

Overview of Data Structures in IP Lookups

David Antoš

xantos@fi.muni.cz

September 19, 2002

1 Introduction

This report sums up various available methods to find the best matching prefix (BMP) of a string especially for the purpose of IP address lookup in routers. The router decides what to do with an incoming packet according to the longest matching prefix in its routing table. This report does not contain any original methods nor results, it is intended purely as an overview.

The method to be used in a router to achieve high performance should satisfy several conceptual conditions (adapted from (Crescenzi et al., 1999)):

- it should work for any placement of the router in the Internet, both edge and backbone routers,
- it should be useful for a long period of time, as hardware implementation is much more difficult and expensive than software one,
- the most important cost is the number of memory accesses,
- it should be easily implemented in hardware.

Note on terminology: we often refer to the address as *key* and to the routing table as *database*. Most of the results presented here are more general and overlap the area of IP lookup problems. The structures are viewed from a very long distance, losing details. Nevertheless the basic ideas should be clear. If not, please contact the author and complain. The same holds for cases when you think we missed some important aspect of the topic described.

2 Trie and its variants

We describe the trie data structure and its numerous improvements.

2.1 The basic trie structure

The trie data structure represents keys as sequences of symbols, not as a whole like conventional structures do. Let us describe now the basic version of trie as shown in (Knuth, 1998). Let us have a finite alphabet Σ . We put $|\Sigma| = m$. Trie is an m -ary tree, its nodes are m -ary vectors indexed by the Σ alphabet. A node in depth l represents a set of keys starting with a prefix of length l . The node represents an m -way branch driven by $(l + 1)$ -st character of the searched word.

Trie look-at¹ starts at the root node branching by the first character. In a general case we progress as follows. We take the next symbol of the word, let it be k . Then the field of the current node indexed by the character k keeps the pointer to the subtrie, that corresponds the look-at in the unread part of the key. Note that if the key is not in the trie, we find at least its longest prefix.

The time is linear wrt. the length of the key, the same holds for inserting and deleting. Memory complexity becomes a problem in practical applications. If we store the nodes one-by-one into a linear field, majority of the nodes is used only sparsely, wasting the memory significantly.

2.2 Dynamic packed trie

Dynamic packed trie is an optimized version of trie that reduces memory complexity of the structure. The price is more complicated look-at and mainly insertion.

The main idea is that the sparse nodes are kept *one mixed into another* into a linear array. A node of the trie uses fields left empty by another one. We of course have to distinguish the fields belonging to the node we are working with. It would be possible if we store the information about the *symbol of the alphabet* corresponding to the field. And we must not pack two nodes on the same starting position, therefore we need to add a bit denoting *node base position*.

The compression was developed by Liang (Liang, 1983), its comparison to other methods in practical application is documented in (Antoš, 2001). Time complexity is still linear wrt. the key length. The complexity of insertions increases depending on the implementation of the finding of a suitable position to store the node. We have implemented a method with simple heuristics (we put a node with more than threshold used fields to the end of the array) with quite satisfactory results.

2.3 Patricia

Patricia (Practical Algorithm to Retrieve Information Coded in Alphanumeric) is a structure derived from a binary trie. The nodes with the only child are removed and each node keeps the number of bits that should be skipped before next comparison is performed. Patricia cannot find exact match but only a *possible match*. At the end of the lookup we have to check if the result is really a match (checking the full string stored in the node/pointed to from the node).

¹ as opposed to tree look-up

The main difference between Patricia and binary trie is that Patricia uses only bits relevant to the decision where to go. We also call this phenomenon *path compression*.

Patricia needs the stored language to be prefix-less, no string may be a prefix of another. This requirement can be easily achieved by inserting a special end-of-word character at the end of each word.

The structure reduces the number of nodes in the tree, nevertheless the expected depth of the tree (lookup time in other words) remains the same as for binary tries (see (Knuth, 1998)). Moreover, it is not possible to extract full keys from Patricia, the compression simply loses some information.

2.4 LC-trie

The main idea of *level compressed* trie, LC-trie in short, is as follows. The highest i complete levels of the trie are replaced with a single node of length m^i and this replacement is propagated top-down. This structure is described by Andersson and Nilsson in (Andersson and Nilsson, 1993), (Andersson and Nilsson, 1994), and (Andersson and Nilsson, 1995).

