Master's thesis

Computer Science

Auxiliary data structures for improved suffix array query performance

Lauri Heino

November 30, 2020

Faculty of Science

University of Helsinki

Supervisor(s)

Dr. Simon J. Puglisi

Examiner(s)

Dr. Simon J. Puglisi, Prof. Veli Mäkinen

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Computer Science

Tekijä — Författare — Author

Lauri Heino

Työn nimi — Arbetets titel — Title

Auxiliary data structures for improved suffix array query performance

Ohjaajat — Handledare — Supervisors

Dr. Simon J. Puglisi

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	November 30, 2020	38 pages

Tiivistelmä — Referat — Abstract

The suffix array is a space-efficient data structure that provides fast access to all occurrences of a search pattern in a text. Typically suffix arrays are queried with algorithms based on binary search.

With a pre-computed index data structure that provides fast access to the relevant suffix array interval, querying can be sped-up, because the binary search process operates over a smaller interval. In this thesis a number different ways of implementing such an index data structure are studied, and the performance of each implementation is measured.

Our experiments show that with relatively small data structures, one can reduce suffix array query times by almost 50%. There is a trade-off between the size of the data structure and the speed-up potential it offers.

ACM Computing Classification System (CCS)
Applied computing → Document management and text processing → Document management → Document searching

Avainsanat — Nyckelord — Keywords

suffix arrays, data structures

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Algorithms specialisation line

Contents

1 Introduction

Suffix arrays were introduced in 1990 by Manber and Myers (1990) as a space-efficient data structure for fast string searching. Independently of them, Gonnet et al. (1992) proposed a similar idea as PAT arrays. One of the main purposes for these data structures is providing fast access to all the occurrences of any pattern in a text.

Suffix arrays are typically used in applications where a large number of different patterns need to be searched within a single text, which frequently arise in bioinformatics, for example (Shrestha et al., 2014; Mäkinen et al., 2015). In these kinds of situations pre-computed indexes are a practical solution.

While suffix arrays themselves are pre-computed index-like data structures, one can also perform further pre-processing on the suffix array itself to accelerate querying. For example, Manber and Myers proposed an augmenting data structure to speed-up querying a suffix array: querying with a straightforward binary search algorithm has a time complexity of $O(|P| \log |T|)$, but with the help of a pre-computed *LCP array* it can be brought down to $O(|P| + \log |T|)$. On the other hand, an LCP array takes twice the space of the suffix array, hence tripling the total size of the data structure – possibly defeating the key selling point of the suffix array. Moreover, constructing an LCP array takes a significant amount of time – almost as much as computing the suffix array itself.

In their original article, Manber and Myers also briefly mentioned another idea: storing a simple pre-computed *bucket array* which is just one quarter of the suffix array in size could help one accelerate query times.

That is also the topic of this thesis: our goal is to understand if suffix arrays could be augmented with a light-weight auxiliary data structure that is much smaller than an LCP array but that could speed-up and possibly provide other improvements for suffix array queries. More specifically, this thesis looks for a data structure that would hypothetically:

1. provide a substantial speed-up for suffix array queries by limiting the number of binary search iterations needed;
2. improve the data locality of the binary search; and
3. potentially reduce the memory footprint of the binary search by allowing us to read

only a part of the suffix array into the memory.

In this thesis we will ignore the possible effect that outputting the occurrences might have on the time or space complexity of the query algorithm as this will be the same for all techniques prestented here. Instead, we are only interested in finding the range in the suffix array that contains the occurrences of the pattern.

The input strings are treated as sequences of characters, and we will not consider the structure within the strings, directly. We also avoid the significant body of techniques on compressed suffix arrays, as these generally slow queries down (Navarro and Mäkinen, 2007).

1.1 Structure of this work

This thesis has been structured as follows:

- Chapter 2 will give an overview on suffix arrays, their properties, and the basic algorithms for querying them.
- Chapter 3 will propose four auxiliary data structures to speed-up querying.
- Chapter 4 will present the results of practical experiments on the performance of the data structures of Chapter 3.
- Chapter 5 will conclude this thesis and suggest future work.

1.2 Notation

We will use the following notation:

- An alphabet Σ of size σ is an ordered set of characters.
- A text on alphabet Σ is a sequence of characters $t_0, t_1, \ldots, t_k$, where $t_i \in \Sigma$. Individual characters of T can be referred to also as $T[i]$. We will also use the notation $T =$ "abc", where $t_0 = a$, $t_1 = b$ etc. In some contexts a text can also be called a string, and the two terms will be used interchangeably.

- The length of a text T is the number of characters in it and is denoted by $|T|$. For example, if $T =$ "abc", $|T| = 3$.
- A substring of a text T is a consecutive run of characters in T. We denote a substring starting at position i and ending at position $j-1$ of T by $T[i..j)$. For example, if $T =$ "$abcdef$", $T[1..4) =$ "bcd". Note that $T = T[0..|T|)$.
- A prefix of text T is a substring that starts at position 0, i.e. $T[0..j)$. For simplicity, we use notation $T[..j] = T[0..j)$ $(j < |T|)$.
- A suffix of text T is a substring that ends at position $|T|-1$, i.e. $T[i..|T|)$. For simplicity, we use notation $T[i..] = T[i..|T|)$ $(i < |T|)$.

In this thesis – and especially in our experiments – the alphabet Σ is the set of all byte-sized characters, i.e. $\sigma = 256$ unless otherwise stated.

1.3 Common definitions

In the following chapters we will be using some commonly known concepts, which we define as follows:

Definition 1.1 (Lexicographical order)**.** *Two strings $A = a_1a_2\cdots a_n$ and $B = b_1b_2\cdots b_m$ are lexicographically ordered, denoted by $S_1 < S_2$, if and only if $a_i < b_i$ for the smallest i such that $a_i \neq b_i$ (where $i < min\{n, m\}$) or, if there is no such i, if and only if $n < m$.*

Definition 1.2 (Longest common prefix)**.** *The longest common prefix of two strings S_1 and S_2, denoted by $lcp(S_1, S_2)$, is the greatest number i such that $S_1[..i) = S_2[..i)$ where $i \leq min(|S_1|, |S_2|)$.*

2 Suffix arrays

i	$T[i]$	$SA_T[i]$	$T[SA_T[i], ..)$
0	a	10	a
1	b	7	a b r a
2	r	0	a b r a c a d a b r a
3	a	3	a c a d a b r a
4	c	5	a d a b r a
5	a	8	b r a
6	d	1	b r a c a d a b r a
7	a	4	c a d a b r a
8	b	6	d a b r a
9	r	9	r a
10	a	2	r a c a d a b r a

Figure 2.1: The suffix array for **abracadabra**.

A suffix array is an array consisting of the starting positions of all the suffixes of a text, sorted according to their lexicographical order. By looking at the example in Figure 2.1 we can immediately see its practical property: all the matching substrings of the text are grouped together. We define the suffix array formally in Definition 2.1.

Definition 2.1 (Suffix array)**.** *A suffix array of text T, denoted by SA_T (or just SA when the context is clear), is an array of integers containing a permutation of $[0, n)$ such that $T[SA[0]..n) < T[SA[1]..n) < \cdots < T[SA[n-1]..n)$, where $n = |T|$.*

Note that even though technically the suffix array is an array of integers, we will say that it contains suffixes (that correspond to those integers), when speaking more informally.

2.1 Construction

A naive way to construct the suffix array SA_T is to initialize an array of length $|T|$ as $0, 1, |T| - 1$, and use any sorting algorithm to sort it in ascending lexicographical order of $T[SA[i]..|T|)$ and $T[SA[j]..|T|)$ when comparing $SA_T[i]$ and $SA_T[j]$. Since the comparison of two suffixes has a time complexity of $O(N)$ (where $N = |T|$), sorting with, for example,

the quick sort algorithm (Cormen et al., 2005, p. 145–150) would have an average-case time complexity of $O(N^2 \log N)$, which is very slow in practice.

However, there is a wide variety of more sophisticated construction algorithms. For example, the original Manber and Myers algorithm (Manber and Myers, 1993) has an average-case time complexity of $O(N \log N)$. More recently, linear-time construction algorithms have been proposed (Kärkkäinen et al., 2006; Li et al., 2018).

Many construction algorithms are surveyed for example by Puglisi, Smyth, and A. H. Turpin (2007).

2.2 Querying

In this thesis we will call the process of finding the suffix array interval corresponding to all occurrences of a pattern in a text *querying*. The literature around suffix arrays often deals with a slightly different problem: finding out if a pattern is a substring of the text. This problem is trivial to solve once we have the interval.

The natural process of searching a suffix array for a pattern uses binary search. The basic search algorithm, as presented, e.g., by Manber and Myers (Manber and Myers, 1993), accesses the text at the index pointed by the middle item of the suffix array search interval, compares the corresponding suffix to the search pattern, and rejects half of the search interval accordingly. The process is repeated until the middle item and the pattern match.

To find all the occurrences of a pattern, the algorithm can be modified to first find the first and then the last occurrence of the pattern, which produces an interval that corresponds to all the occurrences of the pattern. This basic process is explained in detail in Algorithm 2.1.

In Algorithm 2.1, steps 1 and 2 clearly do $\log_2 |T|$ loop iterations. In each one of those iterations at most $|P|$ character comparisons are required to determine which half of the interval to eliminate. Hence, the worst-case time complexity of the algorithm is $O(|P| \log |T|)$.

Algorithm 2.1 sets a baseline to which the other query algorithms in this thesis will be compared.

Algorithm 2.1: A simple binary search algorithm

SearchSA(P, T, SA_T) uses binary search to find the interval $[i, j]$ of the suffix array SA_T that corresponds to the matches of pattern P in text T. A binary search is run to find the lower bound i, followed by another binary search to find the upper bound j.

