PREPRINT. Information Processing Letters, 46:295-300, 1993.

Improved Behaviour of Tries by Adaptive Branching*

Arne Andersson Stefan Nilsson
Department of Computer Science, Lund University,
Box 118, S-221 00 Lund, Sweden

Abstract

We introduce and analyze a method to reduce the search cost in tries. Traditional trie structures use branching factors at the nodes that are either fixed or a function of the number of elements. Instead, we let the distribution of the elements guide the choice of branching factors. This is accomplished in a strikingly simple way: in a binary trie, the i highest complete levels are replaced by a single node of degree 2^i; the compression is repeated in the subtries. This structure, the *level-compressed trie*, inherits the good properties of binary tries with respect to neighbour and range searches, while the external path length is significantly decreased. It also has the advantage of being easy to implement. Our analysis shows that the expected depth of a stored element is $\Theta(\log^* n)$ for uniformly distributed data.

Keywords: Algorithms, data structures, trie, digital search tree, level-compressed trie

1 Introduction

A fundamental, well known, and very well studied technique for storing and retrieving data is to use *tries* [6, 7, 11]. In its original form, the trie is a simple data structure; a set of strings from an alphabet containing k symbols is stored in a natural way in a k-ary tree where each string corresponds to a unique path. In this article, we assume that the elements are strings from a binary alphabet. The generalization to k-ary strings is straightforward.

We say that a string v of length i is the *i-prefix* of a string u if there is a string w such that $u = vw$. The string w is called the *i-suffix* of u. We define a binary trie in the following way:

Definition 1 *A binary trie containing n elements is*

if $n = 0$: an empty leaf;

if $n = 1$: a leaf containing the element;

if $n > 1$: an internal node of degree 2. The left child is a binary trie containing the 1-suffixes of all elements starting with 0 and the right child is a binary trie containing the 1-suffixes of all elements starting with 1.

Note that the strings are assumed to be prefix-free, i.e. no string is a prefix of another string. In particular this implies that there can't be any duplicate strings.

*Published in *Information Processing Letters*, 46:295–300, 1993.

However, if all strings to be stored in the trie are unique, it is easy to ensure that the strings are prefix-free by using a special string terminator. For example, we can append the string "100..." to the end of all strings.

The behaviour of binary tries is well analyzed [3, 4, 11, 12]. For example, in [3] it is shown that the expected average depth of an element is $\Theta(\log n)$ for an independent random sample taken from any distribution with a frequency function $f(x)$ such that $\int f^2(x)\,dx < \infty$. Thus, in many practical cases, the asymptotic behaviour of tries with respect to operations like element retrieval, nearest neighbour search, and range queries is the same as for a comparison-based search tree (or even better if we take into account the fact that a bit-comparison can be performed significantly faster than an element comparison).

A natural way to decrease the search cost in a trie is to use more than one digit for branching. In this way we obtain a *multi-digit trie* as defined below.

Definition 2 *A multi-digit trie containing n elements is*

if $n = 0$: an empty leaf;

if $n = 1$: a leaf containing the element;

if $n > 1$: an internal node of degree 2^i, $i \geq 1$. For each possible i-prefix P there is a child, which is a multi-digit trie, containing the i-suffixes of all elements starting with P.

The definition above does not cover all possible generalizations of the binary trie since we only allow branching factors that are powers of 2. However, this restriction does not constitute a significant limitation and, as pointed out by Tarjan and Yao [14], during a search the computations at the nodes will be simple.

A number of sophisticated variations of multi-digit tries can be found in the literature. The data structure introduced by Tarjan and Yao [14] allows element location in $O(1)$ expected time and $O(\log_n |U|)$ in the worst case, where $|U|$ is the size of the universe. This structure does not support range and neighbour queries efficiently.

