Approximate Pattern Matching Over the Burrows-Wheeler Transformed Text

Nan Zhang[1], Amar Mukherjee[1] Don Adjeroh[2], and Tim Bell[3]

[1] University of Central Florida, Orlando FL 32816, USA, nzhang,amar@cs.ucf.edu,
Partially supported by NSF grant IIS-9977336 and IIS 0207819
[2] West Virginia University, Morgantown WV 26506, USA, don@csee.wvu.edu
[3] University of Canterbury, New Zealand, tim@cosc.canterbury.ac.nz
Supported by a grant from the Hitachinaka Techno Center Inc, Japan.

Abstract. The compressed pattern matching problem is to locate the occurrence(s) of a pattern P in a text string T using a compressed representation of T, with minimal (or no) decompression. In this paper, we consider approximate pattern matching directly on Burrow-Wheeler transformed (BWT) text which is a critical step for a fully compressed pattern matching algorithm on a BWT based compression algorithm. The BWT provides a lexicographic ordering of the input text as part of its inverse transformation process. Based on this observation, pattern matching is performed by text pre-filtering, using a fast q-gram intersection of segments from the pattern P and the text T. Algorithms are proposed that solve the k-mismatch problem in $O(\min\{m|\Sigma|^k \log \frac{u}{|\Sigma|}, mu \log \frac{u}{|\Sigma|}\})$ time worst case, and the k-approximate matching problem in $O(|\Sigma| \log |\Sigma| + \frac{m^2}{k} \log \frac{u}{|\Sigma|} + \alpha k)$ time on average ($\alpha \leq u$), where $u = |T|$ is the size of the text, $m = |P|$ is the size of the pattern, and Σ is the symbol alphabet. Each algorithm requires a few $O(u)$ auxiliary arrays, which are constructed in $O(u)$ time and space.

1 Introduction

The pattern matching problem is to find the occurrence of a given pattern in a given text string. This is an old problem, which has been approached from different fronts, motivated by both its practical significance and its algorithmic importance. Matches between strings are determined based on the *string edit distance*. Given two strings $A : a_1...a_u$, and $B : b_1...b_m$, over an alphabet Σ, and a set of allowed edit operations, the edit distance indicates the minimum number of edit operations required to transform one string into the other. Three basic types of edit operations are used: *insertion, deletion, and substitution* of a symbol. The *exact string matching problem* is to check for the existence of a substring of the text that is an exact replica of the pattern string. That is, the edit distance between the substring of A and the pattern should be zero. In *k-approximate string matching*, the task is to find each substring A_s of A, such that the edit distance between A_s and B is less than or equal to k. Another form of approximate pattern matching is the *k-mismatch problem.* The problem

here is to locate all substrings of A that have a maximum of k mismatches with B. That is, only the substitution operation is allowed. Various algorithms have been proposed for both exact and approximate pattern matching. See [8, 9] for a survey. With the sheer volume of data easily available to an ordinary user and the fact that most of this data is increasingly in compressed format, efforts have been made to address the *compressed pattern matching problem*. Given T a text string, P a search pattern, and Z the compressed representation of T, the problem is to locate the occurrences of P in T with minimal (or no) decompression of Z. Initial attempts on compressed pattern matching were directed at compression schemes based on the LZ family [2], where algorithms have been proposed that can search for a pattern in an LZ77-compressed text string in $O(n \log^2(\frac{u}{n}) + m)$ time, where $m = |P|, u = |T|$, and $n = |Z|$. Some other approaches are proposed to search on Run-length and Huffman coded text.

Our philosophy of compressed pattern matching is to explore the search awareness inherited in the current text compression scheme or create our own compression algorithm by which the compressed data are well organized and can benefit the searching procedure while not sacrificing much on the compression ratio. There are several data structures that can be used to perform fast searching such as sorted link list, sorted array, binary search tree, suffix tree, etc. We will focus on the fast generation of such data structures during the (partial) decompression stage. The block sorting algorithm, Burrows-Wheeler Transform (BWT) [6] is such a algorithm that can provide possible good data organization and facilitate fast searching algorithms because of the sorting stage in the forward transformation. Although there has been substantial work in compressed pattern matching, a recent survey [4] shows that little has been done on searching directly on text compressed with BWT. Even less has been done on compressed domain approximate pattern matching on BWT text. Yet, in terms of data compression BWT is significantly better than the more popular LZ-based methods (such as GZIP and COMPRESS), and is only second to the PPM algorithm. In terms of running time, the BWT is much faster than PPM, but comparable with LZ-based algorithms. So far, the major reported work on searching on BWT-compressed text are those of Sadakane [11] and Ferragina and Manzini [7], who proposed $O(m \log u + \eta_{occ} \log^{\epsilon} u)$ and $O(m + \eta_{occ} \log^{\epsilon} u)$ time algorithms respectively, to locate all η_{occ} occurrences of P in T, where $0 < \epsilon \leq 1$. In [1, 5], methods were reported that can search for exact matches in BWT text in $O(m \log \frac{u}{|\Sigma|})$ time. In this paper, we provide algorithms for solving both the k-mismatch problem and the k-approximate matching problem directly on Burrows-Wheeler transformed text.

