

- The prefix set comprises only those prefixes of P whose length is at most m.
- The target lengths include $i-1$ (for notational convenience, we assume that 0 is a trivial target length for which H_0 is always empty) and j but no length between $i-1$ and j. Hence, prefixes whose length is i, $i+1$, $\cdots$, or j are

expanded to length j. Only prefixes whose length is between $j+1$ and m may leave a marker in H_j.

For $MC(2,4,5)$ (Figure 3–2), $P6$ through $P10$ may leave markers in H_4. The candidate markers are obtained by considering only the first four bits of each of these prefixes. Hence, the candidate markers are 0000*, 0001*, 0011*, and 0100*. However, since the next smaller target length is 1, $P2$, $P3$, and $P4$ will leave a prefix in H_4. The prefixes in H_4 are 0100*, 0101*, 0110*, 0111*, 0010*, and 0011*. So, of the candidate markers, only 0000* and 0001* are different from the prefixes in H_4. Therefore, the marker count $MC(2,4,5)$ is 2.

We may compute all $MC(i,j,m)$ values in $O(nW^3)$ time ($O(nW^2)$ for practical prefix sets) using a local function LMC in each node of the 1-bit trie and a postorder traversal. The method is very similar to that described in Section 3.2.1 for the computation of all EC values. Figure 3–5 shows the LMC and MC values for our example of Figure 3–2.

LMC51[5,j,k]	k = 5
j = 5	0

LMC52[5,j,k]	k = 5
j = 5	0

LMC53[5,j,k]	k = 5
j = 5	0

LMC54[5,j,k]	k = 5
j = 5	0

LMC41[4,j,k]	k = 4	5
j = 4	0	2
5		0

LMC42[4,j,k]	k = 4	5
j = 4	0	1
5		0

LMC43[4,j,k]	k = 4	5
j = 4	0	1
5		0

LMC31[3,j,k]	k = 3	4	5
j = 3	0	0	1
4		0	2
5			0

LMC32[3,j,k]	k = 3	4	5
j = 3	0	0	0
4		0	0
5			0

LMC1[1,j,k]	k = 1	2	3	4	5
j = 1	0	1	1	1	1
2		0	1	1	1
3			0	0	1
4				0	2
5					0

LMC2[2,j,k]	k = 2	3	4	5
j = 2	0	1	1	1
3		0	0	1
4			0	2
5				0

(a) LMC values

MC[1,j,k]	k = 1	2	3	4	5
j = 1	0	1	1	1	1
2	0	1	1	1	1
3		0	0	1	1
4				0	2
5					0

MC[2,j,k]	k = 2	3	4	5
j = 2	0	1	1	1
3		0	0	1
4			0	2
5				0

MC[3,j,k]	k = 3	4	5
j = 3	0	0	1
4		0	2
5			0

MC[4,j,k]	k = 4	5
j = 4	0	4
5		0

MC[5,j,k]	k = 5
j = 5	0

(b) MC values

Figure 3–5: LMC and MC values for Figure 3–2

3.2.3 Algorithm for ECHT

Let $Opt(i, j, r)$ by the storage requirement of the optimal solution to $ECHT(P, r)$ under the following restrictions.

- Only prefixes of P whose length is between i and j are considered.
- Exactly r target lengths are used.
- j is one of the target lengths (even if there is no prefix whose length is j).

Let l_{max}, $l_{max} \leq W$, be the length of the longest prefix in P. We see that $Opt(1, l_{max}, k)$ is the storage requirement of the optimal solution to $ECHT(P, k)$.

When $r = 1$, there is exactly one target length, j. So, all prefixes must be expanded to this length and there are no markers. Therefore,

$$Opt(i, j, 1) = EC(i, j), i \leq j \tag{3.3}$$

When $r = 2$, one of the target lengths is j and the other, say m, lies between i and $j - 1$. Because we assume $mid = \lceil (low + up)/2 \rceil$, the first search is made in H_j and the second in H_m. Consequently, neither H_j nor H_m has any markers. H_j (H_m) includes prefixes resulting from the expansion of prefixes of P whose length is between $m + 1$ and j (i and m). So,

$$Opt(i, j, 2) = \min_{i \leq m < j} \{EC(i, m) + EC(m + 1, j)\}, i < j \tag{3.4}$$

Consider the case $r > 2$. Let the r target lengths be $l_1 < l_2 < \cdots < l_r$. Suppose that the $mid = \lceil (1 + r)/2 \rceil$th target length is v. Let $u - 1$ be the largest target length such that $u - 1 < v$. The first search of the binary search is done in H_v. The number of prefixes and markers in H_v is $EC(u, v) + MC(u, v, j)$. Additionally, the $mid - 1 = \lfloor (r - 1)/2 \rfloor$ target lengths that are less than v define an optimal $(mid - 1)$-target-length solution for prefixes whose length is between i and $u - 1$ subject to

the constraint that $u-1$ is a target length (notice that there are no markers in this solution for prefixes whose length exceeds $u-1$) and the $r-m=\lfloor(r-1)/2\rfloor$ target lengths greater than v define an optimal $(r-m)$-target-length solution for prefixes whose length is between $v+1$ and j subject to the constraint that j is a target length. Hence, we obtain the following recurrence for $Opt(i,j,r)$.

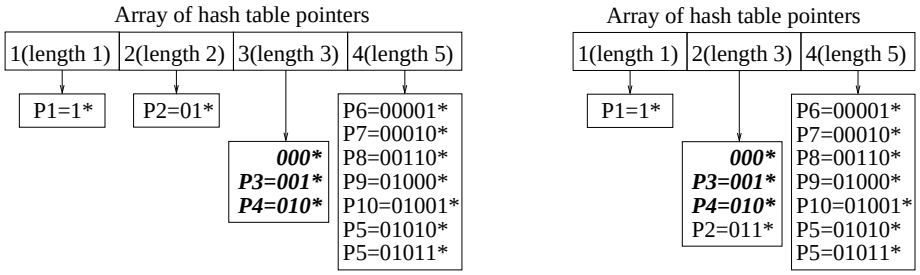
$$\begin{aligned} Opt(i,j,r) &= \min_{i+\lceil(r-1)/2\rceil\le u\le v\le j-\lfloor(r-1)/2\rfloor}\{Opt(i,u-1,\lceil(r-1)/2\rceil) \\ &+ Opt(v+1,j,\lfloor(r-1)/2\rfloor)+EC(u,v) \\ &+ MC(u,v,j)\}, 3\le r\le j-i+1 \end{aligned} \tag{3.5}$$

Using Equations 3.3–3.5 to compute $Opt(1,5,4)$ for the example of Figure 3–2, we get

$$\begin{aligned} Opt(1,5,4) &= \min_{3\le u\le v\le 4}\{Opt(1,u-1,2)+Opt(v+1,5,1)+EC(u,v)+MC(u,v,j)\} \\ &= \min\{Opt(1,2,2)+Opt(4,5,1)+EC(3,3)+MC(3,3,5), \\ &\quad Opt(1,2,2)+Opt(5,5,1)+EC(3,4)+MC(3,4,5), \\ &\quad Opt(1,3,2)+Opt(5,5,1)+EC(4,4)+MC(4,4,5)\} \\ &= \min\{EC(1,1)+EC(2,2)+EC(4,5)+EC(3,3)+MC(3,3,5), \\ &\quad EC(1,1)+EC(2,2)+EC(5,5)+EC(3,4)+MC(3,4,5), \\ &\quad \min\{EC(1,1)+EC(2,3),EC(1,2)+EC(3,3)\} \\ &\quad +EC(5,5)+EC(4,4)+MC(4,4,5)\} \\ &= \min\{1+1+7+2+1,1+1+5+4+2, \\ &\quad \min\{1+3,3+2\}+5+1+4\} \\ &= \min\{12,13,14\}=12. \end{aligned}$$

From the above computations, we see that the optimal expansion lengths are 1, 2, 3, and 5. Figure 3–6(a) shows the CHT structure that results when these four target

lengths are used. The three markers are shown in boldface, two of these markers are also prefixes ($P3$ and $P4$). The storage cost is 12.



(a) 4 target lengths (b) 3 target lengths

Figure 3–6: Optimal-storage CHTs for Figure 3–2

Complexity

To solve Equations 3.3–3.5 for $Opt(1, l_{max}, k)$, we need to compute $O(kW^2)$ $Opt(i, j, r)$ values. Each of these values may be computed in $O(W^2)$ time from earlier computed Opt values. Hence, exclusive of the time needed to compute EC and MC, the time to compute $Opt(1, l_{max}, k)$ is $O(kW^4)$. Adding in the time to compute EC and MC, we get $O(nW^3 + kW^4)$ as the overall time needed to solve the ECHT problem. Of course, on practical data sets, the time is $O(nW^2 + kW^4)$.

3.3 Alternative Formulation

In the $ECHT(P, k)$ problem, we are to find exactly k target lengths for P that minimize the number of (expanded) prefixes and markers (i.e., minimize storage cost). Although k is determined by constraints on required lookup performance, the determined k is only an upper bound on the number of target lengths because using a smaller number of target lengths improves lookup performance. The $ECHT$ problem is formulated with the premise that using a smaller k will lead to increased storage cost, and so in the interest of conserving storage/memory while meeting the lookup performance requirement, we use the maximum permissible number of target lengths. However, this premise is not true. As an example, consider the prefix set

$P = \{P1, P2, P3\} = \{0*, 00*, 010*\}$. The solution for $ECHT(P, 2)$ uses the target lengths 2 and 3; $P1$ expands to 00* and 01* but the 00* expansion is dominated by $P2$ and is discarded; no markers are stored in either H_2 or H_3; and the storage cost is 3. The solution for $ECHT(P, 3)$, on the other hand, uses the target lengths 1, 2, and 3; no prefixes are expanded; H_2 needs a marker 01* for $P3$; and the total storage cost is 4!

With this in mind, we formulate the $ACHT(P, k)$ problem in which we are to find at most k target lengths for P that minimize the storage cost. In case of a tie, the solution with the smaller number of target lengths is preferred, because this solution has a reduced average lookup for the preceding example, the solution to $ECHT(P, 3)$ is $\{1, 2, 3\}$, whereas the solution to $ACHT(P, 3)$ is $\{2, 3\}$. For the example of Figure 3–2, the solution to $ECHT(P, 4)$ is $\{1, 2, 3, 5\}$ resulting in a storage cost of 12; the solution to $ACHT(P, 4)$ is $\{1, 3, 5\}$ resulting in a storage cost that is also 12 (see Figure 3–6(b)).

The $ACHT$ problem may be solved in the same asymptotic time as needed for the $ECHT$ problem by first computing $Opt(i, j, r)$, $1 \leq i < j \leq l_{max}$, $1 \leq r \leq k$ and then finding the r for which $Opt(1, l_{max}, r)$ is minimum, $1 \leq r \leq k$.

3.4 Reduced-Range Heuristic

We first adapt the $ECHT$ heuristic of Srinivasan [80] to the $ACHT$ problem. For this purpose, we define the function C, which is the $ACHT$ analog of T. To get the definition of C, simply replace $ECHT(Q, r)$ by $ACHT(Q, r)$ in the definition of T. Also, we use the same definitions for *ExpansionCost* (now abbreviated to *ECost*) and *Entries* as used in [80] (see Section 3.1). It is easy to see that $C(j, r) \leq C(j, r-1), r > 1$.

A simple dynamic programming recurrence for C is

$$C(j,r) = Entries(j) + \min_{m \in \{0 \ldots j-1\}} \{C(m, r-1) + ECost(m+1, j)\}, j > 0, r > 1 \tag{3.6}$$

$$C(0,r) = 0, C(j,1) = Entries(j) + ECost(1,j), j > 0 \tag{3.7}$$

To see the correctness of Equations 3.6 and 3.7, note that when $j > 0$, there must be at least one target length. If $r = 1$, then there is exactly one target length. This target length is at most j (the target length is j when there is at least one prefix of this length) and so $Entries(j) + ECost(1, j)$ is an upper bound on the storage cost. If $r > 1$, let m and s, $m < s$, be the two largest target lengths in the solution to $ACHT(P, r)$. m could be at any of the lengths 0 through $j - 1$; $m = 0$ would mean that there is only 1 target length. Hence the storage cost is bounded by $Entries(j) + C(m, r-1) + ECost(m+1, j)$. Since we do not know the value of m, we may minimize over all choices for m. $C(0, r) = 0$ is a boundary condition.

We may obtain an alternative recurrence for $C(j, r)$ in which the range of m on the right side is $r - 1 \ldots j - 1$ rather than $0 \ldots j - 1$. First, we obtain the following dynamic programming recurrence for C:

$$C(j,r) = \min\{C(j, r-1), T(j,r)\}, r > 1 \tag{3.8}$$

$$C(j,1) = Entries(j) + ECost(1,j) \tag{3.9}$$

The rationale for Equation 3.8 is that the best CHT that uses at most r target lengths either uses at most $r-1$ target lengths or uses exactly r target lengths. When at most $r - 1$ target lengths are used, the cost is bounded by $C(j, r-1)$, and when

exactly r target lengths are used, the cost is bounded by $T(j,r)$, which is defined by Equation 3.1. Let $U(j,r)$ be as defined in Equation 3.10.

$$U(j,r) = Entries(j) + \min_{m \in \{r-1 \ldots j-1\}} \{C(m, r-1) + ECost(m+1, j)\}, j > 0, r > 1 \tag{3.10}$$

From Equations 3.1 and 3.8 we obtain

$$C(j,r) = \min\{C(j, r-1), U(j,r)\}, r > 1 \tag{3.11}$$

To see the correctness of Equation 3.11, note that for all j and r such that $r \leq j, T(j,r) \geq C(j,r)$. Furthermore,

$$\begin{aligned} Entries(j) \quad + \quad & \min_{m \in \{r-1 \ldots j-1\}} \{T(m, r-1) + ECost(m+1, j)\} \\ \geq \quad & Entries(j) + \min_{m \in \{r-1 \ldots j-1\}} \{C(m, r-1) + ECost(m+1, j)\} \\ = \quad & U(j,r) \end{aligned} \tag{3.12}$$

Therefore, when $C(j, r-1) \leq U(j,r)$, Equations 3.8 and 3.11 compute the same value for $C(j,r)$. When $C(j, r-1) > U(j,r)$, it appears from Equation 3.12 that Equation 3.11 may compute a smaller $C(j,r)$ than is computed by Equation 3.8. However, this is impossible, because

$$\begin{aligned} C(j,r) \quad = \quad & Entries(j) + \min_{m \in \{0 \ldots j-1\}} \{C(m, r-1) + ECost(m+1, j)\} \\ \leq \quad & Entries(j) + \min_{m \in \{r-1 \ldots j-1\}} \{C(m, r-1) + ECost(m+1, j)\} \end{aligned}$$

Therefore, the $C(j,r)$s computed by Equations 3.8 and 3.11 are equal.

In the remainder of this section, we use the reduced ranges $r-1 \ldots j-1$ for C. Heuristically, the range for m (in Equation 3.6) may be restricted to a range that is (often) considerably smaller than $r-1 \ldots j-1$. The narrower range we wish to use is $\max\{M(j-1,r), M(j,r-1), r-1\} \ldots j-1$, where $M(j,r), r>1$, is the smallest m that minimizes

$$C(m, r-1) + ECost(m+1, j)$$

in Equation 3.6. Although the use of this narrower range could result in different results from what we get using the range $r-1 \ldots j-1$, on our benchmark prefix sets, this doesn't happen. In the remainder of this section, we derive a condition Z on the 1-bit trie that, if satisfied, guarantees that the use of the narrower range yields the same results as when the range $r-1 \ldots j-1$ is used.

Let P be the set of prefixes represented by the 1-bit trie. Let $exp(i,j)$, $i \leq j$, be the set of distinct prefixes obtained by expanding the prefixes of P whose length is between i and $j-1$ to length j. Note that $exp(i,i) = \emptyset$ and that $|exp(i,j)| = ECost(i,j)$. We say that $exp(i,j)$ covers a length j prefix p of P iff $p \in exp(i,j)$. Let $n(i,j)$ be the number of length j prefixes in P that are not covered by a prefix of $exp(i,j)$. Note that $n(i,i)$ is the number of length i prefixes in P.

The condition Z that ensures that the use of the narrower range produces the same C values as when the range $r-1 \ldots r-1$ is used is

$$Z = ECost(a,j) - ECost(b,j) \geq 2(n(b,j) - n(a,j))$$

where, $0 < a < b \leq j$.

Lemma 9 *For every 1-bit trie, (a) $ECost(i, j+1) \geq 2ECost(i,j), 0 < i \leq j$, and (b) $ECost(i,j) \geq ECost(i+1,j), 0 < i < j$.*

Proof (a)

$$\begin{aligned} ECost(i, j+1) &= |exp(i, j+1)| \\ &= 2[|exp(i, j)| + n(i, j)] \\ &= 2ECost(i, j) + 2n(i, j) \\ &\geq 2ECost(i, j) \end{aligned}$$

(b) Since, $exp(i+1, j) \subseteq exp(i, j)$, $ECost(i, j) = |exp(i, j)| \geq |exp(i+1, j)| = ECost(i+1, j)$. ■

Lemma 10 $\forall(j > 0, i < j)[Entries(j) + ECost(i, j) \leq Entries(j+1) + ECost(i, j+1)]$.

Proof By definition, $Entries(j)$ = number of prefixes of length j plus the number of nodes at level j of the trie (this latter number equals the number of pointers from level $j-1$ to level j). Since each length j prefix expands to 2 length $j+1$ prefixes, the first term in the sum for $Entries(j)$ is at most $ECost(i, j+1)/2$. Since the subtree rooted at each level j node contains at least one prefix, the second term in the sum for $Entries(j)$ is at most $Entries(j+1)$. So,

$$Entries(j) \leq ECost(i, j+1)/2 + Entries(j+1)$$

From Lemma 9(a), $ECost(i, j) \leq ECost(i, j+1)/2$. So, $Entries(j) + ECost(i, j) \leq ECost(i, j+1) + Entries(j+1)$. ■

Lemma 11 $\forall(j > 0, r > 0)[C[j, r] \leq C[j+1, r]]$.

Proof First, consider the case when $r = 1$. From Equation 3.7, we get $C(j, 1) = Entries(j) + ECost(1, j)$ and $C(j+1, 1) = Entries(j+1) + ECost(1, j+1)$. From Lemma 10, $Entries(j) + ECost(1, j) \leq Entries(j+1) + ECost(1, j+1)$. Hence, $C(j, 1) \leq C(j+1, 1)$.

Next, consider the case $r > 1$. From the definition of $M(j, r)$, it follows that

$$C(j+1, r) = Entries(j+1) + C(b, r-1) + ECost(b+1, j+1),$$

where $0 \leq b = M(j+1, r) \leq j$. When $b < j$, using Equation 3.6 and Lemma 9, we get

$$\begin{aligned} C(j, r) &\leq Entries(j) + C(b, r-1) + ECost(b+1, j) \\ &\leq Entries(j+1) + C(b, r-1) + ECost(b+1, j+1) \\ &= C(j+1, r). \end{aligned}$$

When $b = j$,

$$C(j+1, r) = Entries(j+1) + C(j, r-1) + ECost(j+1, j+1) \geq C(j, r-1).$$

■

The remaining lemmas of this section assume that Z is true.

Lemma 12 $ECost(a, j+1) - ECost(b, j+1) \geq ECost(a, j) - ECost(b, j)]$, $0 < a < b \leq j$.

Proof From the definition of $n(i, j)$, it follows that

$$\begin{aligned} ECost(a, j) - ECost(b, j) &= \sum_{l=a}^{j-1} n(a, l) * 2^{j-l} - \sum_{l=b}^{j-1} n(b, l) * 2^{j-l} \\ &= \sum_{l=a}^{b-1} n(a, l) * 2^{j-l} - \sum_{l=b}^{j-1} [n(b, l) - n(a, l)] * 2^{j-l} \end{aligned}$$

Hence,

$$\begin{aligned}
ECost(a, j+1) &- ECost(b, j+1) \\
&= \sum_{l=a}^{b-1} n(a,l) * 2^{j+1-l} - \sum_{l=b}^{j} [n(b,l) - n(a,l)] * 2^{j+1-l} \\
&= 2[\sum_{l=a}^{b-1} n(a,l) * 2^{j-l} - \sum_{l=b}^{j} [n(b,l) - n(a,l)] * 2^{j-l}] \\
&= 2[\sum_{l=a}^{b-1} n(a,l) * 2^{j-l} - \sum_{l=b}^{j-1} [n(b,l) - n(a,l)] * 2^{j-l}] \\
&\quad -2[n(b,j) - n(a,j)] \\
&= 2[ECost(a,j) - ECost(b,j)] - 2[n(b,j) - n(a,j)]
\end{aligned}$$

Hence, when condition Z is true, $ECost(a, j+1) - ECost(b, j+1) \geq ECost(a, j) - ECost(b, j)$. ■

Lemma 13 $\forall(j > 0, r > 1)[M(j+1, r) \geq M(j, r)]$.

Proof Let $M(j, r) = a$ and $M(j+1, r) = b$. Suppose $b < a$. Then,

$$\begin{aligned}
C(j, r) &= Entries(j) + C(a, r-1) + ECost(a+1, j) \\
&< Entries(j) + C(b, r-1) + ECost(b+1, j)
\end{aligned}$$

since, otherwise, $M(j, r) = b$.

$$\begin{aligned}
C(j+1, r) &= Entries(j+1) + C(b, r-1) + ECost[b+1, j+1] \\
&\leq Entries(j+1) + C(a, r-1) + ECost[a+1, j+1]
\end{aligned}$$

Therefore,

$$ECost(a+1, j) + ECost(b+1, j+1) < ECost(b+1, j) + ECost(a+1, j+1)$$

Hence,

$$ECost(b+1, j+1) - ECost(a+1, j+1) < ECost(b+1, j) - ECost(a+1, j)$$

From Lemma 12, this is not possible when condition Z is true. So, when condition Z is true, $b \geq a$. ■

The next few lemmas use the function Δ, which is defined as $\Delta(j,r) = C(j,r-1) - C(j,r)$. Since, $C(j,r) \leq C(j,r-1), \Delta(j,r) \geq 0$ for all $j > 0$ and all $r > 1$.

Lemma 14 $\forall(j > 0)[\Delta(j,2) \leq \Delta(j+1,2)]$.

