Indexing Variable Length Substrings for Exact and Approximate Matching

Gonzalo Navarro[1,⋆] and Leena Salmela[2]

[1] Department of Computer Science, University of Chile
gnavarro@dcc.uchile.cl
[2] Department of Computer Science and Engineering
Helsinki University of Technology
lsalmela@cs.hut.fi

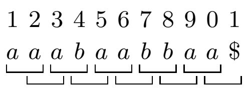
Abstract. We introduce two new index structures based on the q-gram index. The new structures index substrings of variable length instead of q-grams of fixed length. For both of the new indexes, we present a method based on the suffix tree to efficiently choose the indexed substrings so that each of them occurs almost equally frequently in the text. Our experiments show that the resulting indexes are up to 40 % faster than the q-gram index when they use the same space.

1 Introduction

We consider indexing a text for exact and approximate string matching. Given a text $T = t_1t_2 \dots t_n$, a pattern $P = p_1p_2 \dots p_m$, and an integer k, the approximate string matching problem is to find all substrings of the text such that the edit distance between the substrings and the pattern is at most k. The edit distance of two strings is the minimum number of character insertions, deletions, and substitutions needed to transform one string into the other. We treat exact string matching as a subcase of approximate string matching with $k = 0$.

Partitioning into exact search is a popular technique for approximate string matching both in the online case [1,2,11,13], where the text is not preprocessed, and in indexing approaches [3,4,7,10,12], where an index of the text is built. Suppose that the edit distance between two strings, S and R, is at most k. If we split S into $k + 1$ pieces, then at least one piece must have an exact occurrence in R. In the online approach, we thus split the pattern into $k + 1$ pieces, search for all the pieces in the text, and verify all the matches found for approximate occurrences of the whole pattern using a dynamic programming algorithm that runs in $\mathcal{O}(m^2)$ time per verification. In the indexing approach we have two options. If we index all text positions, we can proceed as in the online case: split the pattern into $k + 1$ pieces, search for all the pieces in the index, and verify all the matches found. Another option is to index the text at fixed intervals. Now we search for all pattern substrings of corresponding length in the index and verify

⋆ Supported in part by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.



2-gram	positions	difference coded positions
aa	1, 2, 5, 9	1, 1, 3, 4
ab	3, 6	3, 3
a$	10	10
ba	4, 8	4, 4
bb	7	7

Fig. 1. The 2-grams and the 2-gram index of the text $T =$ "aaabaabbaa$"

the matches found to obtain the approximate occurrences of the whole pattern. The q-gram index of Navarro and Baeza-Yates [7] takes the former approach, while the q-samples index of Sutinen and Tarhio [12] utilizes the latter technique.

A problem of the q-gram index is that some q-grams may be much more frequent than others, which raises verification costs. A technique to choose the $k+1$ optimal pieces [7] was designed to alleviate this problem. In this work we develop two new indexes, the prefix free and the prefix coalesced index, based on the q-gram index [7]. The new indexes index substrings of variable length instead of q-grams of fixed length. The goal is to achieve roughly similar lengths in all position lists. In the prefix free index the set of indexed substrings forms a prefix free set, whereas in the prefix coalesced index this restriction is lifted. The experimental results show that the new indexes are up to 40 % faster than the q-gram index for the same space. Alternatively, the new indexes achieve as good search times as the q-gram index using less space. For example, when $m = 20$ and $k = 2$ the new indexes are as fast as the q-gram index using 30 % less space.

2 *q*-Gram Index

In this section we review previous work on the q-gram index [7], which indexes all q-grams of the text and uses partitioning into exact search to locate occurrences of a pattern. The value of q is fixed at construction time, and the q-grams that occur in the text form the vocabulary of the index. Together with each q-gram the index stores a list of positions where the q-gram occurs in the text. To save space the position lists are difference coded and compressed with variable length integer coding. The q-gram index can be built in $\mathcal{O}(n)$ time [7]. Figure 1 shows the 2-grams of the text "aaabaabbaa$" and the corresponding 2-gram index.

To search for a pattern P with at most k differences, we first extract $k+1$ non-overlapping q-grams from the pattern. We then search for all these q-grams in the index and finally use dynamic programming to verify the positions returned by this search. For example, to search for the pattern $P =$ "abbab" with at most $k = 1$ difference in the 2-gram index of Fig. 1, we first extract two non-overlapping 2-grams from the pattern: "ab" and "ba". Search on the index returns positions 3 and 6 for "ab" and positions 4 and 8 for "ba". Verifying around these positions we obtain the occurrences starting at positions 3 and 6.