2 The Burrows-Wheeler Transform

The BWT performs a permutation of the characters in the text, such that characters in lexically similar contexts will be near to each other. The important procedures in BWT-based compression/decompression are the forward and inverse BWT, and the subsequent encoding of the permuted text.

The forward transform. Given an input text $T = t_1 t_2 \dots t_u$, the forward BWT will block sort the permutation matrix of T. The output is pair, (L, id) where L is the last column of the sorted permutation matrix M, and id is the row number for the row in M that corresponds to the original text string T. Generally, the effect is that the contexts that are similar in T are made to be closer together in L. As an example, suppose $T = mississippi$. Let F and L denote the array of $first$ and $last$ characters respectively. Then, $F = iiiimppssss$ and $L = pssmipissii$. The output of the transformation will be the pair: ($pssmipissii$, 5) (indices are from 1 to u).

The inverse transform. The inverse transformation can be performed using the following steps [6]: 1) Sort L to produce F, the array of first characters, in $O(n)$ time; 2) Compute V, the *transformation vector* that provides a one-to-one mapping between the elements of L and F, such that $F[V[j]] = L[j]$. That is, for a given symbol $\sigma \in \Sigma$, if $L[j]$ is the c-th occurrence of σ in L, then $V[j] = i$, where $F[i]$ is the c-th occurrence of σ in F; 3) Generate the original text T, since the rows in M are cyclic rotations of each other, the symbol $L[i]$ cyclically precedes the symbol $F[i]$ in T. That is, $L[V[j]]$ cyclically precedes $L[j]$ in T. For the example with $mississippi$, we will have V = [6 8 9 5 1 7 2 10 11 3 4]. Given V and L, we can generate the original text by iterating with V. This is captured by a simple algorithm: $T[u+1-i] = L[V^{i-1}[id]], \forall i = 1, 2, \dots u$, where $V^0[s] = s$; and $V^{i+1}[s] = V[V^i[s]]], 1 \leq s \leq u$.

BWT-based compression. Compression with the BWT is usually accomplished in four phases, viz: $bwt \rightarrow mtf \rightarrow rle \rightarrow vlc$, where *mtf* is move-to-front encoding used to further transform L for better compression; *rle* is run length encoding; and *vlc* is variable length coding using entropy encoding methods, such as Huffman or arithmetic coding.

3 Overview of Our Approach

The motivation for our approach is the observation that the BWT provides a lexicographic ordering of the input text as part of its inverse transformation process. The decoder only has limited information about the sorted context, but it is possible to exploit this to perform an initial match on two symbols (a character and its context), and then decode only that part of the text to see if the pattern match continues. Our current algorithms are searching at the the stage of the *bwt* — before the *mtf* and further encoding. The methods can be modified to search directly on the encoded output.

Given F and L, we can obtain a set of *bi-grams* for the original text sequence T. Let $\mathcal{Q}_2^T$ and $\mathcal{Q}_2^P$ be the set of bi-grams for the text string T and the pattern P, respectively. We can use these bi-grams for at least two purposes:

Pre-filtering. To search for potential matches, we consider only the bi-grams that are in the set $\mathcal{Q}_2^T \cap \mathcal{Q}_2^P$. If the intersection is empty, it means that the pattern does not occur in the text, and we don't need to do any further decompression.

Approximate pattern matching. We can generalize the bi-grams to the more usual q-grams, and perform q-gram intersection on $\mathcal{Q}_q^T$ and $\mathcal{Q}_q^P$ — the set

of q-grams from T and P. At a second stage we verify if the q-grams in the intersection are part of a true k-approximate match to the pattern.