Proof If $C(j,2) = C(j,1)$, there is nothing to prove as $\Delta(j+1,2) \geq 0$. The only other possibility is $C(j,2) < C(j,1)$ (i.e., $\Delta(j,2) > 0$). In this case, the solution for $ACHT(j,2)$ uses exactly 2 target lengths. From the recurrence for C (Equations 3.6 and 3.7), it follows that $C(j,1) = Entries(j) + ECost(1,j)$, and

$$\begin{aligned} C(j,2) &= Entries(j) + C(a,1) + ECost(a+1,j) \\ &= Entries(j) + Entries(a) + ECost(1,a) + ECost(a+1,j), \end{aligned}$$

for some a, $0 < a < j$. Therefore,

$$\begin{aligned} \Delta(j,2) &= C(j,1) - C(j,2) \\ &= ECost(1,j) - Entries(a) - ECost(1,a) - ECost(a+1,j). \end{aligned}$$

From Equations 3.6 and 3.7, it follows that

$$\begin{aligned} C(j+1,2) &\leq Entries(j+1) + C(a,1) + ECost(a+1,j+1) \\ &= Entries(j+1) + Entries(a) + ECost(1,a) + ECost(a+1,j+1) \end{aligned}$$

Hence,

$$\Delta(j+1,2) \geq ECost(1,j+1) - Entries(a) - ECost(1,a) - ECost(a+1,j+1).$$

Therefore,

$$\begin{aligned}
\Delta(j+1,2) &- \Delta(j,2) \\
&\geq ECost(1,j+1) - Entries(a) - ECost(1,a) - ECost(a+1,j+1) \\
&\quad -ECost(1,j) + Entries(a) + ECost(1,a) + ECost(a+1,j) \\
&= [ECost(1,j+1) - ECost(a+1,j+1)] \\
&\quad -[ECost(1,j) - ECost(a+1,j)]
\end{aligned}$$

From this and Lemma 12, it follows that when Z is true, $\Delta(j+1,2) - \Delta(j,2) \geq 0$.

■

Lemma 15 $\forall(j > 0, k > 2)[(\Delta(j,k-1) \leq \Delta(j+1,k-1)] \Longrightarrow \forall(j > 0, k > 2)[(\Delta(j,k) \leq \Delta(j+1,k)]$.

Proof Assume that $\forall(j > 0, k > 2)[(\Delta(j,k-1) \leq \Delta(j+1,k-1)]$. We shall show that $\forall(j > 0, k > 2)[(\Delta(j,k) \leq \Delta(j+1,k)]$. Let $M(j,k) = b$ and $M(j+1,k-1) = c$.

Case 1: $c \geq b$.

$$\begin{aligned}
\Delta(j,k) &= C(j,k-1) - C(j,k) \\
&= C(j,k-1) - Entries(j) - C(b,k-1) - ECost(b+1,j) \\
&\leq Entries(j) + C(b,k-2) + ECost(b+1,j) \\
&\quad -Entries(j) - C(b,k-1) - ECost(b+1,j) \\
&= \Delta(b,k-1).
\end{aligned}$$

Also,

$$\begin{aligned}
\Delta(j+1,k) &= C(j+1,k-1) - C(j+1,k) \\
&\geq Entries(j+1) + C(c,k-2) + ECost(c+1,j+1) \\
&\quad -Entries(j+1) - C(c,k-1) - ECost(c+1,j+1) \\
&= \Delta(c,k-1).
\end{aligned}$$

Since $c \geq b$, $\Delta(b, k-1) \leq \Delta(c, k-1)$. Therefore,

$$\Delta(j+1, k) \geq \Delta(c, k-1) \geq \Delta(b, k-1) \geq \Delta(j, k).$$

Case 2: $c < b$.

Let $M[j+1, k] = a$, $M[j, k] = b$, $M[j+1, k-1] = c$, and $M[j, k-1] = d$. From Lemma 13, $a \geq b$ and $c \geq d$. Since $b > c$, $a \geq b > c \geq d$. Also,

$$\begin{aligned}
\Delta(j, k) &= C(j, k-1) - C(j, k) \\
&= [Entries(j) + C(d, k-2) + ECost(d+1, j)] \\
&\quad -[Entries(j) + C(b, k-1) + ECost(b+1, j)]
\end{aligned}$$

and

$$\begin{aligned}
\Delta(j+1, k) &= C(j+1, k-1) - C(j+1, k) \\
&= [Entries(j+1) + C(c, k-2) + ECost(c+1, j+1)] \\
&\quad -[Entries(j+1) + C(a, k-1) + ECost(a+1, j+1)].
\end{aligned}$$

Therefore,

$$\begin{aligned}
\Delta(j+1, k) - \Delta(j, k) &= [C(c, k-2) + ECost(c+1, j+1)] \\
&\quad -[C(d, k-2) + ECost(d+1, j)] \\
&\quad +[C(b, k-1) + ECost(b+1, j)] \\
&\quad -[C(a, k-1) + ECost(a+1, j+1)]. \qquad (3.13)
\end{aligned}$$

Since $j > b > c \geq d = M[j, k-1]$,

$$\begin{aligned}
Entries(j) &+ C(c, k-2) + ECost(c+1, j) \\
&\geq Entries(j) + C(d, k-2) + ECost(d+1, j) \qquad (3.14)
\end{aligned}$$

Furthermore, since $M(j+1,k) = a \geq b$,

$$\begin{aligned} Entries(j+1) &+ C(b,k-1) + ECost(b+1,j+1) \\ &\geq Entries(j+1) + C(a,k-1) + ECost(a+1,j+1) \end{aligned} \quad (3.15)$$

Substituting Equations 3.14 and 3.15 into Equation 3.13, we get

$$\begin{aligned} \Delta(j+1,k) - \Delta(j,k) \geq\ & [ECost(c+1,j+1) - ECost(b+1,j+1)] \\ & -[ECost(c+1,j) - ECost(b+1,j)]. \end{aligned}$$

Lemma 12 and $c < b$ imply that when Z is true, $ECost(c+1,j+1) - ECost(b+1,j+1) \geq ECost(c+1,j) - ECost(b+1,j)$. Therefore, $\Delta(j+1,k) - \Delta(j,k) \geq 0$.

■

Lemma 16 $\forall(j > 0, k \geq 2)[\Delta(j,k) \leq \Delta(j+1,k)]$.

Proof Follows from Lemmas 14 and 15. ■

Lemma 17 *Let* $k > 2$. $\forall(j > 0)[\Delta(j,k-1) \leq \Delta(j+1,k-1)] \Longrightarrow \forall(j > 0)[M(j,k) \geq M(j,k-1)]$.

Proof Assume that $\forall(j > 0)[\Delta(j,k-1) \leq \Delta(j+1,k-1)]$. Suppose that $M(j,k-1) = a$, $M(j,k) = b$, and $b < a$ for some $j, j > 0$. From Equation 3.6, we get

$$\begin{aligned} C(j,k) &= Entries(j) + C(b,k-1) + ECost(b+1,j) \\ &\leq Entries(j) + C(a,k-1) + ECost(a+1,j) \end{aligned}$$

and

$$\begin{aligned} C(j,k-1) &= Entries(j) + C(a,k-2) + ECost(a+1,j) \\ &< Entries(j) + C(b,k-2) + ECost(b+1,j). \end{aligned}$$

Hence,

$$C(b,k-1) + C(a,k-2) < C(a,k-1) + C(b,k-2).$$

Therefore,

$$\Delta(a, k-1) < \Delta(b, k-1).$$

However, $b < a$ and $\forall(j > 0)[\Delta(j, k-1) \leq \Delta(j+1, k-1)]$ imply that $\Delta(b, k-1) \leq \Delta(a, k-1)$. Since our assumption that $b < a$ leads to a contradiction, it must be that there is no $j > 0$ for which $M(j, k-1) = a$, $M(j, k) = b$, and $b < a$. ■

Lemma 18 $\forall(j > 0, k > 2)[M(j, k) \geq M(j, k-1)]$.

Proof Follows from Lemmas 16 and 17. ■

Theorem 2 $\forall(j > 0, k > 2)[M(j, k) \geq max\{M(j-1, k), M(j, k-1), k-1\}]$.

Proof Follows from Lemmas 13 and 18 and the fact that $M(j, k)$ is in the range $r - 1 \cdots j - 1$. ■

Note 2 *From Lemma 16, it follows that whenever* $\Delta(j, k) > 0$*, then* $\Delta(q, k) > 0$*,* $\forall q > j$*.*

Since Theorem 2 holds only when condition Z is true, we must be able to check for this condition in any implementation that attempts to use Theorem 2 to reduce the range for m in Equation 3.6. Unfortunately, the time required to check condition Z exceeds the anticipated gain from using the narrower range. However, we can provide good reason to expect that condition Z will hold on almost all practical data sets (certainly, the condition holds on the practical data sets available to us). Therefore, we propose the use of the narrower range in practice. Even if the condition fails on some data set, the penalty for using the narrower range would be a suboptimal solution. Since C and T are themselves only upper bounds on the cost of optimal solutions, it isn't clear that much is to be lost by solving for T and C inexactly.

The condition Z is

$$ECost(a, j) - ECost(b, j) \geq 2(n(b, j) - n(a, j))$$

where, $0 < a < b \leq j$. We restate this condition in terms of the number of nodes at level $j-1$ of the 1-bit trie for the prefix set. First, put in all missing nodes so that the total number of nodes at level i of the 1-bit trie is 2^i. Call the new nodes *dummy nodes*. Let the number of dummy nodes added to level $j-1$ be $dum(j)$. A node x, dummy or otherwise, at level $j-1$ is covered by a length s, $s < j$, prefix iff x's ancestor at level $s-1$ contains a prefix. Label a node at level $j-1$ iff it is covered by a prefix whose length is between a and $b-1$ and by no prefix whose length is between b and $j-1$ (equivalently, trace a path toward the root from each node x at level j; if the first prefix encountered is at one of the levels $a-1, \cdots, b-2$, label x). Let $N_i(j)$, $0 \leq i \leq 2$ be the number of labeled nodes at level $j-1$ that contain exactly i prefixes (note that a dummy node has no prefix). We see that

$$ECost(a,j) - ECost(b,j) = 2\sum_{i=0}^{2} N_i(j)$$

Further, let $cov(a,j)$ be the number of length j prefixes covered by prefixes whose length is between a and $j-1$. So,

$$\begin{aligned} n(b,j) - n(a,j) &= n(j,j) - cov(b,j) - [n(j,j) - cov(a,j)] \\ &= cov(a,j) - cov(b,j) \\ &= N_1(j) + 2N_2(j) \end{aligned}$$

Therefore,

$$ECost(a,j) - ECost(b,j) - 2(n(b,j) - n(a,j)) = 2(N_0(j) - N_2(j))$$

So, Z is true iff $N_0(j) \geq N_2(j)$. The 1-bit trie for practical data sets has between $2n$ and $3n$ nodes ([71, 73]). So, for large j, $dum(j)$ is fairly close to 2^{j-1} while the number of nodes that have 2 prefixes is at most $n/2$. Since the nodes that comprise $N_0(j)$ are drawn from a much larger pool (the pool is $dum(j)$ plus the empty level $j-1$ nodes of the 1-bit trie) while those that comprise $N_2(j)$ are drawn from a much

```
Heuristic OptimalLengths(W, k)
// W is length of longest prefix.
// k is maximum number of target lengths desired.
// Return C(W,k) and compute M(*,*).
{
    for (j = 1; j <= W; j++) {
       C(j,1) := Entries(j) + ECost(1,j);
       M(j,1) := -1; }
    for (r = 1; r <= k; r++)
       C(0,r) := 0;
    for (r = 2; r ≤ k; r++)
       for (j = r; j <= W; j++) {
       // Compute C(j,r)
          minJ := max(M(j-1,r), M(j,r-1), r-1);
          minCost := Entries(j) + C(minJ, r-1)
                  +ECost(minJ+1, j);
          minCost := C(j,r-1);
          minL := M(j,r-1);
          for (m = minJ+1; m < j; m++) {
             cost := Entries(j) + C(m,r-1) + ECost(m+1,j);
             if (cost < minCost) then
                {minCost := cost; minL := m;}}
          C[j,r] := minCost; M[j,r] := minL;}
    return C[W,k];
}
```

Figure 3–7: Algorithm for binary-search hash tables

smaller pool (the pool comprises the nodes at level $j-1$ that have 2 prefixes), we expect $N_0(j) >> N_2(j)$. For small j, almost all nodes (dummy or otherwise) at level $j-1$ are empty. Again, we expect $N_0(j) \geq N_2(j)$.

Theorem 2 leads to Heuristic *OptimalLengths* (Figure 3–7), which computes $C(W,k)$. The complexity of this algorithm is $O(kW^2)$. Using the M values, the at most k storage-optimal target lengths may be determined in an additional $O(k)$ time. When we add in the time needed to compute the *ECost* and *Entries* values, the asymptotic complexity becomes $O(nW^2)$. On practical data sets, the complexity is $O(nW + kW^2)$.

3.5 More Accurate Cost Estimator

In Section 3.1, we stated three reasons why the actual number of prefixes and markers in H_j may be considerably less than $Entries(j) + ExpansionCost(m+1, j)$. Let $EC(i, j)$ be as defined in Section 3.2 and let $MCost(i, j), 0 < i \leq j \leq W$ be the number of subtrees rooted at level j of the 1-bit trie that contain prefixes that are not covered by a prefix whose length is between i and j. From the reasons stated in Section 3.1, it follows that $EC(i, j) + MCost(i, j)$ is a more accurate estimator of actual number of markers and prefixes in H_j.

Using the more accurate estimator for the number of prefixes and markers, the dynamic programming recurrence for C (Equations 3.6 and 3.7) becomes

$$C(j, r) = \min_{m \in \{0 \ldots j-1\}} \{C(m, r-1) + EC(m+1, j) + MCost(m+1, j)\}, j > 0, r > 1 \tag{3.16}$$

$$C(0, r) = 0, C(j, 1) = EC(1, j) + MCost(1, j), j > 0 \tag{3.17}$$

Since $MCost$ may be computed in the same asymptotic time as needed to compute EC and $ECost$, the asymptotic complexity of the heuristic with the more accurate cost estimator is the same as that of the heuristic of [80].

3.6 Experimental Results

We programmed our space-optimal algorithm, reduced-range heuristic of Section 3.4, and the more accurate cost estimator heuristic of Section 3.5 in C and compared their performance against that of the heuristic of Srinivasan [80]. All algorithms and heuristics were adapted for both the $ECHT$ and the $ACHT$ problems. All codes were compiled using the **gcc** compiler and optimization level -O2. The codes were run on a SUN Ultra Enterprise 4000/5000 computer. For test data, we used the five IPv4 prefix databases of Table 2–1. These databases correspond to

backbone routers. Notice that the number of nodes in the 1-bit trie for each of our databases is between $2n$ and $3n$, where n is the number of prefixes in the database.

Table 3–1 shows the memory (i.e., sum of number of prefixes and markers to be stored in the hash tables) required by the solution to $ECHT(P, k)$ for each of our five databases. The k values used by us are 3, 7, and 15 (corresponding to a lookup performance of 2, 3, and 4 memory accesses per lookup, respectively). For the two heuristics S (heuristic of [80]) and AC (heuristic of Section 3.5, we provide both the memory requirement as estimated by the T function as well as the actual requirement of the solution generated by these two heuristics (since the two heuristics consistently generated solutions with the same actual memory requirement, the actual requirement data is provided in a single column). This latter quantity is obtained by counting the number of prefixes and markers for the k lengths determined by the heuristic. As expected, the use of the more accurate cost estimator in heuristic AC results in smaller T values. However, these smaller values do not translate into a reduced actual memory cost. In all cases, the use of the more accurate cost estimator did not affect the selection of the k lengths and the resulting actual number of prefixes and markers was the same using heuristics S and AC. The space-optimal algorithm of Section 3.2 produces solutions whose memory requirement is up to 15% less than that of the two heuristics. Interestingly, for $k = 3$, the two heuristics generate optimal solutions for all 5 of our databases.

Table 3–2 gives the memory requirements of the solutions obtained by the two heuristics and the optimal algorithm for $ACHT(P, k)$. For the cases $k = 3$ and 7, the memory requirements of the optimal solutions as well as those of the heuristic solutions for the "at most k lengths" version of our problem are the same as those for the "exactly k lengths version". However, for all 5 of our databases the optimal solution for $ACHT(P, 15)$ is superior to that for $ECHT(P, 15)$. The C value of the heuristic solutions for $ACHT(P, 15)$ are smaller than the T values of the heuristic

Table 3–1: Number of prefixes and markers in solution to $ECHT(P, k)$

Database	k	T		Actual	Optimal
		S	AC		
Paix	3	321,189	305,147	299,884	299,884
	7	167,162	158,542	144,617	124,516
	15	167,048	158,428	118,687	107,195
Pb	3	143,648	138,880	133,105	133,105
	7	75,036	72,313	62,598	53,318
	15	74,921	72,198	50,389	44,996
MaeWest	3	140,516	136,194	131,042	13,1042
	7	66,251	63,585	55,006	48,404
	15	66,157	63,491	44,592	39,620
Aads	3	117,908	114,452	109,430	109,430
	7	58,732	56,600	48,436	41,864
	15	58,630	56,498	38,949	34,705
MaeEast	3	103,607	100,599	95,822	95,822
	7	51,075	49,111	41,672	36,151
	15	50,952	48,988	33,651	29,610

solutions for $ECHT(P, 15)$. With the exception of MaeWest, however, the actual cost of the heuristic solutions for $ACHT(P, 15)$ are larger than the actual costs of the heuristic solutions for $ECHT(P, 15)$. This is due to the fact that T and C are only upper bounds on actual cost.

Table 3–3 gives the preprocessing time (i.e., the time to compute $ExpansionCost$, EC, $Entries$, MC, and $MCost$ as needed by the heuristic or optimal algorithm) for our heuristics and optimal algorithm. The preprocessing time for the optimal algorithm is 8 to 9 times that for the heuristic of [80]. The preprocessing time for the more accurate cost-estimator heuristic is about 60% more than that for the heuristic of [80].

Tables 3–4 and 3–5 give the times needed to solve the dynamic programming recurrences for our heuristics and our optimal-space algorithm. In these tables, RRS and RRAC refer to the reduced-range versions of S and AC, respectively. As expected from the analyses of these methods, the preprocessing time is significantly larger than the time needed to solve the dynamic programming recurrences (note that the times in

Table 3–2: Number of prefixes and markers in solution to $ACHT(P, k)$

Database	k	C		Actual	Optimal
		S	AC		
Paix	3	321,189	305,147	299,884	299,884
	7	167,162	158,542	144,617	124,516
	15	167,034	158,414	124,832	105,978
Pb	3	143,648	138,880	133,105	133,105
	7	75,036	72,313	62,598	53,318
	15	74,908	72,185	54,922	44,552
MaeWest	3	140,516	136,194	131,042	131,042
	7	66,251	63,585	55,006	48,404
	15	66,143	63,477	44,449	39,209
Aads	3	117,908	114,452	109,430	109,430
	7	58,732	56,600	48,436	41,864
	15	58,617	56,485	42,735	34,299
MaeEast	3	103,607	100,599	95,822	95,822
	7	51,075	49,111	41,672	36,151
	15	50,937	48,973	37,076	29,255

Table 3–3: Preprocessing time in milliseconds

Database	S	AC	Optimal
Paix	540	820	4,230
Pb	280	440	2,460
MaeWest	260	390	2,210
Aads	230	350	2,049
MaeEast	210	320	1,889

Table 3–4: Execution time, in μsec, for $ECHT(P, k)$

Database	k	S	RRS	AC	RRAC	Optimal
Paix	3	40	18	46	21	400
	7	168	68	223	76	13,660
	15	344	132	442	168	24,650
Pb	3	40	19	63	21	400
	7	172	73	311	97	13,720
	15	329	138	333	159	24,750
MaeWest	3	41	19	39	25	410
	7	168	69	196	84	14,620
	15	335	134	416	146	24,670
Aads	3	38	19	39	21	420
	7	164	72	263	109	13,650
	15	343	139	421	154	24,550
MaeEast	3	39	19	38	22	400
	7	168	74	163	82	13,690
	15	346	141	332	158	25,060

Table 3–3 are in milliseconds while those in Tables 3–4 and 3–5 are in microseconds). Note also that the time for the reduced-range version of each heuristic is less than half that of the original heuristic.

3.7 Summary

We have developed optimal algorithms for the $ECHT(P, k)$ and $ACHT(P, k)$ problems; shown how the dynamic programming recurrence for the heuristic of [80] may be solved in $O(nW + kW^2)$ time on practical data sets (in contrast, the analysis of [80] suggests an $O(nW^2)$ complexity); and proposed a reduced-range heuristic as well as a more accurate cost-estimator heuristic. Experimental results show that the reduced-range heuristic reduces the time to solve the dynamic programming recurrences by more than a factor of 2 while yielding the same result as the original full-range heuristic. We are unable to compare the reduction in run time that results from our methods to do the preprocessing versus that proposed in [80], because the code of [80] is unavailable. Although the more accurate cost-estimator heuristic results in solutions with a better cost estimate than those produced by the heuristic of

Table 3–5: Execution time, in μsec, for $ACHT(P, k)$

Database	k	S	RRS	AC	RRAC	Optimal
Paix	3	40	18	46	21	440
	7	170	66	165	113	14,420
	15	344	127	420	162	26,490
Pb	3	39	19	86	27	440
	7	165	69	210	82	14,500
	15	342	136	401	146	25,800
MaeWest	3	39	19	39	21	430
	7	173	67	231	95	14,570
	15	338	129	462	170	26,060
Aads	3	40	19	39	28	420
	7	168	69	264	81	14,550
	15	343	131	485	149	26,240
MaeEast	3	40	19	39	22	430
	7	167	70	162	109	14,470
	15	346	134	365	183	26,530

[80], the actual costs of the solutions produced by the two heuristics are the same for our test sets. The optimal-space algorithm produces solutions with a memory requirement up to 15% less than that of the solutions produced by the heuristics. However, the optimal-space algorithm takes between 8 and 9 times as much time (preprocessing and recurrence solution time) as does the heuristic of [80] and between 5 and 6 times the time taken by the more accurate cost-estimator heuristic.

Even though the optimal-space algorithm takes significantly more time than is taken by any of the heuristics, its time requirements are very practical. Therefore, the optimal-space algorithm is recommended for applications in which memory conservation is crucial. When a near-optimal solution suffices, our reduced-range heuristic should be used. Finally, although, on our test sets, our more accurate cost estimator produced solutions that have the same cost as those obtained using the cost estimator in [80], there seems to be no reason to use the less accurate estimator.