In real texts some q-grams occur much more frequently than others, and if the pattern is longer than $(k+1)q$, we have several different ways of choosing the $k+1$

```
1. for ($i = 1$; $i \leq m$; $i$++)
2.     $P[i, 0] = R[i, m + 1]$
3.     $C[i, 0] = m + 1$
4. for($r = 1$; $r \leq k$; $r$++)
5.     for($i = 1$; $i \leq m - r$; $i$++)
6.         $P[i, r] = \min(R[i, j] + P[j, r - 1] \mid i < j \leq m - r + 1)$
7.         $C[i, r] = j$ that minimizes the above expression
```

Fig. 2. The dynamic programming algorithm for the optimal partitioning of the pattern

non-overlapping q-grams. Therefore we can speed up verification considerably by choosing the q-grams carefully. Furthermore, the pieces do not need to have the exact length q. If a piece is shorter than q, we find all q-grams starting with the piece in the index and verify all their positions. If the piece is longer than q, we search for the first q-gram of the piece in the index. By allowing pieces shorter than q, we can also search for patterns shorter than $(k + 1)q$.

Navarro and Baeza-Yates [7] give the following method for finding the optimal partitioning of the pattern. It is relatively fast to compute the number of verifications a pattern piece will trigger. We use binary search to locate the q-gram(s) in the index and obtain a contiguous region of q-grams. If we store the accumulated list lengths, the number of verifications can easily be calculated by subtracting the accumulated list lengths at the endpoints of the region. By performing this search for all pattern pieces, we obtain a table R where $R[i, j]$ gives the number of verifications for the pattern piece $p_i \ldots p_{j-1}$. Based on this table, we use dynamic programming to compute the table $P[i, k]$, which gives the total number of triggered verifications for the best partitioning for $p_i \ldots p_m$ with k differences, and the table $C[i, k]$, which gives the position where the next piece starts in order to get $P[i, k]$. We then find the smallest entry $P[\ell_0, k]$ for $1 \leq \ell_0 \leq m-k$, which gives the final number of verifications. The pattern pieces that give this optimal number of verifications begin at $\ell_0, \ell_1 = C[\ell_0, k], \ell_2 = C[\ell_1, k - 1] \ldots \ell_k = C[\ell_{k-1}, 1]$. Figure 2 gives the pseudo code for the dynamic programming algorithm to find the optimal partitioning of the pattern. It runs in $\mathcal{O}(km^2)$ time, whereas R is easily built in $\mathcal{O}(qm \log n)$ time, which can be reduced to $\mathcal{O}(qm \log \sigma)$, where σ is the alphabet size, if the q-gram vocabulary is stored in trie form.

3 Prefix Free Index

Our new variants of the q-gram index index substrings of varying length instead of q-grams of fixed length. The indexed substrings form the vocabulary of the index. The aim is to choose the vocabulary so that each position of the text is indexed and the lengths of the position lists are as uniform as possible. In the first variant, the prefix free index, we further require that the vocabulary is a prefix free set, i.e. no indexed substring is a prefix of another indexed substring.

Let α be the threshold frequency and let the frequency of a string be the number of occurrences it has in the text T. Note that the frequency of the empty

substring	positions	difference coded positions
aaa	1	1
aab	$2, 5$	$2, 3$
$aa\$$	9	9
ab	$3, 6$	$3, 3$
$a\$$	10	10
b	$4, 7, 8$	$4, 3, 1$

(a) Prefix free index

substring	positions	difference coded positions
aa	$1, 9$	$1, 8$
aab	$2, 5$	$2, 3$
ab	$3, 6$	$3, 3$
$a\$$	10	10
ba	$4, 8$	$4, 4$
bb	7	7

(b) Prefix coalesced index

Fig. 3. A prefix free index and a prefix coalesced index for the text T = "aaabaabbaa\$"

string is n. The vocabulary now consists of all such substrings $S = s_1 \dots s_i$ of the text that the frequency of S is at most α and the frequency of the prefix $s_1 \dots s_{i-1}$ is greater than α. This choice ensures that the vocabulary is a prefix free set and no position list is longer than α. Again the position lists are difference coded and compressed with variable length integer coding. Figure 3(a) shows an example of a prefix free index with threshold frequency three.

