Approximate Pattern Matching using the Burrows-Wheeler Transform

Nan Zhang† Amar Mukherjee‡ Don Adjeroh§ Tim Bell¶

February 19, 2003

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 BWT compressed text. 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 \(O(\min\{m(m-k)|\Sigma|^k \log \frac{m}{\alpha k}, mu \log \frac{u}{\Sigma}\})\) time worst case, and the \(k\)-approximate matching problem in \(O(|\Sigma| \log |\Sigma| + \frac{n^2}{\alpha} \log \frac{u}{\Sigma} + \alpha k)\) time on average \((\alpha \leq u)\), where \(u = |T|\) is the size of the text, \(m = |P|\) is the size of the pattern, and \(\Sigma\) is the symbol alphabet. Each algorithm requires an \(O(u)\) 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 \(\Sigma\), 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. Given two strings \(A: a_1...a_u\) and \(B: b_1...b_m\) over an alphabet \(\Sigma\), 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. The parameter \(k\) thus acts as a form of threshold to determine the correctness of a match. Various algorithms have been proposed for both exact and approximate pattern matching. See [Gusfield, 1997, Navarro, 2001a] 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

---

†School of Electrical Engineering & Computer Science, University of Central Florida, Orlando, FL.32816, USA; {nzhang, amar}@cs.ucf.edu
‡Partially supported by a grant from the National Science Foundation IIS-9977336.
§Lane Department of Computer Science & Electrical Engineering, West Virginia University, Morgantown, WV 26506-6109, USA; don@csee.wvu.edu
¶Department of Computer Science, University of Canterbury, New Zealand; {tim}@cosc.canterbury.ac.nz
¶¶Supported by a grant from the Hitachinaka Techno Center Inc, Japan.
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 [Amir et al., 1996, Farach and Thorup, 1998], 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 $n = |Z|$. Bunke and Csirok [Bunke and Csirok, 1995] proposed methods that can search for patterns in run-length encoded files in $O(um_c)$, when $m_c$ is the length of the pattern when it is compressed. In [Moura et al., 2000], $O(n + m\sqrt{u})$ algorithms were proposed for searching Huffman-encoded files. Special compression schemes that facilitate later pattern matching directly on the compressed file have also been proposed [Manber, 1997, Shibata et al., 1999].

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) [Burrows and Wheeler, 1994] 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 [Bell et al., 2001] 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, empirical evidence [Burrows and Wheeler, 1994, Fenwick, 1996, Balkenhol et al., 1999] shows that the BWT is significantly better than the more popular LZ-based methods (such as gzip and compress), and is only second to the PPM* algorithm [Cleary and Witten, 1984]. 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 [Sadakane, 2000] and Ferragina and Manzini [Ferragina and Manzini, 2000], 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 $\eta_{occ}$ occurrences of $P$ in $T$, where $0 < \epsilon \leq 1$. In [Adjeroh et al., 2002, Bell et al., 2002], methods were reported that can search for exact matches in BWT text in $O(m \log \frac{\sqrt{u}}{\epsilon})$ time. The methods, however, considered only exact pattern matching. In general, compressed pattern matching algorithms have focused mainly on exact matching. One exception is [Mäkinen et al., 2001] where they considered RLE-encoded text. Although the one-page abstract of [Adjeroh et al., 2002] discussed $k$-approximate matching very briefly, there was no mention of $k$-mismatches.

A commercial BWT compression algorithm such as bzip2 [[ usually composed of BWT, Move-to-Front (mtf), Run-length encoding (rle) and variable length entropy encoding (vle). 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. To make the algorithm to be fully performed on the compressed text, we also introduce a modified MTF algorithm that brings search awareness to the transformed text. More effort well be taken towards a actual BWT compressed text search algorithm.

