Fast Prefix Matching of Bounded Strings

Adam L. Buchsbaum* Glenn S. Fowler* Balachander Krishnamurthy*
Kiem-Phong Vo* Jia Wang*

Abstract

Longest Prefix Matching (LPM) is the problem of finding which string from a given set is the longest prefix of another, given string. LPM is a core problem in many applications, including IP routing, network data clustering, and telephone network management. These applications typically require very fast matching of bounded strings, i.e., strings that are short and based on small alphabets. We note a simple correspondence between bounded strings and natural numbers that maps prefixes to nested intervals so that computing the longest prefix matching a string is equivalent to finding the shortest interval containing its corresponding integer value. We then present *retries*, a fast and compact data structure for LPM on general alphabets. Performance results show that retries outperform previously published data structures for IP look-up. By extending LPM to general alphabets, retries admit new applications that could not exploit prior LPM solutions designed for IP look-ups.

1 Introduction

Longest Prefix Matching (LPM) is the problem of determining from a set of strings the longest one that is a prefix of some other, given string. LPM is at the heart of many important applications. Internet Protocol (IP) routers [13] routinely forward packets by computing from their routing tables the longest bit string that forms a prefix of the destination address of each packet. Krishnamurthy and Wang [19] describe a method to cluster Web clients by identifying a set of IP addresses that with high probability are under common administrative control and topologically close together. Such clustering information has applications ranging from network design and management to providing on-line quality of service differentiation based on the origin of a request. The proposed clustering approach is *network aware* in that addresses are grouped based on prefixes in snapshots of border gateway protocol (BGP) routing tables.

Telephone network management and marketing applications often classify regions in the country by area codes or combinations of area codes and the first few digits of the local phone numbers. For example, the state of New Jersey is identified by area codes such as 201, 908, and 973. In turn, Morris County in New Jersey is identified by longer telephone prefixes like 908876 and 973360. These applications typically require computing in seconds or minutes summaries of calls originating and terminating at certain locations from daily streams of telephone calls, up to hundreds of millions records at a time. This requires very fast classification of telephone numbers by finding the longest matching telephone prefixes.

Similar to other string matching problems [16, 18, 26] with practical applications [1, 5], LPM solutions must be considered in the context of the intended use to maximize performance. The LPM applications discussed above have some common characteristics.

- Look-ups overwhelmingly dominate updates of the prefix sets. A router may route millions of packets before its routing table changes. Similarly, telephone number classifications rarely change, but hundreds of millions of phone calls are made daily.
- The look-up rate is extremely demanding. IP routing and clustering typically require LPM performance of a couple hundred nanoseconds per look-up. This severely limits the number of machine instructions and memory references allowed.
- Prefixes and strings are bounded in length and based on small alphabets. For example, current IP addresses are 32-bit strings, and U.S. telephone numbers are 10-digit strings.

* AT&T Labs–Research, Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932, USA, {alb,gsf,bala,kpv,jiawang}@research.att.com.

`00100000/3`	`a`	`[32,63]`	`[32,39]`	`00100000/5`	`a`
`00101000/5`	`b`	`[40,47]`	`[40,47]`	`00101000/5`	`b`
			`[48,63]`	`00110000/4`	`a`
`11000000/2`	`c`	`[192,255]`	`[192,207]`	`11000000/4`	`c`
`11010000/4`	`d`	`[208,223]`	`[208,223]`	`11010000/4`	`d`
			`[224,255]`	`11100000/3`	`c`
(a)		(b)		(c)	

Figure 1: (a) An example prefix set, with associated values, for matching 8-bit strings; (b) corresponding nested intervals; (c) corresponding disjoint intervals and the equivalent set of disjoint prefixes.

The first two characteristics mean that certain theoretically appealing solutions based on, e.g., suffix trees [21], string prefix matching [3, 4], or dynamic string searching [12] are not applicable, as their performance would not scale. Fortunately, the third characteristic means that specialized data structures can be designed with the desired performance levels. There are many papers in the literature proposing schemes to solve the IP routing problem [7, 8, 9, 10, 11, 20, 24, 27, 28] with various tradeoffs based on memory consumption or memory hierarchies. We are not aware of any published work that generalizes to bounded strings such as telephone numbers, however.

Work on routing Internet packets [20] exploits a simple relationship between IP prefixes and nested intervals of natural numbers. We generalize this idea to a correspondence between bounded strings and natural numbers, which shows that solutions to one instance of LPM may be usable for other instances. We present *retries*, a novel, fast, and compact data structure for LPM on general alphabets. Simulation experiments based on trace data from real applications show that retries outperform other published data structures for IP routing. By extending LPM to general alphabets, retries also admit new applications that could not exploit prior LPM solutions designed for IP look-ups.

2 Prefixes and Intervals

Let A be an alphabet of finite size $\alpha = \delta + 1$. Without loss of generality, assume that A is the set of natural numbers in the range $[0, \delta]$. Otherwise, map A's elements to their ranks in any fixed, arbitrary order. Then, we can think of elements of A as digits in base α so that a string $s = s_1 s_2 \cdots s_k$ over A represents an integer $v = s_1 \alpha^{k-1} + s_2 \alpha^{k-2} + \cdots + s_k$. We denote $\iota(s) = v$ and $\sigma(v) = s$. When we work with fixed-length strings, we shall let $\sigma(v)$ have enough 0's padded on the left to gain this length. For example, when the string 1001 represents a number in base 2, $\iota(1001)$ is the decimal value 9. Conversely, in base 3 and with prescribed length 6, $\sigma(9)$ is the string 000100. Clearly, for any two strings s and t with equal lengths, $\iota(s) < \iota(t)$ if and only if s precedes t lexicographically.