To search for a pattern P with at most k differences, we first split the pattern into $k+1$ pieces and search for each piece in the index. If the indexed substrings starting with the piece are longer than the piece, we return all positions associated with any substring starting with the piece. If an indexed substring is a prefix of the piece, we return the positions associated with that indexed substring. The positions returned by this search are then verified with the $\mathcal{O}(m^2)$ dynamic programming algorithm to find the approximate occurrences of the pattern. As an example consider searching for the pattern P = "abbab" in the prefix free index of Fig. 3(a) with at most $k = 1$ difference. We start by splitting the pattern into two pieces: "ab" and "bab". The search for "ab" in the index returns positions 3 and 6 and the search for "bab" returns positions 4, 7, and 8. We then verify around these positions and find the occurrences starting at positions 3 and 6.

Although the lengths of the position lists are more uniform than in the q-gram index, we still benefit from computing the optimal partitioning of the pattern. First of all, the lengths of the position lists still vary, and thus the number of verifications can be reduced by choosing pattern pieces with short position lists. Secondly, if the pattern is too short to be partitioned into long enough pieces such that we would get only one position list per pattern piece, it is not clear how to select the pieces without the optimal partitioning technique.

Finding the optimal partitioning of the pattern works similarly to the q-gram index. We first use binary search to locate the indexed substring(s) corresponding to each pattern piece $p_i \dots p_{j-1}$ for $1 \leq i < j \leq m + 1$. This search returns a contiguous region of indexed substrings. If we again store the accumulated position list lengths, we can determine the number of triggered verifications fast. This number is stored in the table $R[i, j]$. We then compute the tables $P[i, k]$ and $C[i, k]$ and obtain from these tables the optimal partitioning of the pattern. The overall time to find the optimal partitioning is $\mathcal{O}(m^2(\log n + k))$.

To choose the vocabulary of the index, we use a simplified version of the technique of Klein and Shapira [5] for constructing fixed length codes for compression of text. Their technique is based on the suffix tree of the text. A cut in a suffix tree is defined as an imaginary line that crosses exactly once each path in the suffix tree from the root to one of the leaves. The lower border of a cut is defined to be the nodes immediately below the imaginary line that forms the cut. Klein and Shapira show that a lower border of a cut forms a prefix free set and a prefix of each suffix of the text is included in the lower border. Thus the lower border of a cut can be used as a vocabulary in the prefix free index.

We choose the vocabulary as follows. First we build the suffix tree of the text and augment it with the frequencies of the nodes. The frequency of a node is the frequency of the corresponding substring of the text. We then traverse the suffix tree in depth first order. If the frequency of a node is at most the threshold frequency α, we add the corresponding string S to the vocabulary and we also add the corresponding leaves to the position list of the string S.

The suffix tree can be built in $\mathcal{O}(n)$ time. The traversal of the tree also takes $\mathcal{O}(n)$ time and we do $\mathcal{O}(1)$ operations in each node. After the traversal the position lists are sorted, which takes $\mathcal{O}(n \log \alpha)$ total time. Finally the position lists are difference coded and compressed taking $\mathcal{O}(n)$ total time. Thus the construction of the prefix free index takes $\mathcal{O}(n \log \alpha)$ time.

4 Prefix Coalesced Index

In the second new variant of the q-gram index, the prefix coalesced index, we require that the vocabulary includes some prefix of each suffix of the text, and if the vocabulary contains two strings S and R such that R is a proper prefix of S, then all positions of the text starting with S are assigned only to the position list of S. Again the position lists are difference coded and the differences are compressed with variable length integer coding.

To choose the vocabulary, we build the suffix tree of the text and augment it with the frequencies of the nodes. The suffix tree is traversed in depth first order so that the children of a node are traversed in descending order of frequency. When we encounter a node whose frequency is at most the threshold frequency α, we add the corresponding string to the vocabulary, subtract the original frequency of this node from the frequency of its parent node and reconsider adding the string corresponding to the parent node to the vocabulary. When a string is added to the vocabulary, we also add the leaves to its position list. Figure 3(b) shows an example of a prefix coalesced index with threshold frequency two.

The suffix tree can again be built in $\mathcal{O}(n)$ time. Because we need to sort the children of each node when traversing the suffix tree, the traversal now takes $\mathcal{O}(n \log \sigma)$ time, where σ is the size of the alphabet. After the traversal the handling of the position lists takes $\mathcal{O}(n \log \alpha)$ as in the prefix free index. Thus the construction of the prefix coalesced index takes $\mathcal{O}(n(\log \alpha + \log \sigma))$ time.

We refine the searching procedure as follows. We again start by splitting the pattern into $k + 1$ pieces and search the index for each piece. If the piece is a prefix of several indexed substrings, we return all position lists associated with these indexed substrings. If an indexed substring is a *proper* prefix of the piece, we return only this position list. Otherwise searching on the index and optimal partitioning of the pattern work exactly the same way as in the prefix free index.