1. (Find lower bound.) Use binary search to find the smallest i such that $T[k, k + |P|) > p$ where $k = SA_T[i]$.
   (a) Initialise $l = 0$ and $r = |SA_T| - 1$.
   (b) While $l < r - 1$, set $m = \lfloor (l + r)/2 \rfloor$ and, if $T[SA_T[m]..) < P$, set $l = m$, or otherwise set $r = m$.
   (c) Set $i = r$.
2. (Find upper bound.) Use binary search to find the greatest j such that $T[SA_T[m]..) \leq P$ similarly to step 1, but this time set $r = m$ if $T[SA_T[m]..) > P$, and otherwise set $l = m$.
3. (Return.) If $i \leq j$, return interval $[i, j]$. Otherwise return `null`.

2.2.1 Utilising longest common prefixes

Binary search in a sorted array of strings can be optimized by the observation that if the bounds of our search interval start with matching characters, we know that all the strings within that interval start with those characters too (Hirschberg, 1978). Hence, we will benefit from maintaining the longest common prefixes of the bounds and the pattern, respectively.

Since a suffix array is effectively a sorted array of strings, the same technique can be applied (Manber and Myers, 1990): we can optimise Algorithm 2.1 by maintaining those longest common prefixes and, when comparing the middle item to the pattern, ignoring the first k characters, where k is the minimum of the two maintained longest common prefixes. Algorithm 2.2 explains this technique in detail.

Intuitively, Algorithm 2.2 might seem to have a worst-case time complexity of $O(|P| + \log |T|)$ as we are making at most one character comparison that does not increase the value of either lcp_l or lcp_r per iteration. However, Manber and Myers (1993) mention the

Algorithm 2.2: An improved binary search algorithm

SearchSAWithLCPs(P, T, SA_T) is similar to the Algorithm 2.1, but it avoids unnecessary character comparisons by maintaining the longest common prefixes between the bounds of the search interval and the pattern, respectively.

The process is similar, but in steps 1 and 2, we maintain $lcp_l = lcp(P, T[SA[l], \ldots)$ and $lcp_r = lcp(P, T[SA[r], \ldots)$. Now, for each iteration, $lcp_m = lcp(P, T[SA[m], \ldots)$ can be computed using the minimum of lcp_l and lcp_r, avoiding unnecessary character comparisons.

With lcp_m one can replace the string comparison required in steps 1(a) and 2 with a single character comparison between $T[SA_T[m] + lcp_m + 1]$ and $p[lcp_m + 1]$.

counter example of $T = ac^{|T|-2}b$ and $P = c^{|P-1|}b$ that shows its worst-case time complexity is still $O(|P| \log |T|)$. The example is easy to understand if $|T|$ is much larger than $|P|$, because then lcp_l – and consequently $\min(lcp_l, lcp_r)$ – remain at zero while the the middle item has all those cs ($|P| - 1$ of them) that will be matched against the pattern.

2.2.2 LCP arrays

i	$T[i]$	$SA_T[i]$	$lLCP_T$ (respective values of l and r in parentheses)	$rLCP_T$	$T[SA_T[i], ..)$
0	a	10	-	-	a
1	b	7	1 (0)	2 (2)	a b r a
2	r	0	1 (0)	0 (5)	a b r a c a d a b r a
3	a	3	1 (2)	0 (5)	a c a d a b r a
4	c	5	1 (3)	0 (5)	a d a b r a
5	a	8	0 (0)	0 (10)	b r a
6	d	1	2 (5)	0 (7)	b r a c a d a b r a
7	a	4	0 (5)	0 (10)	c a d a b r a
8	b	6	0 (7)	0 (10)	d a b r a
9	r	9	0 (8)	2 (10)	r a
10	a	2	-	-	r a c a d a b r a

Figure 2.2: The $lLCP$ and $rLCP$ arrays for **abracadabra**.

Manber and Myers (Manber and Myers, 1990) proposed the use of a pre-computed array of the longest common prefixes of adjacent suffixes as a solution to reach the worst-case time complexity of $O(|P| + \log |N|)$.

In a binary search in SA_T, there are $|T|-2$ possible middle points that can occur, and each one of those is a result of a specific combination of left and right boundaries. Therefore, it is enough if we pre-compute the longest common prefixes for those combinations. More specifically, if we store pre-computed LCPs of the middle points and left boundaries in an array called $lLCP$, and LCPs between middle points and right boundaries in $rLCP$, we can perform the binary search in a fashion similar to Algorithm 2.2 without the unnecessary character comparisons.

The obvious downside of the LCP arrays is their size, as they occupy twice the amount of space the corresponding suffix array consumes. With multi-gigabyte texts, this can make their use impractical.

LCP information of suffix arrays has been used in many different ways besides this approach (Abouelhoda et al., 2002; Kasai et al., 2001).

2.2.3 Biased binary search

Another technique to optimize suffix array querying is selecting the pivot item with a heuristic instead of blindly choosing the middlemost item.

A rather counter-intuitive idea by Hirschberg (1979) suggests that if the longest common prefixes lcp_l and lcp_r differ, we should choose a middle point that *closer* to the bound with a *smaller* LCP, or to be more specific:

$$m = \begin{cases} l + (r-l)/(1 + \max\{1.4, lcp_r - lcp_l\}) & \text{if } lcp_l < lcp_r \\ r - (r-l)/(1 + \max\{1.4, lcp_l - lcp_r\}) & \text{if } lcp_l > lcp_r \\ (l+r)/2 & \text{if } lcp_l = lcp_r \end{cases}$$

More recently, Kärkkäinen and Kempa (2018) have suggested the selection of the pivot item m by weighting the bounds with the difference of their corresponding LCPs (d) and the logarithm of $d+1$:

$$m = \begin{cases} (l \cdot d + r \cdot \log(d+1))/(d + \log(d+1)) & \text{if } d \equiv lcp_r - lcp_l > 0 \\ (r \cdot d + l \cdot \log(d+1))/(d + \log(d+1)) & \text{if } d \equiv lcp_l - lcp_r > 0 \\ (l+r)/2 & \text{if } lcp_l = lcp_r \end{cases}$$

Note that since these biasing techniques depend on the pattern, the use of pre-computed

LCPs becomes difficult due to the increased number of possible combinations of l, m and r that can occur.

3 Pre-computed suffix array indexes

A standard binary search starts the process of reducing the search interval from far away. For example, consider searching an English language novel for the string "*to be or not to be*" using a suffix array. A binary search algorithm would most probably spend quite a few iterations to find the range corresponding to suffixes starting with "*to be*".

Toward the end of their article, Manber and Myers (Manber and Myers, 1993) mention the idea of using a *bucket array* to limit the relevant search interval to speed up the search: for each of the $|\Sigma|^k$ possible k-character strings the index of the first suffix that is not smaller than it is computed and stored in a non-decreasing array, which effectively contains the suffix array interval (the "bucket") for each k-character string. One can now use the bucket array to find the stored interval for the first k characters of the search pattern and use that as the initial interval of the binary search.

While this idea might not provide any asymptotic benefit over a full binary search, it is worth investigating what kind of a practical effect it has on query times.

In this chapter we will be looking at different options to construct an index that would provide fast access to a relevant suffix array interval. We will start by looking at Manber and Myers's bucket array more closely and then propose three other data structures with a similar purpose.

3.1 k-character look-up array

The data structure that Manber and Myers call a bucket array is an integer array consisting of the smallest suffix array indexes for each k-character string. If a k-character string does not occur in the text, the smallest i such that $T[SA_T[i]..) >= P$ is used instead. For clarity, we will call this data structure the *k-character look-up array* and define it formally in Definition 3.1.

Definition 3.1 (k-character look-up array)**.** *A k-character look-up array, denoted by L, array built for suffix array text T and its suffix array SA_T is an array such that $L[P^*] = \arg\min_i T[SA_T[i], ..) >= P$, where $P \in \Sigma^k$ and P^* is an integer encoding of P that maintains the order of strings (e.g. $P^* = P[0] \cdot \sigma^{k-1} + ... + P[k-1] \cdot \sigma^0$).*

One can use the k-character look-up array to find the suffix arrays interval corresponding to suffixes starting with the given k characters in $O(k)$ time.

See Figure 3.3 for example look-up arrays. In the examples, the alphabet is assumed to be $\{\$, a, b, c, \ldots\}$, where \$ is a *null*-character that is added to allow handling of strings with fewer than k characters. \$ is the first character of the alphabet.

i	$T[i]$	$SA_T[i]$	$T[SA_T[i], ..)$
0	a	10	a
1	b	7	a b r a
2	r	0	a b r a c a d a b r a
3	a	3	a c a d a b r a
4	c	5	a d a b r a
5	a	8	b r a
6	d	1	b r a c a d a b r a
7	a	4	c a d a b r a
8	b	6	d a b r a
9	r	9	r a
10	a	2	r a c a d a b r a

$k = 1$:

\$	a	b	c	d	e	⋯	q	r	s	⋯
0	0	5	7	8	8	⋯	8	9	11	⋯

$k = 2$:

\$\$	a\$	aa	ab	ac	ad	ae	⋯	bq	br	bs	⋯
0	0	0	1	3	4	4	⋯	4	5	5	⋯

Figure 3.1: The suffix array for **abracadabra** and corresponding look-up arrays for $k = 1$ and $k = 2$.

3.1.1 Construction algorithm

A k-character look-up array can be constructed with a simple procedure explained in Algorithm 3.1. Interestingly, it can be constructed without the corresponding suffix array. This is because of the observation that one just needs to count all the occurrences of all k-character strings into an array and then compute the cumulative sum of the counts up to each element of the array, respectively.

An important aspect in its construction is the integer encoding of the k-character strings that is used. It needs to maintain the order of the strings and guarantee that two consecutive strings are mapped to two consecutive integers.

Algorithm 3.1 processes the text from left to right twice, and hence, its time complexity is clearly $O(n)$ assuming that in step 2 we maintain a buffer of k characters that we can update in $O(1)$ when we process the next character.

Algorithm 3.1: A construction algorithm for the k-character look-up array.

ConstructLUA(T, k) builds a k-character look-up array by counting the occurrences of all substrings of length k and effectively turning them into suffix array ranges.

1. (Initialise data structure.) Allocate an array with 256^k entries for the look-up array and initialise it with zeros. Call this array LU_T.