2.1 Longest matching prefixes and shortest containing intervals

Let m be some fixed integer. Consider $A^{\leq m}$ and A^m, respectively the sets of strings over A with lengths $\leq m$ and lengths exactly equal to m. Let $P \subseteq A^{\leq m}$, and with each $p \in P$ let there be associated a data value; the data values need not be mutually distinct. We define an LPM instance (A, m) as the problem of finding the data value associated with the longest string in P that is a prefix of some string $s \in A^m$. P is commonly called the *prefix set*, and its elements are called *prefixes*. Following a convention in IP routing, we shall write s/k where s is a string and $k \leq \mathit{len}(s)$ to mean the prefix of s with k digits.

To show examples of results as they develop, we shall use the binary alphabet $A = \{0, 1\}$ and maximum string length $m = 8$. Figure 1(a) shows an example prefix set of four strings and associated values. For example, the first string in the set would best match the string `00100101`, yielding result `a`. On the other hand, the second string would best match `00101101`, yielding `b`.

For any string s in $A^{\leq m}$, let $s^0 = s0 \cdots 0$ and $s^\delta = s\delta \cdots \delta$ be two strings in which enough 0's and δ's are used to pad s to length m. Using the above correspondence between strings and integers, s can be associated with the closed interval of integers $[\iota(s^0), \iota(s^\delta)]$. This interval is denoted $I(s)$, and its *length* is $\iota(s^\delta) - \iota(s^0) + 1$.

Now let v be in $I(s)$, and consider the string $\sigma(v)$, 0-padded on the left to be length m. By construction, $\sigma(v)$ must agree with s^0 and s^δ up to the length of s. On the other hand, if $v < \iota(s^0)$, then $\sigma(v)$ lexicographically precedes s^0, so s cannot be a prefix of $\sigma(v)$. A similar argument applies when $v > \iota(s^\delta)$. Thus, we have:

```
while (lo <= hi)
{   for (i = 0; i < m; ++i)
        if ((lo % A[i+1]) != 0 || (lo + A[i+1] - 1) > hi)
            break;
    itvl2pfx(lo, lo + A[i] - 1);
    lo += A[i];
}
```

Figure 2: Constructing the prefixes covering an interval $[lo, hi]$.

Lemma 1 *Let s be a string in $A^{\leq m}$ and $v < \alpha^m$. Then, s is a prefix of $\sigma(v)$ if and only if v is in $I(s)$.*

For any prefix set P, we use $I(P)$ to denote the set of intervals associated with prefixes in P. Now consider two prefixes p_1 and p_2 and their corresponding intervals $I(p_1)$ and $I(p_2)$. Applying Lemma 1 to the endpoints of these intervals shows that either the intervals are completely disjoint or one is contained in the other. Furthermore, $I(p_1)$ contains $I(p_2)$ if and only if p_1 is a prefix of p_2. Next, when s has length m, $\iota(s^0) = \iota(s^\delta) = \iota(s)$. Lemma 1 asserts that if p is a prefix of s then $I(p)$ must contain $\iota(s)$. The nested property of intervals in a prefix set P then gives:

Theorem 2 *Let P be a prefix set and s a string in A^m. Then, p is the longest prefix matching s if and only if $I(p)$ is the shortest interval in $I(P)$ containing $\iota(s)$.*

Figure 1(b) shows the correspondence between prefixes and intervals. For example, the string `00101101` with numerical value 45 would have `[40,47]` as the shortest containing interval, giving `b` as the matching result.

Two intervals that are disjoint or nested are called *nested intervals*. Theorem 2 enables treating the LPM problem as that of managing a collection of mutually nested intervals with the following basic operations.

Insert. This inserts a new interval $[a, b]$ with some associated data value v. If $[a, b]$ intersects an existing interval, it is required that $[a, b]$ is contained in or contains this interval.

Retract. This deletes an existing interval $[a, b]$ from the current set of intervals.

Get. Given a integer value p, this determines the value associated with the shortest interval, if any, that contains p.

When m and α are small, standard computer integer types suffice to store the integers arising from strings and interval endpoints. This allows construction of practical data structures for LPM based on integer arithmetic.