5 Experimental Results

To save space our implementations of the prefix free and the prefix coalesced indexes use a suffix array [6] instead of a suffix tree when constructing the index. The traversal of the suffix tree is simulated using binary search on the suffix array.

For compressing the position lists in all the indexes, we use bytewise compression of the differences. In this scheme, the highest bit is 0 in the last byte of the coding and 1 in other bytes. The integer is formed by concatenating the seven lowest bits of all the bytes in the coding.

The experiments were run on a 1.0 GHz AMD Athlon dual core processor with 2 GB of memory, running Linux 2.6.23. The code is in C and compiled with `gcc` using `-O2` optimization. We used the 200 MB English text from the *PizzaChili* site, `http://pizzachili.dcc.uchile.cl`, and the patterns are random substrings of the text. For each pattern length, 1,000 patterns were generated.

Figure 4 shows the search times for the q-gram index and the prefix free and prefix coalesced indexes for various pattern lengths and values of k. The q-gram index was tested with 5 values of q: 4, 5, 6, 7, and 8. The prefix free index was tested with 7 values for the threshold frequency α: 200, 500, 1,000, 2,000, 5,000, 10,000, and 20,000. The prefix coalesced index was also tested with 7 values for the threshold frequency α: 100, 200, 500, 1,000, 2,000, 5,000, and 10,000. We see that the new indexes generally achieve the same performance as the q-gram index using less space. The prefix coalesced index allows more flexibility in selecting the vocabulary, and so the the position list lengths are more uniform than in the prefix free index. Thus the search times in the prefix coalesced index are slightly lower than in the prefix free index. However, when we reduce the available space the prefix free index becomes faster than the prefix coalesced index.

Figure 5 shows the vocabulary size for the different indexes. We see that the vocabulary of the q-gram index is much larger than the vocabulary of the prefix free and the prefix coalesced indexes. Some of the q-grams are very frequent in the text and their long position lists compress very efficiently, allowing the q-gram index to use a larger vocabulary. In the prefix free and prefix coalesced indexes the position lists have a more uniform length, and thus these lists do not compress as well, so the vocabulary is much smaller. We can also see that the vocabulary of the prefix free index is larger than the vocabulary of the prefix coalesced index, again reflecting the lengths of the position lists.

Figure 5 also shows the construction time for the different indexes. The construction time of the prefix free and prefix coalesced indexes increases only little when space usage is increased because the most time consuming phase of their construction is the construction of the suffix array, which takes the same time regardless of the space usage of the final index.

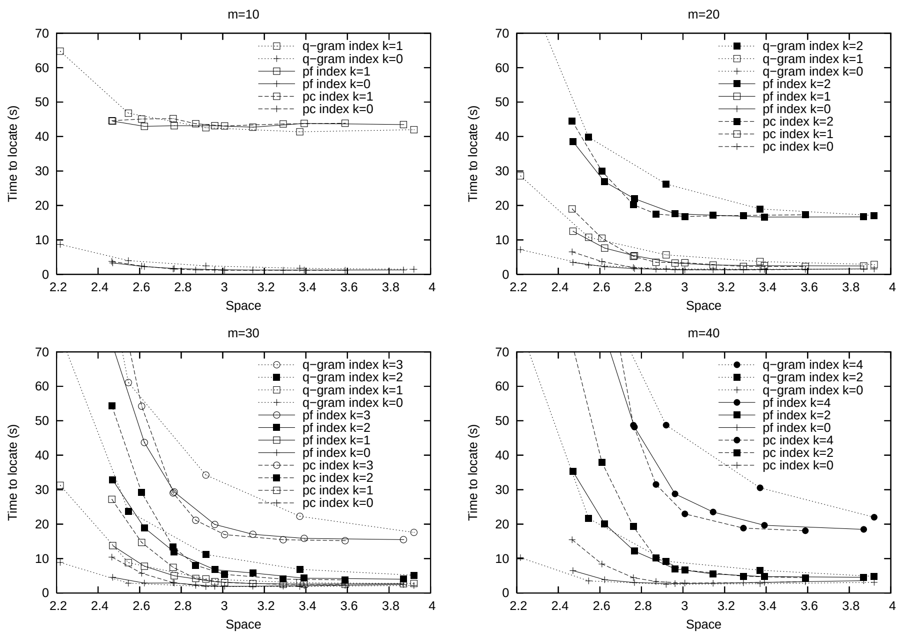


Fig. 4. Search times for the different indexes for various values of k and m. The space fraction includes that of the text, so it is of the form $1 + \frac{\text{index size}}{\text{text size}}$.