## 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 $T = t_1 t_2 \ldots t_u$, the forward BWT is composed of three steps: 1) Form $u$ permutations of $T$ by cyclic rotations of the characters in $T$. The permutations form a $u \times u$ matrix $M'$, with each row in $M'$ representing one permutation of $T$; 2) Sort the rows of $M'$ lexicographically to form another matrix $M$. $M$ includes $T$ as one of its rows; 3) Record $L$, the last column of the sorted permutation matrix $M$, and $id$, the row number for the row in $M$ that corresponds to the original text string $T$. 
The output of the BWT is the pair, \((L, id)\). Generally, the effect is that the contexts that are similar in \(T\) are made to be closer together in \(L\). This similarity in nearby contexts can be exploited to achieve compression. As an example, suppose \(T = \text{mississippi}\). Let \(F\) and \(L\) denote the array of first and last characters respectively. Then, \(F = iiii\text{ppssss}\) and \(L = \text{psmipissii}\). The output of the transformation will be the pair: \((\text{psmipissii}, 5)\) (indices are from 1 to 6).

The inverse transform. The BWT is reversible. It is quite striking that given only the \((L, id)\) pair, the original text can be recovered exactly. The inverse transformation can be performed using the following steps [Burrows and Wheeler, 1994]: 1) Sort \(L\) to produce \(F\), the array of first characters; 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 \(L[j] = \sigma\) is the \(c\)-th occurrence of \(\sigma\) in \(L\), then \(V[j] = i\), where \(F[i]\) is the \(c\)-th occurrence of \(\sigma\) 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 \(\text{mississippi}\), 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: \(T[u + 1 - i] = L[V^{u-1}[id]], \forall i = 1, 2, \ldots, u,\) where \(V^0[s] = s\); and \(V^{i+1}[s] = V[V^i[s]], 1 \leq s \leq u\). In practical implementations, the transformation vector \(V\) is computed by use of two arrays of counts \(C = c_1, c_2, \ldots, c_{|\Sigma|}\), and \(R = r_1, r_2, \ldots, r_u\): \(V[i] = R[i] + C[L[i]], \forall i = 1, 2, \ldots, u\) where, for a given index, \(c, C[c]\) stores the number of occurrences in \(L\) of all the characters preceding \(\sigma_c\), the \(c\)-th symbol in \(\Sigma\), and \(R[j]\) keeps count of the number of occurrences of character \(L[j]\) in the prefix \(L[1, 2, \ldots, j]\) of \(L\) (\(R\) can be calculated on the fly using an array of \(|\Sigma|\) counts, which accumulates the number of times each character has appeared in \(L\) so far.).

With \(V\), we can use the relation between \(L, F, V\) to avoid the sorting required to obtain \(F\). Thus, we can compute \(F\) in \(O(u)\) time.

BWT-based compression. Compression with the BWT is usually accomplished in four phases, viz: \(\text{but} \rightarrow \text{mtf} \rightarrow \text{rle} \rightarrow \text{vlc}\), where \(\text{but}\) is the forward BWT transform; \(\text{mtf}\) is move-to-front encoding [Bentley et al., 1986] used to further transform \(L\) for better compression (this usually produces runs of the same symbol); \(\text{rle}\) is run length encoding of the runs produced by the \(\text{mtf}\); and \(\text{vlc}\) is variable length coding of the \(\text{rle}\) output 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. We give our description in terms of the \(F, L, V\) arrays, (i.e. the output of the \(\text{but}\) transformation — before the \(\text{mtf}\) 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 \(Q_T^F\) and \(Q_T^L\) 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 \(Q_T^F \cap Q_T^L\). 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 \(Q_T^F\) and \(Q_T^L\) — 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.

Example. Suppose \(T = \text{abra}\text{ca}\text{ca}\), and \(P = \text{rac}\). We will have \(L = \text{caraab}\), and \(F = \text{aaabcr}\). Using \(F\) and \(L\), we can obtain the bi-grams: \(Q_T^F = \{ac, ab, br, ca, ra\}\) and \(Q_T^L = \{ra, ac\}\). Intersecting the two, we see that only \(\{ra, ac\}\) are in the intersection. For exact pattern matching, \(ac\) will be eliminated, and thus we will only need to check in the area in \(T\) that contains \(ra\), since any match must contain \(ra\).
3.1 Auxiliary Arrays

The inverse BWT transformation is defined as: \( \forall i = 1, 2, \ldots, u, T[u + 1 - i] = L[V^{i-1}[id]] \), where \( V^i = V[V[\ldots V[s]]] \) (\( i \) times) and \( V^0 = s \). Since \( V^i[z] \) is just one more indirection on \( V^{i-1}[z] \), we can reduce the time required by storing the intermediate results, to be used at the next iteration of the loop. Since \( F \) is already sorted, and \( F[z] = L[V[z]] \), 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 \( V \): \( \forall i, i = 1, 2, \ldots, u \) \( H[i] = V[V^{i-1}[id]] \), and \( T[i] = F[H[u + 1 - i]] \); \( Hr = \text{reverse}(H) \), and \( T[i] = F[Hr[i]] \).