3.1 Auxiliary Arrays

Since F is already sorted, and $F[i] = L[V[i]]$, $\forall i, i = 1, 2, \ldots, u$. we can use a mapping between T and F, (rather than L), so that we can use binary search on F. We can use an auxiliary array H (or its reverse Hr) to hold the intermediate steps of the indexing using V: $H[i] = V[V^{i-1}[id]]$, and $T[i] = F[H[u + 1 - i]]$; $Hr =$ **reverse**(H), and $T[i] = F[Hr[i]]$. Hrs is an index vector to the elements of Hr in sorted order. It is defined as the **inverse** of Hr, that is, $F[i] = T[Hrs[i]]$. It may be observed that Hrs also corresponds to the **suffix array** of the original text T, an important data structure for searching and sequence analysis [8]. Thus, it provides the index to the lexicographically sorted list of all the suffixes from the text.

3.2 Fast q-gram generation

Hr (also H) represents a one-to-one mapping between F and T. By simply using F and Hr, we can access any character in the text string, without using T itself — which is not available without complete decompression. Therefore, there is a simple algorithm to generate the q-grams, for any given q: $\forall_{x=1,2,\ldots u-q+1}$, $\mathcal{Q}_q^T[x] = F[Hr[x]] \ldots F[Hr[x + q - 1]]$;

These q-grams are not sorted. However, we can obtain the sorted q-grams directly by picking out the x's according to their *order* in Hr, and then use F to locate them in T. The index vector Hrs provides information about the ordering, and is defined such that: $\forall i, F[i] = T[Hrs[i]] = F[Hr[Hrs[i]]]$. This means that we have effectively generated the *q-grams* in constant time. Since the availability of F, Hr and Hrs implies constant-time access to any area in the text string T. The x used in the previous description is simply an index on the elements of T.

3.3 Fast q-gram intersection

Based on the nature of the different arrays used by the BWT, and the new auxiliary arrays previously described, algorithms have been proposed to perform fast q-gram intersection [1]. Let $\mathcal{MQ}_q = \mathcal{Q}_q^P \cap \mathcal{Q}_q^T$. We call $\mathcal{MQ}_q$, the *set of matching q-grams*. For each q-gram, we use indexing on F, Hr and Hrs to pick up the required areas in T, and then match the patterns. To compute $\mathcal{MQ}_q$, we need to search for the occurrence of each member of $\mathcal{Q}_q^P$ in $\mathcal{Q}_q^T$.

For the special case when $m = q$ (i.e. exact pattern matching), [1, 5] proposed $O(m \log \frac{u}{|\Sigma|})$ to perform the required set intersection and thus locate the exact matches of P in T, after an $O(u)$ preprocessing of Z, the BWT transformed output T.

4 Locating *k*-mismatches

We propose a k-mismatch algorithm based on the fast q-gram generation algorithm. The pattern matching operation is performed with all possible alignment of the pattern with the text indirectly via the matrix of sorted suffixes specified by the vector Hrs. The characters of the pattern P are compared with characters in successive columns of the permissable q-grams which, in fact, is the suffix matrix S if they exist. If there is a mismatch between the characters at corresponding locations of P and T for a given row in S, the number of the errors or mismatch is incremented by 1 without considering the possibility of inserting or deleting a character in the text to make a match. Since the S matrix is lexicographically sorted, the match or mismatch takes place with the entire group of consecutively located rows in S. We record the number of mismatches (*count*) for the group as well as the start(st) and end (ed) positions for the group in the form of a triplet $(st, ed, count)$. We place the triplet in an output list called **Candidates**. If *count* is still less than k, we will continue to search in the group. If in a given row, the suffix length becomes less than the pattern length, it means that for this alignment of the pattern, the text has ran out of characters. So, for each additional operation a mismatch count in its row triplet has to be added by one as long as $count \leq k$. The operation proceeds until the last character of P is processed yielding a final partition of the suffixes in S having maximum of k mismatch with P. This will correspond to the set of triplets that survive with less than or equal to k mismatches. The triplets remaining in *Candidates* at this point correspond to positions with a maximum of k-mismatches to the pattern. Let C be the arrays of counts $C = c_1, c_2, \ldots c_{|\Sigma|}$, which is computed in the computation of V in inverse BWT. For a given index, c, $C[c]$ stores the number of occurrences in L of all the characters preceding σ_c, the c-th symbol in Σ. The algorithm is given below.

k-mismatch Algorithm
Input: pattern P, the arrays F, C, auxiliary arrays Hr, Hrs, parameter k
Output: **Candidates**, the set of surviving triplets