We need several auxiliary definitions:

i -prefix String v of length i is called i -prefix of string u , if there is string w (possibly empty) so that $u = vw$. The string w is called i -suffix of the string u .

Multi-digit trie Multi-digit trie containing n elements is

- for $n = 0$ an empty leaf,
- for $n = 1$ a leaf containing that element,
- for $n > 1$ an inner node of degree 2^i for $i \geq 1$. For each possible i -prefix P it has a child that is a multi-digit trie and it contains all i -suffixes of all keys starting with the string P .

For $i = 1$ a multi-digit trie becomes an ordinary one. Previous definition does not cover all the generalizations of trie, it allows only “compression factors” as the powers of two.

Multi-digit trie is called *dense*, if it has the same amount of leaves as the corresponding binary trie.

LC-trie Level compressed trie is a multi-digit trie satisfying:

- degree of each node is 2^i , where i is the smallest number such that at least one of the node’s children becomes a leaf,
- each child is a LC-trie.

LC-trie is a binary tree, where the complete levels are replaced with big nodes and this replacement is propagated top-down. Note that this compression adapts nicely to the distribution of stored words.

LC-tries have the smallest external path among all dense tries. Expected time to find a key is $\Theta(\log^* n)$ for uniform distribution of keys², where $\log^* n$ is an iterated logarithm: we put $\log^* n = 1$ and for $n > 1$ we define $\log^* n = 1 + \log^*(\lceil \log n \rceil)$.

It is obvious that level compression and path compression may be combined.

Nilsson and Karlsson (Nilsson and Karlsson, 1999) used LC-trie for best matching prefix lookup in an implementation of IPv4 router. They combine the basic LC-trie with path compression. To make the implementation efficient, they developed a compact representation in the only linear array (containing an exponent of node's branching factor, skip value for path compression, next node pointer/full string and output information pointer). They also give a method for tree build-up in $O(hn)$ time, where h is the depth of resulting LC-trie³. The algorithm works on previously sorted list of strings.

The problem of the basic LC-trie version is, that the only missing string may cause that it is not possible to build a complete level. The solution may be that when finding the branching factor we do not want the levels to be fully complete, but we choose a weaker criterion. We choose a *fill factor* x , where $0 < x \leq 1$. When computing the branching factor for a node covering k prefixes we use the highest possible factor that causes at most $\lceil k(1-x) \rceil$ empty leaves.

Practical test show that the branching factor at the root node affects significantly the overall performance, hence the authors recommend to have a fixed branching factor for the root node, independently on the value of the fill factor x .

Experimental results for IPv4: average depth of the trie is less than 2 and each node needs one memory access. Next access is needed to check whether the found string is really a match (because of the path compression). If so, one more access is needed to read the next-hop value.

In tests made by the authors they are able to perform minimally 0.5 million of lookups per second in a software implementation running on Pentium 133 MHz with fill-factor 0.5 and the first branching of the root on the 16th bit. About 500 kB is needed to keep the structure.⁴

2.5 Grid-of-tries

Grid-of-tries (Srinivasan et al., 1998) is a trie-based memory for multi-dimensional filters, such as source-destination address pairs. The basic scheme can be extended to handle filters on more dimensions, for example with port and protocol numbers.

The structure is motivated with set pruning trees. Let us show the idea behind set pruning trees first. It is a trie of tries, where we first match the destination prefix and the result is the source trie. In the source trie we match the source prefix and obtain the output information. The key is how to connect the source and the destination tries. This simple scheme suffers from memory blowup as a source prefix may appear in several

² this is of course quite a strong condition for us

³ quite a strange measure depending on a result of the algorithm, isn't it?

⁴ I am fully conscious that numbers like this may be much more misleading than theoretical bounds from the complexity theory

tries. A worst case example uses $O(N^2)$ memory, where N is the number of prefixes in the structure.

In order to avoid the memory explosion, we observe that filters associated with a destination prefix D are copied into source trie of D' whenever D is a prefix of D' . We can avoid that by pointing D to a source trie that stores output whose destination field is exactly D . This requires to modify the search strategy so that we must now search the source tries associated with all ancestors of D . Since each output is stored just once, the memory requirement is $O(NW)$, where W is the maximal length of the key. On the other hand, the time cost grows to $O(W^2)$.