2.2 Equivalence among LPM instances and prefix sets

A data structure solving an (A, m) instance can sometimes be used for other instances, as follows. Let (B, n) be another instance of the LPM problem with β the size of B and n the maximal string length. Suppose that $\alpha^m \geq \beta^n$. Since the integers corresponding to strings in $B^{\leq n}$ are less than α^m, they can be represented as strings in $A^{\leq m}$. Furthermore, let $I(p)$ be the interval corresponding to a prefix p in $B^{\leq n}$. Each integer in $I(p)$ can be considered an interval of length 1, so it is representable as a prefix of length m in $A^{\leq m}$. Thus, each string and prefix set in (B, n) can be translated into some other string and prefix set in (A, m). We have shown:

Theorem 3 *Let (A, m) and (B, n) be two instances of the LPM problem in which the sizes of A and B are α and β respectively. Then any data structure solving LPM on (A, m) can be used to solve (B, n) as long as $\alpha^m \geq \beta^n$.*

This use of single values in an interval to generate prefixes is inefficient. Let $[lo, hi]$ be some interval where $lo < \alpha^m$ and $hi < \alpha^m$. Figure 2 shows an algorithm (in C) for constructing prefixes from $[lo, hi]$. A simple induction argument on the quantity $hi - lo + 1$ shows that the algorithm constructs the minimal set of subintervals covering $[lo, hi]$ such that each subinterval corresponds to a single prefix in $A^{\leq m}$. We assume an array `A[]` such that `A[i]` has the value α^i. The function `itvl2pfx()` converts an interval into a prefix by inverting the process of mapping a prefix into an interval described earlier. Note that such a prefix will have length $m - i$.

Given a nested set of intervals I, we can construct a minimal set of prefixes $P(I)$ such that (1) $I(p)$ and $I(q)$ are disjoint for $p \neq q$; and (2) finding the shortest interval in I containing some integer i is the same as finding the longest prefix in $P(I)$ matching $\sigma(i)$. This is done as follows.

1. Sort the intervals in I by their low ends, breaking ties by taking longer intervals first. The nested property of the intervals means that $[i, j] < [k, l]$ in this ordering if $j < k$ or $[i, j]$ contains $[k, l]$.
2. Build a new set of intervals by adding the sorted intervals in order. When an interval $[i, j]$ is added, if it is contained in some existing interval $[k, l]$, then in addition to adding $[i, j]$, replace $[k, l]$ with at most two new disjoint intervals, $[k, i-1]$ and $[j+1, l]$, whenever they are of positive length.
3. Merge adjacent intervals that have the same data values.
4. Apply the algorithm in Figure 2 to each of the resulting intervals to construct the new prefixes.

Figure 1(c) shows how the nested intervals are split into disjoint intervals. These intervals are then transformed into a new collection of prefixes. A property of the new prefixes is that none of them can be a prefix of another.

From now on, we assert that every considered prefix set P shall represent disjoint intervals. If not, we convert it into the equivalent set of prefixes $P(I(P))$ as discussed.

3 The Retrie Data Structure

Theorem 3 asserts that any LPM data structure for one type of string can be used for other LPM instances as long as alphabet sizes and string lengths are within bounds. For example, 15-digit international telephone numbers fit in 64-bit integers, so data structures for fast look-ups of IPv4 32-bit addresses are potentially usable, with proper extensions, for telephone number matching. Unfortunately, many of these are too highly tuned for IP routing to be effective in the other applications that we consider, such as network address clustering and telephone number matching (Section 4). We next describe the *retrie* data structure for fast LPM queries in $A^{\leq m}$. We compare it to prior art in Section 5.

3.1 The basic retrie scheme

Let P be a prefix set over $A^{\leq m}$. Each prefix in P is associated with some data value, an integer in a given range $[0, D]$. We could build a table of size α^m that covers the contents of the intervals in P. Then, the LPM of a string can be looked up with a single index. Such a table would be impossibly large for interesting values of α and m, however. We therefore build a smaller, recursively structured table, which we call a *radix-encoded trie* (or *retrie*). The top level table is indexed by some number of left digits of a given string. Each entry in this table points to another table, indexed by some of the following digits, and so on. As such, there are two types of tables: *internal* and *leaf*. An entry of an internal table has a pointer to the next-level table and indicates whether that table is an internal or leaf table. An entry of a leaf table contains the data associated with the prefix matched by that entry. All internal tables are kept in a single array *Node*, and all leaf tables are kept in a single array *Leaf*.

We show later how to minimize the space used by the entire retrie. The size of a leaf entry depends on the maximum of the data values associated with the given prefixes. For example, if the maximum data value is $< 2^8$, then a leaf entry can be stored in a single byte, while a maximum data value between 2^8 and 2^{16} means that leaf data must be stored using 2 bytes, and so on. For fast computation, the size of an internal table entry is chosen so that the entry would fit in some convenient integer type. We partition the bits of this type into three parts (*pbits*, *ibit*, *bbits*) as follows. The first *pbits* bits define the number of digits used to index the next-level table, which thus has α^{pbits} entries. The single *ibit* bit is either 1, indicating that the next level is internal, or 0, indicating that the next level is leaf. The last *bbits* bits define the offset into the *Node* or *Leaf* array at which the next-level table begins.

In practice, we often use 32-bit integers for internal table entries with the partition $(5, 1, 26)$. Up to 31 digits can thus be used to index a table, and the maximal number of entries in the arrays *Node* and *Leaf* is 2^{26}. This is ample for our applications. For larger applications, 64-bit table entries can be used with some other appropriate bit partitions.

Given a retrie built for some prefix set $P \subseteq A^{\leq m}$, let `root` be a single internal table entry whose bits are initialized to tell how to index the top level table. Let `A[]` be an integer array such that `A[i]` $= \alpha^i$. Now let s be a string in A^m with integer value `sv` $= \iota(s)$. Figure 3 shows the algorithm to compute the data value associated with the LPM of s.