1. **Initialize** *Candidates*:
   for each symbol $\sigma_c \in \Sigma$, (the c-the symbol in Σ), and $\sigma_c \in F$ **do**
   create a triplet with
   $st = C[c] + 1$**;**
   if $c < |\Sigma|$ **then** $ed = C[c+1]$ **else** $ed = u$**;**
   if $\sigma_c = P[1]$ **then** $count$ **= 0 else** $count = 1$**;**
   if $count \leq k$ **then append triplet to** *Candidates*.
   end
2. **for** j **= 2 to m do**
   for each element in *Candidates* **that survive the** $(j-1)$**-th iteration do**
   – **Remove the triplet** $(st, ed, count)$ **from** *Candidates*
   – **for each distinct symbol** $\sigma_c \in F$ **do**

- **locate the start and end position st' and ed' in F between st and ed using binary search on the j-th column of the suffix matrix S**

  Note that we do not actually create the j-gram for each row in the S matrix during binary search. Instead, given the row index pos of a $(j-1)$-gram in S, the j-th symbol s can be accessed in constant time as: $s = F[Hr[Hrs[pos]+j]]$.

- **if $\sigma_c = P[j]$, then add triplet $(st', ed', count)$ to $Candidates$. (Since it is a match to $P[j]$, there is no change to $count$).**
- **else, if $count{+}1 \leq k$, then add triplet $(st', ed', count{+}1)$ to $Candidate$. Increment $count$ by 1.**

end
end
end

3. **Report the m-length patterns between st and ed for each element of $Candidates$ as the k-mismatches. The row positions in F can be converted to the corresponding positions in T using Hrs, as explained earlier.**

Complexity Analysis. The preprocessing cost for preparing the auxiliary arrays is $O(u)$. For each iteration of the innermost loop, binary search is used to locate all the groups with the same j-gram. At most u groups will be generated. Thus each loop takes $O(u \log \frac{u}{|\Sigma|})$ time in the worst case. The maximum number of triplets that can be generated will be in $O(|\Sigma|^k)$. But this cannot be more than u, the text size. In practice, many groups or triplets are dropped because the error count becomes greater than k. The worst case time to search the whole pattern will be $O(\min\{m|\Sigma|^k \log \frac{u}{|\Sigma|}, mu \log \frac{u}{|\Sigma|}\})$. We observe that the maximum number of triplets remains relatively constant for different pattern lengths. The average number of triplets decreases with increasing pattern length, m. Both are, however, relatively small compared to the file size (1.13 MB on average over the test corpus). Typically, the number of triplet increased in the first few, mostly k, iterations and dropped drastically afterwards as more characters from P are checked. The time performance is given at Results section.

5 Locating *k*-approximate matches

There are two phases in our approach. In the first phase, we locate areas in the text that contain potential matches by performing some filtering operations using *appropriately sized* q-grams. In the second phase, we verify the results that are hypothesized by the filtering operations. The verification stage is generally slow, but usually, it will be performed on only a small proportion of the text. Thus, the overall performance depends critically on the number of hypothesis generated.

Locating potential matches. The first phase is based on a known fact in approximate pattern matching:

Lemma 1. k-approximate match [3] *Given a text T, a pattern P, ($m = |P|$), and k, for a k-approximate match of P to occur in T, there must exist at least one r-length block of symbols in P that form an* **exact match** *to some r-length substring in T, where $r = \lfloor \frac{m}{k+1} \rfloor$.* $\diamondsuit$

This is trivially the case for exact pattern matching, in which $k = 0$, and hence $r = m$. With the lemma, we can perform the filtering phase in three steps: 1) Compute r, the minimum block size for the q-grams. 2) Generate $\mathcal{Q}_r^T$ and $\mathcal{Q}_r^P$, the permissible r-grams from the text T, and the pattern P, respectively. 3) Perform q-gram intersection of $\mathcal{Q}_r^T$ and $\mathcal{Q}_r^P$.

Let $\mathcal{MQ}_r = \mathcal{Q}_r^P \cap \mathcal{Q}_r^T$, and $\eta = |\mathcal{MQ}_r|$. Let $\mathcal{MQ}_r^i$ be the i-th matching r-gram. Let $\mathcal{MQ}_r^i[j]$ be the j-th character in $\mathcal{MQ}_r^i$, $j = 1, 2, \ldots r$. Further, let $f[i]$ be the index of the first character of $\mathcal{MQ}_r^i$ in the array of first characters, F. That is, $f[i] = x$, if $F[x] = \mathcal{MQ}_r^i[1]$. We call $\mathcal{MQ}_r$ the *matching r-grams with k*. Its size is an important parameter for the next phase.