CHAPTER 4
$O(\log n)$ DYNAMIC ROUTER-TABLE

In this chapter, we show how to use the range encoding idea of [44] so that longest prefix matching as well as prefix insertion and deletion can be done in $O(\log n)$ time. Despite the intense research that has been conducted in recent years, there is no known way to peform longest prefix matches as well as insertion and deletion of prefixes in $O(\log n)$ time.

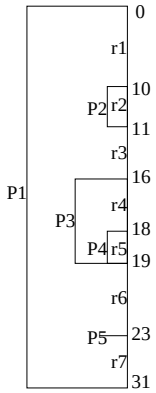
In Section 4.1, we describe the range encoding technique of [44]. We establish a few properties of ranges that represent prefixes in Section 4.2. Our $O(\log n)$ method is described in Section 4.3. In Section 4.4, we present our experimental results. These results, obtained using real IPv4 prefix databases, indicate that the $O(\log n)$ method proposed in this chapter represents a good alternative to existing methods in environments where there is a significant number of insert and/or delete opeations. For example, our method takes more time to find the longest matching prefix than do the variable-stride tries of [82]. However, although these tries are optimized for longest matching prefix searches, they perform very poorly when it comes to insertion and deletion of prefixes. Our proposed method handily outperforms variable-stride tries on these latter operations.

4.1 Prefixes and Ranges

Lampson, Srinivasan, and Varghese [44] have proposed a binary search scheme for longest prefix matching. In this scheme, each prefix is represented as a range $[s, f]$, where s is the start of the range for the prefix and f is the finish of the range for that prefix. For example, when $W = 5$, the prefix $P = 1^*$ matches all destination addresses in the range [10000, 11111] = [16, 31]. So, for prefix P, $s = 16$ and $f = 31$. Figure 4–1 shows a set of five prefixes together with the start and finish of the range

Prefix Name	Prefix	Range Start	Range Finish
P1	*	0	31
P2	0101*	10	11
P3	100*	16	19
P4	1001*	18	19
P5	10111	23	23

Figure 4–1: Prefixes and their ranges



(a) Pictorial representation of prefixes and ranges

End Point	>	=
0	P1	P1
10	P2	P2
11	P1	P2
16	P3	P3
18	P4	P4
19	P1	P4
23	P1	P5
31	-	P1

(b) Table for binary search

Figure 4–2: Pictorial and tabular representation of prefixes and ranges

for each. This figure assumes that $W = 5$. The prefix P1 = *, which matches all legal destination addresses, is called the *default* prefix. Although a real router database may not include the default prefix, *we assume throughout this chapter that this prefix is always present.* This assumption does not, in any way, affect the validity of our work as we may simply augment router databases that do not include the default prefix with a default prefix whose next hop field is null.

Prefixes and their ranges may be drawn as nested rectangles as in Figure 4–2(a), which gives the pictorial representation of the five prefixes of Figure 4–1.

Lampson et al. [44] propose the construction of a table of distinct range end-points such as the one shown in Figure 4–2(b). The distinct end points (range start and finish points) for the prefixes of Figure 4–1 are [0, 10, 11, 16, 18, 19, 23, 31]. Let r_i, $1 \leq i \leq q \leq 2n$ be the q distinct range-end-points for a set of n prefixes. Let $r_{q+1} = \infty$. Let $LMP(d)$ be the longest matching prefix for the destination address

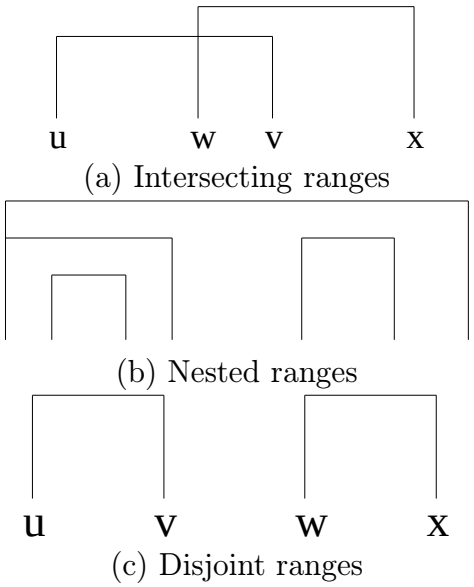
d. With each distinct range end-point, r_i, $1 \leq i \leq q$, the table stores the longest matching prefix for destination addresses d such that (a) $r_i < d < r_{i+1}$ (this is the column labeled ">" in Figure 4–2(b)) and (b) $r_i = d$ (column labeled "="). Now, $LMP(d)$, $r_1 \leq d \leq r_q$ can be determined in $O(\log n)$ time by performing a binary search to find the unique i such that $r_i \leq d < r_{i+1}$. If $r_i = d$, $LMP(d)$ is given by the "=" entry; otherwise, it is given by the ">" entry. For example, since $d = 20$ satisfies $19 \leq d < 23$ and since $d \neq 19$, the ">" entry of the end point 19 is used to determine that $LMP(20)$ is P1. As noted by Lampson et al. [44], the range end-point table can be built in $O(n)$ time (this assumes that the end points are available in ascending order). Unfortunately, as stated in [44], updating the range end-point table following the insertion or deletion of a prefix also takes $O(n)$ time because $O(n)$ ">" and/or "=" entries may change. Although Lampson et al. [44] provide ways to reduce the complexity of the search for the LMP by a constant factor, these methods do not result in schemes that permit prefix insertion and deletion in $O(\log n)$ time.

4.2 Properties of Prefix Ranges

The *length*, $length(P)$, of a prefix P is the number of zeroes and ones in the binary representation of the prefix. For example, P1 of Figure 4–1 has a length of 0 and $length(\text{P4}) = 4$. W is the number of bits in a destination address. Hence, the number of bits in the start and finish points of a prefix also is W. $P = [s, f]$ is a *trivial prefix* iff $length(P) = W$ (equivalently, iff $s = f$). P is a *nontrivial prefix* iff $length(P) < W$ (equivalently, iff $s \neq f$). Prefixes P1–P4 of Figure 4–1 are nontrivial while P5 is a trivial prefix. Let $lsb(x)$ be the least significant bit in the binary representation of x. For example, $lsb(32) = 0$ and $\text{lsb}(3) = 1$.

Lemma 19 *If $P = [s, f]$ is a nontrivial prefix, then $lsb(s) = 0$ and $lsb(f) = 1$.*

Proof Since P is nontrivial, $length(P) < W$. Therefore, s is the bits of P followed by $W - length(P) > 0$ zeroes and f is the bits of P followed by $W - length(P) > 0$ ones. Consequently, $lsb(s) = 0$ and $lsb(f) = 1$. ■



(a) Intersecting ranges

(b) Nested ranges

(c) Disjoint ranges

Figure 4–3: Types of prefix ranges

Two ranges $[u, v]$ and $[w, x]$, $u \leq v$, $w \leq x$, $u \leq w$, *intersect* iff $u < w < v < x$ (see Figure 4–3(a)). The ranges are *nested* iff $u \leq w \leq x \leq v$ (see Figure 4–3(b)). The ranges are *disjoint* iff $v < w$ (see Figure 4–3(c)). Two prefixes intersect, are nested, or are disjoint iff the corresponding property holds with respect to their ranges.

The following lemma is implicit in [44] and other papers on prefix matching.

Lemma 20 *Let $P_i = [s_i, f_i]$ and $P_j = [s_j, f_j]$ be two different prefixes. P_i and P_j are either nested or disjoint (i.e., they cannot intersect).*

Proof When $length(P_i) = length(P_j)$, the destination addresses matched by P_i and P_j are different. So, the ranges of P_i and P_j (and hence the prefixes) are disjoint. When $length(P_i) \neq length(P_j)$, we may, without loss of generality, assume that $length(P_i) < length(P_j)$. If P_i is not a prefix of P_j (i.e., P_i and P_j differ in one of the specified bits), then again, the ranges of P_i and P_j (and hence the prefixes) are disjoint. If P_i is a prefix of P_j, $s_i \leq s_j \leq f_j \leq f_i$. Consequently, P_j is nested within P_i. ■

Lemma 21 *Let $P = [s, f]$, $s \neq f$, be a prefix and let $a = \lfloor (s + f)/2 \rfloor$. P is the longest length prefix that includes*[1] $[a, a + 1]$.

Proof First observe that $f = s + 2^{W-g} - 1$, where $g = length(P)$. Since prefixes do not intersect, any longer (or equal) length prefix $P' = [s', f']$ that includes $[a, a + 1]$ must have $s \leq s' \leq a < a + 1 \leq f' \leq f$. Further, s, s', f, and f' all have the same first g bits and s' and f' have the same first $g + 1$ (or more) bits. Since a and $a + 1$ differ in bit $g + 1$, P' cannot include $[a, a + 1]$. Therefore, no prefix whose length is longer than that of P can include $[a, a + 1]$. If $length(P') = length(P)$, $P' = P$. So, P is the longest length prefix that includes $[a, a + 1]$. ■

4.3 Representation Using Binary Search Trees

4.3.1 Representation

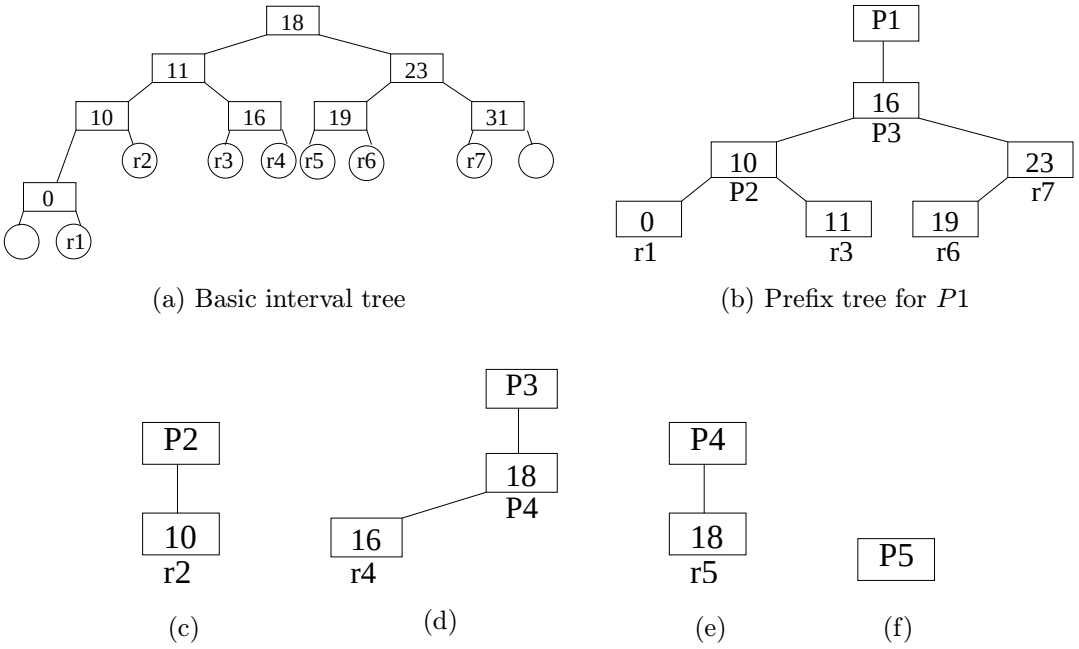
Let r_i, $1 \leq i \leq q \leq 2n$ be the distinct end points of the given set of n prefixes. Assume that these end points are ordered so that $r_i < r_{i+1}$, $1 \leq i < q$. Each of the intervals $[r_i, r_{i+1}]$, $1 \leq i < q$ is called a *basic interval*. The basic intervals of the five-prefix example of Figure 4–1 are [0, 10], [10, 11], [11, 16], [16, 18], [18, 19], [19, 23], and [23, 31]. These basic intervals are labeled $r1$ through $r7$ in Figure 4–2(a).

To perform longest prefix matches, inserts and deletes in $O(\log n)$ time per operation, we use a collection of $n+1$ binary search trees (CBST). Although the $O(\log n)$ performance results only when each of the $n+1$ binary search trees in the CBST is a balanced binary search tree, we introduce the CBST in terms of binary search trees that are not necessarily balanced.

Basic Interval Tree (BIT)

Of the $n + 1$ binary search trees in the CBST, one is called the *basic interval tree (BIT)*. The BIT comprises internal and external nodes and there is one internal node

[1] The prefix $P_i = [s_i, f_i]$ includes the interval $[a, b]$ iff $s_i \leq a \leq b \leq f_i$.



(a) Basic interval tree

(b) Prefix tree for $P1$

(c) (d) (e) (f)

(c)–(f) Prefix trees for $P2$ through $P5$

Figure 4–4: CBST for Figure 4–2(a)

for each r_i. Since the BIT has q internal nodes, it has $q+1$ external nodes. The first and last of these, in inorder, have no significance. The remaining $q-1$ external nodes, in inorder, represent the $q-1$ basic intervals of the given prefix set. Figure 4–4(a) gives a possible (we say possible because, at this time, any binary search tree organization for the internal nodes will suffice) BIT for our five-prefix example of Figure 4–2(a). Internal nodes are shown as rectangles while circles denote external nodes.

The fields of the BIT internal nodes are called *key*, $leftChild$, and $rightChild$. We describe the structure of the BIT external nodes later.

Prefix Trees

The remaining n binary search trees in the CBST are *prefix trees*. For each of the n prefixes in the router table, there is exactly one prefix tree. For each prefix and basic interval, x, define $next(x)$ to be the smallest range prefix (i.e., the longest prefix) whose range includes the range of x. For the example of Figure 4–2(a), the $next()$

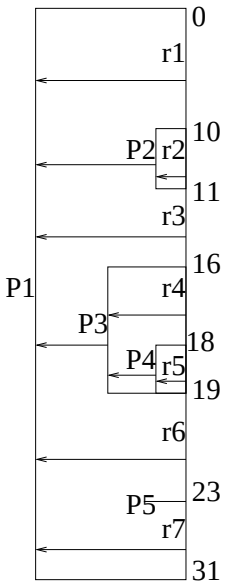


Figure 4–5: Values of $next()$ are shown as left arrows.

values for the basic intervals $r1$ through $r7$ are, respectively, $P1$, $P2$, $P1$, $P3$, $P4$, $P1$, and $P1$. Notice that the next value for the range $[r_i, r_{i+1}]$ is the same as the ">" value for r_i in Figure 4–2(b), $1 \leq i < q$. The $next()$ values for the nontrivial prefixes $P1$ through $P4$ of Figure 4–2(a) are, respectively, "-", $P1$, $P1$, and $P3$. The $next()$ values for the basic intervals and the nontrivial prefixes of Figure 4–2(a) are shown in Figure 4–5 as left arrows.

The prefix tree for prefix P comprises a header node plus one node, called a *prefix node*, for every nontrivial prefix or basic interval x such that $next(x) = P$. The prefix trees for each of the five prefixes of Figure 4–2(a) are shown in Figures 4–4(b)-(f). Notice that prefix trees do not have external nodes and that the prefix nodes of a prefix tree store the start point of the range or prefix represented by that prefix node. In the figures, the start points of the basic intervals and prefixes are shown inside the prefix nodes while the basic interval or prefix name is shown outside the node.

Notice also that nontrivial prefixes and basic intervals do not store the value of next() explicitly. The value of next() is stored only in the header of a prefix tree.[2]

BIT External Nodes

Each of the $q-1$ external nodes of the BIT that represents a basic interval x points to the prefix node that represents this basic interval in the prefix tree for $next(x)$. We call this pointer $basicIntervalPointer$. In addition, an external node that represents the basic interval $x = [r_i, r_{i+1}]$ has a pointer $startPointer$ ($finishPointer$) which points to the header node of the prefix tree for the trivial prefix (if any) whose range start and finish points are r_i (r_{i+1}). For example, $startPointer$ for $r7$ = [23,31] in Figure 4–2(a) points to the header node for the prefix tree of the trivial prefix $P5$; $finishPointer$ for $r6$ = [19, 23] also points to the header node for the prefix tree of $P5$; the remaining start and finish pointers are null.

4.3.2 Longest Prefix Matching

Notice that, because of our assumption that the default prefix is always present, there is always a prefix in our database that matches any W-bit destination address d. The search for the longest prefix that matches d is done in two steps:

Step 1 First we start at the root of the BIT and move down to an appropriate external node. An external node x that represents the basic interval $[r_i, r_{i+1}]$ is *appropriate* for d iff (a) $d = r_i$ and $x.startPointer \neq null$, or (b) $d = r_{i+1}$ and $x.finishPointer \neq null$, or (c) $LMP(d) = next(x)$. Notice that the appropriate node for a given d may not be unique. For instance, for our example BIT, the external nodes for both $r6$ and $r7$ are appropriate when $d = 23$. When $d = 18$, only the external node for $r5$ is appropriate.

[2] If $next()$ values were explicitly stored with basic intervals and trivial prefixes, an update would take $O(n)$ time, because $O(n)$ $next()$ values change following an insert/delete.

Step 2 If cases (a) or (b) of Step 1 apply, then $LMP(d)$ is obtained by following the non-null start or finish pointer. When case (c) applies, the basic interval pointer is followed into the prefix tree corresponding to $next(x)$. The header node of this prefix tree contains the longest matching prefix for d. This header node is located by following parent pointers.

In step 1, we search for an appropriate external node by performing a series of comparisons beginning at the root of the BIT. The search process differs from that employed to search a normal binary search tree (see, for example, [39]) only in how we handle equality between the address d and the key in the current search tree node y. Whenever d equals the key in an internal node y (i.e., $d = y.key$) of the BIT, we know that the basic interval $[r_i, r_{i+1}]$ represented by the rightmost (leftmost) external node in the left (right) subtree of y is such that $r_{i+1} = d$ ($r_i = d$). It is not too difficult to see that one (or both) of these two external nodes is an appropriate external node for d. To determine which, we examine the least significant bit ($lsb(key.y)$) of $key.y$ (equivalently, examine $lsb(d)$). If $lsb(key.y) = 0$, then it follows from Lemma 19 that $y.key = d$ is the start point of some prefix (note that the start and finish points of a trivial prefix are the same). Therefore, the leftmost external node in the right subtree of y is an appropriate node for d (recall that the basic interval for this external node is $[r_i, r_{i+1}]$, where $r_i = d$). When $lsb(y.key) = 1$, $y.key = d$ is the finish point of some prefix and so the rightmost external node in the left subtree of y is an appropriate node for d. This external node has $r_{i+1} = d$.

As an example, suppose we wish to determine $LMP(11)$. We start at the root of the BIT of Figure 4–4(a). Since $d = 11 < root.key = 18$, the current node y become the left child of the root. Now, since $d = y.key$ and $lsb(y.key) = 1$, the appropriate external node for d is the rightmost external node in the left subtree of y. This external node represents the basic interval $r2$. Notice that $next(r2) = P2$. As another example, consider determining $LMP(18)$. Since $d = 18 = root.key$

and $lsb(root.key) = 0$, the appropriate external node is the leftmost external node in the right subtree of the root. This external node represents the basic interval $r5 = [r_i, r_{i+1}] = [18, 19]$. Once again, notice that $LMP(18) = next(r5) = P4$. For $d = 23$, we reach the external node for $r6 = [r_i, r_{i+1}] = [19,23]$. Since $d = r_{i+1}$ and the finish pointer of this external node is non-null, the finish pointer (this points to the header node of the prefix tree for the trivial prefix $P5$) is used to determine $LMP(23) = P5$. Notice that when the router table has a trivial prefix that matches the destination address d, this trivial prefix is $LMP(d)$.

Figure 4–6 gives a high-level statement of the algorithm to determine $LMP(d)$.

Theorem 3 *(a) Algorithm* longestMatchingPrefix *correctly finds* $LMP(d)$.
(b) The complexity of algorithm longestMatchingPrefix *is* $O(height(BIT) + height(prefixTree(d)))$, *where* $prefixTree(d)$ *is the prefix tree for* $LMP(d)$.

Proof Correctness follows from the definition of the BIT and prefix tree data structures. For the complexity, we note that it takes $O(height(BIT))$ time to find the appropriate external node and an additional $O(height(prefixTree(d)))$ time to find $LMP(d)$ in case the function $prefix$ is invoked. ■

4.3.3 Inserting a Prefix

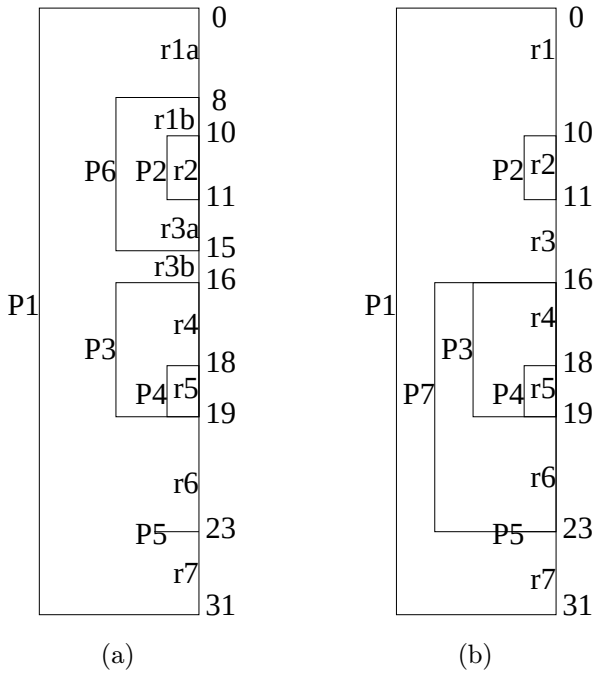
Suppose we wish to add the prefix $P6 = 01^* = [8, 15]$ to the prefix set $P1$-$P5$. Figure 4–7(a) gives the pictorial representation for the prefixes $P1$-$P6$. Relative to the pictorial representation of $P1$-$P5$ (Figure 4–2(a)), we see that the insertion of $P6$ has created two new end points (8 and 15), the basic interval $r1$ has been split into the basic intervals $r1a$ and $r1b$ as a result of the new end point 8, and the basic interval $r3$ has been split into the basic intervals $r3a$ and $r3b$ as a result of the new end point 15.