Figure 4 shows a 3-level retrie for the prefix set shown in Figure 1(c). The *Node* array begins with the top-level internal table. Indices to the left of table entries are in binary and with respect to the head of the corresponding table within the *Node* or *Leaf* array. Each internal table entry has three parts as discussed. All internal table entries without the third part go to the same leaf table entry that has some default value for strings without any matching prefix. For

```
for (node = root, shift = m; ; sv %= A[shift])
{   shift -= node >> (bbits+1);
    if (node & (1 << bbits))
         node = Node[(node & ((1 << bbits) - 1)) + sv/A[shift]];
    else return Leaf[(node & ((1 << bbits) - 1)) + sv/A[shift]];
}
```

Figure 3: Searching a retrie for a longest prefix.

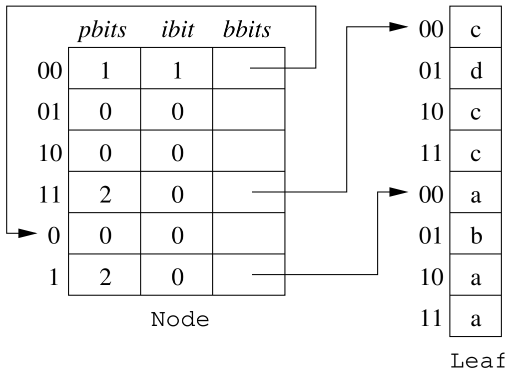

Figure 4: A retrie data structure.

example, the string `00101101` is matched by first stripping off the starting 2 bits `00` to index entry 0 of the top level table. The *ibit* of this entry is 1, indicating that the next level is another internal table. The third part, *bbits*, of the entry points to the base of this table. The *pbits* of the entry indicate that one bit should be used to index the next level. Then, the indexed entry in the next-level table points to a leaf table. The entries of this table are properly defined so that the fourth and fifth bits of the string, `01`, index the entry with the correct data: `b`.

A retrie with k levels enables matching with at most k indexing operations. This guarantee is important in applications such as IP forwarding. Smaller k's mean larger look-up tables, so it is important to ensure that a retrie with k levels uses minimal space. We next discuss how to do this using dynamic programming.

3.2 Optimizing the basic retrie scheme

Given a prefix set P, define $len(P) = \max\{len(p) : p \in P\}$. For $1 \le i \le len(P)$, let $P^{\le i}$ be the subset of prefixes with lengths $\le i$. Then, let $L(P,i)$ be the partition of $P - P^{\le i}$ into equivalence classes induced by the left i digits. That is, each part Q in $L(P,i)$ consists of all prefixes longer than i and having the same first i digits. Now, let $strip(Q,i)$ be the prefixes in Q with their left i digits stripped off. Such prefixes represent disjoint intervals in the LPM instance $(A, m-i)$. Finally, let c_d be the size of a data (leaf table) entry and c_t the size an internal table entry. The following dynamic program computes $S(P,k)$, the optimal size of a retrie data structure for a prefix set P using at most k levels.

$$S(P,k) = \begin{cases} c_d\alpha^{len(P)} & \text{if } P = \emptyset \text{ or } k = 1\text{; otherwise,} \\ \min\left\{c_d\alpha^{len(P)}, \min_{1 \le i < len(P)} \left\{c_t\alpha^i + c_d|P^{\le i}| + \sum_{Q \in L(P,i)} S(strip(Q,i), k-1)\right\}\right\}. \end{cases}$$

The first case states that a leaf table is built whenever the prefix set is empty or only a single level is allowed. The second case recurses to compute the optimal retrie and its size. The first part of the minimization considers the case when a single leaf table is constructed. The second part of the minimization considers the cases when internal tables may be constructed. In these cases, the first term $c_t\alpha^i$ expresses the size of the constructed internal table to be indexed by the first i digits of a string. The second term $c_d|P^{\le i}|$ expresses the fact that each prefix short enough to end at the given level requires a single leaf table entry for its data value. The last term $\sum_{Q \in L(P,i)} S(strip(Q,i), k-1)$ recurses down each set $strip(Q,i)$ to compute a retrie with optimal size for it.

Each set $strip(Q,i)$ is uniquely identified by the string formed from the digits stripped off from the top level of the recursion until $strip(Q,i)$ is formed. The number of such strings is bounded by $|P|len(P)$. Each such set contributes at most one term to any partition of the $len(P)$ bits in a prefix. For $k \leq len(P)$, the dynamic program examines all partitions of $[1, len(P)]$ with $\leq k$ parts. The number of such partitions is $O(len(P)^{k-1})$. Thus, we have:

Theorem 4 *The cost to compute $S(P, k)$ is $O(|P| len(P)^k)$.*

In practice, $len(P)$ is bounded by a small constant, e.g., 32 for IP routing and 10 for U.S. phone numbers. Since the number of levels cannot exceed $len(P)$, the dynamic program essentially runs in time linear in the number of prefixes.

3.3 Superstring lay-out of leaf tables

The internal and leaf tables are sequences of elements. In the dynamic program, we consider each table to be instantiated in the *Node* or *Leaf* array as it is created. We can reduce memory usage by exploiting redundancies, especially in the leaf tables. For example, in IP routing, the leaf tables often contain many similar, long runs of relatively few distinct data values. Computing a short superstring of the tables reduces space very effectively. Since computing shortest common superstrings (SCS) is MAX-SNP hard [2], we experiment with three heuristics.