2. (Count all prefixes.) For each i from 0 to $|T| - 1$, increase $LU_T[j]$ by one, where $j = T[i] * \sigma^{k-1} + \ldots + T[i + k - 1] * \sigma^0$. Use $T[i] = null$ if $i >= |T|$.

3. (Transform counts into minimum indexes.) Initialise $c = LU_T[0]$, set $LU_T[0] = 0$, and for each i from 1 to $|T| - k$, set $LU_T[i] = c$ and $c = c + LU_T[i]$. Now each entry of the look-up array holds the minimal possible index of the corresponding k-character string in the suffix array.

3.1.2 Look-up algorithm

To find the suffix array interval that corresponds to the the suffixes prefixed with $P \in \Sigma^k$ from a k-character look-up array L, we can just read the indexes stored at $L[P^*]$ and $L[P^* + 1]$. The relevant suffix is range is $[L[P^*], L[P^* + 1] - 1]$ (see Algorithm 3.2). The time complexity of the look-up operation depends on the integer encoding used, but if we assume the encoding can be done in $O(k)$, the time complexity of the look-up operation is also $O(k)$.

Algorithm 3.2: A look-up algorithm for the k-character look-up array.

LookUpLUA(L_T, P, k) returns the suffix array range that contains the suffixes starting with the first k characters of P. This range, $[l, r]$, can be computed by reading $l = L_T[P^*]$ and $r = L_T[P^* + 1] - 1$, where i is the integer encoding of P used in Algorithm 3.1.

3.1.3 Size

A k-character look-up array consists of σ^k integers. In practice, this means that the size of the k-character look-up array grows very quickly to an impractical level (see Figure 3.1).

k	Size of the k-character look-up array
1	1 kB
2	262 kB
3	67.1 MB
4	17.2 GB

Table 3.1: Sizes of k-character look-up arrays with one-byte characters and four-byte suffix array indexes.

Inspecting the density of look-up arrays built for some of texts, a great downside can be observed. For example, in a 3-character array built from 1 gigabyte of English text, only less than 1% of the entries are distinct. See Table 3.2 for the proportion of entries that are not equal to their successors in example data sets (which will be discussed in detail in Section 4.1). These entries correspond to non-empty suffix array ranges.

Input file	Fill rate ($k = 1$)	Fill rate ($k = 2$)	Fill rate ($k = 3$)
1 GB of English text	92.6%	23.7%	<1%
1 GB of genome data	1.6%	<1%	<1%

Table 3.2: Fill rates of look-up arrays. Note that an alphabet with $\sigma = 256$ is assumed for both input files.

3.2 k-character look-up hash table

The data sparsity of the k-character look-up array encourages us to improve the data structure used to store suffix array intervals. A natural iteration on the idea is to use another fast look-up structure that is more dense in nature, such a hash table.

The *k-character look-up hash table* is essentially similar to the k-character look-up array, but instead of an array, the ranges are stored in a hash table. If the hashing function and the fill rate are chosen with care, one could expect to end up with a data structure that can be accessed in $O(k)$ time and that does not waste space for empty intervals.

See figure 3.2 for an example.

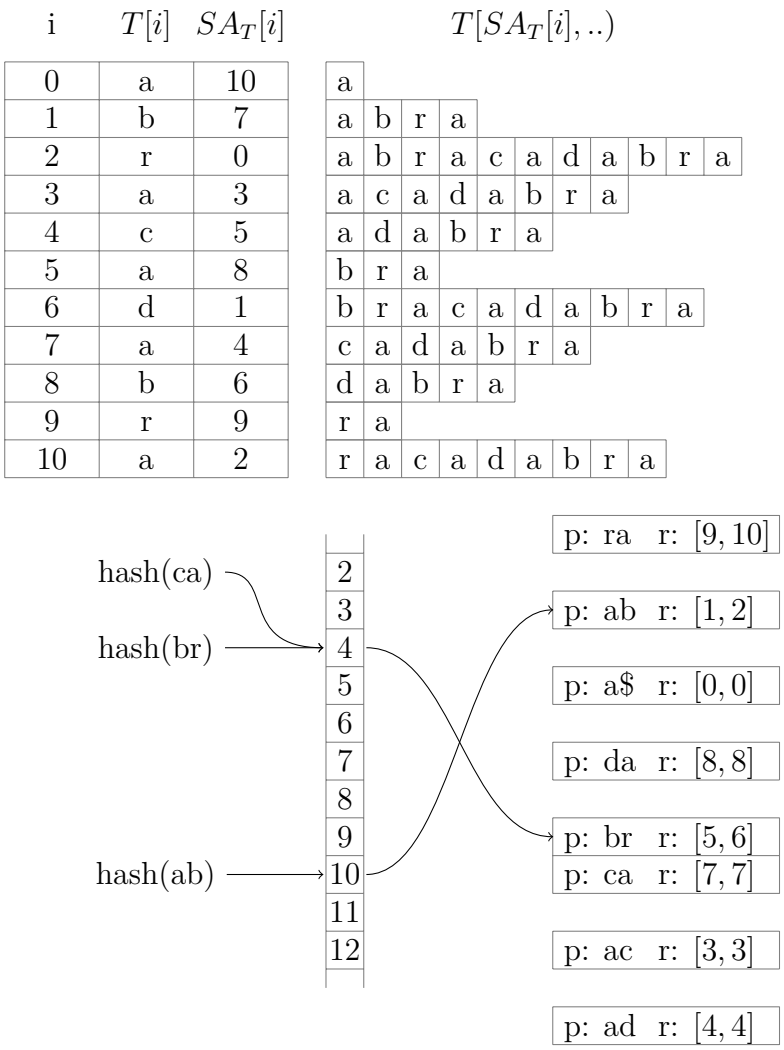


Figure 3.2: The suffix array and a look-up hash table for **abracadabra**.

3.2.1 Construction algorithm

The construction of a k-character look-up hash table requires counting of all the k-character strings in the text and transforming those counts into ranges via a prefix sum, which are then inserted into a hash table. The process of constructing a k-character look-up hash table has been explained in detail in Algorithm 3.3.

3.2.2 Look-up algorithm

A look-up to the k-character look-up hash table can be performed with Algorithm 3.4. Its performance depends greatly on the implementation of the hash table. Especially the

Algorithm 3.3: A construction algorithm for the k-character look-up hash table.

ConstructLUHT(T, k) builds a k-character look-up hash table by counting the occurrences of all substrings of length k and effectively turning them into suffix array ranges. These ranges are stored in a hash table.

1. (Count the occurrences.) Count the occurrences of all k-character substrings in text T. Store the counts in a data structure that allows us to process them in lexicographical order. E.g. a trie can be used to accomplish this.

2. (Build and store ranges.) Process the k-character substrings in lexicographical order while maintaining the cumulative sum of their counts. When processing substring P, the previous cumulative sum equals to the beginning, and next sum, subtracted by one, equals to the end of the corresponding suffix array range. Hash P with a selected hashing function and use the result to store the range in the hash table.

cost of hashing and the likelihood of collision affect the cost of a look-up operation.

Algorithm 3.4: A look-up algorithm for the k-character look-up hash table.

LookUpLUHT(L_T, P, k) returns the suffix array range that contains the suffixes starting with the first k characters of P. This range, $[l, r]$, can be computed by hashing P with the function that has been use to construct L_T and retrieve the value stored for that hash.

3.2.3 Size

Depending on the implementation details, each entry in the hash table needs to store the k-character string and two integers for the range. The total size of the data structure depends also on the hashing function and the collision resolution method. In Figure 3.2 chaining with linked lists (Cormen et al., 2005, p. 237 – 239) is used as the collision resolution mechanism.

The clear benefit of the look-up hash table over the simple look-up table is the density of the data: there are no data entries for non-existent prefixes and the only empty space is the space used for avoiding hash collisions.

3.3 Variable-k look-up trie

One of the weaknesses of the suffix array indexes presented in the previous sections is the fact that we spend an equal amount of space for each possible k-character string no matter how frequent or infrequent it is.

One possible way to address this is to use a trie-like data structure to store suffix array intervals for strings in a way that depends on their frequency. More specifically, we could store intervals so that they have a maximum size, denoted by b. This could be achieved by initially storing the intervals for all 1-character strings and by splitting those intervals recursively until they become smaller than the limit we have set. We call this data structure the *variable-k look-up trie*.

A variable-k look-up trie consists of nodes that have of entries for each character in the alphabet, and each entry stores the minimum position of the corresponding prefix in the suffix array (which effectively correspond to intervals in a way similar to the 1-character look-up array) and a pointer to a child look-up trie. At the root of trie, there is a node the entries of which represent the 1-character strings in the suffix array. If an entry that corresponds to a $(k-1)$-character string p has an interval that is bigger than b, it has a pointer to a child node the entries of which refer to k-character strings with a prefix p. See Figure 3.3 for an example look-up trie.

3.3.1 Construction algorithm

A variable-k look-up trie can be constructed with Algorithm 3.5 that splits the suffix array into intervals of at most b suffixes by recursively making the trie more and more granular.

3.3.2 Look-up algorithm

Give a variable-k look-up trie Algorithm 3.6 can be used to get an initial binary interval. The depth of the returned interval can be returned as an additional piece of information. Since the algorithm processes the pattern character-by-character to make decisions on which child node to proceed to, its time complexity is $O(|P|)$.