**Example.** The mapping vectors are shown below for \( T = \text{abraca}, u = |T| = 6 \). \( Hrs \) 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 [Manber and Myers, 1993, Gusfield, 1997]. Thus, it provides the index to the lexicographically sorted list of all the suffixes from the text. We denote this matrix of sorted suffixes as the \( S \) matrix. The table below shows the auxiliary vectors and the \( S \) matrix for the example text \( T = \text{abraca} \).

<table>
<thead>
<tr>
<th>idx</th>
<th>T</th>
<th>L</th>
<th>F</th>
<th>V</th>
<th>Hr</th>
<th>Hrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>5</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>r</td>
<td>r</td>
<td>a</td>
<td>6</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>a</td>
<td>c</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

The key to a faster approach is to generate only the \( q \)-grams that are necessary, using \( F \) and the auxiliary arrays. We call these \( q \)-grams that are necessary the permissible \( q \)-grams — they are the only \( q \)-grams that are permissible given \( u, m \), and the fact that matching cannot progress beyond the last characters in \( T \) and \( P \).

Further, if we wish to perform exact-pattern matching for a pattern \( P \), where \( |P| = m \), all we need will be the \( m \)-length \( q \)-grams (i.e. the \( m \)-grams) in the text \( T \). The \( m \)-length \( q \)-grams (and excluding the \( q \)-gram from the rotations of the text) are the permissible \( q \)-grams. In general, we have a total of \( u - q + 1 \) permissible \( q \)-grams for a \( u \)-length text. The major problems are to find cheap ways to generate the \( q \)-grams from \( T \), and then how to perform the intersection quickly.

**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 \( q: \forall x = 1, 2, \ldots, u - q + 1 \), \( Q_q^T[x] = F[Hr[x]] \ldots F[Hr[x + q - 1]] \).

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 [Adjeroh et al., 2002]. Let \( M_{Q_q} = Q_q^P \cap Q_q^T \). We call \( M_{Q_q} \), the set of matching q-grams. For each q-gram, we use indexing on \( F, H_r \) and \( H_{rs} \) to pick up the required areas in \( T \), and then match the patterns. To compute \( M_{Q_q} \), we need to search for the occurrence of each member of \( Q_q^P \) in \( Q_q^T \).

For the special case when \( m = q \) (i.e. exact pattern matching), [Adjeroh et al., 2002, Bell et al., 2002] proposed \( O(m \log |\Sigma|) \) 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 \( H_{rs} \). The characters of the pattern \( P \) are compared with characters in successive columns of the permissible 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 (\( \text{count} \)) for the group as well as the start (\( \text{st} \)) and end (\( \text{ed} \)) positions for the group in the form of a triplet (\( \text{st, ed, count} \)). We place the triplet in an output list called \( \text{Candidates} \). If \( \text{count} \) 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 \( \text{count} \leq k \).

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 \( \text{Candidates} \) at this point correspond to positions with a maximum of \( k \)-mismatches to the pattern. The algorithm is given below.

**k-mismatch Algorithm**

**Input**: pattern \( P \), the arrays \( F, C \), auxiliary arrays \( H_r, H_{rs} \), parameter \( k \)

**Output**: \( \text{Candidates} \), the set of surviving triplets

1. **Initialize \( \text{Candidates} \):**
   
   for each symbol \( \sigma_c \in \Sigma \), (the \( c \)-the symbol in \( \Sigma \), and \( \sigma_c \in F \) do
   
   create a triplet with
   
   \( \text{st} = C[c] + 1 \);
   
   if \( c < |\Sigma| \) then \( \text{ed} = C[c + 1] \) else \( \text{ed} = u \);
   
   if \( \sigma_c = P[1] \) then \( \text{count} = 0 \) else \( \text{count} = 1 \);
   
   if \( \text{count} \leq k \) then append triplet to \( \text{Candidates} \).