1. The trivial superstring is formed by catenating the leaf tables.
2. The left and right ends of the leaf tables are merged in a best-fit order.
3. A superstring is computed using a standard greedy SCS approximation [2].

Both methods 2 and 3 effectively reduce space usage (Section 4). In practice, however, the second method gives the best trade-off between computation time and space reduction.

Finally, we note that it is possible to add superstring computation of leaf tables to the dynamic program to estimate more accurately the actual sizes of the leaf tables. This would better guide the dynamic program to select an optimal overall lay-out. The high cost of superstring computation makes this impractical, however. Thus, the superstring lay-out of leaf tables is done only after the dynamic program finishes.

4 Applications and Performance

We consider three applications: IP routing, network clustering, and telephone service marketing. Each exhibits different characteristics. In current IP routing, the strings are 32 bits long, and the number of distinct data values, i.e., next-hops, is small. For network clustering, we merge several BGP tables together and use either the prefixes or their lengths as data values so that after a look-up we can retrieve the matching prefix itself. In this case, either the number of data values is large or there are many runs of data values. For routing and clustering, we compared retries to LC-tries [24], given the conceptual similarity, and to the compressed-table data structure of Crescenzi, Dardini, and Grossi (CDG) [7], given its reported speed. The latter, in particular, is among the fastest IP lookup data structures reported in the literature; cf., Section 5. We used the authors' code for both benchmarks and included in our test suite the FUNET router table and traffic trace used in the LC-trie work [24]. These data structures are designed specifically for IP prefix matching. The third application is telephone service marketing, in which strings are telephone numbers.

4.1 IP routing

Table 1 summarizes the routing tables we used. It reports how many distinct prefixes and next-hops each contained and the sizes of the data structures built on each. Retrie-FL (rsp., -LR; -GR) is a depth-2 retrie with catenated (rsp., left-right, best-fit merge; greedy) record layout. For routing, we limited depth to 2 to emphasize query time. We use deeper retries in Section 4.2. ATT is a routing table from an AT&T BGP router; FUNET is from the LC-trie work [24]; TELSTRA comes from Telstra Internet [29]; the other tables are described by Krishnamurthy and Wang [19].

We timed the LPM queries engendered by router traffic. Since we lacked real traffic traces for the tables other than ATT and FUNET, we constructed random traces by choosing, for each table, 100,000 prefixes uniformly at random (with replacement) and extending them to full 32-bit addresses. We used each random trace as generated and also with the addresses sorted lexicographically to present locality that might be expected in a real router. We generated random traces for the ATT and FUNET tables as well, to juxtapose real and random data. We processed each trace through the

Table 1: Number of entries, next-hops, and data structure sizes for tables used in IP routing experiment.

Routing table	Entries	Next-hops	Data struct. size (KB)				
			retrie			lctrie	CDG
			-FL	-LR	-GR		
AADS	32505	38	1069.49	866.68	835.61	763.52	4446.37
ATT	71483	45	2508.79	2231.89	2180.21	1659.52	15601.92
FUNET	41328	18	506.57	433.00	411.36	967.36	682.93
MAE-WEST	71319	38	1241.14	1040.37	1000.52	1654.06	5520.26
OREGON	118190	33	3828.93	3107.78	3035.73	2711.16	12955.85
PAIX	17766	28	912.94	741.92	723.85	417.74	3241.31
TELSTRA	104096	182	2355.03	2023.08	1971.78	2384.66	9863.96

respective table 100 times, measuring average LPM query time; each data structure was built from scratch for each unique prefix table. We also report data structure build times.

We performed this evaluation on an SGI Challenge and a Pentium. The SGI ran IRIX 6.5 on a 400 MHz R12000 with split 32 KB L1 data and instruction caches, 8 MB unified L2 cache, and 12 GB main memory. The Pentium ran Linux 2.4.6 on a 1 GHz Pentium III with split 16 KB L1 data and instruction caches, 256 KB unified L2 cache, and 256 MB main memory. Each time reported is the median of five runs. Table 2 reports the results.

LC-tries were designed to fit in L2 cache and do so on all the tables on the SGI but none on the Pentium. Retries behave similarly, although they were not designed to be cache resident. CDG fits in the SGI cache on AADS, FUNET, MAE-WEST, and PAIX. Retries uniformly outperformed LC-tries, sometimes by an order of magnitude, always by a factor exceeding three. CDG outperformed retries only on some of the FUNET traces on the Pentium. For that table, the numbers of prefixes and next-hops were relatively low, and CDG is sensitive to these sizes. On the larger tables (OREGON and TELSTRA), retries significantly outperformed CDG. As routing tables are continually growing, with 250,000 entries expected by the year 2005, we expect that retries will outperform CDG on real data. Furthermore, the FUNET trace was filtered to zero out the low-order 8 bits of each address for privacy purposes [23] and thus is not a true trace for the prefix table, which contains some entries longer than 24 bits.

The data suggest that the non-trivial superstring retrie variations significantly reduce the space of the data structure. As might be expected, the greedy superstring approximation is comparatively slow, but the best-fit merge runs with little time degradation over the trivial superstring method and still provides significant space savings.