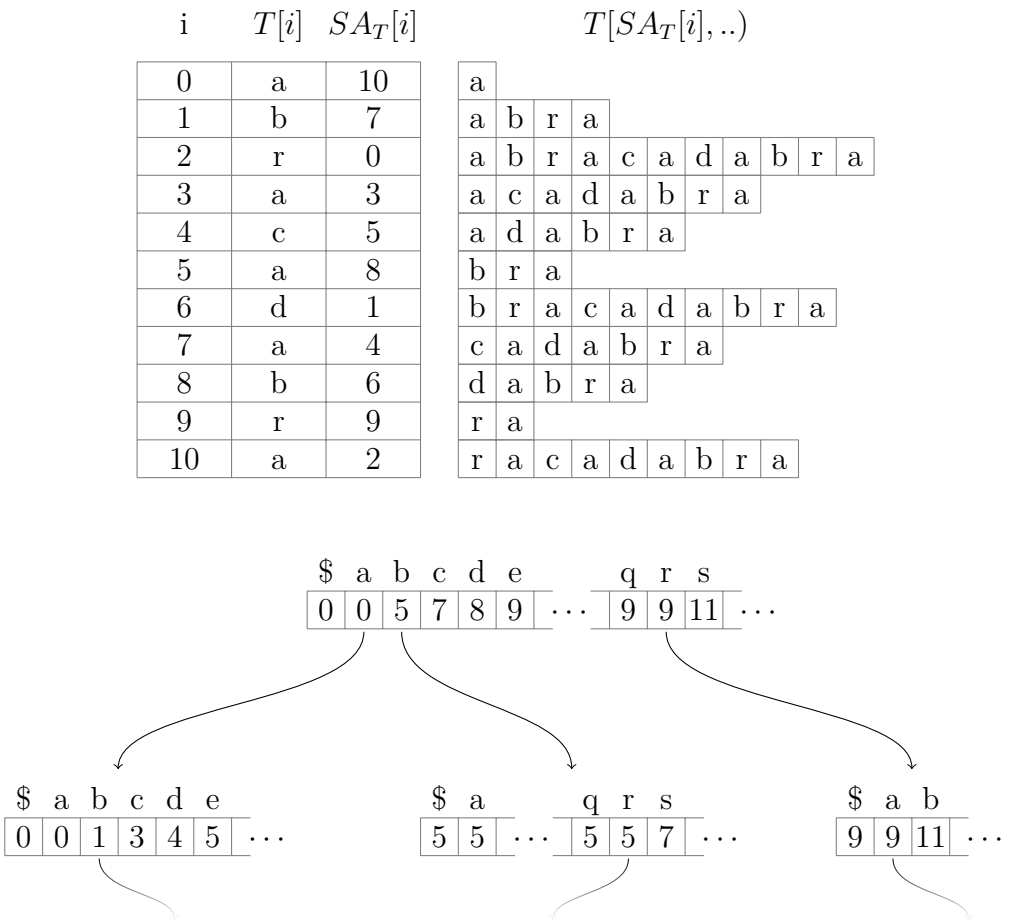


Figure 3.3: A variable-k look-up trie for **abracadabra** with $b = 1$.

3.3.3 Size

The minimum number leaf entries (entries without a child) in a variable-k look-up trie equals to $|T|/b$. If no space is wasted for empty entries, $|T|/(b \cdot \sigma)$ leaf nodes are needed to store them. On the other, for instance with $T = a^{n \cdot b}$ most of the nodes would have $\sigma - 1$ empty entries. Hence, we can say that the size of the trie greatly depends on the input text and the way it can be split into distinct intervals.

3.4 k-character prefix table

A completely different approach for building an index for a suffix array is sampling it at frequent intervals to provide a small data structure where one could perform the first most expensive iterations of the binary search. See Definition 3.2 for a formal definition and

Algorithm 3.5: A construction algorithm for the variable-k look-up trie.

ConstructLookUpTrie(SA_T, T, b) builds a variable-k look-up trie in a recursive manner.

1. (Initialisation.) We start computing the minimum positions of all 1-character strings in SA_T, but for each entry, we also store a pointer that is initialised as *null*. We store all this information into a *node*.

2. (Splitting.) If any of the ranges in the array are bigger than b items, we *split* those ranges by calling the same function in such a way that it will construct another 1-character look-up array but only for the corresponding interval and considering the characters positioned at $SA_T[T[i] + k]$, where i is a value within the interval being split and k is the depth of recursion we are at. Store a pointer to the result of this recursive call to the corresponding entry.

Figure 3.4 for an example.

Definition 3.2 (k-character prefix table)**.** *A k-character prefix table is a data structure that stores k-character prefixes of sampled suffix array entries. The parameters of a prefix table are the prefix length (later denoted by k) and the sampling interval (denoted by b). Prefixes are stored for SA positions 0, b, $2b$, ..., $(\lfloor n/b \rfloor - 1) \cdot b$ so that for a prefix table position $l = n \cdot k$, the entries l, $l + 1$, ..., $l + k - 1$ are occupied by the first k characters of the suffix represented by the suffix array entry $SA[n \cdot b]$.*

3.4.1 Construction algorithm

A k-character prefix table can be built by simply sampling the text at positions corresponding to regular suffix array positions (see Algorithm 3.7). The running time of Algorithm 3.7 is clearly $\Theta(kn/b)$ where n is the length of the suffix array.

3.4.2 Look-up algorithm

The prefix table can be searched with a binary search algorithm similar to what we have used in earlier sections for the suffix array. The goal of the search is to find the first entry which is smaller than the search pattern (l) and the last entry which is not greater than

Algorithm 3.6: A look-up algorithm for the variable-k look-up trie.

LookUpLUT(L_T, P) returns a suffix array range that contains (but is not limited to) occurrences of pattern P using look-up trie L_T. We also return the number of matched characters k which can be used to initialise the LCP values of the bounds for the binary search.

It starts by reading the entry for $P[0]$ in L_T.

- If the entry in L_T for $P[0]$ has been split (and $P > i$), we call the same function recursively with the corresponding child trie and $P[1..)$. If the recursive call returns range R and value k, we return R and $k + 1$.
- Otherwise we return $k = 1$ and the range $[l, r - 1]$ that can be computed by reading the position stored in the entry (l) and next item in the array (r). (If the entry is the last one in the array, we need to read the next item from the parent node.)

the pattern (r). The interval $[l, r]$, or to be more specific, the corresponding suffix array interval ($[l \cdot b, r \cdot (b+1) - 1]$) represents *buckets* that contain all the possible suffix matches. Algorithm 3.8 presents the process in detail. A binary search in a k-character prefix table has a time complexity of $O(k \log N)$ where $N = |T|/b$.

One can observe that the prefix table lookup offers no asymptotic advantage over a direct suffix array search. However, its hypothetical benefit stems from the idea that if one chooses the parameters k and b with care, the resulting array could fit into processor's cache, possibly leading to fewer cache misses compared to the direct search, which is likely to access large parts of the suffix array.

3.4.3 Size

The size of a prefix table is kn/b bytes (assuming one-byte characters).

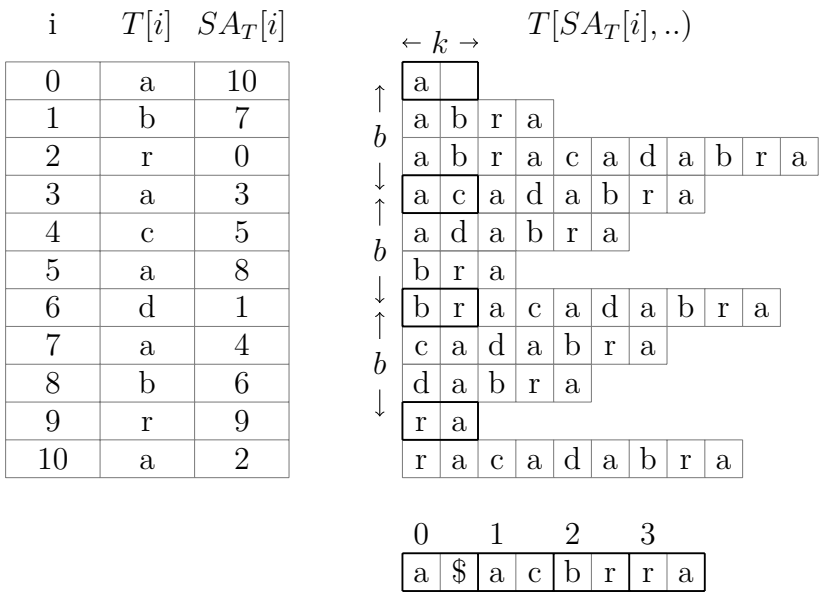


Figure 3.4: The 2-character prefix table for **abracadabra**, built with $b = 3$.

3.5 Using index data structures to speed-up suffix array querying

One can use the pre-computed data structures presented in this chapter to speed-up suffix array querying by using the intervals they return as the initial binary search interval. Additionally, each one of the data structures make it unnecessary to match some of the characters. To be more specific, with the k-character look-up array and hash table, one can initialise lcp_l and lcp_r values of Algorithm 2.2 as k. With the variable-k look-up trie, one can use the depth of the returned interval to initialise them, and finally, with the k-character prefix table, one can use the LCPs of the returned bounds and the pattern as initial values of lcp_l and lcp_r.

The overall procedure to query a suffix array with the help of an index data structure is explained in detail in Algorithm 3.9.

When the initial interval of the binary search is limited, one can expect improved data locality in the search algorithm. It is also worth mentioning that in Algorithm 3.9 one can limit the use of resources by reading only the the interval $[l, r]$ of the suffix array to memory, which can enable the use of the suffix array in resource-limited environments.

Algorithm 3.7: A construction algorithm for the k-character prefix table.

ConstructPrefixTable(k, b, T, SA_T) uses the sampling interval b and prefix length k to build a prefix table P for text T using its suffix array SA_T.

1. (Initialise data structure.) Allocate space for $k \cdot \lfloor n/b \rfloor$ characters for P.
2. (Fill the data structure.) For each i from 0 to $\lfloor n/b \rfloor - 1$, copy k characters, starting at position $SA_T[i \cdot b]$, from T to the next available slot in P, i.e. if $m = SA_T[i * b]$, set $P[ib, ib + k) = T[m, m + k)$.
3. (Return.) Return P.

Algorithm 3.8: A lookup algorithm for the k-character prefix table.

LookUpPrefixTable(k, b, PT_T, P, SA_T) searches the prefix table PT_T (built for it with parameters k and b) for pattern P. It returns an interval where one can expect to find all suffixes starting with P in the suffix array SA_T.

1. (Binary search.) Find l and r using a binary search procedure similar to Algorithm 2.2 but instead of accessing the text T, the comparisons are done between strings $PT_T[ib, ib + k)$ and $PT_T[jb, jb + k)$ when comparing positions i and j.
2. (Return.) Return the range $[l \cdot b, r \cdot (b + 1) - 1]$ and corresponding LCPs.