2. **for** \( j = 2 \) to \( m \) **do**
   
   for each element in \( \text{Candidates} \) that survive the \((j-1)\)-th iteration do
   
   Remove the triplet (\( \text{st, ed, count} \)) from \( \text{Candidates} \)
   
   for each distinct symbol \( \sigma_c \in F \) do
   
   - locate the start and end position
   
   \( \text{st}' \) and \( \text{ed} \) in \( F \) between \( \text{st} \) and \( \text{ed} \) 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 \( \text{pos} \) of a \((j-1)\)-gram in \( S \), the \( j \)-th symbol \( s \) can be accessed in constant time as: \( s = F[H_r[H_r[\text{pos}] + j]] \).

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

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 \( H_r \), as explained earlier.

We demonstrate the algorithm with an example: \( P = \text{“ssis”}, T = \text{“mississippi”}, k = 2 \).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>FS</td>
<td>FST</td>
<td>FSTT</td>
</tr>
<tr>
<td>1 i</td>
<td>i</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>2 i</td>
<td>(1,4,1)</td>
<td>(2,3,2)</td>
<td>ipp</td>
</tr>
<tr>
<td>3 i</td>
<td>is</td>
<td>(3,4,1)</td>
<td>(3,4,2)</td>
</tr>
<tr>
<td>4 i</td>
<td>is</td>
<td>is</td>
<td>issi</td>
</tr>
<tr>
<td>5 m</td>
<td>(5,5,1)</td>
<td>mi</td>
<td>(5,5,2)</td>
</tr>
<tr>
<td>6 p</td>
<td>(6,7,1)</td>
<td>pi</td>
<td>(6,6,2)</td>
</tr>
<tr>
<td>7 p</td>
<td>pp</td>
<td>(7,8,2)</td>
<td>(7,8,2)</td>
</tr>
<tr>
<td>8 s</td>
<td>si</td>
<td>(8,9,1)</td>
<td>sip</td>
</tr>
<tr>
<td>9 s</td>
<td>(8,11,0)</td>
<td>si</td>
<td>sis</td>
</tr>
<tr>
<td>10 s</td>
<td>ss</td>
<td>(10,11,0)</td>
<td>ssi</td>
</tr>
<tr>
<td>11 s</td>
<td>ss</td>
<td>ssi</td>
<td>ssis</td>
</tr>
</tbody>
</table>

When the first character of the pattern ‘s’ is checked, the first column of q-gram is partitioned into four groups. These are represented by the four triplets recording the start and end position in the sorted matrix as well as the number of errors that occurred. For example, row 8 to row 11 has the character ‘s’ matching \( P[1] \). This has triplet \((8,11,0)\). Row 1 to row 4 has the character ‘i’ that is a mismatch to \( P[1] \), hence the triplet \((1,4,1)\). Then continue matching with the next character in the pattern by checking the second character of the bigrams in the text. The first row corresponds to the suffix starting from the last character of the text, thus the bigram beginning at this position is not a valid or permissible bigram. The match operation, therefore has to be terminated. Terminated matches are indicated with ‘X’ in the above example. Then suffixes that belong to rows two to four are split into 2 groups, one with a mismatch giving the triplet \((2,3,2)\) and the other a match giving triplet \((3,4,1)\). Continuing with the third character of the pattern, the suffix in row two has a mismatch so that the error count is greater than \( k \). The corresponding triplet is removed from the candidate set, denoted by ‘X’. This also happens for the suffix in row 5. Finally, suffixes in rows 9,10 and 11 lead to pattern match with the error count 2,1 and 0, respectively.

**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 \( O(u \log \frac{u}{m}) \) 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 \( O(\min\{m(n-k)/\Sigma^k \log \frac{n}{10}, mu \log \frac{n}{10}\}) \). Figure 1 shows experimental results of the maximum and average number of triplets generated over the test corpus for different word lengths.

