A Linear Lower Bound on Index Size for Text Retrieval

Erik D. Demaine* Alejandro López-Ortiz†

Abstract

Most information-retrieval systems preprocess the data to produce an auxiliary index structure. Empirically, it has been observed that there is a tradeoff between query response time and the size of the index. When indexing a large corpus, such as the web, the size of the index is an important consideration. In this case it would be ideal to produce an index that is substantially smaller than the text.

In this work we prove a linear lower bound on the size of any index that reports the location (if any) of a substring in the text in time proportional to the length of the pattern. In other words, an index supporting linear-time substring searches requires about as much space as the original text. Here "time" is measured in the number of bit probes to the text; an arbitrary amount of computation may be done on an arbitrary amount of the index. Our lower bound applies to inverted word indices as well.

1 Introduction

Text retrieval is crucial in such contexts as searching the web, news, and medical databases. The most basic problem, used as a subroutine in most search engines, is to search for a given substring (keyword or phrase) in a corpus of text. Because the text database changes infrequently relative to the frequency and abundancy of queries, fundamental to any search technique is a preprocessing step to prepare an *index* for fast searches.

It has been observed empirically that the larger an index the better the query time. To illustrate with an extreme example, in the absence of an index, it is necessary to examine the entire text to see if the query string is present. For example, the Knuth-Morris-Pratt algorithm [KMP77] requires no index and runs in time proportional to the length of the text plus the pattern. In practice such a search is done often with the UNIX utility `grep`.

On the other end of the spectrum, a query can be answered in time that is linear in the length of the pattern. This bound is obtained by the popular *suffix tree* data structure [Wei73, McC76, Ukk95, GK97]. The index consists of $O(n)$ words, where n is the number of bits in the text. Recently, Grossi and Vitter [GV00] have shown that the space can be improved to $O(n)$ bits (proportional to the size of the text) for a slight sacrifice in query time, an additive factor of $O(\lg^\epsilon n)$.

An important but relatively unstudied field of research is to determine the asymptotic tradeoff between fast queries and a small index. As time is currently the most important issue in most systems, a natural problem in this field is to determine the minimum amount of space required by an index supporting linear-time substring searches. In this paper we prove the first nontrivial lower bound on this problem. Specifically, we show that any such indexing structure requires space proportional to the text itself. This bound applies in a powerful bit-probe model in which the search algorithm has free access to the *entire* index and can perform unlimited computation—only probes to the text are charged. We also show that the same lower bound holds in the practical case of *inverted word indices*, where the text is broken into n/w allowed query "words" each of length w.

1.1 Model of Computation. More formally, we consider the problem of determining the location, if any, of a given query string P (the *pattern*) in the text string T, in time bounded in terms of the length of the pattern, $|P|$, using an index I. The index is a static structure precomputed before query time, and for the purposes of lower bounds we do not consider the preprocessing time needed to build I. The search algorithm has access to P, T, and I, and the standard model is that it can perform $O(|P|)$ total internal computations and accesses (probes) to P, T, and I. We impose one restriction to this model, that each probe to the text T only retrieves a single bit from T.

Our lower bound applies to a much stronger model. The search algorithm is allowed an unlimited amount of computation, and can read the entire index I and pattern P; we only count the number of bit probes made to the text T. This model is stronger than

*Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, email: eddemaine@uwaterloo.ca.

†Faculty of Computer Science, University of New Brunswick, P. O. Box 4400, Fredericton, N. B. E3B 5A3, Canada, email: alopezo@unb.ca. Supported by NSERC.

the powerful bit-probe model [BMRS00, Mil93, Yao81], which in particular is stronger than a RAM in which probes to the text are restricted to bit probes. Our lower bounds of course apply to these models as well.

2 Main Result

In this section we prove the following theorem under the preceding model of computation:

THEOREM 2.1. *If I is an index supporting a search for the substring P in $o(\lg^2 |P|)$ bit probes to the text T, then $|I| = \Omega(|T|)$. This bound applies even when all query substrings P have length roughly $\lg |T|$, and do not overlap.*

See Theorem 2.2 on page 5 for a precise statement of the constant factor.