Algorithm 3.9: The suffix array query algorithm that uses index data structures.

Query(P, SA_T, T, L) returns the interval $[l, r]$ that corresponds to the occurrences of P in suffix array SA_T using the index data structure L.

1. (Index look-up.) Use the look-up method of L to find the initial interval $[l, r]$ for the binary search and the number of already possibly matched characters at the bounds of this initial range (lcp_l and lcp_r).
2. (Binary search.) Use the procedure presented in Algorithm 2.2 to find the interval in SA_T that corresponds to occurrences of P but initialise l and r and lcp_l and lcp_r according to the result of step 1 to avoid repeated character comparisons.

4 Experiments

In this chapter we will describe the results of the experiments that were run to compare the query performance of the basic algorithms presented in Chapter 2 and the speed-up the data structures presented in Chapter 3 provide.

4.1 Data sets

The experiments were performed on two data sets: an English language text and genome data. The English text consisted of concatenated texts from the Gutenberg project and prepared by the Pizza & Chili Corpus project[†]. A sample from the text looks like this:

> *He began his preparations by making huge bows and arrows without number, but he had no arrow heads. At last his grandmother, Noko, told him that an old man who lived at some distance could furnish him with some, and he sent her to get them. Though she returned with her wrapper full, he told her that he had not enough and sent her again for more.*

The genome data was the human reference genome (build 38, patch release 12) [‡]. All the header data was removed from the file, so that it only consisted of symbols `A`, `C`, `G` and `T`. A sample from the data looks like this:

```
ACTTCTGGGTATATAGCCAAAGGAATTGAAATCAATATGTCAAAGGGATATCTGCACTCCTATG
TTATTGCAGCATGTTCACAATGGCCAAGATATAGAATCAACCTAACTGTTCATAGACAGATGAA
TGGATAAATGAAATGTGATATGGAAAATTATTCAGCCTTAAAAACAGTAGGAAATTCTGTCATT
TGAGACAACGTGGATGAACCTAGAGGACATTAAGCTAAGTGAAATAAGCTAGACACAGAAAGAC
```

For both of the data sets, only the first gigabyte was used, and the size of the suffix array was 4 gigabytes.

[†] `http://pizzachili.dcc.uchile.cl/index.html`

[‡] `https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.38/`

4.2 Pattern generation

Sets of patterns were generated for each data set to gain a good understanding of the query performance with patterns of different sizes.

The pattern sets were generated using the Algorithm 4.1. The sets were stored in separate files so that each file contains n patterns of length k. Those parameters n and k were written in the beginning of each pattern file to simplify processing.

Patterns were generated for each data set with lengths $k = 8, 16, \ldots, 1024$. For each k, $n = 1000$ was used.

4.3 Computer environment

The experiments were run on the Google Cloud infrastructure using a compute-optimized `c2-standard-4`* machine which has 16 gigabytes of RAM and a 3.1 MHz Intel Xeon (Cascade Lake) processor with 32 kB (L1), 1 MB (L2) and 1.375 MB (L3) caches. The machine was running the Debian 9.0 operating system.

4.4 Software

To run experiments, all the algorithms and data structures presented in Chapter 2 and Chapter 3 were implemented using the C11 programming language and compiled using the *clang*† compiler. Google's *googletest*‡ framework was used to validate the functionality of the key parts of the source code.

The suffix arrays and the pattern sets were prepared on a laptop and transferred to Google Cloud's Cloud Storage service along with the data sets. Bash scripts were written to automate the process of downloading the required data from Cloud Storage to the machine, running the performance measurement, and uploading the result files in the CSV (comma-separated values) format back to Cloud Storage.

For each algorithm and data structure measured, 1,000 queries for each generated pattern was performed. For each algorithm and data structure, this totals in 1,0000,0000 queries

*`https://cloud.google.com/compute/docs/machine-types#c2_machine_types`
†`https://clang.llvm.org/`
‡`https://github.com/google/googletest`

for each pattern length k. To measure the total time to perform all those queries, the `clock()`* function declared in `time.h` was used.

Algorithm 4.1: The pattern generation algorithm.

GeneratePatterns(T, n, k) returns n randomly sampled patterns of length k from text T by randomizing n integers from range $[0, |T| - 1)$ and outputting the corresponding strings.

4.5 The effect of biasing and maintaining LCPs on query performance

First, we wanted to gain an understanding on how the basic optimisation techniques presented in Chapter 2 affect the suffix array query performance. We measured the query performance of four different algorithms (see Table 4.1).

The query times relative to Algorithm 2.1 are shown in Figure 4.1 and Figure 4.2. With both of the data sets, the LCP-aware algorithms perform clearly better than the Algorithm 2.1. A bit surprisingly, biasing does not seem to offer a performance benefit over the simpler LCP-aware Algorithm 2.2 with our data sets.

When comparing the results between the two versions of biasing, the both perform quite similarly but the one by Kärkkäinen and Kempa (2018) behaves more consistently than the one by Hirschberg (1979).

Name used in results	Algorithm
`simple`	Algorithm 2.1
`lcp-aware`	Algorithm 2.2
`biased-hirschberg`	Algorithm 2.2 with biasing by Hirschberg (1979)
`biased-kk`	Algorithm 2.2 with biasing of Kärkkäinen and Kempa (2018)

Table 4.1: The query algorithms that were compared.

*`https://www.gnu.org/software/libc/manual/html_node/CPU-Time.html`

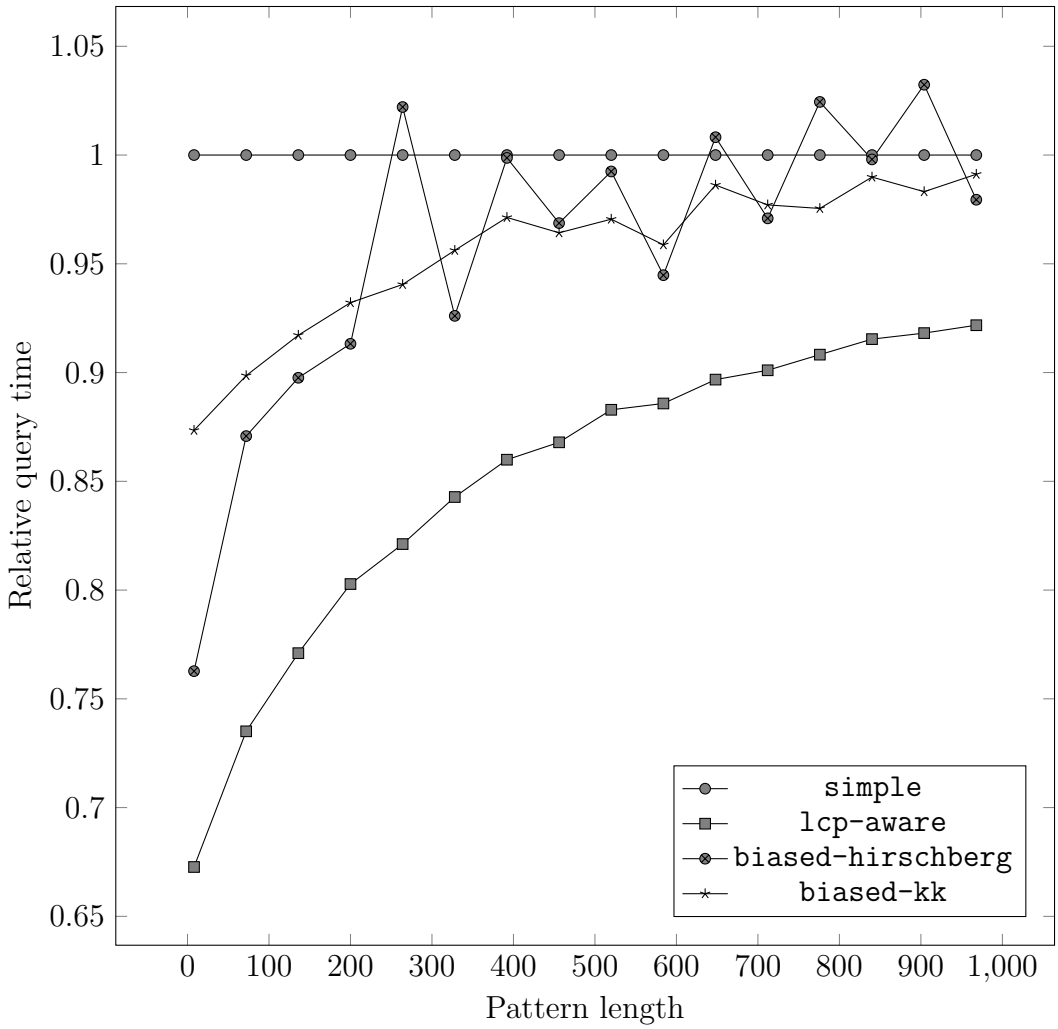


Figure 4.1: The relative query times with the English text data set.

4.6 Query performance using look-up data structures

To compare the suffix array query performance using the look-up data structures, Algorithm 3.9 was used.

With the original goal of a finding light-weight index data structure in mind, the size of the data structures was limited to 100 megabytes. For each data structure, we wanted to measure the effect of its parameters to understand the trade-off between the speed-up offered and the data structure size.

The data structure configurations used for the English text data set are listed in Table 4.2 and the ones used for the genome data set in Table 4.3.