The P-fast trie and the Q-fast trie by Willard [17] support element retrieval and nearest neighbour search in

$$O(\sqrt{\log |U|})$$

time in the worst case using $O\left(n\sqrt{\log |U|}2^{\sqrt{\log |U|}}\right)$ and $O(n)$ space respectively. A range search requires $O(\sqrt{\log |U|} + s)$ time, where s is the size of the output.

The Y-fast trie by Willard [16] is a trie structure that uses perfect hashing [1]. This trie requires $O(n)$ space and supports element retrieval and nearest neighbour search in $O(\log\log |U|)$ time in the worst case and range search in $O(\log\log |U| + s)$ time.

The reason why rather sophisticated methods are required in order to obtain efficient multi-digit tries is that a careless use of large branching factors will introduce a large number of empty leaves. This may cause a multi-digit trie to perform much worse than a binary trie with respect to space requirement and the cost of neighbour and range searches. In particular, we have the following facts:

1. no multi-digit trie has fewer leaves than the corresponding binary trie (i.e. the binary trie containing the same set of elements);

2. during a range or neighbour search, the number of leaves that has to be examined in a multi-digit trie is never less than in the corresponding binary trie.

Thus, among all multi-digit tries, the binary trie has some favourable properties.

In an attempt to find trie structures which support fast retrieval of elements and retain these good properties we focus our interest on multi-digit tries that do not contain any superfluous leaves.

Definition 3 *A multi-digit trie is called dense if it contains the same number of leaves as the corresponding binary trie.*

In this article we study the class of dense tries. In particular, we introduce *level-compressed tries*, which have the smallest external path length among all dense tries. Our analysis will show that these tries support element retrieval in $\Theta(\log^* n)$ time in the expected case for an independent random sample taken from the uniform distribution. The function $\log^* n$ is the iterated logarithm function, which is defined as follows. $\log^* 1 = 1$. For any $n > 1$, $\log^* n = 1 + \log^*(\lceil \log n \rceil)$. This function has an extremely slow growth rate. Thus, level-compressed tries are an attractive alternative to other data structures for storage and retrieval of data.

2 Data Structure

As pointed out in the introduction, dense tries have the same favourable properties as binary tries with respect to space efficiency and proximity operations. Thus, by finding a dense trie with significantly higher branching factors than a binary trie, we may be able to improve the cost of search operations without loosing any of the nice properties. We say that a dense trie is *optimal* if no other dense trie containing the same elements has smaller external path length. As we will show, the level-compressed trie defined below is an optimal dense trie.

Definition 4 *A level-compressed trie, LC-trie, is a multi-digit trie with the following properties:*

- *the degree of the root is 2^i, where i is the smallest number such that at least one of the children becomes a leaf;*
- *each child is a level-compressed trie.*

The LC-trie may be viewed as a binary trie where the i highest complete levels are replaced by a single node of degree 2^i; the replacement being made top-down. An example of a binary trie and the corresponding LC-trie is given in Figure 1. It follows immediately from the definition that an LC-trie is dense. Furthermore, in the next theorem we prove that an LC-trie is optimal.

Theorem 1 *A dense trie is optimal if and only if it is an LC-trie.*

Proof: Let S be a set of elements. The theorem follows from the following three facts:

- there is at least one optimal dense trie containing S;
- there is exactly one LC-trie containing S;
- an optimal dense trie is an LC-trie.

First, we note that an optimal trie exists, since there are only finitely many dense tries that can contain S.

The second fact follows from the definition of LC-tries.

The proof of the third fact will be by contradiction. Assume that the dense trie T is optimal and not an LC-trie. Since T is not an LC-trie, there is a node v in T such that each child of v has a degree of at least 2. Now, we construct a new dense trie T' in the following way:

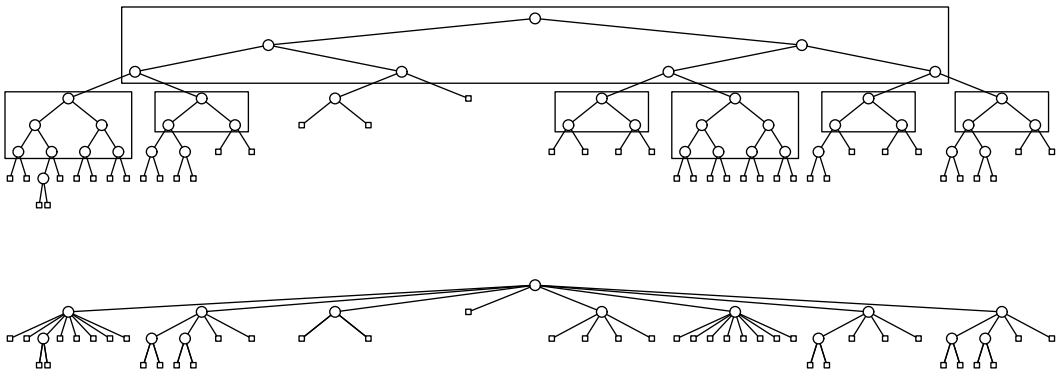

Figure 1: *Compressing a trie. Compressed levels are marked by rectangles.*

1. increase the degree of v by a factor of 2;
2. for each child w of v do
   (a) replace w by two nodes w_1 and w_2, both having half the degree of w;
   (b) place the subtries of w below w_1 and w_2 in the obvious way.

It is easily seen that T' is dense and that no leaf in T' has a larger depth than the corresponding leaf in T. Thus, the external path length of T' is not larger than that of T. Now, the procedure above is repeated until we arrive at a dense trie T'', where at least one of v's children is a leaf. Let x be such a leaf. In T'' the distance between v and x is 1, while this distance is larger in T. Thus, T'' has smaller external path length than T. This contradicts the fact that T is optimal. □

3 Implementation

An LC-trie can be implemented efficiently using an array where each node is represented by a record containing two nonnegative integers *bits* and *adr*. The method is similar to the implementation of compressed tries described in [7].

- If $bits > 0$ the record represents an internal node of degree 2^{bits}. The children of this node are stored in positions $adr + i$, $0 \leq i < 2^{bits}$.
- If $bits = 0$ the record represents a leaf. If $adr = 0$ the leaf is empty, otherwise *adr* contains a reference to the element represented by the leaf.

From the definition of LC-tries it follows that the degree of a node can be at most $2^{\lceil \log n \rceil}$. Therefore, the value of the integer *bits* can never exceed $\lceil \log n \rceil$ and hence $\lceil \log(1 + \lceil \log n \rceil) \rceil$ bits are sufficient to represent *bits*. Thus, in a practical application *bits* will easily fit into one byte.

The fact that we use *adr* both as a reference into the array and as a pointer to an element causes no problem since integers and pointers are represented in the same way.

Below, we give the algorithm for the search operation. Let S be the string searched for and let EXTRACT(S, k, m) be a function that returns the number

given by the m bits starting at position k in S. It should be clear that if m bits fit into one machine word this operation can be performed in constant time (in the unit cost RAM model) by simple arithmetic operations. We denote the array representing the tree by T. The root is stored in T[1]. The algorithm uses two temporary variables k and *node*.

```
k := 1
node := 1
while T[node].bits > 0 do
      node := T[node].adr + EXTRACT(S, k, T[node].bits)
      k := k + T[node].bits
endwhile
return T[node].adr
```

The value returned by EXTRACT will never exceed the degree of the node. Thus, from the discussion above it follows that a call to EXTRACT can be performed in constant time. Hence, the search time is proportional to the number of nodes visited on the traversed search path.

In the next section we show that this path will be significantly shortened by level-compression.

4 Analysis

In the following analysis we give a tight asymptotic bound on the expected average depth of an element stored in an LC-trie.