Based on the discussion in 3.2, the indices j and $f[i]$ can be generated in O(1) time. Similarly, step 2 above can be done in constant time and space. The cost of step 3, will grow slower than $\frac{m^2}{k+1} \log u$. The time for hypothesis generation is simply the time needed for r-gram intersection, where r is given by **Lemma 1**. Let $\mathcal{Z}_i^{\mathcal{F}}$ be the number of q-grams in the text starting with the symbol with index i in $|\Sigma|$. Then, $\mathcal{Z}_i^{\mathcal{F}} = C[i+1] - C[i], \forall_{i=1,2,\ldots|\Sigma|-1}$, and $\mathcal{Z}_i^{\mathcal{F}} = u - C[i]$ **if** $i = |\Sigma|$. Similarly, let $\mathcal{Z}_i^{\mathcal{P}}$ be the number of q-grams in the pattern starting with the symbol with index i in $|\Sigma|$. Let $\mathcal{Z}_{d_P}$ be the number of q-grams in $\mathcal{Q}_q^P$ that started with *distinct* characters — simply, the number of non-empty partitions in $\mathcal{Q}_q^P$. Thus, $\mathcal{Z}_{d_P} \leq m - q + 1$ and $\mathcal{Z}_{d_P} \leq |\Sigma|$. Plugging these into the analysis for QGRAM algorithm [1], we have the following:

Lemma 2. Locating potential k-approximate matches. *Given $T = t_1 t_2 \ldots t_u$, transformed with the BWT, $P = p_1 p_2 \ldots p_m$, an alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$, and the arrays F, Hr and I. Let k be given. The hypothesis phase can be performed in time proportional to: $\mathcal{Z}_{d_P} \log |\Sigma| + \lfloor \frac{m}{k+1} \rfloor \sum_{i \in \Sigma} \mathcal{Z}_i^{\mathcal{P}} \log \mathcal{Z}_i^{\mathcal{F}}$* $\diamondsuit$

Verifying the matches. Here, we need to verify if the r-blocks that were hypothesized in the first phase are true matches. The time required will depend critically on η_h. We perform the verification in two steps: 1) Using Hr and F determine the matching neighborhood in T for each r-block in $\mathcal{MQ}_k$. The maximum size of the neighborhood will be $m + 2k$; 2) Verify if there is a k-approximate match within the neighborhood.

Let $\mathcal{N}_i$ be the neighborhood in T for $\mathcal{MQ}_k^i$, the i-th matching q-gram. Let t be the position in T where $\mathcal{MQ}_k^i$ starts. That is, $t = Hr[F[\overline{\mathcal{MQ}}_k^{\mathcal{F}}[i]]]$. The neighborhood is defined by the left and right limits: t_{left} and t_{right} viz: $t_{left} = t - k$ **if** $t - k \geq 1$; $t_{left} = 1$ otherwise; $t_{right} = t + m + k$ **if** $t + m + k \leq u$; $t_{right} = u$ otherwise. Hence, the i-th matching neighborhood in T is given by: $\mathcal{N}_i = T[t_{left} \ldots t \ldots t_{right}]$. Thus, $|\mathcal{N}_i| \leq m + 2k, \forall_i, i = 1, 2, \ldots \eta_h$. We then obtain a set of matching neighborhoods $\mathcal{S}_{\mathcal{MQ}} = \{\mathcal{N}_1, \mathcal{N}_2, \ldots \mathcal{N}_{\eta_h}\}$. Verifying a match within any given $\mathcal{N}_i$ can be done with any fast algorithm for k-approximate

matching, for instance, Ukkonen's $O(ku)$ algorithm [12]. The cost of the fist step in the verification will be in $O(\eta_h)$. The cost of the second step will thus be in $O(\eta_h k(m+2k)) \leq O(\eta_h k(3m)) \approx O(\eta_h km)$.

Example. Let $T = abraca$ and $P = brace$, with $k = 1$. Then, $r = 2$, and permissible q-grams will be $\mathcal{Q}_2^P = \{ac, br, ce, ra\}$, $\mathcal{Q}_2^T = \{ab, ac, br, ca, ra\}$, yielding $\mathcal{MQ}_1 = \{ac, br, ra\}$, and $\mathcal{N}_1 = [3 \ldots 6]; \mathcal{N}_2 = [1, \ldots, 6]; \mathcal{N}_3 = [2, \ldots, 6]$. Matches will be found in $\mathcal{N}_1$ and $\mathcal{N}_2$ at positions 1 and 2 in T, respectively. $\diamondsuit$

We conclude this section with the following theorem:

Theorem 1. k-approximate matching in Burrows-Wheeler transformed text *Given a text string T, a pattern P, and symbol alphabet Σ. Let Z be the* **bwt** *output when T is the input. There is an algorithm that can locate the k-approximate matches of P in T, using only Z, (i.e. without full decompression nor with off-line index structures) in $O(|\Sigma| \log |\Sigma| + \frac{m^2}{k} \log \frac{u}{|\Sigma|} + k|\psi_{\mathcal{MQ}}|)$ time on average, ($|\psi_{\mathcal{MQ}}| \leq u$), and in $O(ku + |\Sigma| \log |\Sigma| + \frac{m^2}{k} \log \frac{u}{|\Sigma|})$ worst case, after an $O(u)$ preprocessing on Z.* $\diamondsuit$

The proof is based on using the $O(m \log \frac{u}{|\Sigma|})$ BWT-based search algorithm [1, 5] to match all the $m - r + 1$ r-grams that will be generated, where $r = \lfloor \frac{m}{k+1} \rfloor$. We omit the detailed proof for brevity. With the q-gram approach, we can treat exact pattern matching as no different from k-approximate pattern matching. We just have k=0, and hence no verification stage.

6 Results

To test the proposed methods for compressed approximate pattern matching, we selected 133 files from three different corpora. The files included the text files in the Canterbury corpus, *html* and C program files from the Calgary corpus, and the AP, DOE and FR files from Disk 1 in the TREC-TIPSTER corpus. The file sizes ranged from 11,150 characters (`fields.c`) to 4,161,115 characters (`FR89011` in TREC-TIPSTER Corpus). The average file size is 936 Kbytes.

Experiments were performed using 10 sets of sample patterns (words). Each set has 100 words with the same length, $m = |P| = 2, 3, \ldots 11$. The words are the most frequently used from the English dictionary. The test were run on Sun Unltra-5 work station (360MHz) and Solaris 2.5 Operating System with 256Mb memory. For the compression performance, the commercial $O(n \log n)$ implementation of `Bzip2` using ternary tree has a compression time of 3.0 seconds per megabytes over the corpus, while our normal quick sort $O(n^2 \log n)$ implementation is 55.82 seconds per megabytes. However, compression is off-line in compressed pattern matching. We can replace our BWT with the optimized one. For decompression time which is critical to the pattern matching performance, `Bzip2` is about 0.76 second per megabytes on average and ours is 1.28 seconds per megabytes, among which 0.94 second is spent on building auxiliary arrays. Time is measured in seconds in all reported results. The decompression is of linear time. So our decompression is slightly worse than `Bzip2`. But our amortized gain will be shown in the searching period.

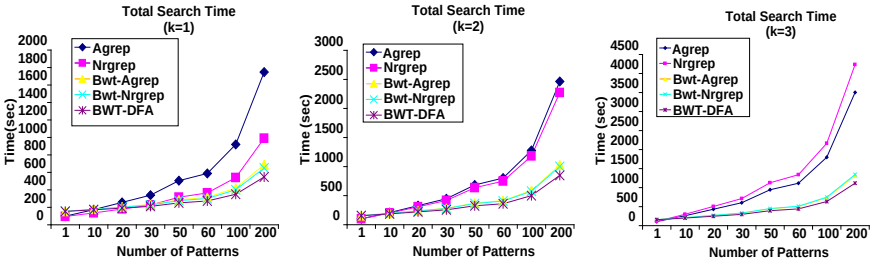

Fig. 1. Total search time including decompression overhead

We compare the proposed compressed domain k-mismatch algorithm with an algorithm described in Gusfield's book [8] which is based on suffix trees. Here, the k-mismatch check at any position i in T is performed using at most k longest common extension computation. Each computation can be performed in constant time, after the longest common ancestor table has been constructed. The suffix tree, however, usually requires a large storage, (about $21u$ bytes), although the construction is in $O(u)$ time. The search time used by the two algorithms are shown in Figure 2 (left column).

For k-approximate matches, we tested the proposed method with two popular approximate pattern matching algorithms: AGREP [13], and NRGREP [10]. Both algorithms are based on bit-wise operations using the patterns and text. The two algorithms operate on the raw (uncompressed) text. The results for the search time are shown in Figure 2, right column. Results for the proposed BWT-based approach is labeled BWT-DFA. Here, the hypothesis phase is performed by finding the exact matches for the r-grams using the QGRAM algorithm [1], while the verification phase is performed using Ukkonen's DFA [12]. We have also included two other results: BWT-AGREP and BWT-NRGREP. These correspond to when we used the proposed q-gram filtering approach, but with the verification phase performed with AGREP and NRGREP respectively. The construction time for Ukkonen's DFA is shown in Table 1.