Figures 4.3 and 4.4 show the relative running times of querying with the look-up data

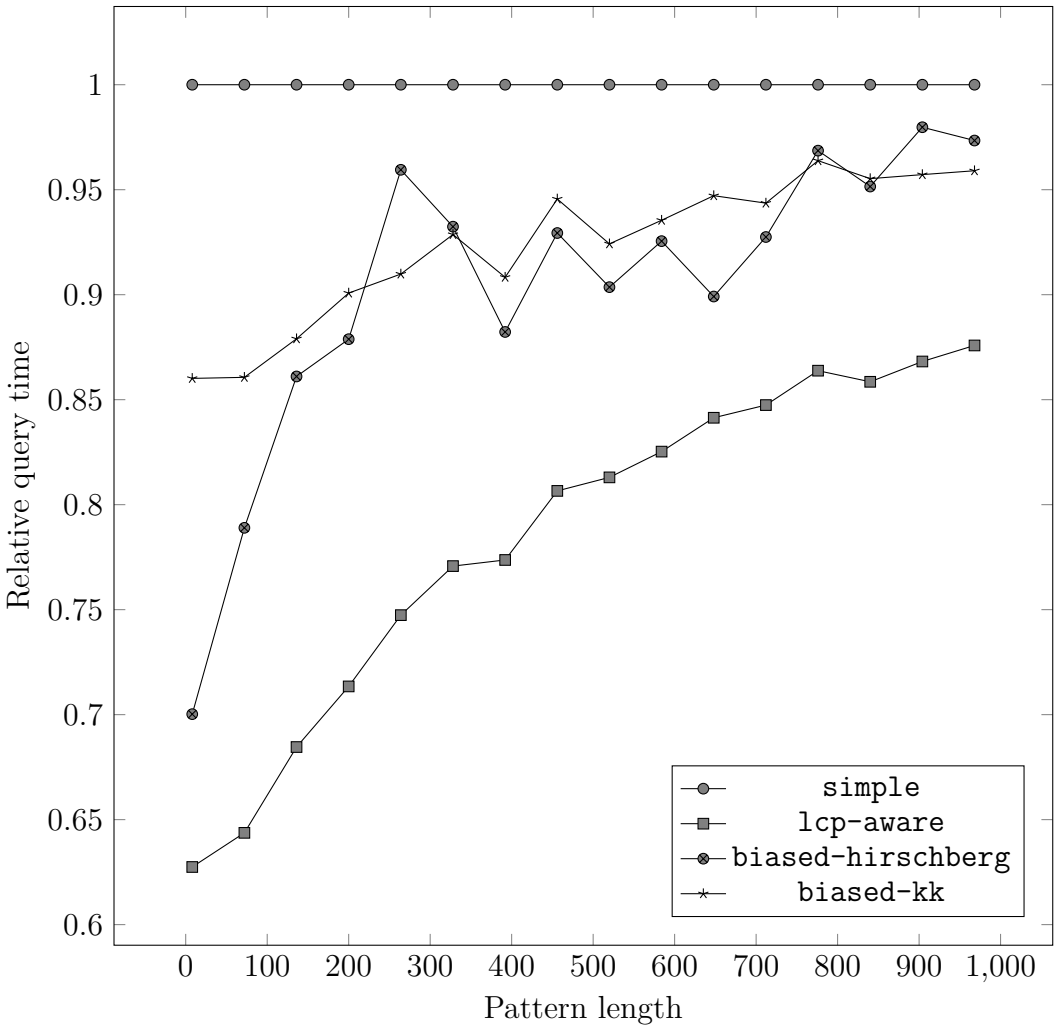


Figure 4.2: The relative query times with the genome data set.

structures.

When querying the English text, the results clearly show that the look-up tries perform better than the arrays and the hash tables. The only exception is the 5-character hash table, `LUHT-5`, which offers a speed-up similar to the look-up trie with the fewest nodes. On the other hand, the hash table is almost 8 times bigger.

When querying the genome data, the results are somewhat similar, but one interesting difference is that the 8- and 10-character look-up hash tables, `LUHT-8` and `LUHT-10`, provide bigger speed-ups than the look-up tries of similar sizes. This could be explained by the fact that the character distribution in the genome data can be expected to be much more even than English text, and hence, the look-up hash tables do not have the problem of spending bytes for infrequent k-character strings.

Name used in results	Data structure	Size	Notes
`LUA-2`	2-character look-up array	262 KB	
`LUA-3`	3-character look-up array	67.1 MB	
`LUHT-2`	2-character look-up hash table	279 KB	
`LUHT-3`	3-character look-up hash table	3.1 MB	
`LUHT-4`	4-character look-up hash table	20.7 MB	
`LUHT-5`	5-character look-up hash table	91.6 MB	
`LUT-65536`	variable-k look-up trie	12.5 MB	$b = 65,536$, 9,865 nodes
`LUT-32768`	variable-k look-up trie	57.4 MB	$b = 32,768$, 44,815 nodes
`LUT-12288`	variable-k look-up trie	77.9 MB	$b = 12,288$, 60,852 nodes

Table 4.2: The look-up data structures that were used for the English data set.

When comparing `LUA-2` and `LUA-3` to `LUHT-2` and `LUHT-3`, respectively, a somewhat expected observation can be made in the results with the English text: the look-up arrays are slightly faster. On the other hand, when comparing their sizes, the 2-character versions are very similar (both in the range of 260–280 kB) whereas the 3-character array is more than 20 times bigger than its hash table counterpart (67.1 MB vs. 3.1 MB). A similar phenomenon can observed in the results with the genome data.

When comparing the sizes of the data structures and the respective query times (Figure 4.5), there seems to be a clear trade-off between the space used for a look-up data structure and the speed-up it offers. The large look-up arrays (`LUA-3`) stick out as outliers that provide a rather small speed-up considering the size they consume.

The results give a clear signal that one should aim at a data structure that uses space in an efficient way. In many cases, the look-up trie is probably a good candidate, while with more evenly distributed data, a look-up hash table can be a feasible solution too.

4.7 Query performance using prefix tables

To measure the suffix array query performance using the k-character prefix table, a variety of configurations were used (see Table 4.4) with Algorithm 3.9. The results are presented

Name used in results	Data structure	Size	Notes
`LUA-2`	2-character look-up array	262 KB	
`LUA-3`	3-character look-up array	67.1 MB	
`LUHT-2`	2-character look-up hash table	326 B	
`LUHT-4`	4-character look-up hash table	5 kB	
`LUHT-6`	6-character look-up hash table	90 kB	
`LUHT-8`	8-character look-up hash table	1.6 MB	
`LUHT-10`	10-character look-up hash table	27.3 MB	
`LUT-65536`	variable-k look-up trie	16.7 MB	$b = 65,536$, 12,706 nodes
`LUT-32768`	variable-k look-up trie	31.7 MB	$b = 32,768$, 24,788 nodes
`LUT-12288`	variable-k look-up trie	93.2 MB	$b = 12,288$, 72,848 nodes

Table 4.3: The look-up data structures that were used for the genome data set.

in Figures 4.6 and 4.7.

The hypothesis for the benefit of the prefix table was that it could allow the first iterations of the binary search to be completed with fewer cache misses, and this is supported by the results that clearly indicate that the configurations smaller in size perform better. To understand the trade-off between the parameters k and b, three different 80-kilobyte configurations were compared. The results give a weak signal that higher values of k would yield better results.

However, none of the configurations were able to reach the query performance of the look-up data structures or even the plain Algorithm 2.2.

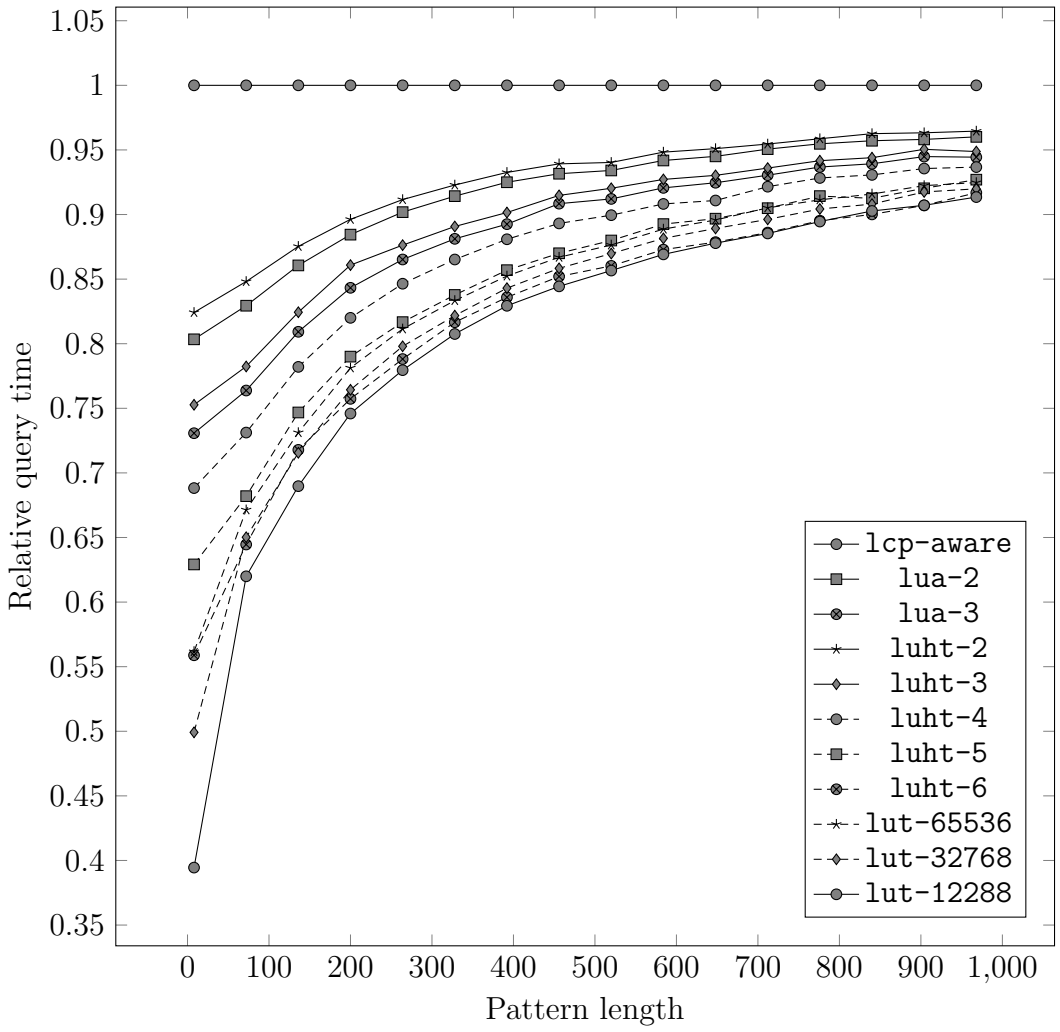


Figure 4.3: The relative query times using look-up arrays, hash tables and tries with the English text data set.