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). Figure 2 shows a typical evolution of the number of triplets during a search for $k$-mismatches, (with $m = |P| = 7$, $u = 125$KB and $k = 2, 3, 4$). The number of triplets approaches a maximum at the $k$-th iteration, then and drops quickly as more characters from $P$ are checked.

![Figure 1: Number of triplets generated: left column: peak triplets, right column: average triplets](image1)

![Figure 2: Evolution of peak triplets generated (k=2,3,4)](image2)

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 2: k-approximate match [Baeza-Yates and Perleberg, 1992] 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 $r = \lfloor \frac{m}{k+1} \rfloor$.

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 $Q^T_r$ and $Q^P_r$, the permisable r-grams from the text $T$, and the pattern $P$, respectively. 3) Perform q-gram intersection of $Q^T_r$ and $Q^P_r$.

Let $M_{Q_k} = Q^T_r \cap Q^P_r$, and $\eta_h = |M_{Q_k}|$. Let $M_{Q^i_{k}}$ be the i-th matching q-gram. Let $M_{Q^{i}_{k}}[j]$ be the j-th character in $M_{Q^i_k}$, $j = 1, 2, \ldots r$. Further, let $\overline{M_{Q^i_k}}[i]$ be the index of the first character of $M_{Q^i_k}$ in the array of first characters, $F$. That is, $\overline{M_{Q^i_k}}[i] = x$, if $F[x] = M_{Q^i_k}[1]$. We call $M_{Q_k}$ the matching q-grams at $k$. Its size is an important parameter for the next phase.

Based on the discussion in 3.2, the indices $j$ and $\overline{M_{Q^i_k}}[i]$ 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 q-gram intersection, where $q$ is given by Lemma 2. Plugging these into the analysis for qgram algorithm [Adjeroh et al., 2002], we have the following:

Lemma 3: Locating potential k-approximate matches. Given $T = t_1t_2 \ldots t_u$, transformed with the BWT, $P = p_1p_2 \ldots p_m$, 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: $2d_u \log |\Sigma| + \frac{m}{k+1} \sum_{\sigma \in \Sigma} Z^T_\sigma \log Z^F_\sigma$.

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 $\eta_h$. We perform the verification in two steps: 1) Using $Hr$ and $F$ determine the matching neighborhood in $T$ for each r-block in $M_{Q_k}$. The maximum size of the neighborhood will be $m + 2k$; 2) Verify if there is a k-approximate match within the neighborhood.

Let $N_i$ be the neighborhood in $T$ for $M_{Q^i_k}$, the i-th matching q-gram. Let $t$ be the position in $T$ where $M_{Q^i_k}$ starts. That is, $t = Hr[F[\overline{M_{Q^i_k}}[i]]]$. 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: $N_i = T[t_{left} \ldots t_{right}]$. Thus, $|N_i| \leq m + 2k, \forall i, i = 1, 2, \ldots \eta_h$. We then obtain a set of matching neighborhoods $S_{M_Q} = \{N_1, N_2, \ldots, N_{\eta_h}\}$. Verifying a match within any given $N_i$ can be done with any fast algorithm for k-approximate matching, for instance, Ukkonen’s $O(ku)$ algorithm [Ukkonen, 1985]. 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 $Q^T_2 = \{ac, br, ce, ra\}$, $Q^P_2 = \{ab, ac, br, ca, ra\}$, yielding $M_{Q_1} = \{ac, br, ra\}$, and $N_1 = [3, 5]; N_2 = [1, 2]; N_3 = [2, 4, 6]$. Matches will be found in $N_1$ and $N_2$ at positions 1 and 2 in $T$, respectively.

We conclude this section with the following theorem:

Theorem 2: k-approximate matching in Burrows-Wheeler transformed text. Given a text string $T = t_1t_2 \ldots t_u$, a pattern $P = p_1p_2 \ldots p_m$, and symbol alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$. 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 $O(|\Sigma| \log |\Sigma| + \frac{m^2}{k+1} \log \frac{|\Sigma|}{m} + k|\psi_{M_Q}|)$ time on average, ($|\psi_{M_Q}| \leq u$), and in $O(ku + |\Sigma| \log |\Sigma| + \frac{m^2}{k} \log \frac{|\Sigma|}{m})$ worst case, after an $O(u)$ preprocessing on $Z$.