2.1 Construction. Consider a random permutation of the integers $0, \ldots, n-1$, random in the Kolmogorov sense [LV97], i.e., the Kolmogorov complexity of the permutation is $\lg n! - O(1)$. From such a permutation we construct the text T by writing the numbers in binary, using exactly $\lg n$ bits each, in the order given by the permutation, and terminating each binary representation with a special character #. For example, for $n = 16$, the permutation $5, 12, 1, \ldots, 15, 10$ would generate the string

$$T = 0101\#1100\#0001\# \ldots \#1111\#1010\#.$$

Define $N = |T| = n(\lg n + 1)$.

Note that while we use a ternary alphabet for convenience, this can easily be avoided by a standard trick. Our only requirement is that the substrings such as 0101# appear exactly once in the text. Thus, we can replace the # symbol with the special string 1^k, and modify all previously existing occurrences of the prefix 1^{k-1} to $1^{k-1}0$. The original string T is easily recovered from this new string. Because each binary string occurs with essentially uniform probability throughout T, the length of the new string is N for the original T, plus kn for the total length of the separators, plus roughly $N/2^{k-1}$ for the extra 0's added to occurrences of 1^{k-1}. Choosing $k = 1 + \lg \ln n$ minimizes the length of the new string, achieving

$$N\left(1 + \frac{\lg \ln N - \frac{3}{2}}{\ln N}\right) = N + o(N) \text{ bits.}$$

In the remainder of the proof, we assume that the alphabet is ternary for simplicity of presentation.

2.2 Proof Outline. Because the permutation and hence the text T has Kolmogorov complexity $\lg n! - O(1) = n \lg n - O(n)$ by Stirling's approximation, any encoding of T requires at least $N - O(N/\lg N)$ bits.[1]

The heart of the proof is to show that the text T can be reconstructed from the index I plus two small auxiliary structures A and B. That is, we will construct structures A and B so that, together, I, A, and B form an encoding of T. We can combine I, A, and B into one string by first encoding the length of I, so that I and AB can later be partitioned. (Separating A and B will be easy.) By the Kolmogorov argument above, this proves that

$$\lg |I| + |I| + |A| + |B| \geq N - O(N/\lg N). \quad (2.1)$$

Thus, if we can build A and B sufficiently small, say $o(N)$, we obtain a lower bound of $N - o(N)$ on the size of I. But note that A and B do not even have to be sublinear in the size N of the text. Kolmogorov complexity gives us a tight bound on the encoding size, specifying that the leading constant on N is 1. Thus, provided $|A| + |B| \leq cN$ for some constant $c < 1$, we obtain a linear lower bound on $|I|$, namely $|I| \geq (1 - c)N - O(N/\lg N)$.

2.3 Ideas for Encoding. If the search algorithm made no probes to the text T, it follows trivially that the index I encodes all positional information for each pattern. Specifically, suppose there were a search algorithm Search that computes the location of a pattern P using only the index I. Then the original text T can be reconstructed by sequentially querying all patterns, as follows:

Algorithm Decoding

- For P from $00\cdots0\#$ to $11\cdots1\#$:
  1. Let b denote the beginning of the occurrence of the pattern P found by Search (P).
  2. Write the pattern P into positions $b, b + 1, \ldots, b + \lg n$ of the reconstructed text T.

Of course, our task is not this easy: the query algorithm can perform as many as $c|P|$ probes to T for some constant c. We can think of the Search algorithm as having the following outline:

Generic Algorithm Search (P, I, T)

1. Initialize our "knowledge" to the pattern P and index I.
2. For i from 1 up to at most $c|P|$,

[1] Although formally there is no difference between $+O(f)$ and $-O(f)$, we find $-O(f)$ more suggestive in these situations.

(a) Using current knowledge, compute which bit to next probe the text T, call the position $p_{P,i}$.

(b) Probe $T[p_{P,i}]$, and add it to our knowledge.