Figure 4–7(b) shows the pictorial representation for the case when $P1$-$P5$ are augmented by the prefix $P7 = 10^* = [16, 23]$. In this case no new end points are created and none of the basic intervals of $P1$-$P5$ split.

```
algorithm longestMatchingPrefix(d)
{// return header node for LMP(d)
   // find appropriate external node
   y = root of BIT;
   while (y is an internal node)
      if (d < y.key) y = y.leftChild;
      else if (d > y.key) y = y.rightChild;
      else // d equals y.key
         if (lsb(y.key) is 0)
            {
               eNode = leftmost external node in right subtree of y;
               if (eNode.startPointer is null)
                  return (prefix(eNode.basicIntervalPointer));
               else return (eNode.startPointer);
            }
         else //lsb(y.key) is 1
            {
               eNode = right most external node in left subtree of y;
               if (eNode.finishPointer is null)
                  return (prefix(eNode.basicIntervalPointer));
               else return (eNode.finishPointer);
            }
   return (prefix(y.basicIntervalPointer));
}

algorithm prefix(pNode)
{// return prefix in header node of prefix tree that contains node pNode
   y = pNode;
   while (y is not a header node)
      y = y.parent;
   return (y);
}
```

Figure 4–6: Algorithm to find $LMP(d)$



(a) Figure 4–2(a) after inserting $P6 = 01*$
(b) Figure 4–2(a) after inserting $P7 = 10*$

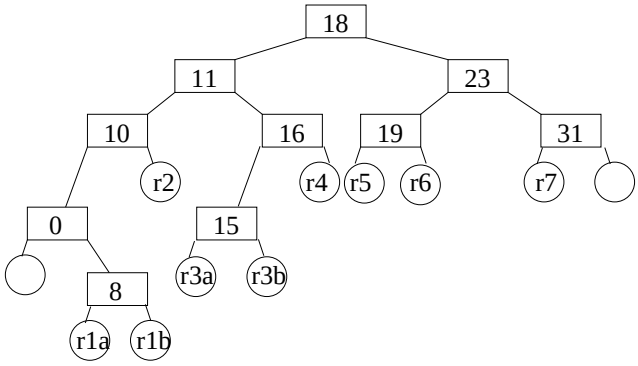
Figure 4–7: Pictorial representation of prefixes and ranges after inserting a prefix

In addition to possibly increasing the number of distinct end points, the insertion of a new prefix changes the $next()$ value of certain prefixes and basic intervals. The insertion of $P6$ into $P1$-$P5$ changes $next(P2)$ from $P1$ to $P6$ ($next(r1b)$ and $next(r3a)$ become $P6$). The insertion of $P7$ into $R1$-$R5$ changes $next(P3)$ and $next(r6)$ from $P1$ to $P7$.

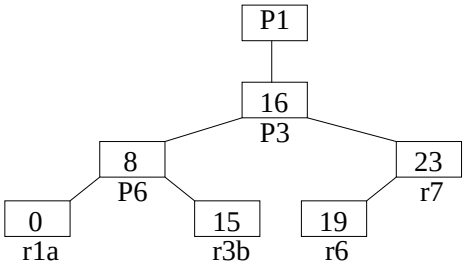
Updating the BIT

Since the default prefix $*$ is always present, we need not be concerned with insertion into an empty BIT.

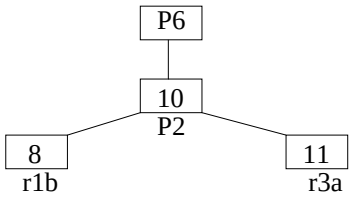
It is easy to verify that the insertion of a new prefix will increase the number of distinct end points by 0, 1, or 2. Correspondingly, the number of basic intervals will increase by 0, 1, or 2. Because the number of internal (external) nodes in a BIT equals (is one more than) the number of distinct end points, the number of internal and external nodes in the BIT increases by the same amount as does the number of



(a) BIT for $P1$-$P5$ and $P6$



(b) prefix tree for $P1$



(c) prefix tree for $P6$

Figure 4–8: Basic interval tree and prefix trees after inserting $P6 = 01*$ into Figure 4–4

distinct end points. Figure 4–8(a) shows the BIT for $P1$-$P6$. Since the insertion of $P7$ into the prefix set $P1$-$P5$ does not change the set of distinct end points, the BIT for $P1$-$P5$ and $P7$ has the same structure as does that for $P1$-$P5$.

Lemma 22 *Let $P = [s, f]$ be a new prefix that is inserted into a router database. Assume that the insertion of P creates no new end points.*

(a) If $length(P) < W$, the BIT is unchanged. (Even though the next value may change for several basic intervals and prefixes, these changes do not affect the BIT.)

(b) If $length(P) = W$, the structure of the BIT is unchanged. However, the start pointer in the external node for the basic interval $[r_i, r_{i+1}]$, where $r_i = s = f$ and the finish pointer in the external node for the basic interval $[r_i, r_{i+1}]$, where

```
algorithm insertEndPoint(u)
{// insert the end point u into the BIT
   y = root of BIT;
   while (y is an internal node)
      if (u < y.key) y = y.leftChild;
      else if (u > y.key) y = y.rightChild;
      else // u equals y.key
      {// u is not a new end point
         if (length of new prefix is W)
            {
               eNode = leftmost external node in right subtree of y;
               update eNode.startPointer to point to header node for new prefix;
               eNode = right most external node in left subtree of y;
               update eNode.finishPointer to point to header node for new prefix;
            }
            return;
      }
   // u is a new end point
   insert a new internal node z with z.key = u between y and its
    parent and create a new external node for the remaining child of z;
   return;
}
```

Figure 4–9: Algorithm to insert an end point

$r_{i+1} = s = f$ change (both now point to the header node for the prefix tree of P).

Proof Straightforward. ■

To update the BIT as required by the insertion of the prefix $P = [s, f]$, we insert the end points s and f into the BIT using algorithm *insertEndPoint* of Figure 4–9. Of course, when $s = f$, we invoke *insertEndPoint* just once.

The fields of the two external node children of the newly created internal node z are easily changed/set to their correct values. When a new internal node z is created, a basic interval $[r_i, r_{i+1}]$ is split into the two basic intervals $[r_i, u]$ and $[u, r_{i+1}]$. Let $e1$ and $e2$, respectively, be the external nodes that represent these basic intervals. Let e be the external node that represents the original interval $[r_i, r_{i+1}]$ (note that

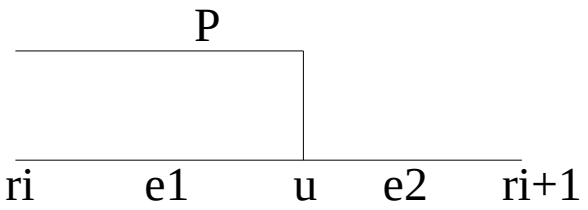


Figure 4–10: Splitting a basic interval when $lsb(u) = 1$

e is either $e1$ or $e2$). The start pointer of $e1$ is the start pointer of e and the finish pointer of $e2$ is the finish pointer of e. When the length of the new prefix P is W, the basic interval pointers of $e1$ and $e2$ are the same as that of e and the finish pointer of $e1$ and the start pointer of $e2$ point to the header node of the prefix tree of the new prefix. When $length(P) \neq W$, the finish pointer of $e1$ and the start pointer of $e2$ are null. Further, when $length(P) \neq W$ and $lsb(u) = 1$ (see Figure 4–10), the basic interval pointer of $e2$ is the same as that of e and the basic interval pointer of $e1$ points to a new node that is to go into the prefix tree of the new prefix P. The case when $length(P) \neq W$ and $lsb(u) = 0$ is similar.

Theorem 4 *(a) Algorithm* insertEndPoint *correctly inserts an end point into the BIT.*

(b) The complexity of the algorithm is $O(height(BIT))$.

Proof Correctness follows from the definition of a BIT. For the complexity, we see that it takes $O(height(BIT))$ time to exit the **while** loop. The ensuing insert (if any) of a new internal and external node takes $O(1)$ time if the BIT is not to be balanced and $O(height(BIT))$ time if the BIT is to be balanced. ■

Updating Prefix Trees

When the prefix $P = [s, f]$ is inserted, we must create a new prefix tree for P. Additionally, when $length(P) < W$ or when $length(P) = W$ and s is a new end point, we must update the prefix tree for the longest prefix $Q = [a, b]$ such that $a \leq s \leq f \leq b$ (i.e., the prefix Q such that $next(P) = Q$). Note that because of our assumption that the default prefix $*$ is always present, Q exists whenever P is not

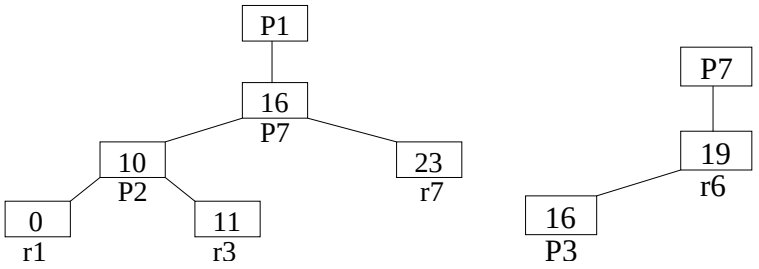
the default prefix. We assume that whenever a request is made to insert a prefix that is already in the database, we need only update the next-hop information associated with this prefix. Therefore, the only time that Q does not exist, we are to simply locate the header node for the default prefix and update the next-hop information. For the remainder of this subsection, we assume that Q exists. Additional work that is to be done includes the insertion of up to two new basic interval nodes. These nodes go into the prefix trees for P and/or Q.

Consider the insertion of $P6 = [8, 15]$ into $P1$-$P5$ (Figures 4–2(a) and 4–7(a)). When $P6$ is inserted, $Q = P1$. Let Z be the set of prefixes and basic intervals x for which $next(x) = Q = P1$ and the range of x is contained within that of $P6$ (i.e., $P2$). The $next()$ value for the prefixes and basic intervals in Z changes from $Q = P1$ to $P6$. The basic intervals ($r1$ and $r3$) that intersect the range of $P6$ (recall from Lemma 20 that no prefix can intersect $P6$) get split into four basic intervals with two of these having *next* value Q and the other two having *next* value $P6$. The prefix trees for prefixes other than Q and $P6$ are unaffected by the insertion of $P6$.

To make the above changes, we use the split and join operations [39] of a binary search tree. For binary search trees T, *small*, and *big*, these operations are defined below.

- $T.split(u)$**:** Split T into two binary search trees *small* and *big* such that *small* has all keys/elements of T that are less than u and *big* has those that are greater than or equal to u.
- $join(small, big)$**:** This operation starts with two binary search trees *small* and *big* with the property that all keys in *small* are less than every key in *big* and creates a binary search tree that includes all keys in *small* and *big*.

To determine, the basic intervals and prefixes in the prefix tree of $Q = P1$ whose *next* value changes to $P6$, we first split the prefix tree of $P1$ by invoking $split(8)$ (8 is the start point of the new prefix $P6$). The resulting binary search trees *small*1



(a) prefix tree for $P1$ after the insertion of $P7$

(b) prefix tree for $P7$

Figure 4–11: Prefix trees after inserting $P7 = 10*$ into $P1$-$P5$

and $big1$ have the keys $\{0\}$ and $\{10, 11, 16, 19, 23\}$, respectively. Next, we split the binary search tree $big1$ by invoking $split(15)$ (15 is the finish point of $P6$) to get the binary search trees $small2$ and $big2$, which have the keys $\{10, 11\}$ and $\{16, 19, 23\}$, respectively. We now have three binary search trees $small1$ with key $\{0\}$ representing the basic interval $\{r1a\}$, $small2$ with keys $\{10, 11\}$ representing $\{r1b, P2\}$, and $big2$ with keys $\{16, 19, 23\}$ representing $\{P3, r6, r7\}$. To construct the new prefix tree for $P1$, we join $small1$ and $big2$ and then insert the basic interval $r3b$ as well as the new prefix $P6$. To get the prefix tree for $P6$, we insert the basic interval $r1b$ into $small2$. The resulting $P1$ and $P6$ prefix trees are shown in Figures 4–8(b) and (c).

Now, consider the insertion of $P7 = [16, 23]$ into $P1 - P5$. Once again, $Q = P1$. Following $split(16)$, $small1$ has the keys $\{0, 10, 11\}$ and $big1$ has the keys $\{16, 19, 23\}$. When $big1$ is split using $split(23)$, we get $small2$ with keys $\{16, 19\}$ and $big2$ with the key $\{23\}$. To get the new prefix tree for $P1$, we join $small1$ and $big2$ and then insert the new prefix $P7$. The resulting tree has the keys $\{0, 10, 11, 16, 23\}$ (the key 16 represents $P7$). $small2$ is the tree for $P7$. These prefix trees for $P1$ and $P7$ are shown in Figures 4–11(a) and (b).

To complete the discussion of the insertion operation, we need to describe how the prefix Q is determined. When $length(P) < W$, Q may be determined using Lemma 23. When $length(P) = W$ and s is a new end point, Q is $LMP(s)$.

Lemma 23 *Let R be a prefix set that includes the default prefix $*$. Let $P = [s, f]$, $s \neq f$ (i.e., $length(P) < W$), $P \notin R$, be a prefix that is to be inserted into R. Let $a = \lfloor (s + f)/2 \rfloor$.*

(a) *There is a unique basic interval x of R that contains $[a, a + 1]$.*

(b) *The longest prefix $Q \in R$ that includes the interval $[s, f]$ is $next(x)$.*

Proof (a) Since the default prefix $*$ is in R, the distinct end points of R are $0 = r_1 < r_2 < ... < r_q = 2^W - 1$. Therefore, there is a unique i such that $r_i \leq a < a+1 \leq r_{i+1}$. So, $x = [r_i, r_{i+1}]$ is the unique basic interval of R that contains $[a, a_{i+1}]$.

(b) By definition, $next(x)$ is the smallest range prefix (i.e., longest prefix) $P' = [s', f']$ of R that includes the basic interval $[r_i, r_{i+1}]$. Therefore, P' is the longest prefix of R that includes $[a, a + 1]$. From Lemma 21 and $P \notin R$ (so $P \neq P'$), it follows that $length(P') < length(P)$. Since prefixes do not intersect and since both P and P' include $[a, a + 1]$, $s' \leq s \leq a < a + 1 \leq f \leq f'$. Further, since P' is the longest prefix of R with this property, $Q = P' = next(x)$. ■

Figure 4–12 gives a high-level description of our algorithm to update the prefix trees.

Theorem 5 *(a) Algorithm* updatePrefixTrees *correctly updates a prefix tree.*
(b) The complexity of the algorithm is $O(height(BIT)+split(pt)+join(pt)+insert(pt))$, where $split(pt)$, $join(pt)$, and $insert(pt)$ are, respectively, the times to split a prefix tree, join two prefix trees, and insert into a prefix tree.

Proof Correctness follows from the definition of a prefix tree. For the complexity, we see that it takes $O(height(BIT))$ time to determine Q. In addition to determining Q, at most 2 splits, 1 join, and 3 insertions into prefix trees are done. ■

Theorem 6 *The complexity of the insert-prefix operation is $O(height(BIT)+split(pt) +join(pt) + insert(pt))$.*

```
algorithm updatePrefixTrees(s, f)
{// update the prefix trees when the prefix P = [s, f] is inserted
   if (s == 0&&f == 2^W − 1)
   {// P is the default prefix
      Update next-hop field of default prefix;
      return;
   }
   if (s == f)
   {// length(P) = W
      if (P is not a new prefix) Update next-hop field for P;
      else
      {
         Create a header node for P's prefix tree;
         if (s is a new point)
         {
            Q = LMP(s);
            Insert the basic interval that begins at s into Q;
         }
      }
      return;
   }
   // P is a nontrivial prefix
   Determine Q using Lemma 23;
   if (P == Q) Update next hop of Q and return;
   (small1, big1) = Q.split(s);
   (small2, big2) = big1.split(f);
   Q = join(small1, big2);
   Insert s (i.e., prefix P) into Q;
   if (f < finish point of prefix represented by Q)
      Insert f into Q;
   Insert basic intervals into Q as needed;
   Insert basic intervals into small2 as needed;
   small2 is the prefix tree for P;
}
```

Figure 4–12: Algorithm to update prefix trees

Proof Follows from the complexities of $insertEndPoint$ and $updatePrefixTrees$ and the observation that when a prefix is inserted, we make at most 2 invocations of $insertEndPoint$ and 1 of $updatePrefixTrees$. ■

4.3.4 Deleting a Prefix

To delete $P6 = [8, 15]$ from the database $P1 - P6$ of Figure 4–7(a), we must do the following:

1. Delete 8 and 15 from the BIT and merge the basic intervals $r1a$ and $r1b$ as well as $r3a$ and $r3b$.
2. Move the prefix-tree node for $P2$, which is presently in the in prefix tree for $P6$ to the prefix tree for $P1$ and discard the remainder of the prefix tree for $P6$.

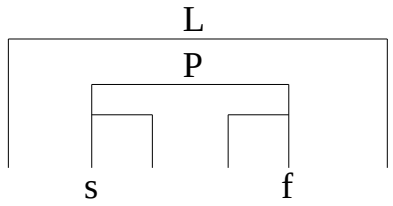
To delete $P7 = [16, 23]$ from the database $P - P5$ and $P7$ of Figure 4–7(b), we must move the prefix-tree nodes for $P3$ and $r6$ from the prefix tree for $P7$ to the prefix tree for $P1$ and discard the header node of the prefix tree for $P7$. To delete $P5 = [23, 23]$ from $P1 - P5$, we must remove 23 from the BIT, merge the basic intervals $r6$ and $r7$, and discard the prefix tree for $P5$. The deletion of the default prefix $*$ requires us to simply change the next-hop field for this prefix to null (recall that the default prefix must be retained in the database at all times).

In the remainder of our discussion, we assume that the prefix to be deleted is not the default prefix. We see that the deletion of a prefix $P = [s, f]$, $P \neq *$, requires us to perform some or all of the following tasks:

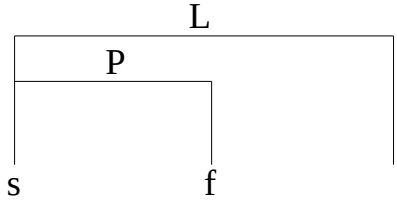
1. Locate the prefix tree for P.
2. Determine the longest prefix L whose range includes $[s, f]$ ($L = P1$ in our preceding examples).
3. Determine whether s and/or f are to be deleted from the BIT. If so, delete them.
4. Move a portion of the prefix tree for P into that of L and discard the remainder.
5. Merge pairs of external nodes in the BIT.

To perform task 1, we observe that when $s = f$, the prefix tree for P may be located by first determining an external node e of the BIT that represents a basic interval $[r_i, r_{i+1}]$ with either $r_i = s$ or $r_{i+1} = f$. In the former case, $e.startPointer$ gives us the desired prefix tree and in the latter case, $e.finishPointer$ does this. In case the pointer is null, P is not a prefix of the database. When $s \neq f$, task 1 may be performed using Lemma 23 to determine prefix Q using s and f. If $Q \neq P$, then P is not in the prefix database. In case the prefix to be deleted is not in the database, the deletion algorithm terminates.

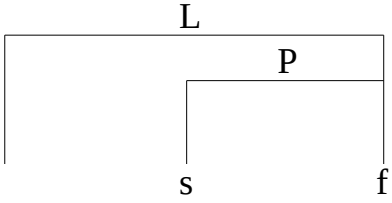
A simple strategy for task 2 is to add a prefix-node pointer $prefixNode$ to the header node of every prefix tree. The prefix-node pointer for the prefix S points to the unique node N that is in one of the prefix trees and represents prefix S. By following parent pointers from N, we reach the header node for the prefix L. The prefix-pointer in the header node of the prefix tree for S is set when S is inserted into the database. Once set, this pointer does not change. A slightly more involved strategy is described now. This strategy does not require us to make any changes to the BIT or prefix-trees structures. First note that since the prefix database contains the default prefix $*$ and since $P \neq *$, the database contains a unique prefix L of longest length whose range includes $[s, f]$. To determine L, let U denote the subset of database prefixes that either start at s or finish at f (or both). Since $P \in U$, U is not empty. Let S be the shortest prefix in U. We consider the following three cases, which are exhaustive: (1) $P = S$, (2) $P \neq S$ and S starts at s, and (3) $P \neq S$ and S finishes at f. These three cases are shown pictorially in Figure 4–13. Let x be the basic interval (if any) that includes $[s-1, s]$ (note that when $s = 0$, there is no such x) and let y be the basic interval (if any) that includes $[f, f+1]$. We see that, in all cases, L is the shorter of the prefixes $next(x)$ and $next(y)$. We may determine $next(x)$ ($next(y)$) by following the basic interval pointer in the BIT external node for



(a) P is shortest



(b) P is not shortest



(c) P is not shortest

Figure 4–13: $P = S$; $P \neq S$ and S starts at s; and $P \neq S$ and S finishes at f

x (y) to the prefix tree for $next(x)$ ($next(y)$) and then following parent pointers to the header node for $next(x)$ ($next(y)$).

The easiest way to perform task 3 is to augment the BIT structure so that with each distinct end point we maintain a count of the number of prefixes in the database that start (finish) at that end point. When this count is 1, the deletion of $P = [s, f]$ requires us to remove s (f) from the BIT. The insert algorithm is easily modified to update the count fields whenever a prefix is inserted. An alternative strategy, which doesn't require us to augment either the BIT or prefix-trees structure, is described now. When $s = f$, s is to be deleted from the BIT iff there is no other prefix for which s is an end point. To determine this, compute $next(x)$, where x is the basic interval that includes $[s, s + 1]$ in case $lsb(s) = 0$ and includes $[f - 1, f]$, otherwise. When $lsb(s) = 0$, s is to be deleted iff the start point of $next(x) \neq s$. When $lsb(s) = 1$, s is to be deleted iff the finish point of $next(x) \neq s$. When $s \neq f$, s (f) is to be deleted iff none of the following is true (1) there is a prefix in the database whose start and

finish points are s (f), (2) $next(x) \neq P$, where x is the basic interval (if any) that includes $[s, s+1]$ ($[f-1, f]$), or (3) start (finish) point of L (task 2) equals s (f).

For task 4, we first delete the header node of the prefix tree for P as well as the basic interval nodes for the up to two basic intervals in the prefix tree of P that are to be merged with adjacent basic intervals. Call the resulting binary search tree $PT'(P)$. Next, we split the prefix tree $PT(L)$ for P as in $(small, big) = PT(L).split(s)$. The new prefix tree for L is $join(join(small, PT'(P)), big)$.

Task 5 is to be done only when either s or f or both are to be deleted. This task is easily integrated into the delete s (f) task (task 3).