The proof is based on using the $O(m \log \frac{|\Sigma|}{m})$ BWT-based search algorithm [Adjeroh et al., 2002, Bell et al., 2002] 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 Pattern Matching with the Encoded Output

The algorithms described so far operate basically on the transformed BWT text, rather than the encoded output after the \textit{mtf} stage. In practice, it will be useful to perform the pattern matching directly on the encoded strings, with minimal decoding. The Move-to-front (\textit{mtf}) algorithm described in [Bentley et al., 1986] is used to transform the BWT output ($L, id$) so that the resulting sequence has a lot of small numbers. The sequence can be well compressed using a variable length coding method such as Huffman or Arithmetic coding. However, the current \textit{mtf} has a disadvantage for our compressed pattern matching. We cannot compute the auxiliary arrays directly from the \textit{mtf} output. We have to totally decode the \textit{mtf} results so that we can obtain the $C$ and $R$ arrays to compute $F$, $V$, $Hr$ and $Hrs$, etc. Therefore we propose a modified version of \textit{mtf} so that we can have a faster way to compute the auxiliary arrays.

The forward \textit{mtf} is modified as follows: We will output two sequences ($M_1, M_2$) for string $L$. Given the BWT output sequence, for each symbol, if the symbol is a new symbol compared to the previous one, we output the number '1' to the sequence $M_1$ and output its position $\delta_i$ in $\Sigma$ to $M_2$. Otherwise, output the number '0' to $M_1$ only. Therefore, $M_1$ is a binary sequence including new symbol information with '1' and repeating symbol information with '0'. $M_2$ maintains the ordering information for the distinct symbol appearance. For example, $\Sigma = \{a, b, c\}$ and the position codes representing the alphabet are $\{0, 1, 2\}$, the modified Move-to-Front coding for string "bbabaccebbabab" are the strings ("1010010011010100", "102101"). There is no actual move-to-front operation for the most recently appeared symbol. The decoding is straightforward. Based on string $M_1$, if a '1' is encountered, look for the next symbol in $M_2$ and output the corresponding symbol in $\Sigma$. If a '0' is encountered, simply repeat the previous symbol. At first glance, $M_1$ is of length $u$ and $M_2$ is the extra sequence that makes no compression at all. But $M_1$ is a binary sequence which can be further compressed by a high efficiency algorithm such as Run-Length Encoding or Dynamic Markov Coding (DMC). We have shown that this gives better compression in [\textit{]}.

As stated in [\textit{]}, we need arrays $C$ and $R$ to computer $V$ and then for $Hr$ and $Hrs$. $C = (c_1, c_2, ..., c_{|\Sigma|})$ is the character count array. $C(c)$ is the number of occurrence in $L$ of all the character preceding , $\delta_i$ the $c$-th symbol in $\Sigma$. $R = (r_1, r_2, ..., r_u)$ is the number of occurrence of $L(j)$ in the prefix $L(1, 2, ..., j)$ of $L$. Let $m2 = |M_2|$. $C$ can be computed in $O(m2 + |\Sigma|)$ time.

Given $|\Sigma| = s$. The algorithm to compute $C$ becomes:

```plaintext
cum_count = 0;
C(0) = 0;
for i = 1 to s do
    C(i) = C(i-1) + cum_count;
    Cum_count = C(i);
end
```

The algorithm to compute $R$ is:

```plaintext
seq = 1;
for i = 1 to s do count(i)=0;
for j = 1 to m2 do
    for k = 1 to M1[j]
        R(seq) = count(M2[j])+1;
        count(M2[j]) = R(seq);
    seq = seq + 1;
end
```

Both of the algorithms are of complexity $O(u)$. Since $V(i) = R(i) + C(L(i))$, we still need to know $L(i)$, for each $i$. Our initial idea is to avoid recovering string $L$ to obtain auxiliary arrays directly from \textit{mtf} results. However, in this modified \textit{mtf}, with $M_1$ and $M_2$, $L(j)$ can be computed in $O(u)$ time, compared to the normal \textit{mtf} which has an complexity of $O(|\Sigma|u)$. $|\Sigma|$ is generally taken as 256 which is quite a large number.
in actual running. Our method has a small constant for $O(u)$ computation because we avoid the alphabet adjustment for each symbol.