Here we have isolated the probes to the text. Our goal is to encode in auxiliary structures the probed bits $T[p_{P,1}], \ldots, T[p_{P,c|P|}]$, for if these were somehow known for every pattern P, the Search algorithm could run without access to T, so we could recover the text by applying Algorithm Decoding. Note that we do not have to encode where the Search algorithm probes (the $p_{P,i}$'s). One trivial option would be to simply write down the bits in the order they would be probed when executing Decoding, i.e., store the string

$$T[p_{00\cdots0,1}]T[p_{00\cdots0,2}]\cdots T[p_{00\cdots0,c|P|}]\cdots\cdots T[p_{11\cdots1,\lg n}]$$

as an auxiliary structure. (Define $p_{P,i}$ arbitrarily if fewer than i probes were made during the call to Search (P, I, T).) Of course, this auxiliary structure is too large, taking $cn \lg n = cN$ bits.

Slightly more sophisticated is to encode just the *new* probed bits. In other words, the encoding algorithm for T simulates the execution of Decoding, and whenever its subroutine Search probes position p, the encoder first checks whether it has already encoded the bit $T[p]$. If so, the encoder simply omits the bit, with no explicit code; otherwise, the bit is simply written in the encoded string. The decoder, on the other hand, has perfect knowledge of what bits it has probed so far, so whenever it comes across a new bit to probe, it plays the same game: if it has never probed that bit before, it reads the next unread bit in the encoded string, taking that as the probe result; and otherwise, it uses its knowledge of the past to determine the result of the probe. This scheme is somewhat more efficient than the trivial one above, although still insufficient; however, this idea will be used heavily in our encoding.

2.4 Our Encoding Scheme. We use two auxiliary structures, A and B, to efficiently encode the probed bits during the simulated execution of Decoding. The basic strategy is that when searching for a pattern P, we decide whether to record in A the new bits probed in T. After all queries have been made and A has been filled, any remaining probed bits in T that remain unrecorded will be recorded in B.

More precisely, the auxiliary structure A is a table with n rows, one per pattern from $00\cdots0$ to $11\cdots1$. Algorithm Encoding (detailed below) loops over the patterns, following a simulated execution of Decoding. Each row of table A contains a bit specifying whether the row is "present." If it is present, the row effectively records the bits probed by Search (P, I, T) that were not already probed (as in the second method above). The idea is that we will record the probed bits *if* they will give us a significant bonus in implicitly revealing other bits. The advantage of recording the probe values for a Search is that the decoder can later simulate the execution of Search and actually determine the location of the pattern P: this reveals $\lg n + 1$ bits of the text to be the pattern followed by a # symbol. Some of these bits may already be known to the decoder, but some may not. If a constant fraction is not known, say at least $r \lg n$ out of the $\lg n + 1$ bits are newly revealed, we make the row of A present. We will optimize the constant r ($0 \le r \le 1$) later.

If a row is absent, it consists only of the presence bit, 0. If the row is present, it continues with two additional fields. Again, the goal of these fields is just to encode the bits probed by Search (P, I, T) in order to reveal the bits where the pattern P occurs. However, Search might probe a bit (or several) that is discovered, at the end of the search, to occur within the found instance of P. We need to avoid encoding these bits, for otherwise our "bonus" will be lost.

Thus, if Search probes a bit that occurs within the to-be-found instance of P, the first field of the row encodes the first probe with this property (encoded in binary using exactly $\lg(c \lg n)$ bits), followed by the distance from the beginning of the instance of P to the probed bit (encoded in binary using exactly $\lg(\lg n + 1)$ bits). Then the second field encodes any bits probed by Search that were not probed or revealed before and are not within the occurrence of the pattern P. On the other hand, the decoder reads the answers from probes made by Search using the second field, until it reaches the probed bit specified in the first field. Then it determines the location of the pattern P, and adds those bits to its knowledge. Finally, the decoder continues reading the second field as usual until reaching the next pattern P and row of A.

The simpler case is when Search does not probe any bit in the to-be-found instance of P. Then the first field just encodes a zero bit to state this fact, and the second field encodes all probed bits that were not already encoded in A.