Theorem 7 *The complexity of the delete operation is $O(height(BIT) + height(pt) + split(pt) + join(pt) + delete(pt))$, where $height(pt)$ is the height of a prefix tree and $delete(pt)$ is the time to delete from a prefix tree.*

Proof Task 1 is done by searching down the BIT and then (possibly) going up a prefix tree. This takes $O(height(BIT) + height(pt))$ time. Task 2 requires us to go down the BIT and up a prefix tree once for each of x and y. So, task 2 also takes $O(height(BIT) + height(pt))$ time. For task 3, we must determine whether the end points s and f of the prefix that is to be deleted are also to be deleted and then delete these end points if so determined. For each of s and f, we must find a $next()$ value and then (possibly) delete the point from the BIT. It takes $O(height(BIT) + height(pt))$ time to determine $next()$ and $O(height(BIT))$ to delete a point. For task 4, we must do up to 3 deletions from a prefix tree, perform 1 split, and 2 joins. So, task 4 takes $O(delete(pt) + split(pt) + join(pt))$. Finally, task 5 is integrated into task 3 without any increase in asymptotic complexity. ■

4.3.5 Complexity

The red-black tree [39] is a good choice of data structure for the binary search trees of the CBST. The following properties [39] of red-black trees are important to us:

- The height of a red-black tree is logarithmic in the number of nodes in the tree.
- We may insert into, delete from, and split a red-black tree in O(height of tree) time.
- Two red-black trees with n_1 and n_2 nodes, respectively, may be joined in $O(\log(n_1 n_2))$ time.

From these properties and the earlier stated complexities of the search, insert, and delete algorithms for our proposed CBST structure, it follows that we can perform longest prefix matches as well as prefix insertion and deletion in $O(\log n)$ time, where n is the number of prefixes in the database. When the trees of the CBST structure are implemented as red-black trees, the resulting structure is called CRBT (collection of red-black trees).

Although the use of AVL trees in place of red-black trees also results in $O(\log n)$ router-table operations, red-black trees are generally believed to be faster than AVL trees by a constant factor. When unbalanced binary search trees are used in place of red-black trees, the complexity of the match/insert/delete algorithms becomes $O(n)$ (though the expected complexity is $O(\log n)$). Using splay trees in place of red-black trees results in router-table operations whose amortized complexity is $O(\log n)$. As for the space complexity, the BIT has at most $2n$ internal and $2n+1$ external nodes. Further, the n prefix trees together have n header nodes, $n-1$ prefix nodes (there is no prefix node for the default prefix), and at most $2n-1$ basic interval nodes. So, the BIT and the prefix trees together have at most $8n$ nodes. Therefore, the space complexity is $O(n)$.

4.3.6 Comments

Our algorithms assume that prefixes are given by the start and finish points of their ranges. In practical databases, this may not be the case; a prefix may be specified by its start point and length. In this case, the finish point of the prefix may be

Table 4–1: Statistics of prefix databases obtained from IPMA project on Sep 13, 2000

Database	Paix	Pb	MaeWest	Aads	MaeEast
Num of prefixes	85,988	35,303	30,719	27,069	22,713
Num of 32-bit prefixes	1	0	1	0	1
Num of end points	167,434	69,280	60,105	53,064	44,463
Max nesting depth	7	6	6	6	7
Avg nesting depth	2.13	1.90	1.90	1.86	1.82
Max prefix tree	76,979	44,333	38,469	36,201	32,437
Avg prefix tree	2.95	2.96	2.96	2.96	2.96

computed in $O(1)$ time provided we precompute the values $A(i) = 2^i - 1$, $0 \leq i \leq W$. The finish point of a prefix P whose start point is s is $s + A(W - length(P))$.

4.4 Experimental Results

We programmed the CRBT scheme in C++ and measured its performance using IPv4 prefix databases. The codes were run on a SUN Ultra Enterprise 4000/5000 computer. The g++ compiler with optimization level -O3 was used. For test data, we used the five IPv4 prefix databases of Table 4–1 [53]. Interestingly, the number of distinct end points is almost twice the number of prefixes in each database. The *depth of nesting* is the number of prefixes that cover a given basic interval. For example, the depth of nesting for the basic interval r1 of Figure 4–2(a) is 1, because prefix P1 is the only prefix that covers r1. The depth of nesting for r5 is 3, because P1, P3 and P4 cover r5. The maximum depth of nesting is surprisingly almost the same for all five of our databases. Note that the depth of nesting reported in Table 4–1 includes the default prefix that we have added to the database. The average nesting depth is obtained by summing the nesting depth for all basic intervals and dividing by the number of basic intervals. For our sample data, the average nesting depth is very small. In fact, if we eliminate the default prefix added by us to the original databases, the average depth of nesting becomes about 1. So, most of the basic intervals are covered by at most 1 prefix!

Table 4–2: Memory for data structure (in KBytes)

Database	Paix	Pb	MaeWest	Aads	MaeEast
Num of prefixes	85,988	35,303	30,719	27,069	22,713
Memory	16,139	6,664	5,786	5,106	4,280

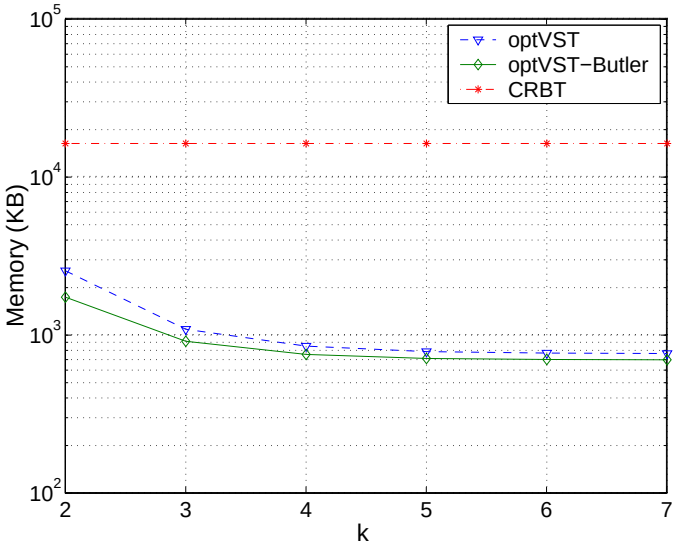


Figure 4–14: Memory required (in KBytes) by best k-VST and CRBT for Paix

Max prefix tree is the maximum number of nodes in any of the constructucted prefix trees. This number does not include the header node. Avg prefix tree is the average number of nodes in a prefix tree. Although the prefix tree for the default prefix has a very large number of nodes (this prefix tree was always the largest), the majority of the prefix trees are rather small.

Table 4–2 shows the amount of memory used by our data structure. Figure 4–14 compares the memory used by our data structure and that used by the the optimal variable-stride tries (VST) of Srinivasan and Varghese [82]. CRBT is our collection of red-black trees data structure, optVST is the optimal variable-stride trie of [82], and optVST-Butler is the optimal variable-stride trie of [82] augmented with Butler nodes. k is the height of the VST, and is a user-specified parameter. The data for VSTs are taken from [72]. Our CRBT structure takes 6.4 times the memory required by an optimal VST whose height is 2.

To measure the search, insert, and delete times for our data structure, we first obtained a random permutation of the prefixes in the databases of [53]. For each database, we started with a CRBT that included the first 75% of the prefixes (order is determined by the random permutation). Then, the remaining 25% were inserted and the time to insert these 25% was measured. The average time for one of these inserts is reported in Table 4–3. For the delete time, we started with the CRBT for 100% of the prefixes in a database and measured the time to delete the last 25% of these prefixes. The average time for one of these delete operations is reported. Finally, for the search time, we measured the time to perform a search for a destination address in each of the basic intervals, and averaged over the number of basic intervals. The columns labeled Dyn (dynamic) give the times for the case when the insert and delete codes use C++'s `new` and `delete` methods to create and free nodes as required by the insert and delete operations, respectively. The columns labeled Sta (static) is for codes that do not use dynamic memory allocation/deallocation during insert and delete operations. Instead, we begin by allocating the maximum number of prefix trees as well as the maximum number of internal, external, and prefix nodes that may be needed. These allocated nodes are linked into four different chains, one for each node type. During an insert, nodes are taken from these chains, and during a delete, nodes are returned to these chains. As the run times of Table 4–3 show, dynamic allocation/deallocation accounts for a significant portion of the run time. Although one would, in theory, expect the time for a search to be the same when dynamic and static allocation and deallocation are used, the search times reported in Table 4–3 differ for three of the five databases. We suspect that this difference is largely due to caching differences resulting from the differences in node addresses in the two schemes. It is interesting to note that even though search, insert, and delete are $O(\log n)$ operations, an insert or delete takes about 25 times as much time as does a search when dynamic allocation and deallocation are used. When static

Table 4–3: Execution time (in μsec) for randomized databases

	Search		Insert		Delete	
Database	Dyn	Sta	Dyn	Sta	Dyn	Sta
Paix	1.97	2.20	47.45	36.29	46.99	36.29
Pb	1.73	1.88	44.19	28.33	44.19	33.99
MaeWest	1.83	2.00	44.28	27.25	42.97	31.25
Aads	1.51	1.88	44.33	28.08	41.38	32.51
MaeEast	1.57	1.80	42.27	28.18	40.51	29.94

allocation and deallocation are used, this ratio is about 16. In either case, the ratio is far more than the less than two factor between the time to insert/delete from a red-black tree and that to search a red-black tree. This order of magnitude jump in the ratio of insert/delete time and search time is due to the several join and split operations needed to insert/delete into/from a CRBT.

The times of Table 4–3 cannot be compared with the times for corresponding operations on an optimal VST as reported by Sahni and Kim in [72]. This is because in the experiments conducted in [72], the database prefixes were considered in the order they appear in each database rather than in a random order. Further, in the experiments of [72], we started with an optimal VST that contained the first 90% of the database prefixes and then inserted the remaining 10%. The average time for each of these latter inserts is reported in [72]. The delete times are similarly obtained by removing the last 10% of the prefixes from an optimal VST that initially has all 100% of the prefixes. The run times for our CRBT structure for the experiment conducted in [72] are shown in Table 4–4. Notice that in this experiment, the cost of an insert/delete is only 15 times that of a search when dynamic allocation and deallocation are used. When static allocation and deallocation are used, this ratio drops to about 7.

Figures 4–15 through 4–17 compare the run times for the search, insert, and delete operations using the Paix database and the CRBT and optimal VST structures. The search time using the CRBT structure is about 4 times that when an optimal

Table 4–4: Execution time (in μsec) for original databases

	Search		Insert		Delete	
Database	Dyn	Sta	Dyn	Sta	Dyn	Sta
Paix	2.33	2.21	27.91	13.96	30.24	18.61
Pb	1.98	1.98	28.33	14.16	31.16	19.83
MaeWest	2.28	2.28	29.31	13.03	29.31	16.28
Aads	2.22	1.85	29.56	14.78	29.56	18.48
MaeEast	1.76	2.20	26.42	17.61	30.82	17.61

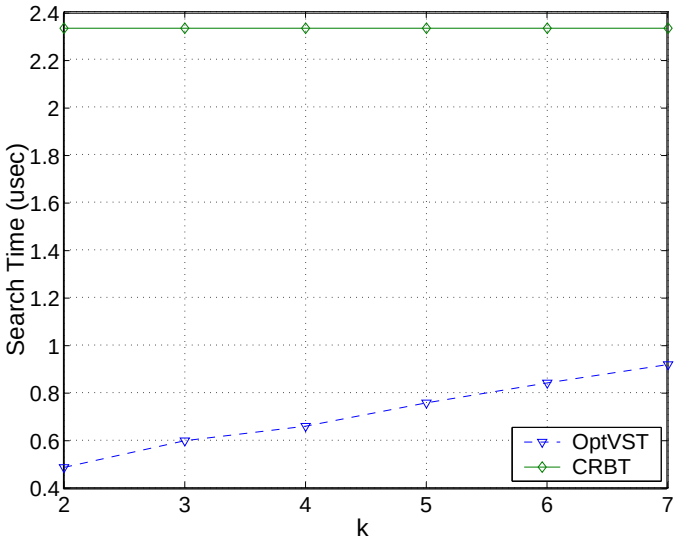


Figure 4–15: Search time (in μsec) comparison for Paix

VST of height $k = 2$ is used. However, when $k = 2$, each insert takes about 6 times the time taken by our CRBT structure with dynamic allocation/deallocation and 12 times the time taken by our CRBT structure with static allocation/deallocation! For the delete operation, these ratios are 26 and 43, respectively. Note that these ratios increase as we increase k. So, although the CRBT is slower than optimal VSTs for the search operation, it is considerably faster for the insert and delete operations!

4.5 Summary

The collection of red-black search trees (CRBT) data structure developed by us provides the first known way to perform longest-prefix matches, as well as prefix insert and delete in $O(\log n)$ time. The CRBT is interesting from both the theoretical and practical viewpoints. From the theoretical viewpoint, it represents the first data

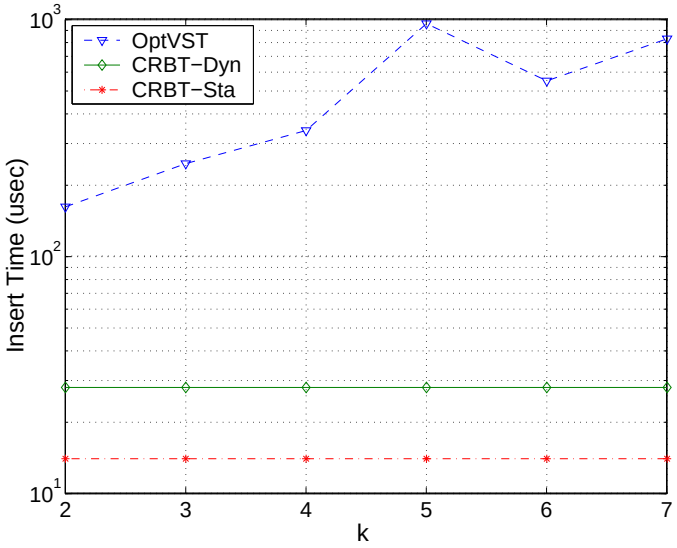


Figure 4–16: Insert time (in μsec) comparison for Paix

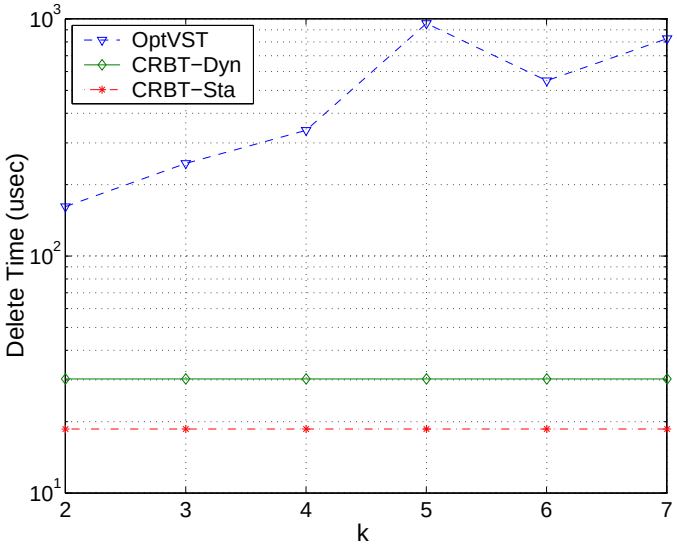


Figure 4–17: Delete time (in μsec) comparison for Paix

structure to support dynamic router-table operations in $O(\log n)$ time each. From the practical viewpoint, we note that the CRBT permits updates to be performed in much less time than when structures such as the VST, which are optimized for search, are used. In a security-conscious environment, our router would need to operate in a blocking mode (i.e., an insert/delete must complete before any inbound/outbound packets are forwarded). In such an environment, the CRBT would block traffic for about 1/10th the time the VST would. On the other hand, when traffic is not blocked due to an insert/delete in progress, the VST would process packets at 4 to 5 times the rate of the CRBT. In another application environment, our concern may be the total time to process a stream of search/insert/delete requests. Suppose that for every pair of insert and delete requests, there are m search requests. Further, suppose that the search/insert/delete times for the optimal VST are 0.5/170/800 micro seconds and that the times for the CRBT are 2.2/14/19 micro seconds (these are approximately the times for Paix). Then, when $m > 551$, the optimal VST would perform better than the CRBT.

It is worth noting that the technique developed here may be used to extend the biased skip list scheme of Ergun et al. [25, 26] so that lookups, inserts, and deletes may all be done in $O(\log n)$ expected time, while providing good expected performance for bursty access patterns (see Sahni and Kim [74]).

Finally, as noted in the introduction, when a compressed binary trie is used to represent a dynamic router table, each of the dynamic router-table operations takes $O(W)$ time. Since the compressed trie algorithms have much smaller constants than do the CRBT algorithms and since $n \leq 2^W$, the CRBT is expected to outperform the compressed binary trie structure for relatively small values of n. The threshold at which the compressed binary trie gives better overall performance is higher for IPv6 than for IPv4.

CHAPTER 5
DYNAMIC LOOKUP FOR BURSTY ACCESS PATTERNS

In this chapter, we focus on the management of router tables for a dynamic environment (i.e., search, insert, and deletes are performed dynamically) in which the access pattern is bursty. In a *bursty* access pattern the number of different destination addresses in any subsequence of q packets is $<< q$. That is, if the destination of the current packet is d, there is a high probability that d is also the destination for one or more of the next few packets. The fact that Internet packets tend to be bursty has been noted in [18, 46], for example.

Although the CRBT structure proposed in Chapter 4 can be used for our proposed environment to perform search, insert, and delete in $O(\log n)$ time each, the CRBT structure does not make explicit use of the assumed bursty nature of the access pattern. Despite the fact that the CRBT was not developed with bursty access in mind, the cache properties of contemporary computers make the CRBT perform better when the access pattern is bursty than when it is not. An alternative to the CRBT, the ACRBT, is proposed in this chapter. This alternative requires more memory than is required by the CRBT. However, experiments indicate that the search, insert, and delete operations are faster in the ACRBT than in the CRBT.

In this chapter, we develop also two data structures that explicitly account for the assumed bursty access pattern. The first, biased skip lists with prefix trees (BSLPT), extends the biased skip list (BSL) structure proposed in [25] for static router tables. This extension allows us to find longest matching prefixes as well as to insert and delete prefixes in $O(\log n)$ expected time (n is the number of prefixes in the router table). The second data structure, which is a splay-tree data structure called collection of splay trees (CST), is an adaptation of the CRBT data structure proposed

by us in Chapter 4. Using this structure, we can find longest matching prefixes and insert and delete prefixes in $O(\log n)$ amortized time. Experiments conducted by us indicate that the CST structure has best performance when the access pattern is very bursty. Otherwise, the ACRBT structure performs best. It should be noted that BSLPTs have $O(\log n)$ expected complexity per operation, CSTs have $O(\log n)$ amortized complexity, and CRBTs and ACRBTs have $O(\log n)$ worst-case complexity. The effectiveness of using supernode implementations of red-black trees to implement the ACRBT structure was evaluated experimentally. Our tests show that using a supernode implementation for the front end of the ACRBT generally reduces the search time while keeping the insert and delete time relatively unchanged.

In Section 5.1, we extend the BSL structure of [25] to BSLPT. Our proposed CST is described in Section 5.2. In Section 5.3, we describe the comparison of BITs and ABITs in both time complexity and space complexity. In Section 5.4, we present our experimental results.

5.1 Biased Skip Lists with Prefix Trees

Ergun et al. [25] propose a biased skip list structure for bursty access. Their structure is suitable for static router tables; that is, router tables in which the prefix set does not change (no prefixes are inserted or deleted). To construct the initial BSL, the n prefixes of the router table are processed to determine the up to $2n - 1$ basic intervals they define. The longest matching prefix for destination addresses that fall within each basic interval as well as for destination addresses that equal an end point is determined. A master list of basic intervals along with the determined longest matching prefix information is constructed. This list is indexed into using a skip list structure (see [62] or [70] for a description of skip lists). Figure 5–1 shows a possible skip list structure for the basic intervals of Figure 4–2(a).

In a biased skip list, ranks are assigned to the basic intervals. $rank(a) < rank(b)$ whenever interval a is accessed more recently than interval b. Basic interval ranks are

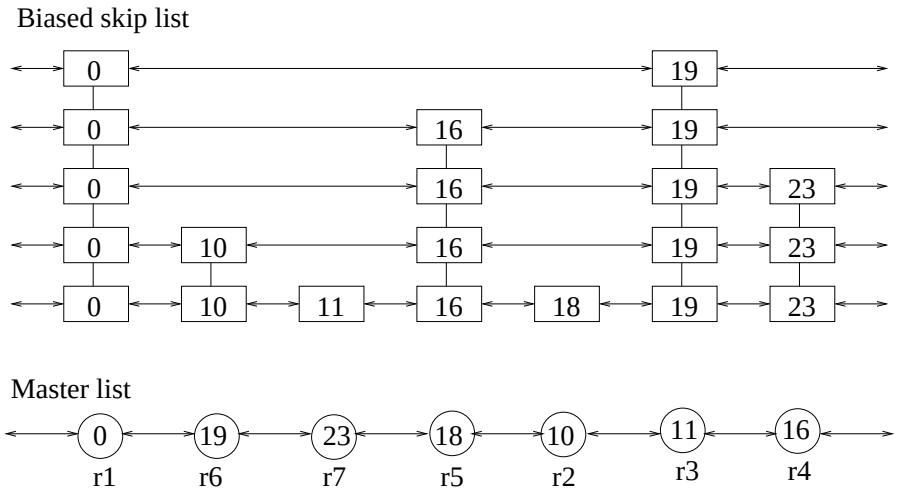


Figure 5–1: Skip list representation for basic intervals of Figure 4–2(a)

used to bias the selection of intervals that are represented in skip list chains that are close to the top. Hence, searching for a destination address that is matched by a basic interval of smaller rank takes less time than when the matching interval has higher rank. If the destination of the current packet is d and the matching basic interval for d as determined by the skip list search algorithm is a, then $rank(a)$ becomes 1 and all previous ranks between 1 and $oldRank(a)$ are increased by 1. The skip list structure is updated following each search to reflect the current ranks. Consequently, searching for basic intervals that were recently accessed is faster than searching for those that were last accessed a long time ago. This property makes the BSL structure of [25] perform better on bursty access patterns than on random access patterns. Regardless of the nature of the access, the expected time for a search in a BSL is $O(\log n)$.