An alternative algorithm to obtain $L(i)$ but avoiding computation of the whole $L$ is through $M_1$ and $M_2$. If $M_1$ is represented by the number of repetitions of the corresponding symbols in $M_2$, $M_1$ in the above example becomes “2,4,4,3,3,3”. By $M_1$ we can obtain the accumulation count “2,6,10,13,16,19” for the distinct symbol list in $M_2$ in $O(m2)$ time. Then the symbol corresponding to the $i$-th position in string $L$ can be found by the binary search in the accumulation array and the corresponding symbol code in $M_2$ can be directly obtained. Thus the complexity becomes $O(u \log m2)$. To estimate log $m2$, we do the test on the whole corpus. $u/m2$ is approximately 3. For simplicity, let $u/m2 = 2$ and $u$ is a 1Mb byte block size. Then $O(u \log m2) = O(19u)$. Compared to the $O(|\Sigma|u)$ time for $mtf$, it is still a significant improvement.

### 7 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 (Table ??). 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 Ultra-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 of-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.

We compare the proposed compressed domain $k$-mismatch algorithm with an algorithm described in Gusfield’s book [Gusfield, 1997] 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

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>File</th>
<th>Size</th>
<th>File</th>
<th>Size</th>
<th>File</th>
<th>Size</th>
<th>File</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>book1</td>
<td>76177</td>
<td>book2</td>
<td>80854</td>
<td>file1</td>
<td>20644</td>
<td>file2</td>
<td>25788</td>
<td>file3</td>
<td>54738</td>
</tr>
<tr>
<td>book3</td>
<td>50615</td>
<td>book4</td>
<td>109288</td>
<td>file4</td>
<td>164513</td>
<td>file5</td>
<td>116765</td>
<td>file6</td>
<td>104512</td>
</tr>
<tr>
<td>news1</td>
<td>377199</td>
<td>news2</td>
<td>801016</td>
<td>file7</td>
<td>82303</td>
<td>file8</td>
<td>80405</td>
<td>file9</td>
<td>500143</td>
</tr>
<tr>
<td>paper1</td>
<td>84299</td>
<td>paper2</td>
<td>845836</td>
<td>file10</td>
<td>500760</td>
<td>file11</td>
<td>1395657</td>
<td>file12</td>
<td>2184569</td>
</tr>
<tr>
<td>paper3</td>
<td>46022</td>
<td>paper4</td>
<td>251452</td>
<td>file13</td>
<td>544686</td>
<td>file14</td>
<td>2184569</td>
<td>file15</td>
<td>2348176</td>
</tr>
<tr>
<td>paper5</td>
<td>11326</td>
<td>paper6</td>
<td>281421</td>
<td>file16</td>
<td>25852</td>
<td>file17</td>
<td>1534119</td>
<td>file18</td>
<td>4282176</td>
</tr>
<tr>
<td>paper7</td>
<td>20619</td>
<td>paper8</td>
<td>364720</td>
<td>file19</td>
<td>27885</td>
<td>file20</td>
<td>1042927</td>
<td>file21</td>
<td>2882176</td>
</tr>
<tr>
<td>prog1</td>
<td>30661</td>
<td>prog2</td>
<td>1058164</td>
<td>file22</td>
<td>769726</td>
<td>file23</td>
<td>1085895</td>
<td>file24</td>
<td>2328238</td>
</tr>
<tr>
<td>prog3</td>
<td>43579</td>
<td>prog4</td>
<td>710594</td>
<td>file25</td>
<td>846199</td>
<td>file26</td>
<td>1085670</td>