There is a nearly uniform improvement in look-up times from retrie-FL to retrie-LR to retrie-GR even though each address look-up performs exactly the same computation and memory accesses in all three cases. This suggests beneficial effects from the hardware caches. We believe that this is due to the overlapping of leaf tables in retrie-LR and retrie-GR, which both minimizes space usage and increases the hit rates for similar next-hop values.

The data also suggest that LPM queries on real traces run significantly faster than on random traces. Again this suggests beneficial cache effects, from the locality presented by IP routing traffic. Real trace data is thus critical for accurate measurements, although random data seem to provide an upper bound to real-world performance.

Finally, while retries take longer to build than LC-tries (and sometimes CDG), build time (for -FL and -LR) is acceptable, and query time is more critical to routing and on-line clustering, which we assess next.

4.2 Network clustering

For clustering, we combined the routing tables used above. There were 168,161 unique network addresses in the tables. The goal of clustering is to recover the actual matching prefix for an IP address, thereby partitioning the addresses into equivalence classes [19]. PREF assigns each resulting prefix itself as the data value. LEN assigns the length of each prefix as its data value, which is sufficient to recover the prefix, given the input address. PREF, therefore, has 168,161 distinct data values, whereas LEN has only 32.

We built depth-2 and -3 retries and LC-tries for PREF and LEN. Table 3 details the data structure sizes. Note the difference in retrie sizes for the two tables. The relative sparsity of data values in LEN produces a much smaller *Leaf* array, which can also be more effectively compressed by the superstring methods. Also note the space reduction

Table 2: Build and query times for routing.

Routing table	Data struct.	SGI				Pentium			
		Build	Query (ns)			Build	Query (ns)		
		(ms)	Traffic	Sort. rand.	Rand.	(ms)	Traffic	Sort. rand.	Rand.
AADS	retrie-FL	195		15	20	150		25	48
	retrie-LR	225		15	20	180		24	42
	retrie-GR	4157		15	20	3010		22	40
	lctrie	113		160	215	100		153	380
	CDG	214		17	144	220		25	69
ATT	retrie-FL	459	18	16	31	370	20	32	83
	retrie-LR	518	17	16	22	440	18	31	68
	retrie-GR	9506	17	16	22	7220	19	31	66
	lctrie	245	163	159	214	270	146	181	452
	CDG	1011	64	39	224	1010	42	43	85
FUNET	retrie-FL	166	14	14	18	120	14	20	28
	retrie-LR	190	14	14	17	130	14	21	24
	retrie-GR	1650	14	14	17	1020	14	19	24
	lctrie	136	134	149	199	140	111	153	381
	CDG	102	15	14	20	70	8	14	26
MAE-WEST	retrie-FL	340		14	21	270		28	65
	retrie-LR	388		15	20	310		26	52
	retrie-GR	8241		15	21	6950		26	49
	lctrie	233		155	210	250		176	454
	CDG	325		19	152	310		33	73
OREGON	retrie-FL	447		15	67	350		34	76
	retrie-LR	512		16	23	390		31	59
	retrie-GR	8252		16	23	7740		31	56
	lctrie	383		155	341	420		206	464
	CDG	949		26	142	1050		41	81
PAIX	retrie-FL	118		15	20	90		24	36
	retrie-LR	132		14	18	90		22	30
	retrie-GR	2100		14	18	1360		22	28
	lctrie	66		162	213	60		146	284
	CDG	136		15	123	150		22	58
TELSTRA	retrie-FL	508		16	27	420		37	86
	retrie-LR	572		16	21	480		31	62
	retrie-GR	10000		15	21	8510		31	56
	lctrie	343		155	311	390		201	458
	CDG	599		26	180	610		47	83

achieved by depth-3 retries compared to depth-2 retries. Depth-3 retries are smaller than LC-tries for this application, yet, as we will see, outperform the latter. CDG could not be built on either PREF or LEN. CDG assumes a small number of next-hops and exceeded memory limits for PREF on both machines. CDG failed on LEN, because the number of runs of equal next-hops was too large. Here the difference between the IP routing and clustering applications of LPM becomes striking: retries work for both applications, but CDG cannot be applied to clustering.

We timed the clustering of Apache and EW3 server logs. Apache contains IP addresses recorded at www.apache.org. The 3,461,361 records gathered in late 1997 had 274,844 unique IP addresses. EW3 contains IP addresses of clients visiting three large corporations whose content were hosted by AT&T's Easy World Wide Web. Twenty-three million entries were gathered during February and March, 2000, representing 229,240 unique IP addresses. The experimental setup was as in the routing assessment. In an on-line clustering application, such as in a web server, the data struc-

Table 3: Data structure sizes for tables used in clustering experiment.

Routing table	Data struct. size (KB)						
	depth-2 retrie			depth-3 retrie			lctrie
	-FL	-LR	-GR	-FL	-LR	-GR	
PREF	13554.87	13068.88	13054.80	3181.63	2878.24	2889.90	3795.91
LEN	5704.37	4938.74	4785.66	1400.84	1045.53	990.33	3795.91

Table 4: Build and query times for clustering.