When a router table has to support the insertion and deletion of prefixes, the BSL structure of [25] becomes prohibitively expensive. This is because, an insert or delete can require us to change the longest matching prefix information stored in $O(n)$ basic intervals. To overcome this problem, we enhance the BSL structure of [25] by adding the back-end CPT in Chapter 4. Notice that a BSL is functionally very similar to the front-end BIT of the CRBT structure in Chapter 4. Both are used to search for the matching basic interval. Unlike the CRBT, which uses a

basicIntervalPointer to index into a back-end CPT from which $LMP(d)$ may be determined for d values that are not matched by prefixes whose length is W, a BSL retains $LMP(d)$ within the master-list node for the matching interval. This is the reason insertion and deletion operations in a BSL take $O(n)$ time. To overcome this problem, we augment the BSL structure with a back-end CPT structure and configure the master-list nodes exactly the same as the external nodes of a BIT. Note that each master-list node of a BSL, like each external node of a BIT, represents a basic interval. In an augmented BSL, each master-list node has three pointers: *startPointer*, *finishPointer*, and *basicIntervalPointer*. For a master-list node that represents the basic interval $[r_i, r_{i+1}]$, *startPointer* (*finishPointer*) points to the header node of the prefix tree (in the back-end structure) for the prefix (if any) whose range start and finish points are r_i (r_{i+1}). *basicIntervalPointer* points to the prefix node for the basic interval $[r_i, r_{i+1}]$. This prefix node is in a prefix tree of the back-end structure.

To search a BSLPT for $LMP(d)$, we use the BSL search scheme of [25] to search the front-end BSL for a matching basic interval. This search gets us to the master-list node Q for a matching basic interval. When d equals the left (right) end-point of the matching basic interval and *startPointer* (*finishPointer*) is not null, $LMP(d)$ is pointed to by *startPointer* (*finishPointer*). Otherwise, we follow the *basicIntervalPointer* in Q to get into a prefix tree. Then, we follow parent pointers to the header of the prefix tree to determine the prefix that is $LMP(d)$.

To insert a new prefix $P = [s, f]$ (s and f are, respectively, the range start and finish points of P), we note that:

1. When neither s nor f is a new end point, the number of basic intervals is unchanged. Further, since the *basicIntervalPointer* in each master-list node points to the prefix node that represents the same basic interval, none of the *basicIntervalPointer*s in the BSL change (what does change, is the prefix tree

these prefix nodes need to be in, but, this change is handled by the insert algorithm for the back-end structure). Only the *startPointer*s and *finishPointer*s in master-list nodes may change when neither s nor f is a new end-point. Further, since these pointers are set only when there is a prefix with $s = f =$ an end point of the basic interval, at most two of these pointers may change when $|P| = W$ ($|P|$ is the length or number of bits in the prefix P). The value of these changed pointers is the header node of the prefix tree for the new prefix P. When $|P| < W$, $s \neq f$ and no pointer in the BSL changes.

2. When s is a new end point, the matching basic interval $[a, b]$ for s splits into two. Note that, because of our assumption that the default prefix * is always present, every destination address d has a matching basic interval. The replacement basic intervals for $i = [a, b]$ are $i1 = [a, s]$ and $i2 = [s, b]$. It is easy to see the following:

   - $i1.startPointer = i.startPointer$ and $i2.finishPointer = i.finishPointer$.
   - When $|P| = W$, $i1.finishPointer = i2.startPointer =$ pointer to header node of prefix tree for P; $i1.basicIntervalPointer$ and $i2.basicIntervalPointer$ point to prefix nodes for $i1$ and $i2$, respectively. Both of these prefix nodes are in the same prefix tree as was the prefix node for i. This prefix tree can be found by following parent pointers from $i.basicIntervalPointer$.
   - Figure 5–2 shows the situation when $|P| < W$. Since, $|P| < W$,

     $$i1.finishPointer = i2.startPointer = \textbf{null}.$$

     Also, from Figure 5–2, we see that $i1.basicIntervalPointer = i.basicIntervalPointer$ (note that the prefix node $i.basicIntervalPointer$ now represents the smaller basic interval $i1$) and $i2.basicIntervalPointer$ points to a new prefix node for the interval $i2$. This new prefix node is to be in the prefix tree for P.

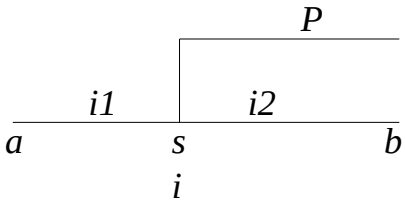


Figure 5–2: Start point s of P splits the basic interval $[a, b]$

3. The case when f is a new end point is similar to the preceding case.
4. When both s and f are new end points, both cases (2) and (3) apply.

Figure 5–3 gives the insert algorithm.

The steps for $modifyCPT$ are not provided, because these steps are identical to those used to modify the CPT back-end structure of a CRBT. The interested reader is referred to Chapter 4 for a statement of these steps.

The algorithm to delete a prefix uses the delete from CPT algorithm in Chapter 4 to modify the back-end structure appropriately together with an algorithm to make the necessary changes to the BSL front end. Since these changes to the BSL are the inverse of what happens during an insert, we do not detail them here. When the back-end prefix trees are red-black trees, the expected complexity of a search, insert, and delete is $O(\log n)$.

Ergun et al. [25] also propose a dynamic skip list structure (DSL) for static router tables. Although this structure is expected to perform better than the BSL structure, Ergun et al. have not implemented it, but have chosen to stay with the simpler BSL structure. We note that the DSL structure of [25] may also be augmented with a back-end prefix-tree structure as described here. The result is a structure in which search, insert, and delete all take $O(\log n)$ expected time each. The unaugmented DSL structure of [25] performs a search in $O(\log n)$ expected time. However, insert and delete take $O(n)$ time each.

```
algorithm insertPrefix(s, f)
{// insert a prefix
   handlePoint(s);
   if (s ≠ f) handlePoint(f);
   modifyCPT(s, f)
}
 algorithm handlePoint(u)
{// modify BSL to account for the end point u
   Search BSL for a matching basic interval Q for u.
   if (u is an end point of Q){
      if (|P| = W){
         Update startPointer in master-list node (if any) for basic
          interval that starts at u.
         Update finishPointer in master-list node (if any) for basic
          interval that finishes at u.
         }
      return;
      }
   // u is a new end point
   Replace the basic interval Q = [a, b] with the intervals
    [a, u] and [u, b] and update pointers as described in text.
   return;
}
```

Figure 5–3: BSLPT insert algorithm

5.2 Collection of Splay Trees

Splay trees [79] are binary search trees that self adjust so that the deepest encountered surviving node in any operation becomes the root following the operation. This self-adjusting property makes splay trees a promising data structure for bursty applications. Unfortunately, we cannot simply replace the use of red-black trees in the CRBT structure in Chapter 4 with splay trees. This is because, the red-black tree used to represent the BIT in Chapter 4 uses two types of nodes: internal and external. Every search has to reach an external node from which it may progress into a back-end prefix tree. Following the splay operation, the reached external node should become the root. This isn't permissible. With this realization, we were faced with the option of either modifying the splay step so that the parent of the reached external node became the root or changing the BIT structure so that the new structure did not have external nodes. The former option was rejected as this would result in a front-end tree whose root always has an external node as one of its children. So, we propose an alternative BIT structure (ABIT) in which we have only internal nodes. When the BIT structure of a CRBT is replaced by the ABIT structure, we get, what we call, the ACRBT structure.

Each (internal) node of the ABIT represents a single basic interval. Therefore, each ABIT node has fields for the left and right end points of a basic interval, a left and right child pointer, pointers to $LMP(d)$ when d equals either the left or right end point of the basic interval, and a pointer to the corresponding basic-interval node in a prefix tree for the case when d lies between the left and right end points of the basic interval (this is the same as the *basicIntervalPointer* used in CRBT and BSLPT). Figure 5–4 shows the ABIT for the basic intervals of Figure 4–2(a). In this figure, only the left and right end points of each basic interval are shown (the $LMP(d)$ values and the *basicIntervalPointers* are not shown).

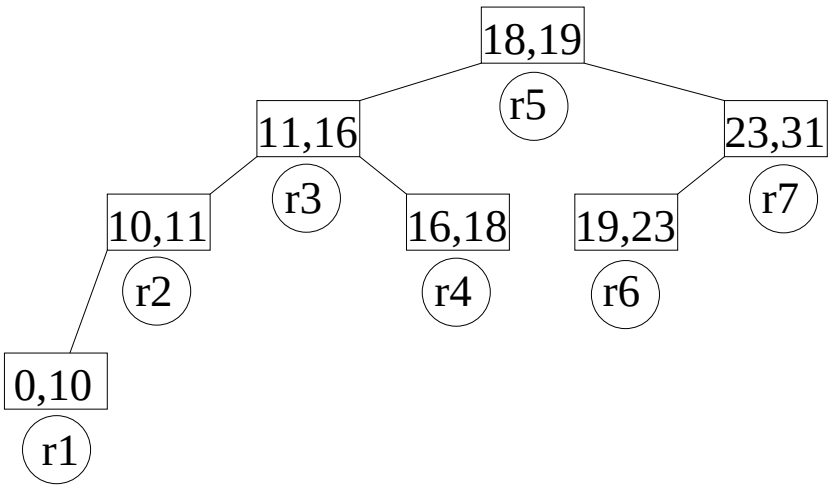


Figure 5–4: Alternative Base Interval Tree corresponding to Figure 4–2(a)

The CST structure proposed by us for bursty applications of dynamic router tables uses the ABIT structure as the front end. The back end is the same prefix tree structure proposed by us in Chapter 4 except that each prefix tree is implemented as a splay tree rather than as a red-black tree. The algorithms to search, insert, and delete for our proposed CST structure are simple adaptations of those for the proposed BSLPT structure. Therefore, further details are not provided. We note that the amortized complexity of these algorithms is $O(\log n)$.

Notice that splay trees also may be used to implement the back-end prefix trees of the BSLPT structure (rather than red-black trees).

5.3 Comparison of BITs and ABITs

Space Complexity

A BIT has two types of nodes, internal and external. For an n-prefix router-table, the number, I, of internal nodes is at most $2n$; the number, E, of external nodes is $I - 1$.[1] Each internal node has a field for an interval end-point plus three pointer

[1] Although a binary tree with m internal nodes has $m + 1$ external nodes, the first and last of these have no significance in a BIT. So, these two external nodes are not explicitly represented.

fields (two children and one parent field) and each external node has three pointer fields (into the back-end structure). Additionally, every node has a type field that enables us to distinguish between internal and external nodes and a color field to distinguish between red and black nodes. Suppose that d bytes are needed to store an interval end-point, p for a node pointer, t for the type field, and c for the color field. We see that the storage requirement of a BIT is $(d+t+c+3p)I+(t+3p)(I-1)$ $= (d + 2t + c + 6p)I - t - 3p$ bytes. The ABIT for the same n-prefix router-table has $I - 1$ nodes, each of which has two end-point, one color, and 6 pointer fields (1 parent, 2 children, and 3 to back-end nodes). So, the storage required by the ABIT is $(2d + c + 6p)(I - 1) = (2d + c + 6p)I - 2d - c - 6p$ bytes.

In theory, $t = c = 1/8$ (i.e., 1 bit) is sufficient and $p = 4$ bytes. For IPv4, $d = 4$ bytes. So, $memory(BIT) = 28.375I - 12.125$ bytes and $memory(ABIT) = 32.125I - 32.125$ bytes. The BIT structure requires about 12% less space than is required by the ABIT. For IPv6, the differential is about 25%. In practice, however, when coding in a language such as C++, we find it convenient to implement the type and color field using the data type `byte` (note that the alternative type `boolean` is 4 bytes long in the g++ implementation of C++). So, in practice, $t = c = 1$. So, in practice, for IPv4 router tables, $memory(BIT) \approx 31I$ bytes and $memory(ABIT) \approx 33I$ bytes. So, for IPv4 router tables, using the convenient 1-byte per color and type field implementation, BIT takes 6% less memory than is taken by ABIT!

Time Complexity

Since the number of internal nodes in the BIT and ABIT for a given n-prefix router table is almost the same and since a BIT has external nodes, whereas an ABIT does not, $height(ABIT) \approx height(BIT) - 1$. When searching for the longest matching prefix, the worst-case number of key comparisons in the BIT is $height(BIT)$. In an ABIT, this number is approximately $2height(ABIT)$. So, in the worst case, the ABIT requires approximately twice as many key comparisons as are required by the

BIT. However, on average, the differential is somewhat less, becuase in a BIT every search neccessarily goes all the way to an external node, whereas in an ABIT, a search may terminate at any internal node. Further, in contemporary computers, the larger number of key comparisons performed required by an ABIT may not manifest itself as an increase in measured run time. This is because, the run time is dominated by cache misses and not by key comparisons. If $height(BIT) = 10$ and $height(ABIT) = 9$, the worst-case cache-misses during a search is 10 for a BIT and 9 for an ABIT. So, despite the larger number of key comprisons, a search in an ABIT may take 10% (on average, the search goes to one level above the lowest level; the average search in an ABIT, assuming all node accesses result in cache misses, causes 20% fewer cache misses than is the case for a BIT) less time than taken by a search in a BIT.

5.4 Experimental Results

We programmed[2] the three bursty-access-schemes described in this chapter in C++ and measured the performance of these schemes as well as that of the CRBT scheme in Chapter 4 using IPv4 prefix databases. Two implementations of the ACBRT scheme were used. In the first, the BIT structure was implemented using conventional red-black trees (i.e., binary search trees with one element per node). We continue to refer to this implementation as ACRBT. In the second implementation, the BIT structure of the ACRBT was implemented using a supernode red-black tree [40]. This supernode implementation is referred to as SACRBT. In the SACRBT implementation, the BIT structure is a binary red-black search tree, each node of which has up to 5 basic intervals (i.e., up to 6 end points). These end points are stored in ascending order.

[2] We are grateful to the authors of [25] for providing us their C code for the BSL structure. Our BSLPT code utilized a C++ translation of the BSL code of [25] for the front end and our own code for the back end.

For the CST structure, we used top-down splay trees when implementing the ABIT and bottom-up splay trees for the back-end prefix trees. Top-down splay trees were used for the ABIT, because past experimental studies show that these perform better than bottom-up splay trees in normal applications (see [40], for example). In the case of prefix trees, the search operation begins at the bottom of the splay tree (in normal search applications, this search would begin at the root). So, search time is optimized by using bottom-up rather than top-down splay trees. Our implementation of the BSLPT structure also employed bottom-up splay trees in the back-end structure.

The codes were run on a SUN Ultra Enterprise 4000/5000 computer. The g++ compiler with optimization level -O2 was used. For test data, we used the five IPv4 prefix databases of Table 4–1.

Total Memory Requirement

Table 5–1 shows the amount of memory used by each of the four data structures. As predicted by our analysis of Section 5.3, the front-end structure of the CRBT (i.e., the BIT) takes about 6% less memory than is taken by the ABIT of ACRBT. The back-end structure is identical in the CRBT and ACRBT. Therefore, these take the same amount of memory. Both the front and back ends of the CST structure takes less memory than their counterparts in the CRBT and ACRBT. This is because, the front-end splay tree requires no parent pointer, no type field, and no color field and the back end splay tree requires no color field. The total memory required by the CST structure is about 12% less than that required by the ACRBT structure and about 9% less than that required by the CRBT structure. The BSLPT, on the other hand, requires about twice the memory required by each of the other structures. As noted in [40], the use of a supernode structure reduces memory requirement relative to the meory required by the corresponding binary tree structure. The front end of the SACRBT takes about 36% less memory than is required by the ACRBT front

Table 5–1: Memory requirement (in KB)

Schemes		Paix	Pb	MaeWest	Aads	MaeEast
CRBT	Front End	5,068	2,097	1,819	1,606	1,346
	Back End	6,465	2,664	2,315	2,041	1,712
	Total	11,534	4,761	4,134	3,648	3,056
ACRBT	Front End	5,395	2,232	1,936	1,710	1,432
	Back End	6,465	2,664	2,315	2,041	1,712
	Total	11,861	4,897	4,252	3,751	3,145
SACRBT	Front End	3,467	1,438	1,246	1,101	921
	Back End	6,465	2,664	2,315	2,041	1,712
	Total	9,932	4,102	3,561	3,142	2,633
CST	Front End	4,578	1,894	1,643	1,450	1,215
	Back End	5,970	2,460	2,137	1,885	1,580
	Total	10,548	4,354	3,781	3,336	2,796
BSLPT	Front End	17,530	7,698	7,015	6,496	4,578
	Back End	5,970	2,460	2,137	1,885	1,580
	Total	23,508	10,158	9,152	8,381	6,159

end. Similar reductions in front and back end memory requirements are expected when supernode implementations are used for the other data structures considered here. Figure 5–5 histograms the total memory required by each data structure.

Search Time

To measure the average search time, we first constructed the data structure for each of our five prefix databases. Eleven sets of test data were used. The destination addresses in the first set, NODUP, comprised the end points of the basic intervals corresponding to the database being searched. These end points were randomly permuted. The data set, DUP10 (DUP20), was constructed from NODUP by making 10 (20) consecutive copies of each destination address. For the data set RAN10 (RAN20), we randomly permuted every block of 50 (100) destination addresses from DUP10 (DUP20) (note that each such block has only 5 different destination addresses). The remaining 6 data sets were constructed from the 6 trace sequences obtained from `http://ita.ee.lbl.gov/html/contrib/DEC-PKT.html` and `http://ita.ee.lbl.gov/html/contrib/LBL-PKT.html`. Four of these trace sequences represent all wide-area traffic between Digital Equipment Corporation and the rest

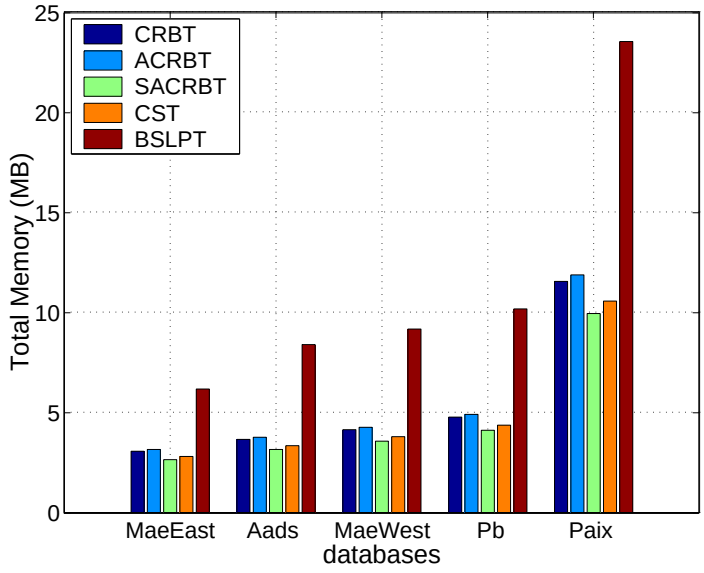


Figure 5–5: Total memory requirement (in MB)

Table 5–2: Trace sequences

Trace	#packets	#addresses
DEC-PKT-1	3,062,997	8,084
DEC-PKT-2	3,574,364	7,328
DEC-PKT-3	4,086,844	11,855
DEC-PKT-4	5,244,322	11,280
LBL-PKT-4	910,385	1,539
LBL-PKT-5	757,629	1,621

of the world for four different one-hour intervals in 1995. The remaining 2 trace sequences represent all wide-area traffic between Lawrence Berkeley Laboratory and the rest of the world for two different one-hour intervals in 1994. Table 5–2 gives the characteristics of these 6 traces.[3] In this table, #packets is the number of packets in the trace sequence and #addresses is the number of different destination addresses for these packets.

[3] Only the TCP, UDP, and TCP SYN/FIN/RST packets in the original traces are reported and used by us. The raw traces include other packets that have insufficient associated information to be useful.

These data sets represent different degrees of burstiness in the access pattern. In NODUP, all search addresses are different. So, this access pattern represents the lowest possible degree of burstiness. In DUP20, every block of 20 consecutive IP packets has the same destination address, a high degree of burstiness. In RAN20, even though each destination address occurs 20 times, the destination addresses are not necessarily in consecutive packets; there is some measure of temporal spread among the recurring addresses. Since the number of different destination addresses in the trace data is 2 to 3 orders of magnitude less than the number of packets, these traces represent a much higher degree of burstiness than do the DUP and RAN data sets. For example, in DEC-PKT-1, each destination address occurs approximately 379 times. However, temporal burstiness (i.e., inverse of the time between the first and last packets that have the same destination address) is far greater in the DUP and RAN data sets.

For the NODUP, DUP, and RAN data sets, the total search time for each data set was measured and then averaged to get the time for a single search. This experiment was repeated 10 times. So, 10 average search times were obtained. The average of these averages together with the standard deviation (SD) in the averages is given in Tables 5–3 and 5–4. Since packet destination-addresses given in the trace sequences were sanitized for privacy reasons, we randomly mapped the distinct addresses in the trace data, which are consecutive integers beginning at 1, into random 32-bit addresses. For our experiment, this random mapping to 32-bit addresses was done 10 times for each trace and the average search time for each mapping as well as the average of the averages and the standard deviation of the averages computed. Tables 5–5 and 5–6 give the times for the trace sequences. Figures 5–6 and 5–7 histogram the average times for all data sets.