The search time can be improved to $O(W)$ while the memory requirement is kept linear. The key idea is using precomputation and switch pointers. The stuff is quite tricky and needs pictures, therefore we recommend the interested reader to have a look at the Section 5 of (Srinivasan et al., 1998).

2.6 Multi-bit trie with expansion

Just another version of trie memory a bit shifted to use in hardware is presented by (Moestedt and Sjödin, 1998). They use a multi-bit trie with number of bits in levels, typically chosen in order to break the key into three to five peaces. The prefixes are expanded up to the nearest level. There are three types of nodes: *valid*, *part*, and *invalid* ones. A valid node represents an entry in the database. A part node represents a prefix of a valid entry, and an invalid node does not represent a valid entry in the database, nevertheless it has a special meaning for the lookup algorithm.

To simplify the lookup process, we want the tree to satisfy two conditions: All possible children of a part node are present in the trie (so called *prefix group*; the added prefixes that do not appear in the database are marked is invalid), and no node is allowed to be valid and part at the same time. If there is a prefix that is both valid and prefix of a valid prefix, it appears several times in the trie, once as a part node and then expanded into a prefix group where all entries are valid.

To find an output we search the trie from the shortest prefix up to the first valid or invalid node. The lookup time is linear wrt. the number of levels of the trie. There is a trade-off between memory and lookup time as having less trie levels causes more auxiliary entries to be added. The authors provide only measurements in a practical application.

Such a structure may be represented using separate linear arrays, number of them equals to the number of levels. An entry either represents a valid prefix and then it points to the output information or it represents a part of a prefix and then it contains pointer to a new table on lower level. For lookup, the key is divided into subfields corresponding the levels of the trie. Comparison of the first level gives the result or a pointer to a table on level two and so on.

Memory consumption depends on the size of the table entries, for tables of practical measurements we refer to the original article. As a result of prefix expansion we often see a prefix group that is filled with just two kinds of entries. By storing one entry of each kind, we can save memory. We introduce a special representation of this special kind of

prefix groups. This is easy to implement in hardware (with one AND-gate, a comparator, and two multiplexers per stage).

3 Binary search using prefix lengths

Binary search over prefix lengths is based on three basic ideas: hashing to check if the address matches a prefix of given length. Then, binary search through prefix lengths is used and finally some precomputing is done to prevent backtracking. The technique is described deeply in (Waldvogel et al., 2001).

First we show the linear lookup on tables containing prefixes of the same length and later we will refine this basic scheme. First we divide the prefix database according to their lengths. We now have a table for each length.

To find the longest prefix for the key D we start from the table of the longest prefix, we extract the appropriate number of bits from D and we try to find it in the hash table. If it succeeds we have the best matching prefix. Otherwise, we continue testing tables of the nearest smallest prefix length.

Having a hash function computable in linear time (it means perfect in the best case), this lookup is linear wrt. the number of distinct prefix lengths in the database.⁵

It brings the possibility to go through the distinct prefix lengths using a binary search. The problem is that the classical terms ‘lesser’ and ‘bigger’ in binary searches mean ‘shorter’ and ‘longer’ in terms of prefix lengths. However, such kind of information cannot be obtained from hashing. Hashing says only ‘hit’ and/or ‘miss’ but nothing in between.

From the other side, when we find a matching field, it does not mean that it is the best matching one. There may be a longer one. Hence, when we match, we have to continue testing longer prefixes, too. More practically, when we find a match we remember it and go on checking longer tables. We keep a track of the so far best matching candidate. On the contrary, if we miss, it only has sense to continue among shorter prefixes.

It is easy to find an example that the strategy above is not correct. To repair it we have to add prefixes of all longer lengths to all tables of smaller lengths (if they are not already there) to lead the search. Those dummy prefixes (called *markers*) allow the algorithm above to get to longer prefixes. Of course, they have no associated output values.