Machine	Table	Operation	Times (build: ms) (query: ns)						
			depth-2 retrie			depth-3 retrie			lctrie
			-FL	-LR	-GR	-FL	-LR	-GR	
SGI	PREF	build	1732	2028	25000	6919	7478	43000	801
		query (Apache)	20	19	19	36	35	35	136
		query (EW3)	20	19	19	36	36	36	139
	LEN	build	1299	1495	28000	4299	4476	25000	588
		query (Apache)	15	16	16	32	32	32	135
		query (EW3)	15	16	16	33	32	32	139
Pentium	PREF	build	1300	1570	19000	4670	5060	32000	750
		query (Apache)	26	26	26	41	40	40	121
		query (EW3)	27	27	27	43	42	42	129
	LEN	build	990	1170	25000	2860	3070	18000	640
		query (Apache)	21	21	21	35	34	34	117
		query (EW3)	23	21	21	37	35	35	124

tures are built once (e.g., daily), and addresses are clustered as they arrive. Thus, query time is paramount. Retries significantly outperform LC-tries for this application, even at depth 3, as shown in Table 4.

4.3 Telephone service marketing

In our telephone service marketing application, the market is divided into regions, each of which is identified by a set of telephone prefixes. Given daily traces of telephone calls, the application classifies the callers by regions, updates usage statistics, and generates a report. Such reports may help in making decisions on altering the level of advertisement in certain regions. For example, the set of prefixes identifying Morris County, NJ, includes `908876` and `973360`. Thus, a call originating from a telephone number of the form `973360XXXX` would match Morris County, NJ.

Table 5 shows performance results (on the above SGI) from using a retrie to summarize telephone usage in different regions of the country for the first six months of 2001. The second column shows the number of call records per month that we use in the experiment. Since this application is off-line, we consider the total time required to classify all the numbers. The third column shows this time (in seconds) for retries, and the fourth column contrasts the time using a binary search approach for matching. This shows the benefit of using retries in this application. Previous IP look-up data structures, which we review next, do not extend to this alphabet.

5 Comparison to Related Work

The popularity of the Internet has made IP routing an important area of research. Several LPM schemes for binary strings were invented in this context. The idea of using ranges induced by prefixes to perform IP look-ups was suggested by Lampson, Srinivasan, and Varghese [20] and later analyzed by Gupta, Prabhakar, and Boyd [15] to guarantee worst-case performance. Ergun et al. [10] considered biased access to ranges. Feldman and Muthukrishnan

Table 5: Time to classify telephone numbers.

Month	Counts	Retrie (s)	Bsearch (s)
1	27,479,712	24.35	83.51
2	25,510,814	22.37	74.73
3	28,993,583	25.49	84.60
4	28,452,823	24.94	80.76
5	29,786,302	26.11	84.86
6	28,874,669	25.27	80.79

[11] generalized the idea to packet classification. We generalized this idea to non-binary strings and showed that LPM techniques developed for strings based on one alphabet can also be used for strings based on another. Thus, under the right conditions, the data structures invented for IP routing can be used for general LPM.

Retries are in the general class of multi-level table look-up schemes used for both hardware [14, 17, 22] and software [8, 9, 24, 27, 28] implementations for IP routing. Since modern machines use memory hierarchies with sometimes dramatically different performance levels, some of these works attempt to build data structures conforming to the memory hierarchies at hand. Both the LC-trie scheme of Nilsson and Karlsson [24] and the multi-level table of Srinivasan and Varghese [28] attempt to optimize for L2 caches by adjusting the number of levels to minimize space usage. Efficient implementations, however, exploit the binary alphabet of IP addresses and prefixes.

Cheung and McCanne [6] took a more general approach to dealing with memory hierarchies that includes the use of prefix popularity. They consider a multi-level table scheme similar to retries and attempt to minimize the space usage of popular tables so that they fit into the fast caches. Since the cache sizes are limited, they must solve a complex constrained optimization problem to find the right data structure. L1 caches on most machines are very small, however, so much of the gain comes from fitting a data structure into L2 caches. In addition, the popularity of prefixes is a dynamic property and not easy to approximate statically. Thus, we focus on bounding the number of memory accesses and minimizing memory usage. We do this by (1) separating internal tables from leaf tables so that the latter can use small integer types to store data values; (2) using dynamic programming to optimize the lay-out of internal and leaf tables given some bound on the number of levels, which also bounds the number of memory accesses during queries; and (3) using a superstring approach to minimize space usage of the leaf tables. The results in Section 4 show that we achieve both good look-up performance and small memory usage.

Crescenzi, Dardini, and Grossi [7] proposed a compressed-table data structure for IP look-up. The key idea is to identify runs induced by common next-hops among the 2^{32} implicit prefixes to compress the entire table with this information. This works well when the number of distinct next-hops is small and there are few runs, which is mostly the case in IP routing. The compressed-table data structure is fast, because it bounds the number of memory accesses per match. Unfortunately, in network clustering applications, both the number of distinct next-hop values and the number of runs can be quite large. Thus, this technique cannot be used in such applications. Section 4 shows that retries generally outperform the compressed-table data structure for IP routing and use much less space.

6 Conclusions

We considered the problem of performing LPM on short strings with limited alphabets. We showed how to map such strings into the integers so that small strings would map to values representable in machine words. This enabled the use of standard integer arithmetic for prefix matching. We then presented retries, a novel, multi-level table data structure for LPM. Experimental results were presented showing that retries outperform other comparable data structures.