We use a statistical model where real numbers are drawn at random and their representations as binary fractions are used to decide where in the trie they are stored. Note that with probability one no rational numbers occur, which corresponds to finite or periodic binary fractions. Let $x_1, \ldots, x_n$ be an independent random sample taken from the uniform distribution on $[0, 1)$. We choose the indices in such a way that $x_1 < \cdots < x_n$, and write $x_0 = 0$ and $x_{n+1} = 1$. In our analysis we will use the random variable

$$M_n = \max_{0 \le i \le n} (x_{i+1} - x_i),$$

the maximal spacing between adjacent elements in the sample. The distribution of M_n is well studied; already 1897 Whitworth [15] found the distribution function. Devroye [2] and Slud [13], among others, have studied tail estimates for M_n. We will use the following version of a lemma by Slud [13]:

Lemma 1 *If δ is a positive constant, then*

$$P(|nM_n/\ln n - 1| > \delta) = O(n^{-\delta}).$$

Theorem 2 *If an independent random sample taken from the uniform distribution on the unit interval is stored in a level-compressed trie, the expected depth of an element is $\Theta(\log^* n)$.*

Proof: First we determine an upper bound. We will show that the degree at the root is $\Omega(n/\log n)$ with high probability. This also holds true at the lower levels of the trie, since the elements in a subtree are a random sample of independently chosen points in an interval of some length 2^{-j}; thus, the analysis applies with due alteration of details.

Let x be an element and let T_x denote the number of elements that are stored in the same subtrie as x. Choose the integer i such that $\frac{2\ln n}{n} < 2^{-i} \le \frac{4\ln n}{n}$ and split

the interval $[0,1)$ into subintervals of size 2^{-i}. Let I denote the subinterval into which x falls. Let y be any of the other $n-1$ elements and let p be the probability that y falls into I. Then, $p = 2^{-i} \leq \frac{4 \ln n}{n}$.

In the expected case, the effect of choosing i like this will be that $M_n \leq 2^{-i}$. Therefore every subinterval of size 2^{1-i} will contain at least two elements and hence the number of children of the root will be at least 2^i. This implies that the elements that are stored in the same subtrie as x will be a subset of the elements in I. From this we will deduce that for some constant C, $P(T_x > C \ln n)$ is small. This is done by studying the random variables M_n and n_I, where n_I denotes the number of elements (not counting x) falling into I.

Clearly, n_I obeys the binomial distribution $B(n-1,p)$. Let $C = 2e$. We use the Chernoff bound [8] to get a tail estimate for the binomial function.

$$P(n_I > C \ln n) \leq e^{-(n-1)p} \left(\frac{e(n-1)p}{C \ln n} \right)^{C \ln n} = O(n^{-1}) \tag{1}$$

If $M_n \leq 2^{-i}$ then, as explained above, $T_x \leq n_I$. In particular

$$M_n \leq 2^{-i} \text{ and } n_I \leq C \ln n \Rightarrow T_x \leq C \ln n$$

and hence

$$P(M_n \leq 2^{-i} \text{ and } n_I \leq C \ln n) \leq P(T_x \leq C \ln n). \tag{2}$$

From Lemma 1 we have

$$P\left(M_n > \frac{\ln n}{n} (1+\delta) \right) = O(n^{-\delta}).$$

Hence

$$P(M_n > 2^{-i}) = P\left(M_n > \frac{\ln n}{n} \left(1 + \frac{n 2^{-i}}{\ln n} - 1 \right) \right) = O(n^{-\varepsilon}), \tag{3}$$

where $\varepsilon = \frac{n 2^{-i}}{\ln n} - 1 \geq 1$.

Combining these results, we get

$$\begin{aligned} P(T_x > C \ln n) &\leq P(\neg(M_n \leq 2^{-i} \text{ and } n_I \leq C \ln n)) \\ &\leq P(M_n > 2^{-i}) + P(n_I > C \ln n) \\ &= O(n^{-1}). \end{aligned} \tag{4}$$

From Theorem 1 it follows that the expected depth of an element in an LC-trie is never larger than that of the corresponding binary trie. It has been shown by Devroye [3] that the expected external path length of a binary trie is $O(n \log n)$. Therefore, we can use the estimate $O(\log n)$ for the expected depth of an element in an LC-trie when $T_x > C \ln n$.