m	k=1	k=2	k=3
2	0.0005	0.0003	0.0003
3	0.0008	0.0008	0.0004
4	0.0013	0.0021	0.0019
5	0.0018	0.0045	0.0053
6	0.0023	0.0081	0.0133
7	0.003	0.0135	0.0293
8	0.0039	0.0188	0.0568
9	0.0049	0.0237	0.0942
10	0.0063	0.0296	0.1401
11	0.0079	0.035	0.1844
15	0.0118	0.056	0.287
20	0.0167	0.08	0.58
34	0.1	0.52	3.13

Table 1. Construction time for Ukkonen's DFA

The k-approximate matching results in Figure 2 give the average single pattern search time over the whole corpus. When $k = 1$, NRGREP performs better than AGREP when pattern size getting larger. They have similar time measurement for $k = 2$. But AGREP performs slightly better than the NRGREP for $k = 3$. For all the algorithms, report only one pattern occurrence for each line in the text, which is the sam as the original AGREP and NRGREP.

The proposed methods perform constantly better than the AGREP and NRGREP performing on the raw text. For our current test when $m \leq 11$ and $k \leq 3$, the cost for DFA construction is minimal comparing to the search time. For each pattern, the DFA need to be computed only once regardless to the number of the files to search. For a single verification of r-gram extension, the time is almost constant using AGREP, NRGREP or DFA since they are linear and the candidate string is of the size $m + 2k$ only. The fluctuation of the search time main comes from the number of the r-grams (mostly, $r = 1, 2, 3$ in our case) found and the number of verifications failed in a line. We observe that, the search time increased at a rate from 1.41 to 1.83 when k increased from 1 to 2 and 2 to 3, except that NRGREP has an average of 3.10 time increment for k changes from 2 to 3.

BWT-AGREP , BWT-NRGREP, and BWT-DFA produced shorter search times than the traditional AGREP and NRGREP algorithms, which perform matches on the de-compressed text. Their reported search time does not include the time required for decompression. When multiple patterns search is considered, both decompression time and searching time plays important roles in the search. Figure 1 shows the multiple pattern searching time. It indicates that the amortized cost of our algorithms are lower when the number of search patterns exceeds around 20 for $k = 1$, 10 for $k = 2$, and less than 10 for $k = 3$. This property make sure that for frequent text retrieval, the proposed methods performs better.

7 Conclusion

The BWT with its sorted contexts however provides an ideal platform for compressed domain pattern matching. This paper has described algorithms for approximate pattern matching directly on BWT-transformed text and shows the advantage over the decompression and search scheme. The proposed algorithms could be further improved. For instance, the space requirement could be reduced by considering the compression blocks typically used in BWT-based compression schemes, while the time requirement could be further reduced by using faster pattern matching techniques for the q-gram intersection. One way to reduce the relatively high overheads will be to consider pattern matching on blocked BWT-compressed files, since these will typically involve smaller text sizes.

The overhead of DFA construction is rather high when k grows big (such as DNA sequence) compared to the search time. But once a DFA is constructed for a pattern, it can be used to find any other pattern with edit distance less than or equal to k without changing the DFA. The proposed approach will thus be very effective in dictionary matching, where one may wish to search for a

pattern in multiple text sequences. One possible improvement on the hypothesis verification phase could be to use dynamic construction of the DFA. The basic concept in the method does not depend on any particular verification algorithm. This could also be seen from the results with BWT-AGREP and BWT-NRGREP. Thus, one could abandon Ukkonen's DFA altogether, and look for alternative faster verification algorithms.

We note that the methods as described basically operate on the output of the BWT transformation stage. One challenge is to extend the approach to operate beyond the BWT output, i.e. after the later encoding stages in the BWT compression process. A modified move-to-front that has the search awareness property is currently under research and the compression ratio is better then our implementation of $bwt \rightarrow mtf \rightarrow rle \rightarrow vlc$ work flow and is slightly worse than the optimized `Bzip2`.