A number of open problems remain. A dynamic LPM data structure that performs queries empirically fast remains elusive. Build times for static structures are acceptable for present applications, but the continual growth of routing tables will likely necessitate dynamic solutions in the future. As with general string matching solutions, theoretically appealing approaches, e.g., based on interval trees [25], do not exploit some of the peculiar characteristics of these applications. Feldman and Muthukrishnan [11] report partial progress. We have a prototype based on nested-interval maintenance but have not yet assessed its performance.

Our results demonstrate that LPM data structures perform much better on real trace data than on randomly generated data. Investigating the cache behavior of LPM data structures on real data thus seems important. Absent common benchmark suites of real data, which seem unlikely given the proprietary nature of such data, work on modeling IP address traces for experimental purposes is also worthwhile.

Acknowledgment. We thank John Linderman for describing the use of prefixes in telephone service marketing.

References

[1] K. C. R. C. Arnold. *Screen Updating and Cursor Movement Optimization: A Library Package*. 4.2BSD UNIX Programmer's Manual, 1977.

[2] A. Blum, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. *J. ACM*, 41(4):630–47, 1994.

[3] D. Breslauer. Fast parallel string prefix-matching. *Theor. Comp. Sci.*, 137(2):268–78, 1995.

[4] D. Breslauer, L. Colussi, and L. Toniolo. On the comparison complexity of the string prefix-matching problem. *J. Alg.*, 29(1):18–67, 1998.

[5] Y.-F. Chen, F. Douglis, H. Huang, and K.-P. Vo. TopBlend: An efficient implementation of HtmlDiff in Java. In *Proc. WebNet'00*, 2000.

[6] G. Cheung and S. McCanne. Optimal routing table design for IP address lookup under memory constraints. In *Proc. 18th IEEE INFOCOM*, volume 3, pages 1437–44, 1999.

[7] P. Crescenzi, L. Dardini, and R. Grossi. IP address lookup made fast and simple. In *Proc. 7th ESA*, volume 1643 of *LNCS*, pages 65–76. Springer-Verlag, 1999.

[8] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *Proc. ACM SIGCOMM '97*, pages 3–14, 1997.

[9] W. Doeringer, G. Karjoth, and M. Nassehi. Routing on longest-matching prefixes. *IEEE/ACM Trans. Netwk.*, 4(1):86–97, 1996. *Err.,* **5**(1):600, 1997.

[10] F. Ergun, S. Mittra, S. C. Sahinalp, J. Sharp, and R. K. Sinha. A dynamic lookup scheme for bursty access patterns. In *Proc. 20th IEEE INFOCOM*, volume 3, pages 1444–53, 2001.

[11] A. Feldmann and S. Muthukrishnan. Tradeoffs for packet classification. In *Proc. 19th IEEE INFOCOM*, volume 3, pages 1193–202, 2000.

[12] P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search. In *Proc. 27th ACM STOC*, pages 693–702, 1995.

[13] V. Fuller, T. Li, J. Yu, and K. Varadhan. *Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy*. Internet Engineering Task Force (www.ietf.org), 1993. RFC 1519.

[14] P. Gupta, S. Lin, and M. McKeown. Routing lookups in hardware and memory access speeds. In *Proc. 17th IEEE INFOCOM*, volume 3, pages 1240–7, 1998.

[15] P. Gupta, B. Prabhakar, and S. Boyd. Near-optimal routing lookups with bounded worst case performance. In *Proc. 19th IEEE INFOCOM*, volume 3, pages 1184–92, 2000.

[16] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977.

[17] N.-F. Huang, S.-M. Zhao, J.-Y. Pan, and C.-A. Su. A fast IP routing lookup scheme for gigabit switching routers. In *Proc. 18th IEEE INFOCOM*, volume 3, pages 1429–36, 1999.

[18] G. Jacobson and K.-P. Vo. Heaviest increasing/common subsequence problems. In *Proc. 3rd CPM*, volume 644 of *LNCS*, pages 52–65. Springer-Verlag, 1992.

[19] B. Krishnamurthy and J. Wang. On network-aware clustering of web clients. In *Proc. ACM SIGCOMM '00*, pages 97–110, 2000.

[20] B. Lampson, V. Srinivasan, and G. Varghese. IP lookups using multiway and multicolumn search. *IEEE/ACM Trans. Netwk.*, 7(3):324–34, 1999.

[21] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–72, 1976.

[22] A. Moestedt and P. Sjödin. IP address lookup in hardware for high-speed routing. In *Proc. Hot Interconnects VI*, Stanford Univ., 1998.

[23] S. Nilsson. Personal communication. 2001.

[24] S. Nilsson and G. Karlsson. IP-address lookup using LC-tries. *IEEE J. Sel. Area. Comm.*, 17(6):1083–92, 1999.

[25] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1988.

[26] D. Sankoff and J. B. Kruskal. *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparisons*. Addison Wesley, Reading, MA, 1983.

[27] K. Sklower. A tree-based routing table for Berkeley UNIX. In *Proc. USENIX Winter 1991 Tech. Conf.*, pages 93–104, 1991.

[28] V. Srinivasan and G. Varghese. Fast address lookup using controlled prefix expansion. *ACM Trans. Comp. Sys.*, 17(1):1–40, 1999.

[29] Telstra Internet. http://www.telstra.net/ops/bgptab.txt.