The first thing to notice is that even though the CRBT and ACRBT are not designed to perform better on bursty access paterns than on non-bursty ones, they

Table 5–3: Search time (in μsec) for CRBT, ACRBT, and SACRBT structures on NODUP, DUP, and RAN data sets

Schemes		Paix	Pb	MaeWest	Aads	MaeEast
CRBT	NODUP	6.51	5.77	5.50	5.29	5.15
	SD	0.30	0.09	0.05	0.09	0.12
	DUP10	1.55	1.41	1.36	1.36	1.31
	SD	0.00	0.01	0.03	0.01	0.03
	DUP20	1.29	1.21	1.17	1.17	1.13
	SD	0.02	0.01	0.01	0.02	0.01
	RAN10	2.08	1.54	1.58	1.39	1.49
	SD	0.00	0.00	0.04	0.03	0.05
	RAN20	1.89	1.44	1.37	1.28	1.22
	SD	0.01	0.01	0.01	0.02	0.02
ACRBT	NODUP	4.59	3.69	3.54	3.39	3.14
	SD	0.05	0.10	0.08	0.08	0.00
	DUP10	1.13	1.08	1.06	1.04	1.02
	SD	0.01	0.00	0.02	0.00	0.01
	DUP20	0.98	0.91	0.89	0.88	0.86
	SD	0.00	0.00	0.00	0.00	0.00
	RAN10	1.74	1.54	1.40	1.37	1.26
	SD	0.00	0.00	0.01	0.16	0.00
	RAN20	1.44	1.10	1.05	1.00	0.98
	SD	0.11	0.01	0.02	0.01	0.01
SACRBT	NODUP	4.76	3.96	3.64	3.56	3.46
	SD	0.06	0.23	0.25	0.16	0.11
	DUP10	1.16	0.98	0.95	0.94	0.91
	SD	0.01	0.03	0.00	0.01	0.00
	DUP20	0.96	0.86	0.85	0.84	0.83
	SD	0.00	0.00	0.00	0.00	0.00
	RAN10	1.72	1.41	1.50	1.28	1.52
	SD	0.01	0.33	0.16	0.00	0.00
	RAN20	1.15	1.00	0.98	0.97	0.97
	SD	0.00	0.00	0.00	0.00	0.03

Table 5–4: Search time (in μsec) for CST and BSLPT structures on NODUP, DUP, and RAN data sets

Schemes		Paix	Pb	MaeWest	Aads	MaeEast
CST	NODUP	9.91	8.68	8.45	8.23	7.78
	SD	0.38	0.06	0.07	0.09	0.11
	DUP10	1.23	1.13	1.11	1.09	1.04
	SD	0.00	0.00	0.00	0.00	0.01
	DUP20	0.76	0.71	0.69	0.68	0.66
	SD	0.00	0.00	0.00	0.00	0.00
	RAN10	1.69	1.59	1.56	1.53	1.50
	SD	0.00	0.00	0.00	0.00	0.01
	RAN20	1.24	1.18	1.18	1.16	1.12
	SD	0.00	0.00	0.00	0.01	0.00
BSLPT	NODUP	88.20	77.70	75.16	73.10	66.14
	SD	3.75	0.15	0.35	0.30	0.32
	DUP10	9.36	8.28	8.13	7.90	7.13
	SD	0.36	0.01	0.28	0.18	0.03
	DUP20	5.01	4.38	4.26	4.15	3.81
	SD	0.10	0.07	0.01	0.01	0.01
	RAN10	9.73	8.50	8.22	8.21	7.43
	SD	0.46	0.03	0.01	0.51	0.02
	RAN20	5.39	4.79	4.72	4.60	4.34
	SD	0.02	0.01	0.01	0.01	0.01

Table 5–5: Search time (in μsec) for CRBT, ACRBT, and SACRBT structures on trace sequences

Schemes		Paix	Pb	MaeWest	Aads	MaeEast
CRBT	DEC-PKT-1	3.91	3.44	3.03	2.91	2.95
	SD	0.16	0.13	0.17	0.16	0.13
	DEC-PKT-2	3.87	3.43	3.05	2.85	3.07
	SD	0.25	0.20	0.21	0.17	0.21
	DEC-PKT-3	4.17	3.57	3.26	3.05	3.25
	SD	0.27	0.14	0.18	0.25	0.11
	DEC-PKT-4	3.99	3.49	3.13	2.92	3.18
	SD	0.20	0.13	0.16	0.14	0.11
	LBL-PKT-4	4.03	3.54	3.16	3.01	3.22
	SD	0.19	0.14	0.17	0.15	0.19
	LBL-PKT-5	4.03	3.46	3.21	3.05	3.19
	SD	0.15	0.09	0.10	0.10	0.08
ACRBT	DEC-PKT-1	3.53	3.08	2.87	2.61	2.77
	SD	0.17	0.16	0.19	0.15	0.12
	DEC-PKT-2	3.61	3.03	2.90	2.58	2.85
	SD	0.22	0.19	0.21	0.17	0.19
	DEC-PKT-3	3.96	3.21	2.99	2.74	2.93
	SD	0.24	0.16	0.15	0.20	0.08
	DEC-PKT-4	3.73	3.14	2.94	2.65	2.75
	SD	0.21	0.14	0.14	0.14	0.09
	LBL-PKT-4	3.65	3.19	2.97	2.68	2.97
	SD	0.16	0.17	0.18	0.15	0.16
	LBL-PKT-5	3.71	3.09	2.93	2.71	2.84
	SD	0.16	0.09	0.11	0.11	0.11
SACRBT	DEC-PKT-1	2.81	2.47	2.16	2.07	2.05
	SD	0.15	0.08	0.16	0.09	0.11
	DEC-PKT-2	2.83	2.51	2.14	2.06	2.09
	SD	0.22	0.13	0.14	0.11	0.15
	DEC-PKT-3	2.97	2.65	2.32	2.10	2.19
	SD	0.24	0.15	0.10	0.13	0.10
	DEC-PKT-4	2.82	2.58	2.20	2.09	2.21
	SD	0.16	0.11	0.12	0.10	0.07
	LBL-PKT-4	2.91	2.56	2.22	2.14	2.13
	SD	0.15	0.10	0.17	0.13	0.14
	LBL-PKT-5	2.81	2.56	2.17	2.06	2.19
	SD	0.12	0.08	0.08	0.10	0.05

Table 5–6: Search time (in μsec) for CST and BSLPT structures on trace sequences

Schemes		Paix	Pb	MaeWest	Aads	MaeEast
CST	DEC-PKT-1	1.35	1.41	1.46	1.43	1.42
	SD	0.07	0.08	0.08	0.08	0.08
	DEC-PKT-2	1.35	1.41	1.39	1.46	1.41
	SD	0.09	0.09	0.09	0.09	0.11
	DEC-PKT-3	1.45	1.54	1.53	1.52	1.51
	SD	0.04	0.05	0.06	0.06	0.06
	DEC-PKT-4	1.43	1.48	1.46	1.51	1.55
	SD	0.18	0.07	0.08	0.07	0.07
	LBL-PKT-4	1.21	1.27	1.26	1.28	1.30
	SD	0.09	0.10	0.11	0.10	0.11
	LBL-PKT-5	1.53	1.25	1.29	1.23	1.24
	SD	0.09	0.07	0.05	0.05	0.07
BSLPT	DEC-PKT-1	5.27	5.03	4.95	4.74	4.98
	SD	0.45	0.40	0.45	0.48	0.45
	DEC-PKT-2	5.19	4.97	4.79	4.58	4.75
	SD	0.62	0.67	0.59	0.59	0.57
	DEC-PKT-3	5.94	5.48	5.36	5.18	5.45
	SD	0.39	0.41	0.39	0.38	0.45
	DEC-PKT-4	5.49	5.36	5.13	4.90	5.28
	SD	0.40	0.45	0.37	0.36	0.45
	LBL-PKT-4	4.65	3.77	4.01	3.68	3.94
	SD	0.33	0.39	0.41	0.43	0.39
	LBL-PKT-5	3.95	3.74	3.89	3.53	3.76
	SD	0.38	0.44	0.45	0.38	0.44

actually do so. For example, searching the database Paix takes about four times as much time per search using the data set NODUP (non-bursty) as it does using the data set DUP10! This is because of the cache effect–the first search for destination d in CRBT potentially causes h cache misses, where h is the height of the BIT. Subsequent searches have (almost) no cache misses because the BIT and back-end nodes needed for the search are still in cache! So, computer caches automatically result in improved performance for bursty access patterns! The performance improvement between RAN10 and NODUP isn't as much (a factor of 3 rather than 4), because of increased cache conflicts caused by intermediate searches for different destinations. Still, it appears that most of the accessed nodes remain in cache long enough to service all repeated searches for the same destination in RAN10. The reduction in average search-time in going from DUP10 to DUP20 and from RAN10 to RAN20 isn't quite as dramatic—a mere 9% to 17%. The use of supernodes in the SACRBT generally reduces the search time by a small amount. Similar reductions in search times are seen when the trace data sets are used together with the CRBT, ACRBT, and SACRBT structures. In the case of the trace data sets, the SACRBT provides a much greater reduction in search time over the ACRBT than was the case with the NODUP, DUP, and RAN data sets.

The data structures, CST and BSLPT, that are designed to perform better, through a reduction in work (number of comparisons and number of pointers followed) on bursty access patterns show a much greater performance improvement when the access pattern is bursty than when it is not. For example, on the database Paix, CST and BSLPT took about 8 to 9 times as much time on the NODUP data set as they did on the DUP10 data set. A further 26% to 46% reduction in average search-time was evidenced when going from DUP10 to DUP20 and from RAN10 to RAN20. The average search times for the trace data sets was comparable to that for the DUP and RAN data sets. On the database Paix, for example, CST took about 7 times as much

time on the NODUP data set as on the DEC-PKT-1 data set. This ratio was about 17 for BSLPT!

Although a worst-case and average search in an ACRBT requires more comparisons than does a similar search in a CRBT, our experiments show that the average search time in an ACRBT is less than that in a CRBT. In fact, in our tests, the ACRBT was generally faster than the CRBT and in most of our tests, the search time in an ACRBT was less than that in a CRBT by between 16% and 29%. Although the CST and ACRBT were competitive on the DUP10 and RAN10 data sets, the CST took about 50% more time than did ACRBT on the NODUP test, and took about 20% less time than taken by the ACRBT on the DUP20 test. On the trace data, the CST took about one-third to one-half the time taken by the CRBT, ACRBT, and SACRBT structures. The BSLPT was no match for the other structures on any of the data sets!

Insert Time

To measure the average insert time for each of the data structures, we first obtained a random permutation of the prefixes in each of the databases. Next, the first 75% of the prefixes in this random permutation were inserted into an initially empty data structure. The time to insert the remaining 25% of the prefixes was measured and averaged. This timing experiment was repeated 10 times. Table 5–7 shows the average of the 10 average insert times as well as the standard deviation (SD) in the measured average time. Figure 5–8 histograms the average times of Table 5–7.

Our insert experiments show that the BSLPT structure is the clear loser for this operation. The average insert in a BSLPT takes about twice as much time as it does in any of the three tree-based front-end structures. The insert operation is faster in the CST than in the ACRBT, and inserts are faster in the ACRBT than in the CRBT. An insert in the Paix database, for example, takes 18% less time when an ACRBT is used than when a CRBT is used. The time for the CST is 14% less than

Table 5–7: Average time to insert a prefix (in μsec)

Schemes		Paix	Pb	MaeWest	Aads	MaeEast
CRBT	AVG	35.12	32.86	31.51	31.62	29.94
	SD	1.35	0.00	0.54	1.03	0.00
ACRBT	AVG	28.65	26.17	25.26	24.82	23.42
	SD	0.39	0.35	0.67	0.62	0.85
SACRBT	AVG	33.58	30.82	29.69	29.70	28.00
	SD	0.36	0.71	1.82	0.83	0.99
CST	ADV	24.60	23.68	22.39	22.60	21.66
	SD	0.14	0.35	0.54	1.40	0.85
BSLPT	ADV	64.10	58.47	67.84	67.82	51.60
	SD	0.65	1.52	0.73	0.46	0.85

Table 5–8: Average time to delete a prefix (in μsec)

Schemes		Paix	Pb	MaeWest	Aads	MaeEast
CRBT	AVG	38.89	36.03	34.90	35.31	33.81
	SD	1.30	0.71	0.54	0.83	0.74
ACRBT	AVG	30.98	28.55	27.34	27.33	25.71
	SD	0.39	0.47	0.00	0.77	0.90
SACRBT	AVG	33.95	31.27	29.82	30.88	28.70
	SD	0.43	0.79	1.43	0.83	0.85
CST	AVG	32.88	31.72	30.86	31.18	29.41
	SD	0.31	0.00	0.62	1.62	0.85
BSLPT	AVG	74.01	65.49	63.68	63.10	57.76
	SD	0.77	0.89	0.96	0.99	1.11

for the ACRBT. The insert time for the ACRBT is about 15% less than that for the supernode implementation SACRBT.

Delete Time

To measure the average delete time, we started with the data structure for each database and removed the last 25% of the prefixes in the database. These prefixes were determined using the permutation generated for the insert time test. Once again, the test was run 10 times and the average of the averages computed. Table 5–8 shows the average time to delete a prefix as well as the standard deviation in this time over the 10 test runs. Figure 5–9 histograms the average times of Table 5–8.

As was the case for the search and insert operations, for the delete operation too, the BSLPT structure is the clear loser. A delete in the BSLPT structure takes more than twice the time it takes in any of the tree-based front-end structures. Among the tree-based front-end structures, the delete time is the least for the ACRBT structure. For example, on the Paix database, a delete using CST takes about 6% more time than when an ACRBT is used; a delete using CRBT takes 25% more time than when an ACRBT is used. The use of supernodes increases the average delete time by 10%.

5.5 Summary

We have modified the CRBT structure proposed in Chapter 4 by changing the structure of the front end. Although, the modified structure, called the ACRBT, requires more memory than does the CRBT, experiments conducted by us indicate that the ACRBT is generally faster than the CRBT for the search, insert, and delete operations.

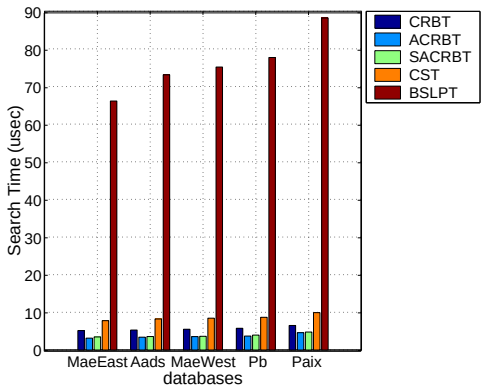
By replacing the front-end red-black trees of the CRBT structure in Chapter 4 with top-down splay trees and the back-end red-black trees with bottom-up splay trees, we arrive at the CST data structure that takes less memory than taken by either the CRBT and ACRBT. For acutely bursty access patterns (e.g., DUP20 and the trace data sets), searches in the CST structure are much faster than in the ACRBT structure. When the access pattern is not bursty (e.g., NODUP), searches in a CST take about twice as much time as they do in an ACRBT. For the insert operation, CSTs are slightly faster than ACRBTs, but for the delete operation, the reverse is the case.

We can add the CRBT back-end to the BSL structure of [25] to obtain a biased skip list structure that permits the insertion and deletion of prefixes in $O(\log n)$ expected time. However, our experiments indicate that the resulting data structure is highly non-competitive with CRBTs, ACRBTs, and CSTs for the search, insert, and delete operations. The BSLPT is always significantly inferior to the CST on

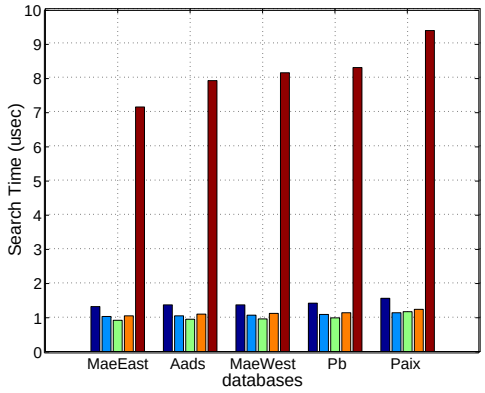
all operations. Therefore, biased skip lists cannot be recommended for even highly bursty access applications.

Of the structures we tested, the ACRBT is recommended for non-bursty to moderately bursty applications, the CST is recommended for highly bursty applications. Finally, we observe that the add-on data structures, such as a cache list of most-recent destination addresses, suggested in [25] for further improvement in search performance may also be used in conjunction with the data structures of this chapter.

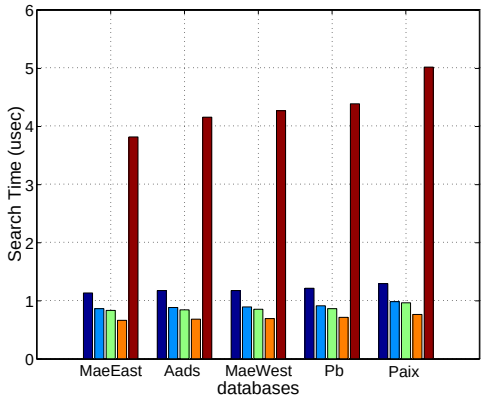
Finally, by using supernode binary trees [40] in place of traditional one-element-per-node binary trees, we can improve the search performance of our data structures. Although we did this only for the front-end of the ACRBT structure, we expect the results to carry over for the remaining structures.



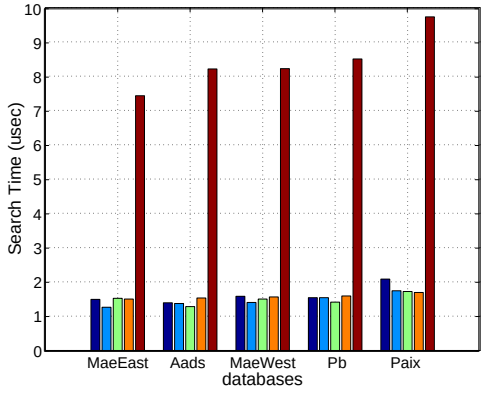
(a) NODUP



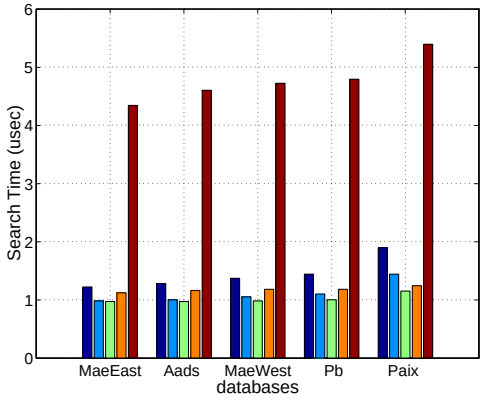
(b) DUP10



(c) DUP20

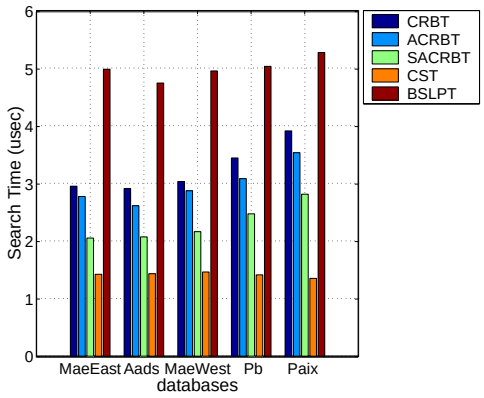


(d) RAN10

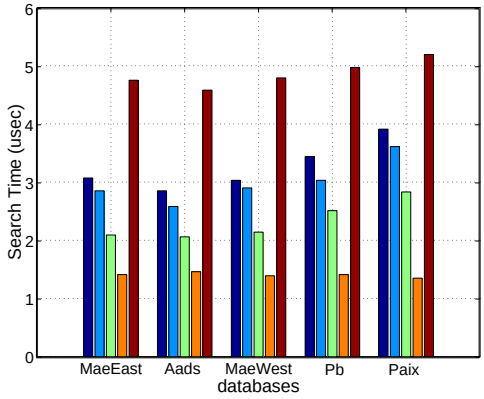


(e) RAN20

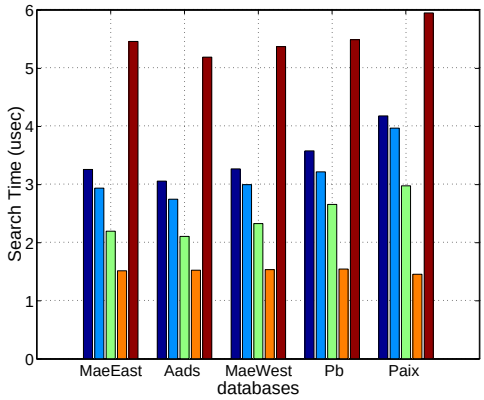
Figure 5–6: Average search time for NODUP, DUP, and RAN data sets



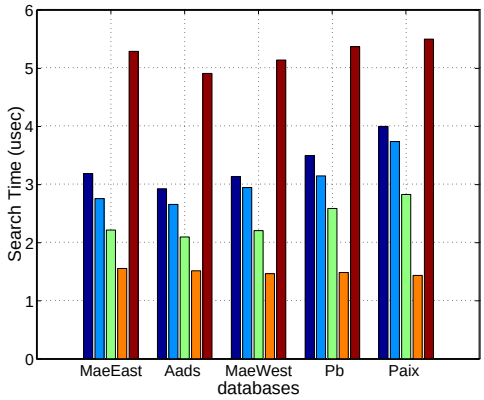
(a) DEC-PKT-1



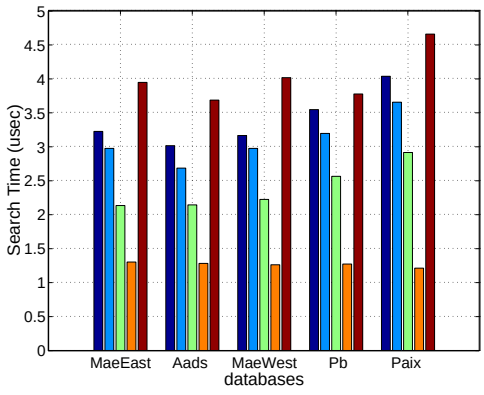
(b) DEC-PKT-2



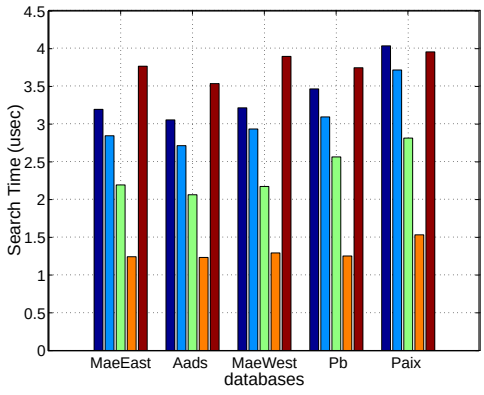
(c) DEC-PKT-3



(d) DEC-PKT-4



(e) LBL-PKT-4



(f) LBL-PKT-5

Figure 5–7: Average search time for trace sequences

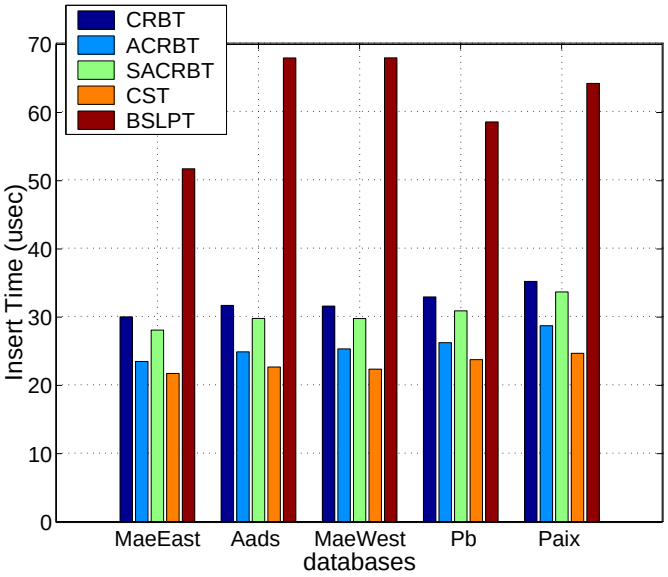


Figure 5–8: Average time to insert a prefix

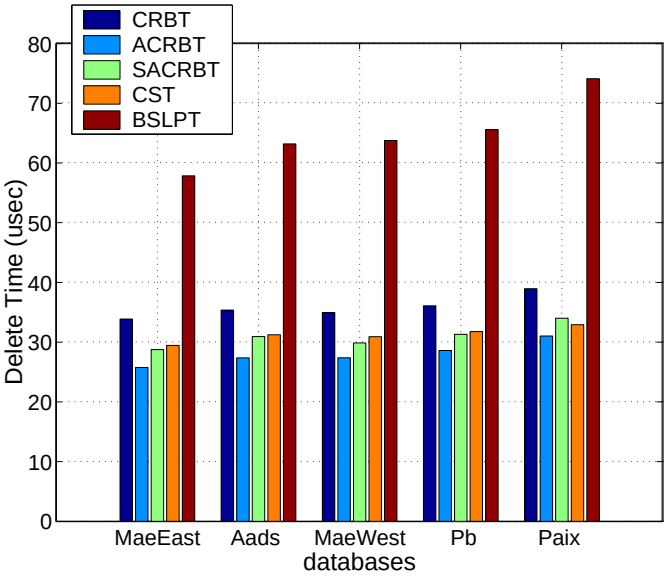


Figure 5–9: Average time to delete a prefix

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

In this dissertation, we have developed several efficient algorithms for IP lookup in Internet routers. Through analyzing and experimenting with them, we have seen that the proposed algorithms work well compared with other known algorithms. We summarize our contributions in Section 6.1 and discuss directions for future work in Section 6.2.