References

1. Adjeroh, D. A., Mukherjee, A., Bell, T., Powell, M., and Zhang, N. (2002). Pattern matching in bwt-transformed text. *Proceedings, IEEE Data Compression Conference*, page 445.
2. Amir, A., Benson, G., and Farach, M. (1996). Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52:299–307.
3. Baeza-Yates, R. and Perleberg, C. (1992). Fast and practical approximate string matching. *Proceedings, Combinatorial Pattern Matching, LNCS 644*, pages 185–192.
4. Bell, T., Adjeroh, D., and Mukherjee, A. (2001). Pattern matching in compressed text and images. Technical report, Department of Computer Science, University of Canterbury.
5. Bell, T., Powell, M., Mukherjee, A., and Adjeroh, D. A. (2002). Searching bwt compressed text with the boyer-moore algorithm and binary search. *Proceedings, IEEE Data Compression Conference, 2002*, pages 112–121.
6. Burrows, M. and Wheeler, D. (1994). A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, Palo Alto, California.
7. Ferragina, P. and Manzini, G. (2000). Opportunistic data structures with applications. *Proceedings, 41st IEEE Symposium on Foundations of Computer Science, FOCS'2000*.
8. Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press.
9. Navarro, G. (2001a). A guided tour of approximate string matching. *ACM Computing Surveys*, 33(1):31–88.
10. Navarro, G. (2001b). Nr-grep: A fast and flexible pattern matching tool. *Software – Practice and Experience*, (31):1265–1312.
11. Sadakane, K. (2000). Compressed text databases with efficient query algorithms based on the compressed suffix array. *Proceedings, ISAAC'2000*.
12. Ukkonen, E.,. Finding approximate patterns in strings. *Journal of Algorithms* (6)1985, 132-137.
13. Wu, S. and Manber, U. (1992a). agrep — A fast approximate pattern-matching tool. In USENIX Association, editor, *Proceedings of the Winter 1992 USENIX Conference: January 20 — January 24, 1992, San Francisco, California*, pages 153–162, Berkeley, CA, USA. USENIX.

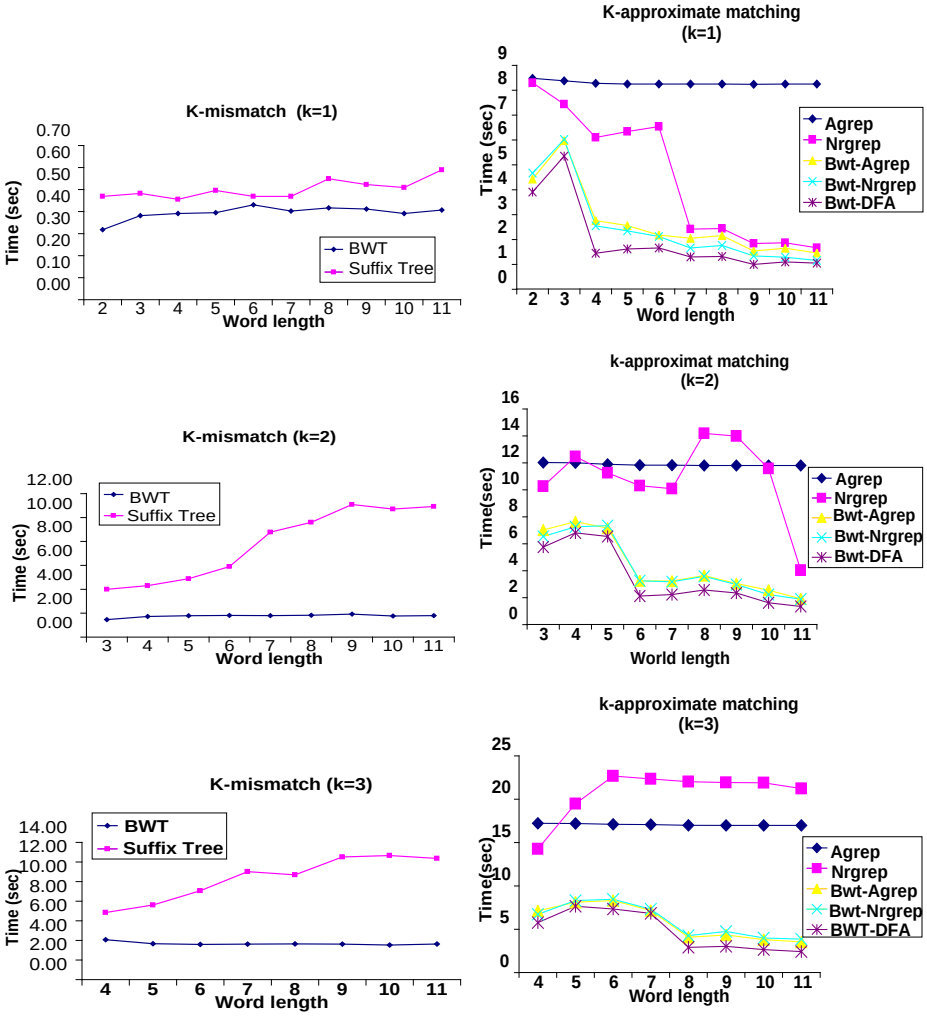

Fig. 2. Results for k-mismatches (left column) and k-approximate match (right column)