If we add such an auxiliary information, natural question is how much memory we will have to pay. It turns out that markers do not have to be added to all nodes but only to a logarithmic number of them. Using a sophisticated precomputation (its description is in the article noticed above) we can achieve lookup time $O(\log W_d)$, where W_d is the number of distinct prefix lengths in the database. We recall that this upper bound is based on the presumption that we have a constant-time hashing function.

This basic scheme may be improved. One of the possibilities is to move the top-speed lookup towards the most probable case (paying with some performance loss in the worst case). This can be done bringing some asymmetry into the structure, pushing the most probable length to the root (for IPv4 typically 24 bit length). Obviously, some paths may degenerate to a pure linear search which is unacceptable.

⁵ this assumption is very important and of course quite difficult to achieve, the weakest point of the technique

Another improvement may be *mutating binary search*. Its main idea—as opposed to the implementation—is quite simple. When we get a matching entry in the hash table and we move to a new subtree, it is sufficient to stay in this subtree. It creates a whole network of binary trees, getting a match causes changing the tree to a more specialized one. In other words, the binary tree changes, mutates, dynamically during a search.

Each field in the hash table can contain a description of the new tree specialized for prefixes of this table. The authors do not show any theory for this method, they only declare that in all analyzed databases the number of hash lookups decreased from five to four in the worst case and the average number of hash table lookups was two.

The precomputation of the tree network can be quite a complex operation, the same is true for memory complexity of the auxiliary structures.

4 Controlled prefix expansion

Controlled prefix expansion transforms a set of prefixes into an equivalent set of prefixes with fewer prefix lengths. The technique is described in (Varghese and Srinivasan, 1999).

The idea is—as usually—simple. The prefixes are padded with all possible strings of zeroes and ones that are missing in the database to reach higher prefix length. Of course if a string we should add with this process is already in the database, we do not add it, we let the original one stay in its place. It is obvious that when performed naively, prefix expansion may significantly increase storage. For example, the whole prefix lookup problem may be converted into the ordinary array search if we expanded all the prefixes in the database to the full length. The search would be especially easy, nevertheless this solution is unacceptable as the array would require 2^{128} fields for IPv6.

On the other hand, prefix expansion can improve many lookup schemes. We want the number of distinct prefix length to be as small as possible. Some authors recommend the initial lookup (16 or 24 bits for IPv4) as the start of the process. The model may be more general, picking the lengths dynamically. The goal is to minimize total storage needed.

4.1 Choosing optimal levels

Using “brute force” to choose the optimal solution is not possible due to the enormous number of combinations to test. This naturally leads to using dynamic programming. In the first step, we compute the distribution of prefix lengths in the database. The last level must be the total key length. In the second step, we reduce the problem to placing next-to-last level and covering the remaining lengths. If we compute the storage required for each value, the third step we can choose the minimal value.

The above indicated recursive algorithm reduces the optimal level picking to Wk sub-problems, where W is the key length and k is the number of desired levels. Time complexity is $O(W^2k)$. The algorithm minimizes the number of expanded prefixes. The final goal is to reduce the total storage required. We are not able to give a general framework for that as it depends on the actual storage method. For example, in tries some added prefixes may increase the storage significantly while others may share portions of the trie structure. Therefore an application specific solution is needed.

Let us now consider a multibit trie with prefix lengths expanded to certain levels. To search for the best matching prefix, we break the key into chunks corresponding to the lengths of the trie and use the chunks to follow the path in the trie until we reach a null pointer. Following the path, we keep track of the last output associated with the path. The last information we discovered is the longest matching prefix. The search time is $O(k)$, where k is the maximum path through the expanded trie.

When inserting, we first simulate search on the string up to the last chunk. We either terminate by reading the last chunk or reading a null pointer. In both cases we finish the chunk inserting the remaining prefix bits there.

For deletion, there is no way to find out which expanded prefixes belong to a certain prefix. The easiest way to handle deletions is to keep an auxiliary one-bit trie in the memory containing the prefixes in the pure, unmodified form. It could serve both insertions and deletions as a initial data structure from which the “productive” compressed version is derived. The complexity of insertion and deletion is $O(W)$ to search plus the time to reconstruct trie node $O(S_m)$, where S_m is the maximum size of the node.