6 Conclusions and Further Work

We have presented two new indexes for exact and approximate string matching based on the q-gram index. They index variable length substrings of the text to achieve more uniform lengths of the position lists. The indexed substrings in the prefix free index form a prefix free set, whereas in the prefix coalesced index this restriction is lifted. Our experiments show that the new indexes are up to 40 % faster than the q-gram index for the same space. This shows that lists of similar length are definitely beneficial for the search performance, although they are not as compressible and thus shorter substrings must be indexed.

Our techniques receive a parameter α, giving the maximum allowed list length, and produce the largest possible index that fulfills that condition. Instead, we could set the maximum number of substrings to index, and achieve the most uniform possible index of that vocabulary size. For this we would insert the suffix tree nodes into a priority queue that sorts by frequency, and extract the most frequent nodes (with small adaptations depending on whether the index is prefix free or prefix coalesced). The construction time becomes $\mathcal{O}(n \log n)$.

Future work involves extending more sophisticated techniques based on q-grams and q-samples, such as those requiring several nearby pattern pieces to be found before triggering a verification, and/or based on approximate matching of the pattern pieces in the index [8,9].

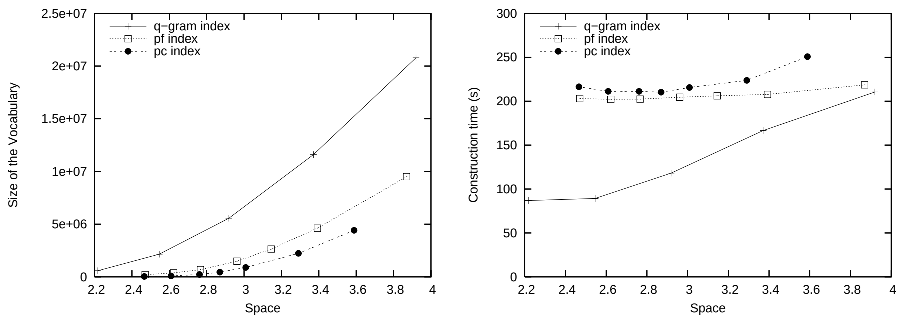


Fig. 5. The vocabulary size and the construction time of the q-gram index and the prefix free and prefix coalesced indexes as a function of space usage.

References

1. Baeza-Yates, R., Navarro, G.: Faster approximate string matching. Algorithmica 23(2), 127–158 (1999)
2. Baeza-Yates, R., Perleberg, C.: Fast and practical approximate string matching. Information Processing Letters 59(1), 21–27 (1996)
3. Holsti, N., Sutinen, E.: Approximate string matching using q-gram places. In: Proc. Finnish Symp. on Computer Science, pp. 23–32. University of Joensuu (1994)
4. Jokinen, P., Ukkonen, E.: Two algorithms for approximate string matching in static texts. In: Tarlecki (ed.) MFCS 1991. LNCS, vol. 520, pp. 240–248. Springer, Heidelberg (1991)
5. Klein, S.T., Shapira, D.: Improved variable-to-fixed length codes. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 39–50. Springer, Heidelberg (2008)
6. Manber, U., Myers, G.: Suffix arrays: A new method for online string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
7. Navarro, G., Baeza-Yates, R.: A practical q-gram index for text retrieval allowing errors. CLEI Electronic Journal 1(2) (1998)
8. Navarro, G., Baeza-Yates, R., Sutinen, E., Tarhio, J.: Indexing methods for approximate string matching. IEEE Data Engineering Bulletin 24(4), 19–27 (2001)
9. Navarro, G., Sutinen, E., Tarhio, J.: Indexing text with approximate q-grams. Journal of Discrete Algorithms 3(2–4), 157–175 (2005)
10. Shi, F.: Fast approximate string matching with q-blocks sequences. In: Proc. WSP 1996, pp. 257–271. Carleton University Press (1996)
11. Sutinen, E., Tarhio, J.: On using q-gram locations in approximate string matching. In: Spirakis, P. (ed.) ESA 1995. LNCS, vol. 979, pp. 327–340. Springer, Heidelberg (1995)
12. Sutinen, E., Tarhio, J.: Filtration with q-samples in approximate string matching. In: Hirschberg, D., Myers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 50–63. Springer, Heidelberg (1996)
13. Wu, S., Manber, U.: Fast text searching allowing errors. Communications of the ACM 35(10), 83–91 (1992)