6.1 Conclusions

We have proposed data structures for one-dimensional packet classification in this dissertation. The main contributions of the dissertation are discussed as follows.

Multibit Tries: In Chapter 2, we improved on the dynamic programming algorithms of [82], which determine the strides of optimal multibit fixed-stride and variable-stride tries, by providing alternative dynamic programming formulations for both fixed- and variable-stride tries. While the asymptotic complexities of our algorithms are the same as those of the corresponding algorithms of [82], experiments using real IPv4 routing table data indicate that our algorithms run considerably faster. An added feature of our variable-stride trie algorithm is the ability to insert and delete prefixes taking a fraction of the time needed to construct an optimal variable-stride trie from scratch.

Binary Search on Prefix Length: Chapter 3 considers the collection of hash tables (CHT) organization, which was proposed in [87] for an IP router table. Srinivasan et al. [80] have proposed the use of controlled prefix-expansion to reduce the number of distinct prefix-lengths (also equal to the number of hash tables in the CHT). The complexity of their algorithm is $O(nW^2)$, where n is the number of prefixes, and W is the length of the longest prefix. We have developed an algorithm that

minimizes storage requirement but takes $O(nW^3 + kW^4)$ time, where k is the desired number of distinct lengths. Also, we have proposed improvements to the heuristic of [80].

$O(\log n)$ **Dynamic Router-Table:** In Chapter 4, we proposed a data structure in which prefix matching, prefix insertion, and deletion can each be done in $O(\log n)$ time, where n is the number of prefixes in the router table. For W-bit destination addresses, the use of binary tries enables us to determine the longest matching prefix as well as to insert and delete a prefix in $O(W)$ time, independent of n. Since $n << 2^W$ in real router tables, it is desirable to develop a data structure that permits these three operations in $O(\log n)$. Although the proposed data structure is slower than optimized variable-stride tries for longest prefix matching, the proposed data structure is considerably faster for the insert and delete operations.

Dynamic Lookup for Bursty Access Patterns: We have developed data structures for dynamic router-tables for bursty access-patterns in Chapter 5. In this chapter, we first formulated a variant, ACRBT, of the CRBT data structure proposed in Chapter 4 for dynamic router-tables. By replacing the red-black trees used in the ACRBT with splay trees, we obtain the CST structure in which search, insert, and delete take $O(\log n)$ amortized time per operation, where n is the number of prefixes in the router table. By replacing the front end of the CST with biased skip lists, we obtain the BSLPT structure in which search, insert, and delete take $O(\log n)$ expected time. The CST and BSLPT structures are designed so as to perform much better when the access pattern is bursty than when it is not. For extremely bursty access patterns, the CST structure is best. Otherwise, the ACRBT is recommended. A supernode implementation of the ACRBT usually has better search performance than does the traditional one-element-per-node implementation.

Table 6–1 summarizes the performance characteristics of various data structures for the longest matching-prefix problem.

Table 6–1: Performance of data structures for longest matching-prefix

Data Structure	Search	Update	Memory Usage
Linear List	$O(n)$	$O(n)$	$O(n)$
End-point Array	$O(\log n)$	$O(n)$	$O(n)$
Sets of Equal-Length Prefixes	$O(\alpha + \log W)$ expected	$O(\alpha \sqrt[s]{n} W \log W)$ expected	$O(n \log W)$
1-bit tries	$O(W)$	$O(W)$	$O(nW)$
s-bit Tries	$O(W/s)$	-	$O(2^s nW/s)$
CRBT	$O(\log n)$	$O(\log n)$	$O(n)$
ACRBT	$O(\log n)$	$O(\log n)$	$O(n)$
BSLPT	$O(\log n)$ expected	$O(\log n)$ expected	$O(n)$
CST	$O(\log n)$ amortized	$O(\log n)$ amortized	$O(n)$

6.2 Future Work

Although much effort has been devoted to the IP router tables, this area has not yet reached complete maturity. We discuss some issues for future research directions as follows.

IPv6 Extensions: Some router table structures that give a good lookup performance for IPv4 prefixes may face scalability problems when applied to IPv6. For instance, a large-stride multibit trie to accomodate a hundred thousand IPv6 prefixes may exponentially increase the amount of memory; whereas, with small-stride nodes, it may take high execution time for worst-case lookup operations. Even with moderate strides, the multibit trie may suffer from either the tremendous amount of memory or the number of cache misses, or both.

Future Routers: Gigabit routers can typically provide total line capacity of up to tens of gigabits per second, and support port interface speeds up to OC-48 (2.5 Gbps). Current terabit routers are designed with the aggregate line capacity to handle thousands of gigabits per second and to provide high-scalable performance and high port density [1]. These routers can support port interface speeds as high as OC-192 (10 Gbps) and beyond. To support lookups in future routers, order of magnitude faster lookup times than those of current routers are required. We may

need to use on-chip memory and/or SRAM memory to allow such a speed. Thus, some scalable and efficient data structures should be developed to fit more than hundered of thousands prefixes into on-chip SRAM.

Switching versus Routing: The basic difference between switching and routing is that switching uses *indexing* to determine the next hop for a packet in the address table whereas routing uses *searching* (or *lookup*). Since indexing is $O(1)$ operation, it may be much faster than any search teachnique. Because of this, many people started thinking about replacing routers with switches wherever possible and vendors introduced several products into the market. There are two very different approaches that combine layer 2 switching and layer 3 routing. The first approach (e.g., IP switching [56] and multi-protocol over ATM (MPOA) [2]) aims at improving routing performance by separating the transmission of network control information from the normal data traffic. Control traffic passes through the routers to initiate a call or connection setup, whereas normal data traffic can be switched through the already established path. The other approach (e.g., tag switching [64] and MPLS [68]) addresses WAN (wide area network) route scalability issues [58]. Routing decisions are performed once at the entry point to the WAN and the remaining forwarding decisions within the WAN switch infrastructure are based on switching techniques.

Packet Classification: Despite the vast amount of attention to the packet classification [4, 5, 11, 28, 34, 35, 43, 63, 81, 83, 88], there is a scaling problem even for medium size databases when rules contain more than 2 fields. Both QoS and security guarantees require a finer discrimination of packets based on fields other than the destination address. Classifiers have evolved from firewalls [15] that filter out unwanted packets at the edge routers of networks using rule tables of up to 500 rules. The advent of DiffServ [7] and policing applications has given an anticipation that classifiers could support a few hundred thousand rules at edge routers [49]. However, while the current best classification algorithms [5, 34, 35] work well for

databases of up to (say) 1000 rules, they require very large amounts of memory for larger databases.

We hope the IP lookup algorithms described in this dissertation will provide a brigehead for exploring these areas further.

REFERENCES

[1] D. Allen, Terabit routing: Simplifying the core, *Telecommunications Online*, http://www.telecommagazine.com/, May 1999. Site last visited June 2003.

[2] ATM Forum, Multi-protocol over ATM specification, version 1.1, af-mpoa-0114.000, http://www-comm.itsi.disa.mil/atmf/mpoa.html, May 1999.

[3] J. Aweya, IP router architectures: An overview, *Journal of Systems Architecture*, Vol. 46, 2000, 483-511.

[4] F. Baboescu, S. Singh, and G. Varghese, Packet classification for core routers: Is there an alternative to CAMs?, *Proceedings of IEEE INFOCOM 03*, April 2003.

[5] F. Baboescu and G. Varghese, Scalable packet classification, *Proceedings of ACM SIGCOMM 01*, September 2001.

[6] F. Baker, Requirements for IP version 4 routers, RFC 1812, IETF, http://www.ietf.org/rfc.html, June 1995.

[7] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, An architecture for differentiated services, RFC 2475, IETF, http://www.ietf.org/rfc.html, December 1998.

[8] R. Braden, D. Borman, and C. Partridge, Computing the Internet checksum, RFC 1071, IETF, http://www.ietf.org/rfc.html, September 1988.

[9] B. Braden, L. Zhang, S. Berson, S. Herxog, and S. Jamin, Resource reservation protocol (RSVP) – version 1 functional specification, RFC 2205, IETF, http://www.ietf.org/rfc.html, September 1997.

[10] A. Bremler-Barr, Y. Afek, and S. Har-Peled, Routing with a clue, *Proceedings of ACM SIGCOMM 99*, September 1999, 203-214.

[11] M. Buddhikot, S. Suri, and M. Waldvogel, Space decomposition techniques for fast layer-4 switching, *Proceedings of ACM SIGCOMM 01*, September 2001.

[12] V. Cerf, Computer networking: Global infrastructure for the 21st century, http://www.cs.washington.edu/homes/lazowska/cra/networks.html, 1995. Site last visited June 2003.

[13] G. Chandranmenon and G. Varghese, Trading packet headers for packet processing, *IEEE Transactions on Networking*, April 1996.

[14] T. Chaney, A. Fingerhut, M. Flucke, and J. Turner, Design of a gigabit ATM switch, *Proceedings of IEEE INFOCOM 97*, March 1997.

[15] W. Cheswick, S. Bellovin, and A. Rubin, Firewalls and Internet security: Repelling the wily hacker, Addison-Wesley Professional, 2nd ed., 2002, 464 pages.

[16] G. Cheung and S. McCanne, Optimal routing table design for IP address lookups under memory constraints, *Proceedings of IEEE INFOCOM 99*, March 1999.

[17] T. Chiueh and P. Pradhan, High-performance IP routing table lookup using CPU caching, *Proceedings of IEEE INFOCOM 99*, March 1999.

[18] K. Claffy, H. Braun, and G. Polyzos, A parameterizable methodology for Internet traffic flow profiling, *IEEE Journal of Selected Areas in Communications*, 1995.

[19] D. Comer, Computer networks and Internets with Internet applications, 3rd ed., Prentice Hall, NJ, 2001, 683 pages.

[20] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, Router plugins: A software architecture for next generation routers, *Proceedings of ACM SIGCOMM 98*, August 1998, 191-202.

[21] S. Deering and R. Hinden, Internet protocol, version 6 (IPv6) specification, RFC 2460, IETF, http://www.ietf.org/rfc.html, December 1998.

[22] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, Small forwarding tables for fast routing lookups, *Proceedings of ACM SIGCOMM 97*, October 1997, 3-14.

[23] W. Doeringer, G. Karjoth, and M. Nassehi, Routing on longest-matching prefixes, *IEEE/ACM Transactions on Networking*, 4, 1, 1996, 86-97.

[24] R. Draves, C. King, V. Srinivasan, and B. Zill, Constructing optimal IP routing tables, *Proceedings of IEEE INFOCOM 99*, March 1999.

[25] F. Ergun, S. Mittra, S. Sahinalp, J. Sharp, and R. Sinha, A dynamic lookup scheme for bursty access patterns, *Proceedings of IEEE INFOCOM 01*, 2001.

[26] F. Ergun, S. Sahinalp, J. Sharp, and R. Sinha, Biased skip lists for highly skewed access patterns, *3rd Workshop on Algorithm Engineering and Experiments*, 2001.

[27] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei, Protocol independent multicast-sparse mode (PIM-SM): Protocol specification, RFC 2362, IETF, http://www.ietf.org/rfc.html, June 1998.

[28] A. Feldman and S. Muthukrishnan, Tradeoffs for packet classification, *Proceedings of IEEE INFOCOM 2000*, 2000.

[29] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, Hypertext transfer protocol – HTTP/1.1, RFC 2616, IETF, http://www.ietf.org/rfc.html, June 1999.

[30] V. Fuller, T. Li, J. Yu, and K. Varadhan, Classless inter-domain routing (CIDR): An address assignment and aggregation strategy, RFC 1519, IETF, http://www.ietf.org/rfc.html, September 1993.

[31] M. Gray, Internet growth summary, http://www.mit.edu/people/mkgray/net/internet-growth-sumary.html, 1996. Site last visited June 2003.

[32] P. Gupta, S. Lin, and N. McKeown, Routing lookups in hardware at memory access speeds, *Proceedings of IEEE INFOCOM 98*, April 1998.

[33] P. Gupta and N. McKeown, Dynamic algorithms with worst-case performance for packet classification, *IFIP Networking*, 2000.

[34] P. Gupta and N. McKeown, Packet classification on multiple fields, *Proceedings of ACM SIGCOMM 99*, September 1999.

[35] P. Gupta and N. McKeown, Packet classification using hierarchical intelligent cuttings, *Hot Interconnects VII*, August 1999.

[36] P. Gupta, B. Prabhakar, and S. Boyd, Near-optimal routing lookups with bounded worst case performance, *Proceedings of IEEE INFOCOM 2000*, 2000.

[37] C. Hedrick, Routing information protocol, RFC 1058, IETF, http://www.ietf.org/rfc.html, June 1988.

[38] R. Hinden, Applicability statement for the implementation of classless inter-domain routing (CIDR), RFC 1517, IETF, http://www.ietf.org/rfc.html, September 1993.

[39] E. Horowitz, S. Sahni, and D. Mehta, Fundamentals of data structures in C++, W.H. Freeman, NY, 1995, 653 pages.

[40] H. Jung and S. Sahni, Supernode binary search trees, *International Journal on Foundations of Computer Science*. To appear.

[41] K. Kim and S. Sahni, IP lookup by binary search on prefix length, *Journal of Interconnection Networks*, Vol. 3, No. 3 & 4, 2002, 105-128.

[42] J. Klensin, Simple mail transfer protocol, RFC 2821, IETF, http://www.ietf.org/rfc.html, April 2001.

[43] T.V. Lakshman and D. Stiliadis, High-speed policy-based packet forwarding using efficient multi-dimensional range matching, *Proceedings of ACM SIGCOMM 98*, August 1998.

[44] B. Lampson, V. Srinivasan, and G. Varghese, IP lookup using multi-way and multicolumn search, *Proceedings of IEEE INFOCOM 98*, April 1998.

[45] B. Leiner, V. Cerf, D. Clark, R. Kahn, L. Kleinrock, D. Lynch, J. Postel, L. Roberts, and S. Wolff, A brief history of the Internet, http://www.isoc.org/internet/history/brief.shtml, August 2000. Site last visited June 2003.

[46] S. Lin and N. McKeown, A simulation study of IP switching, *Proceedings of IEEE INFOCOM 2000*, 2000.

[47] K. Lougheed and Y. Rekhter, A border gateway protocol (BGP), RFC 1163, IETF, http://www.ietf.org/rfc.html, June 1990.

[48] T. Mallory and A. Kullberg, Incremental updating of the Internet checksum, RFC 1141, IETF, http://www.ietf.org/rfc.html, January 1990.

[49] C. Matsumoto, CAM vendors consider algorithmic alternatives, *EE Times*, http://www.eetimes.com/story/OEG20020520S0014, May 2002. Site last visited June 2003.

[50] A. McAuley and P. Francis, Fast routing table lookups using CAMs, *Proceedings of IEEE INFOCOM 93*, 1993, 1382-1391.

[51] N. McKeown, A fast switched backplane for a gigabit switched router, *Business Communications Review*, Vol. 27, No. 12, December 1997.

[52] N. McKeown, M. Izzard, A. Mekkittikul, B. Ellersick, and M. Horowitz, The tiny tera: A packet switch core, *IEEE Micro*, January 1997.

[53] Merit, IPMA statistics, http://nic.merit.edu/ipma, (snapshot on September 13, 2000), 2000.

[54] D. Milojicic, Trend wars: Internet technology, http://www.computer.org/concurrency/articles/trendwars_200_1.htm, 2000.

[55] J. Moy, OSPF version 2, RFC 1247, IETF, http://www.ietf.org/rfc.html, July 1991.

[56] P. Newman, G. Minshall, and L. Huston, IP switching and gigabit routers, *IEEE Communications Magazine*, January 1997.

[57] S. Nilsson and G. Karlsson, Fast address look-up for Internet routers, *IEEE Broadband Communications*, 1998.

[58] D. Passmore and J. Bransky, Route once switch many, http://www.burtongroup.com/Public/WhitePapers/Route_Once_wp.html, July 1997. Site last visited June 2003.

[59] J. Postel, Internet protocol, RFC 791, IETF, http://www.ietf.org/rfc.html, September 1981.

[60] J. Postel, Transmission control protocol, RFC 793, IETF, http://www.ietf.org/rfc.html, September 1981.

[61] J. Postel, User datagram protocol, RFC 768, IETF, http://www.ietf.org/rfc.html, August 1980.

[62] W. Pugh, Skip lists: a probailistic alternative to balanced trees, *Comm. of the ACM*, 33, 6, 1990.

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

[64] Y. Rekhter, B. Davie, D. Katz, E. Rosen, and G. Swallow, Cisco systems' tag switching architecture overview, RFC 2105, IETF, http://www.ietf.org/rfc.html, February 1997.

[65] Y. Rekhter and T. Li, A border gateway protocol 4 (BGP-4), RFC 1771, IETF, http://www.ietf.org/rfc.html, March 1995.

[66] Y. Rekhter and T. Li, An architecture for IP address allocation with CIDR, RFC 1518, IETF, http://www.ietf.org/rfc.html, September 1993.

[67] J. Reynolds and J. Postel, Assigned numbers, RFC 1700, IETF, http://www.ietf.org/rfc.html, October 1994.

[68] E. Rosen, A. Viswanathan, and R. Callon, Multiprotocol label switching architecture, RFC 3031, IETF, http://www.ietf.org/rfc.html, January 2001.

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

[70] S. Sahni, *Data structures, algorithms, and applications in Java*, McGraw Hill, NY, 2000.

[71] S. Sahni and K. Kim, Efficient construction of fixed-stride multibit tries for IP lookup, *Proceedings of 8th IEEE Workshop on Future Trends of Distributed Computing Systems*, 2001, 178-184.

[72] S. Sahni and K. Kim, Efficient construction of variable-stride multibit tries for IP lookup, *Proceedings of IEEE Symposium on Applications and the Internet (SAINT)*, 2002, 220-227.

[73] S. Sahni and K. Kim, Efficient construction of multibit tries for IP lookup, *IEEE/ACM Transactions on Networking*. To appear.

[74] S. Sahni and K. Kim, Efficient dynamic lookup for bursty access patterns. Submitted.

[75] S. Sahni and K. Kim, $O(\log n)$ dynamic packet routing, *Proceedings of IEEE Symposium on Computers and Communications*, 2002, 443-448.

[76] S. Sahni, K. Kim, and H. Lu, Data structures for one-dimensional packet classification using most-specific-rule matching, *Proceedings of International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN)*, 2002, 3-14.

[77] S. Sahni, K. Kim, and H. Lu, Data structures for one-dimenstional packet classification using most-specific-rule matching, *International Journal of Foundations of Computer Science*. To appear.

[78] K. Sklower, A tree-based routing table for Berkeley Unix, Technical Report, University of California, Berkeley, 1993.

[79] D. Sleator and R. Tarjan, Self-adjusting binary search trees, *Journal of the ACM*, 32, 1985.

[80] V. Srinivasan, Fast and efficient Internet lookups, *CS Ph.D Dissertation*, Washington University, August 1999.

[81] V. Srinivasan, S. Suri, and G. Varghese, Packet classification using tuple space search, *Proceedings of ACM SIGCOMM 99*, September 1999.

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

[83] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, Faster and scalable layer four switching, *Proceedings of ACM SIGCOMM 98*, August 1998.

[84] S. Suri, G. Varghese, and P. Warkhede, Multiway range trees: Scalable IP lookup with fast updates, *Proceedings of GLOBECOM 01*, 2001.

[85] A. Tammel, How to survive as an ISP, *Networld Interop*, 1997.

[86] D. Waitzman, C. Partridge, and S. Deering, K. Lougheed and Y. Rekhter, Distance vector multicast routing protocol, RFC 1075, IETF, http://www.ietf.org/rfc.html, November 1988.

[87] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, Scalable high speed IP routing lookups, *Proceedings of ACM SIGCOMM 97*, October 1997, 25-36.

[88] T. Woo, A modular approach to packet classification: Algorithms and results, *Proceedings of IEEE INFOCOM 2000*, 2000.

BIOGRAPHICAL SKETCH

Kun Suk Kim received the B.E. and M.E. degrees in Computer Engineering from the Kyungpook National University, Korea, in 1992 and 1994, respectively. He was a research staff member at the Electronics and Telecommunications Research Institute, Korea, for five and a half years from 1994. Since 1999, he has been taking the Ph.D. degree course in the Computer and Information Science and Engineering department at the University of Florida, Gainesville, FL. He is interested in network-based systems, telecommunications, and computer networks. He is currently working on algorithms for networks.