The application of this method to multibit trie leads to a method based on dynamic programming that allows to the complexity of finding the optimal level picking in time $O(kW^2)$, where k is the number of levels desired and W is the key length. For a detailed description we again refer to paper (Varghese and Srinivasan, 1999).

4.2 Allowing adaptive level cutting

The next natural step is to allow the expansion lengths differ in various subtrees. It turns out that the complexity of the compression algorithm increases with a multiplicative factor of n —number of prefixes in the database, the total time complexity is therefore $O(nW^2k)$. Nevertheless if we compare this method with LC-tries, we discover that various paths lead to the same result. This structure is nothing else than just LC-trie. (Moreover the authors declare that LC-trie is a special case of their structure with the special property that all levels are completely packed. If they read the articles about LC-trie carefully, they should know there is also a more sophisticated version allowing to set fill factor for packed nodes and producing much better results.)

4.3 Applying the ideas to other methods

The ideas of prefix expansion are quite general and do not have to be related only to trie searches. They can also be applied to binary search on levels (see Section 3).

The time complexity of binary search on levels depends logarithmically on the number of distinct lengths in the database. By expansion, we can decrease the number of distinct lengths, paying with some increase of storage. Nevertheless we needed markers to make the searches work correctly. The number of markers actually decreases. When n prefixes are expanded so that only $W/2$ lengths are left in the database the number of prefixes doubles to $2n$. The worst case number of markers changes from $n(\log_2 W/2 - 1)$ to $n(\log_2(W/2) - 1) = n(\log_2 W - 2)$. It means that the total number of hashes can be reduced by one without changing the worst case storage requirement.

Compressing the levels for hash table search produces the serious problem of finding a hash function. The requirement of having perfect hash function is very strong. The authors use a bit weaker criterion, semi-perfect hash function. *Semi-perfect hash function* is a hash function that guarantees that the number of collisions in a bucket is bounded from above by a constant natural number (known in advance). This allows using fixed memory for the list of conflicting items. Obviously, this approach wastes memory as the lists may be used in a small number of cases.

The problem is that when reducing the number of levels causing the hash tables big, finding a semi-perfect hash function may be a very costly operation, it can take even minutes to do.

5 Expansion/compression approach

Let us now have m -bit key to find the longest prefix. This method starts from a fully expanded table with 2^m entries and tries to compress them. Between the 2^m entries in the table there are not many different outputs. Representing the relation between keys and outputs as strings, the strings can be compressed in order to provide an implicit representation of the strings in the table. The technique is described in detail in (Crescenzi et al., 1999).

In the expansion phase, we derive implicitly the outputs for all the 2^m keys in the obvious manner (see also Section 4, the process is similar). In the compression phase, we fix a value k satisfying $1 \leq k \leq m$ and we find two statistical parameters α_k and β_k related to some properties of the data in the tables.

We are now given the expanded table T' and we want to build tables to represent the same information in less space by a *run length encoding* scheme (RLE). We start with clustering pairs in T' according to the first k bits of their strings. The cluster corresponding to a string $x \in \Sigma^k$ is the set of pairs of all suffixes that follow x in the expanded table and their outputs. Note that the cluster has 2^{m-k} elements.

Now we define *row k -size* α_k as the number of distinct cluster corresponding to strings of length k .

The RLE compression takes part now. It takes a cluster $T'_{(x)}$ and returns a RLE sequence $s_{(x)}$ as follows. We sort the cluster in ascending manner and transform it into RLE sequence by replacing each maximal run with the same output information with the output and number of repetitions. The α_k is the number of sequences produced. We further process them to obtain an equal number of equivalent RLE sequences. The main goal is to obtain sequences satisfying for any i that the i -th pair of any two such sequences have the same run length value. This can be done by cutting the sequences in appropriate places. The result of the process is a set of sequences where the run lengths are common to all of them.

What we have now are the α_k sequences s_1, \dots, s_{α_k} obtained from the distinct clusters $T'_{(x)}$ with $x \in \Sigma^k$, unified to the same run lengths. We define the *column k -size* β_k , where $1 \leq k \leq m$ as the (equal) length for the RLE sequence resulting from the process above. It can be shown that for $1 \leq k \leq m$ is $\beta_k \leq 3|T|$.