We inductively assert that the expected depth $A(n)$ can be bounded by

$$A(n) \leq D \log^* n \tag{5}$$

for some constant D. This is obviously true for $n = 1$. For $n \geq 2$ there is a constant D' such that

$$\begin{aligned} A(n) &\leq 1 + \sum_{m=1}^{C \ln n} P(T_x = m) \cdot A(m) + P(T_x > C \ln n) \cdot O(\log n) \\ &\leq 1 + \sum_{m=1}^{C \ln n} P(T_x = m) \cdot D \log^*(C \ln n) + O(1) \\ &\leq D(\log^*(\log n)) + D'. \end{aligned} \tag{6}$$

It can be arranged that $D' \leq D$. Then the last expression can be bounded from above by $D(\log^* n - 1) + D' \leq D \log^* n$ and hence $A(n) = O(\log^* n)$.

The lower bound can be established in a similar way. By choosing the integer i such that $\frac{\ln n}{4n} < 2^{-i} \leq \frac{\ln n}{2n}$ we can ensure that $P(T_x \leq c \ln n) \cdot A(c \ln n) = o(1)$ for some positive constant c and the result follows from an argument similar to the the one in the upper bound analysis. □

5 Comments

The way the branching factors are chosen in a level-compressed trie is sound; use a high degree only when it pays off. In particular, we can not imagine any situation where a binary trie behaves better than the LC-trie, at least for static structures.

A well known method to shorten the paths in a binary trie is to compress long single paths into single nodes, resulting in a *Patricia tree* [7]. The path compression is performed by removing each binary node with an empty child. In each internal node, we store an index indicating the bit used for branching. However, the average depth of a Patricia tree is still $\Omega(\log n)$. The behaviour of Patricia trees for the uniform distribution has been studied extensively [5, 9, 10, 11]. As we have shown, the level compression technique gives a much better improvement in search time than the Patricia technique, at least for uniformly distributed data. Of course, path compression and level compression can be combined into a compact, simple, and efficient data structure.

6 Acknowledgements

We would like to thank the anonymous referee for much helpful advice.

References

[1] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[2] L. Devroye. Laws of the iterated logarithm for order statistics of uniform spacings. *The Annals of Probability*, 9(5):860–867, 1981.

[3] L. Devroye. A note on the average depth of tries. *Computing*, 28:367–371, 1982.

[4] Ph. Flajolet. On the performace evaluation of extendible hashing and trie searching. *Acta Informatica*, 20:345–369, 1983.

[5] Ph. Flajolet. Digital search trees revisited. *SIAM Journal on Computing*, 15(3):748–767, 1986.

[6] E. H. Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

[7] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1991.

[8] T. Hagerup and C. Rüb. A guided tour of chernoff bounds. *Information Processing Letters*, 33(6):305–308, 1990.

[9] P. Kirschenhofer and H. Prodinger. Some further results on digital search trees. In *Proc. 13th ICALP*, pages 177–185. Springer Verlag, 1986. Lecture Notes in Computer Science vol. 26.

[10] P. Kirschenhofer, H. Prodinger, and W. Szpankowski. On the balance property of patricia tries: External path length viewpoint. *Theoretical Computer Science*, 68:1–17, 1989.

[11] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[12] B. Pittel. Paths in a random digital tree: limiting distributions. *Advances in Applied Probability*, 18:139–155, 1986.

[13] E. Slud. Entropy and maximal spacings for random partitions. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, 41:341–352, 1978.

[14] R. E. Tarjan and A. C.-C. Yao. Storing a sparse table. *Communications of the ACM*, 22(11), 1979.

[15] W. A. Whitworth. *Choice and Chance*. Cambridge University Press, 1897.

[16] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17:81–84, 1983.

[17] D. E. Willard. New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28:379–394, 1984.