Name used in results	Data structure	b	Size
`PT-8-10000`	8-character prefix table	$10,000$	80 kB
`PT-16-5000`	16-character prefix table	$5,000$	80 kB
`PT-32-2500`	32-character prefix table	$2,500$	80 kB
`PT-8-100000`	8-character prefix table	$100,000$	800 kB
`PT-8-1000000`	8-character prefix table	$1,000,000$	8 MB
`PT-8-10000000`	8-character prefix table	$10,000,000$	80 MB

Table 4.4: The prefix table configurations that were used.

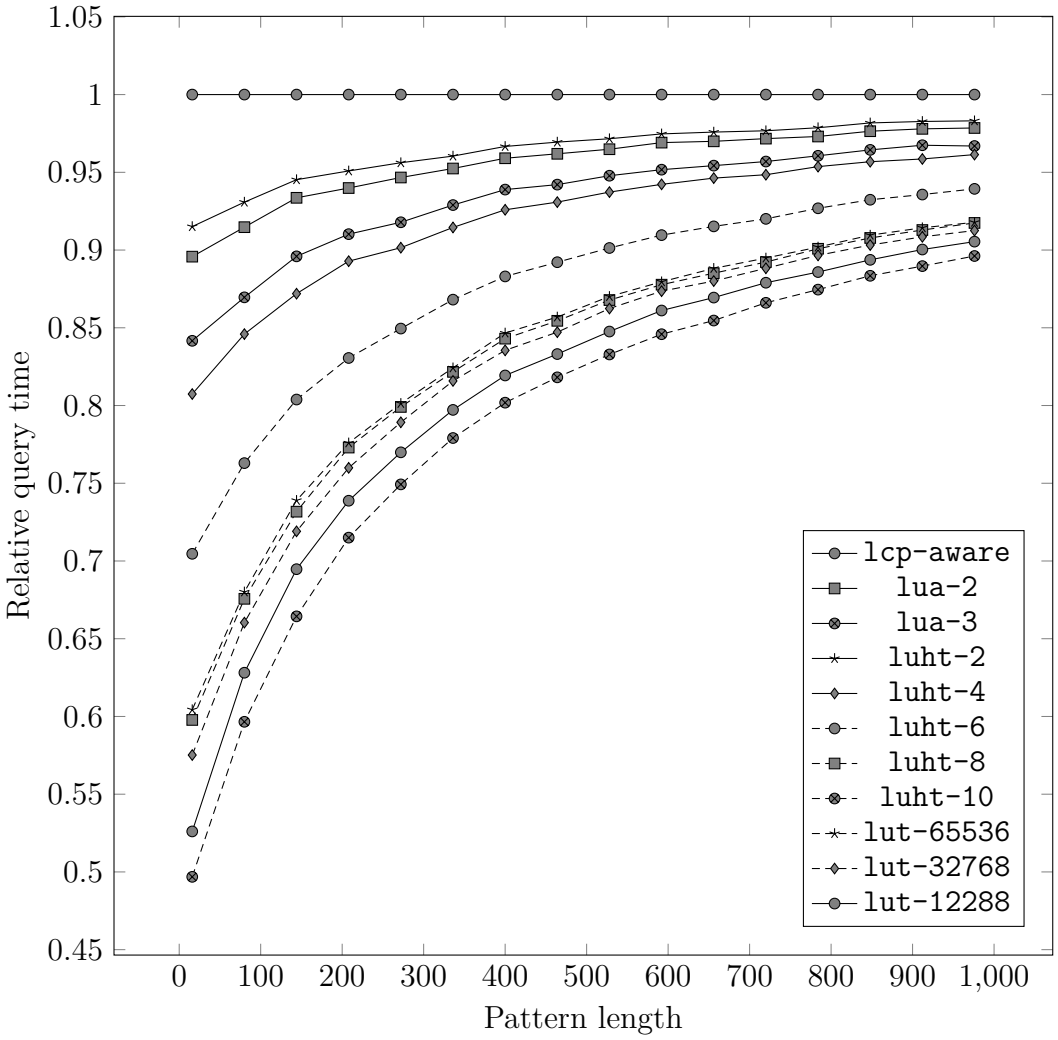
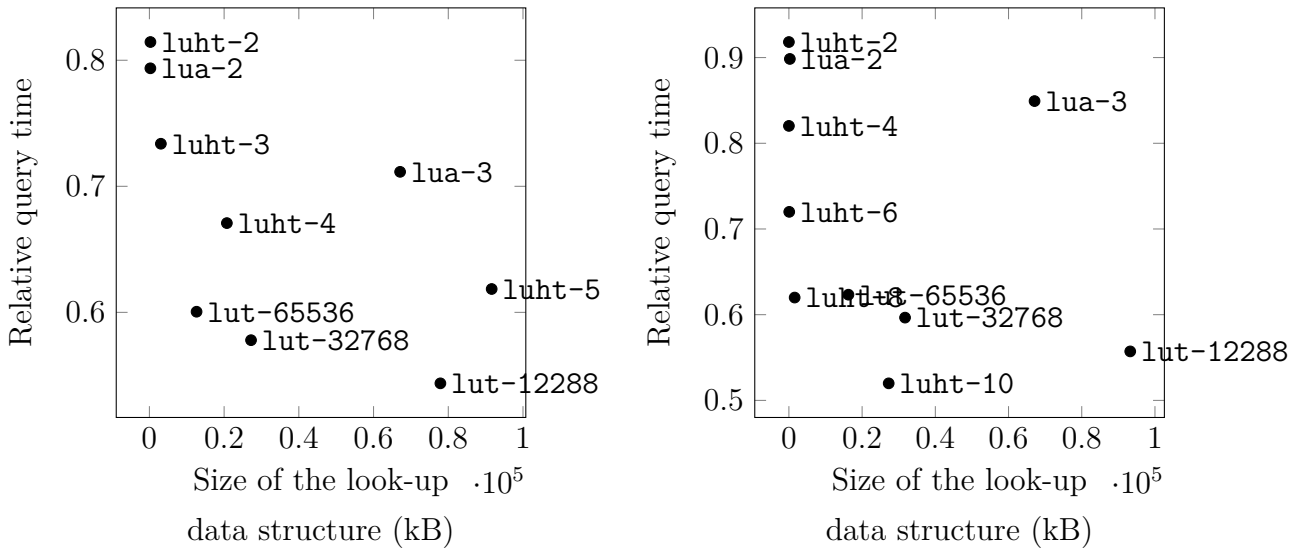


Figure 4.4: The relative query times using look-up arrays, hash tables and tries with the genome data set.



(a) English text

(b) Genome data

Figure 4.5: Query times relative to Algorithm 2.2 for a 32-character pattern versus the size of the look-up data structure.

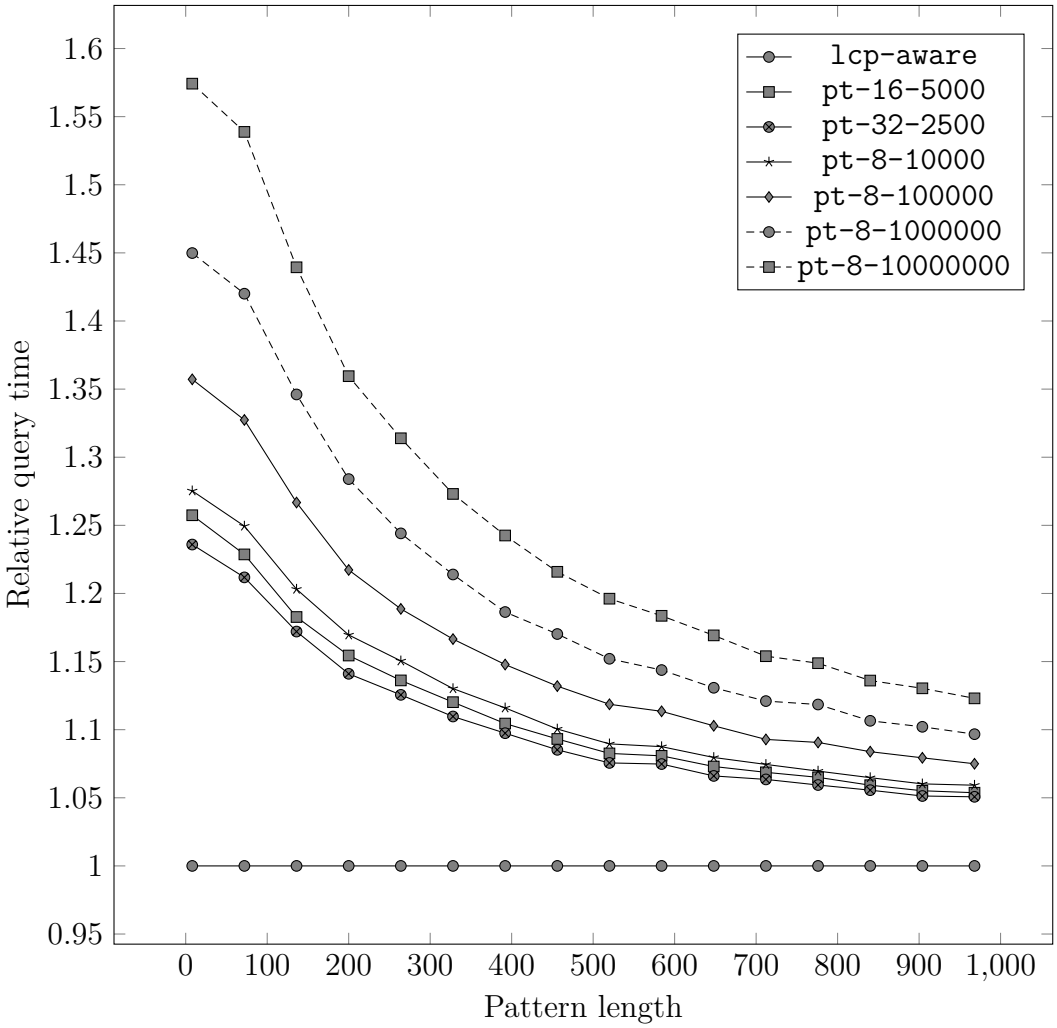


Figure 4.6: The relative query times using prefix tables with the English text data set.