After the simulated execution of Algorithm Decoding, some of the probed bits have been encoded in A, and other bits have been revealed by knowing the locations of patterns. Any unknown bits of T are now written into B, ordered left-to-right as they occur in T. Thus it is clear that by applying essentially the same process, we can decode A and B to reconstruct the original text T. The remaining question is how efficient our encoding is.

Algorithm Encoding (P)

- For P from $00\cdots0\#$ to $11\cdots1\#$:

  1. Apply Search $(P,\ I,\ T)$, let b denote the beginning of the occurrence of the pattern P, and let $p_{P,1},\ldots,p_{P,k}$ denote the probed bits in order.
  2. If more than $r\lg n$ of the bits $T[b]$, ..., $T[b+\lg n]$ are either marked known or were just probed by the search,
     - Write "0" (row of A is absent).
  3. Otherwise:
     (a) Write "1" (row of A is present).
     (b) If one of the probed bits is in between $T[b]$ and $T[b+\lg n]$ inclusive:
        i. Write "1" (bit of P was probed).
        ii. Let $p_{P,i}$ denote the first probed bit that is in between $T[b]$ and $T[b+\lg n]$.
        iii. Write $i-1$ in binary using exactly $\lg(c\lg n)$ bits.
        iv. Write $p_{P,i}-b$ in binary using exactly $\lg(\lg n+1)$ bits.
     (c) Otherwise, write "0" (no bit of P was probed).
     (d) For i from 1 to k:
        - If $T[p_{P,i}]$ is not in between $T[b]$ and $T[b+\lg n]$ inclusive, and $T[i]$ is not marked as known:
          i. Write $T[i]$ (as a new probed bit).
          ii. Mark $T[i]$ as known.
     (e) Mark the bits $T[b]$, ..., $T[b+\lg n]$ as known.

- For i from 1 to N,

  4. If $T[i]$ is not marked as known, write $T[i]$ (as a bit of B).

2.5 Size of Encoding. It may be surprising that our carefully worded encoding method actually saves space in the worst case over the simpler encoding algorithms, but we will show that the savings are significant. The basic idea is two-fold. First, as mentioned, if most of the bits in the instance of P are unknown, then it pays to encode the probe bits, because then we learn where P is and hence learn most of those $\lg n$ bits for free. Second, we will show that a significant portion of pattern instances will have this property, i.e. A will have many rows, inducing much savings.

Equation (2.1) tells us that the encoding of the triple (I,A,B) has size at least $n\lg n-O(n)$. Equivalently, we have

$$|I|+\lg|I| \geq n\lg n-|A|-|B|-O(n). \tag{2.2}$$

Now we need to estimate the sizes of A and B. Let a be the number of present rows in A, and let b be the number of absent rows in A. Thus we have

$$a+b=n. \tag{2.3}$$

Each row in table A consists of the present bit, sometimes (a out of n times) followed by the values of the newly probed bits optionally prefixed by some information about the location of P. Thus,

$$\begin{aligned} |A| &\leq n[2\text{ bits}]+a[\lg(c\lg n)+\lg(\lg n+1)\text{ bits}] \\ &+ [\text{number of probed bits in } A]. \end{aligned}$$

Applying this to Equation (2.2) gives us

$$\begin{aligned} |I|+\lg|I| &\geq n\lg n-(2n+2a\lg(\lg n+1)+a\lg c) \\ &- \underbrace{([\text{number of probed bits in } A]+|B|)} \\ &- O(n) \end{aligned} \tag{2.4}$$

The work so far has simply converted the encoding method into algebra. The important part that remains is marked with a brace in Equation (2.4): bound the total number of bits encoded explicitly in A and B, i.e., the number of probe bits encoded in A plus the entire size of B.

Consider any bit in the text T. The bit is within exactly one of the query patterns, say P. We distinguish two cases:

Case 1: The row of A corresponding to P is absent. There are $b(\lg n+1)$ such bits.