Let $\#bytes(n)$ denote the number of bytes needed to store a non-negative integer n and let $word$ be large enough to hold $\max\{\log |T| + 1, \log H\}$ bits.

Given a table T with m -bit key and H distinct outputs, we can store T into three tables `row_index`, `col_index`, and `output` of total size $2^k \cdot \#bytes(\alpha_k) + 2^{m-k} \cdot \#bytes(\beta_k) + \alpha_k \cdot \beta_k \cdot \#bytes(H)$ bytes, or $O(2^k + 2^{m-k} + |T|^2)$ words for $1 \leq k \leq m$. A lookup can be performed using *exactly three* accesses. The proof of the theorem can be found in the article referred above.

6 Other improvements

In this section we show some tricks that are sometimes used to optimize the lookup process.

6.1 Using fields for the “first jump”

Sometimes it may be useful to fix several first bits of the key and find them in a simple indexed array. Each field of the array points into a full lookup structure to the corresponding prefix.

6.2 Converting address ranges into prefixes

We often have to deal with *ranges* of values. As shown in (Srinivasan et al., 1998), an arbitrary range can be converted into a union of prefix ranges, and the prefix range can be expressed by a prefix.

Suppose we want to convert an arbitrary range X that lies in a range $[0, 2^k]$. We define an *anchored range* as a range having at least one endpoint at the end of the enclosing range. Then the range X can be split into at most two anchored ranges that lie within $[0, 2^{k-1}]$ and $[2^{k-1}, 2^k]$. Each anchored range can be split into a logarithmic number of prefix ranges. Therefore we can represent an arbitrary subrange of $[0, 2^k]$ with at most $2k$ prefix ranges. For example, the range > 1023 can be expressed with six prefixes 0000001, 00001, 0001, 001, 01, and 1.

6.3 Internal cache

Caching is a popular method for IPv4. The question is whether four times longer address in IPv6 together with arbitrary division by a mask allows often local appearance of same prefixes in the network traffic. I guess it will not, nevertheless the question of using cache should be discussed more deeply.

7 References

Andersson, A. and Nilsson, S. (1993). Improved Behaviour of Tries by Adaptive Branching. *Information Processing Letters*, 46:295–300.

- Andersson, A. and Nilsson, S. (1994). Faster Searching in Tries and Quadrees—An Analysis of Level Compression. In *Proceedings of Second Annual European Symposium on Algorithms*, pages 82–93.
- Andersson, A. and Nilsson, S. (1995). Efficient Implementation of Suffix Trees. *Software—Practice and Experience*, 25(2):129–141.
- Antoš, D. (2001). PatLib, Pattern Manipulating Library. Master’s thesis, Faculty of Informatics.
- Crescenzi, P., Dardini, L. and Grossi, R. (1999). IP Address Lookup Made Fast and Simple. Technical Report TR-99-01, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy.
- Knuth, D. E. (1998). *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition.
- Liang, F. M. (1983). Word Hy-phen-a-tion by Com-pu-ter. Technical Report STAN-CS-83-977, Stanford University.
- Moestedt, A. and Sjödin, P. (1998). IP Address Lookup in Hardware for High-Speed Routing. In *Proc. IEEE Hot Interconnects 6 Symposium*, pages 31–39, Stanford, California, USA.
- Nilsson, S. and Karlsson, G. (1999). IP-Address Lookup Using LC-Tries. In *IEEE Journal on Selected Areas in Communications*, pages 1083–1092.
- Srinivasan, V., Varghese, G., Suri, S. and Waldvogel, M. (1998). Fast and Scalable Layer Four Switching. In *Proceedings of the ACM SIGCOMM 98*.
- Varghese, G. and Srinivasan, V. (1999). Fast Address Lookups using Controlled Prefix Expansion. *Transactions on Computer Systems*, 1(17):1–40.
- Waldvogel, M., Varghese, G., Turner, J. and Plattner, B. (2001). Scalable High-Speed Prefix Matching. *ACM Transactions on Computer Systems*, 19(4):440–482.
- Waldvogel, M., Varghese, G., Turner, J. and Plattner, B. (1997). Scalable High Speed IP Routing Lookups. In *Proceedings SIGCOMM 97*.