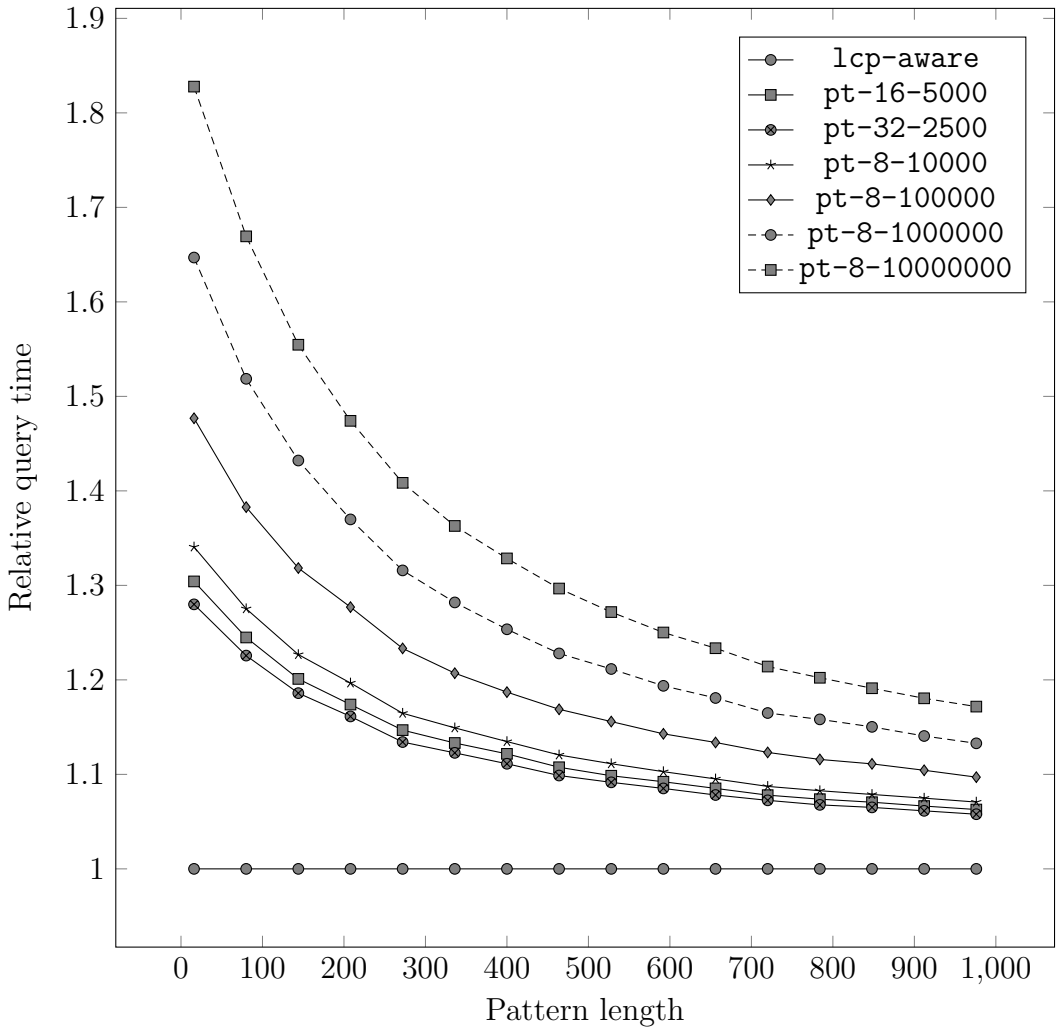


Figure 4.7: The relative query times using prefix tables with the genome data set.

5 Conclusions

We have presented a number of pre-computed index data structures that can be used for finding a good initial range for a binary search in a suffix array. They were all presented at a level of detail that should allow anyone to implement them and investigate them further.

We used an iterative approach to improve our hypothesis on the optimal look-up data structure. For example, the main weakness of the look-up array was addressed in the look-up hash table, the deficiencies of which were addressed in the look-up trie.

We measured the query speed-up with four different data structures, and the results provided clear signals that the idea of augmenting suffix array with a light-weight index can offer meaningful speed-up and other benefits.

Our approach to the topic was reasonably practical, and there remains space for further analysis of the properties of the presented data structures. For example, their size and density can be studied further and the time complexity of their construction and look-up operations can be analyzed more thoroughly.

5.1 Future work

This thesis considered the pre-computed indexes only as a means to improve binary searching of suffix arrays. There are other possibilities to use those indexes, for example:

- If query pattern P is split into k-character strings, and for each of those strings, we fetch the corresponding range from a suffix array index, we can expect to find the indexes $i, i+k, \ldots$ within the respective ranges for each match, i.e. if we denote those k-character strings by $S_0, S_1, \ldots, S_n$ (and P is their concatenation) and the corresponding suffix array ranges by $R_0, R_1, \ldots, R_n$, we can merge those ranges into a set $R = \{i | (i+k) \in R_k\}$ which consists of the matches.

  This kind of range merging technique offers a way to query the suffix array without never referencing the original text. One could also consider embedding the contents of those suffix array ranges directly in the suffix array index.

  The challenge of this method lies in the efficiency of merging algorithm. An efficient

algorithm for merging has been proposed by Fischer et al. (2016). See also (Puglisi, Smyth, and A. Turpin, 2006).

- If a pre-computed suffix array index returns a somewhat small initial binary search range, one could access the text directly at the corresponding indexes to do direct string matching, instead of using binary search.
- Our ideas could be used with other data structures and string search algorithms, for example the ones by Ferragina and Manzini (2005) and Kim et al. (2003).

These ideas may be worthly of further investigation.

Bibliography

Abouelhoda, M. I., Ohlebusch, E., and Kurtz, S. (2002). "Optimal Exact String Matching Based on Suffix Arrays". In: *String Processing and Information Retrieval*. Ed. by A. H. F. Laender and A. L. Oliveira. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 31–43. ISBN: 978-3-540-45735-0.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2005). *Introduction to algorithms, 2nd ed.* MIT press.

Ferragina, P. and Manzini, G. (July 2005). "Indexing Compressed Text". In: *J. ACM* 52.4, pp. 552–581. ISSN: 0004-5411. DOI: 10.1145/1082036.1082039. URL: https://doi.org/10.1145/1082036.1082039.

Fischer, J., Köppl, D., and Kurpicz, F. (2016). "On the Benefit of Merging Suffix Array Intervals for Parallel Pattern Matching". In: *Combinatorial Pattern Matching*.

Gonnet, G. H., Baeza-Yates, R. A., and Snider, T. (1992). "New Indices for Text: Pat Trees and Pat Arrays." In: *Information Retrieval: Data Structures & Algorithms* 66, p. 82.

Hirschberg, D. S. (1978). "A lower worst-case complexity for searching a dictionary". In: *Rice University ECE Technical Report* 7808.

– (1979). "Searching a dictionary within tighter time bounds". In: *Rice University Technical Report*.

Kärkkäinen, J. and Kempa, D. (2018). "Skewed String Binary Search". Unpublished Manuscript.

Kärkkäinen, J., Sanders, P., and Burkhardt, S. (2006). "Linear work suffix array construction". In: *Journal of the ACM (JACM)* 53.6, pp. 918–936.

Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K. (2001). "Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications". In: *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings*. Ed. by A. Amir and G. M. Landau. Vol. 2089. Lecture Notes in Computer Science. Springer, pp. 181–192.

Kim, D. K., Sim, J. S., Park, H., and Park, K. (2003). "Linear-Time Construction of Suffix Arrays". In: *Combinatorial Pattern Matching*. Ed. by R. Baeza-Yates, E. Chávez, and M. Crochemore. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 186–199. ISBN: 978-3-540-44888-4.

Li, Z., Li, J., and Huo, H. (2018). "Optimal in-place suffix sorting". In: *International Symposium on String Processing and Information Retrieval*. Springer, pp. 268–284.

Mäkinen, V., Belazzougui, D., Cunial, F., and Tomescu, A. I. (2015). *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press.

Manber, U. and Myers, G. (1990). "Suffix Arrays: A New Method for on-Line String Searches". In: *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '90. San Francisco, California, USA: Society for Industrial and Applied Mathematics, pp. 319–327. ISBN: 0898712513.

– (1993). "Suffix arrays: a new method for on-line string searches". In: *siam Journal on Computing* 22.5, pp. 935–948.

Navarro, G. and Mäkinen, V. (2007). "Compressed full-text indexes". In: *ACM Comput. Surv.* 39.1, p. 2.

Puglisi, S. J., Smyth, W. F., and Turpin, A. (2006). "Inverted Files Versus Suffix Arrays for Locating Patterns in Primary Memory". In: *String Processing and Information Retrieval*. Ed. by F. Crestani, P. Ferragina, and M. Sanderson. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 122–133. ISBN: 978-3-540-45775-6.

Puglisi, S. J., Smyth, W. F., and Turpin, A. H. (July 2007). "A Taxonomy of Suffix Array Construction Algorithms". In: *ACM Comput. Surv.* 39.2, 4–es. ISSN: 0360-0300. DOI: 10.1145/1242471.1242472. URL: https://doi.org/10.1145/1242471.1242472.

Shrestha, A. M. S., Frith, M. C., and Horton, P. (Jan. 2014). "A bioinformatician's guide to the forefront of suffix array construction algorithms". In: *Briefings in Bioinformatics* 15.2, pp. 138–154. ISSN: 1467-5463. DOI: 10.1093/bib/bbt081. eprint: https://academic.oup.com/bib/article-pdf/15/2/138/562332/bbt081.pdf. URL: https://doi.org/10.1093/bib/bbt081.