In this case, the bit is either encoded as a probe bit in another row of A, or it will be encoded in B. Indeed, this accounts for all bits encoded in B, and some of the probe bits encoded in A.

Case 2: The row of A corresponding to P is present.

In this case, the bit is implicitly encoded by the discovered location of P. This means that the bit will not be explicitly encoded in any future rows of A or in B. However, it may have already been encoded in an earlier row of A.

Fortunately, by the definition of A, we have a bound on the number of such bits. Namely, this row of A was chosen to be present because the number of already known bits of P was at most $r\lg n$. Thus, the number of bits of this type that are encoded as probe bits in A (over all present rows of A) is at most $ar\lg n$.

In total, we have the estimate

$$[\text{number of probed bits in } A] + |B| \leq b\lg n + b + ar\lg n.$$

Combining this equation with (2.3) and (2.4), we obtain

$$\begin{aligned} |I| + \lg|I| &\geq n\lg n - 2n - 2a\lg(\lg n + 1) - a\lg c \\ &\quad - b\lg n - b - ar\lg n - O(n) \\ &= (n - b - ar)\lg n - 2a\lg\lg n - a\lg c \\ &\quad - O(n) \\ &= (1-r)a\lg n - a(2\lg\lg n + \lg c) - O(n) \end{aligned} \tag{2.5}$$

Lastly, for this bound to be useful, we must bound a from below. To do this, we recount which of the $n\lg n$ bits of T are encoded where. On the one hand, each row of A encodes at most $c\lg n$ probe bits, plus implicitly encoding at most $\lg n + 1$ pattern bits. Thus,

$$\text{number of bits encoded by } A \leq a\Big((1+c)\lg n + 1\Big).$$

On the other hand, whenever a portion of a pattern P is encoded in B, at least $r\lg n$ of the bits were already explicitly or implicitly encoded in A, leaving at most $(1-r)\lg n + 1$ bits of the pattern to be encoded in B. Thus,

$$\text{number of bits encoded by } B = |B| \leq b\Big((1-r)\lg n + 1\Big).$$

Combining these two equations,

$$\begin{aligned} n\lg n &= \text{total number of bits encoded} \\ &\leq a(1+c)\lg n + b(1-r)\lg n + n. \end{aligned}$$

Dividing both sides by $\lg n$ and simplifying, we obtain

$$\begin{aligned} n &\leq a(1+c) + b(1-r) + n/\lg n \\ &= a(1+c) + (n-a)(1-r) + n/\lg n \\ &= a(c+r) + n(1-r) + n/\lg n \end{aligned}$$

Solving for a, we obtain the desired bound:

$$a \geq \frac{r}{c+r}\, n - n/\lg n.$$

Substituting this into Equation (2.5), we obtain

$$\begin{aligned} |I| + \lg|I| &\geq (1-r)\left(\frac{r}{c+r}\, n - n/\lg n\right)\lg n \\ &\quad - \left(\frac{r}{c+r}\, n - n/\lg n\right)(2\lg\lg n + \lg c) \\ &\quad - O(n) \\ &= \frac{r(1-r)}{c+r}\, n\lg n - \frac{r}{c+r}\, n(2\lg\lg n + \lg c) \\ &\quad - O(n) \end{aligned}$$

Focusing just on the lead $n\lg n$ term, this expression is maximized when $r = \sqrt{c(1+c)} - c$, giving us the bound

$$\begin{aligned} |I| + \lg|I| &\geq \left(1 + 2c - 2\sqrt{c(1+c)}\right) n\lg n \\ &\quad - \left(1 - \frac{\sqrt{c}}{\sqrt{1+c}}\right) n(2\lg\lg n + \lg c) \\ &\quad - O(n) \\ &= \left(1 + 2c - 2\sqrt{c(1+c)}\right) n\lg n \\ &\quad - O(n\lg\lg n + n\lg c) \end{aligned}$$

We have been careful to treat c as a variable, not as a constant. Thus, if the search algorithm is allowed $t(n)$ bit probes, we can define $c = t(n)/\lg n$, and obtain the following bound:

THEOREM 2.2. *If there is an algorithm for substring searches making $t(n)$ bit probes to a text of size $N = n\lg n$, then the following bound on the index size must hold:*

$$\begin{aligned} |I| &\geq \left(1 + 2\,\frac{t(n)}{\lg n} - 2\sqrt{\frac{t(n)}{\lg n}\left(1 + \frac{t(n)}{\lg n}\right)}\right) n\lg n \\ &\quad - O(n\lg\lg n + n\lg t(n)) \end{aligned}$$

For example, if $t(n) = \lg n$, we have

$$|I| \geq (3 - 2\sqrt{2})\, n\lg n.$$

COROLLARY 2.1. *If $t(n) = o(\lg^2 n)$, then $|I| = \Omega(n\lg n) = \Omega(N)$.*

3 Conclusion

This work starts an important area in text retrieval of determining the size of the smallest index to support fast substring searches. We have proved the first nontrivial lower bound on this problem. Namely, we have shown that if the search algorithm makes only $o(|P|^2)$ bit probes into the text for a query substring P, then the size of the index must be proportional to the text.

Our bound of $\Omega(|T|)$ on the size of the index is tight up to a constant factor in our model because probes to the index are free: a trivial strategy is to store an exact copy of the text in the index, and answer queries by scanning the index. However, it is natural to ask for the exact constant factor. It seems difficult to improve past this trivial strategy, so we conjecture that the constant is 1:

CONJECTURE 3.1. *For any search algorithm that makes $O(|P|)$ bit probes to the text T, the index must have size at least $|T| - o(|T|)$.*

Another open problem is how small the index can be if the algorithm is allowed $O(|P|)$ probes into the text, when a probe retrieves a *word* instead of a bit. Here we follow the standard model of a word being $\lg n$ bits or so. Our result comes close to answering this question, by showing that the index must have linear size even for $o(|P| \lg n)$ bit probes, none of which have to be consecutive. It seems that with $\lg n$ clusters of $\lg n$ consecutive bit probes, no improvement can be made, and again the size of the index must be linear in the size of the text.

Finally, there is work to be done on the upper-bound side. For a more reasonable model in which probes to the index are not free, is there a strategy that matches the trivial space bound of $|T|$ above, or even achieving $O(|T|)$ bits? For the text and queries used in the lower-bound proof in this paper, the optimal bounds can be achieved: an inverted word list (for words such as 0101#) can be stored in $|T|$ bits, and suffices to answer queries in $O(|P|)$ time (with no probes to the text). Can this bound be obtained in the more general setting of searching for arbitrary substrings instead of just words? Currently the best-known structures either take $O(|T|)$ words of space [Wei73, McC76, Ukk95, GK97], or take $O(|T|)$ bits of space but require an additional $O(\lg^{\epsilon} n)$ time for queries [GV00].

Acknowledgments

This work was initiated at a Schloss Dagstuhl seminar on Data Structures, organized by Susanne Albers, Ian Munro, and Peter Widmeyer. During the open problem session, Roberto Grossi posed the problem at hand. We thank Prosenjit Bose, Andrej Brodnik, Roberto Grossi, and Ian Munro for helpful discussions during the meeting.

References

[BMRS00] Harry Buhrman, Peter Bro Miltersen, Jaikumar Radhakrishnan, and Venkatesh Srinivasan. Are bitvectors optimal? In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, Portland, Oregon, May 2000.

[GK97] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[GV00] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, Portland, Oregon, May 2000.

[KMP77] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[LV97] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer-Verlag, New York, second edition, 1997.

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the Association for Computing Machinery*, 23(2):262–272, 1976.

[Mil93] Peter Bro Miltersen. The bit probe complexity measure revisited. In P. Enjalbert, A. Finkel, and K. W. Wagner, editors, *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, pages 662–671, Würzburg, February 1993.

[Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[Wei73] Peter Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Iowa City, Iowa, 1973.

[Yao81] Andrew Chi Chih Yao. Should tables be sorted? *Journal of the Association for Computing Machinery*, 28(3):615–628, 1981.