<td>file27</td>
<td>1059412</td>
</tr>
<tr>
<td>trans1</td>
<td>73068</td>
<td>trans2</td>
<td>914089</td>
<td>file28</td>
<td>806484</td>
<td>file29</td>
<td>1039742</td>
<td>file30</td>
<td>1079556</td>
</tr>
<tr>
<td>alice-99.txt</td>
<td>152099</td>
<td>alice-100.txt</td>
<td>551312</td>
<td>file31</td>
<td>924493</td>
<td>file32</td>
<td>1078926</td>
<td>file33</td>
<td>2109269</td>
</tr>
<tr>
<td>apache-99.txt</td>
<td>125192</td>
<td>apache-100.txt</td>
<td>300143</td>
<td>file34</td>
<td>267142</td>
<td>file35</td>
<td>1080091</td>
<td>file36</td>
<td>1097424</td>
</tr>
<tr>
<td>ap html</td>
<td>24693</td>
<td>ap html2</td>
<td>1001421</td>
<td>file37</td>
<td>316974</td>
<td>file38</td>
<td>1070072</td>
<td>file39</td>
<td>871195</td>
</tr>
<tr>
<td>bin html</td>
<td>11360</td>
<td>bin html2</td>
<td>938082</td>
<td>file40</td>
<td>317114</td>
<td>file41</td>
<td>1009278</td>
<td>file42</td>
<td>1279327</td>
</tr>
<tr>
<td>bibtex1.txt</td>
<td>129754</td>
<td>bibtex2.txt</td>
<td>1014885</td>
<td>file43</td>
<td>1062484</td>
<td>file44</td>
<td>1067043</td>
<td>file45</td>
<td>1081019</td>
</tr>
<tr>
<td>bibtex41.txt</td>
<td>481861</td>
<td>bibtex42.txt</td>
<td>903785</td>
<td>file46</td>
<td>1077484</td>
<td>file47</td>
<td>1095098</td>
<td>file48</td>
<td>1165306</td>
</tr>
<tr>
<td>table.xlsx</td>
<td>604732</td>
<td>table1.xlsx</td>
<td>298325</td>
<td>file49</td>
<td>1067664</td>
<td>file50</td>
<td>1097680</td>
<td>file51</td>
<td>1220298</td>
</tr>
<tr>
<td>AP9890101</td>
<td>214705</td>
<td>AP989012</td>
<td>506406</td>
<td>file52</td>
<td>1087462</td>
<td>file53</td>
<td>1937196</td>
<td>file54</td>
<td>2638208</td>
</tr>
<tr>
<td>AP9890102</td>
<td>590710</td>
<td>AP989012</td>
<td>251138</td>
<td>file55</td>
<td>1087542</td>
<td>file56</td>
<td>1944118</td>
<td>file57</td>
<td>2888248</td>
</tr>
<tr>
<td>AP9890103</td>
<td>784558</td>
<td>AP989012</td>
<td>809697</td>
<td>file58</td>
<td>1098406</td>
<td>file59</td>
<td>1952805</td>
<td>file60</td>
<td>3079352</td>
</tr>
<tr>
<td>AP9890104</td>
<td>695243</td>
<td>AP989012</td>
<td>916256</td>
<td>file61</td>
<td>1085538</td>
<td>file62</td>
<td>1960369</td>
<td>file63</td>
<td>3093222</td>
</tr>
<tr>
<td>AP9890105</td>
<td>879625</td>
<td>AP989012</td>
<td>957654</td>
<td>file64</td>
<td>1087113</td>
<td>file65</td>
<td>1972303</td>
<td>file66</td>
<td>3121403</td>
</tr>
<tr>
<td>AP9890106</td>
<td>761518</td>
<td>AP989012</td>
<td>880244</td>
<td>file67</td>
<td>1076801</td>
<td>file68</td>
<td>1979314</td>
<td>file69</td>
<td>3128500</td>
</tr>
<tr>
<td>AP9890107</td>
<td>543407</td>
<td>AP989012</td>
<td>899590</td>
<td>file70</td>
<td>1079618</td>
<td>file71</td>
<td>1985533</td>
<td>file72</td>
<td>3135600</td>
</tr>
</tbody>
</table>

| Table 1: File size of the benchmark |
Table 2: Construction time for Ukkonen’s DFA

<table>
<thead>
<tr>
<th>m</th>
<th>k=1</th>
<th>k=2</th>
<th>k=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.0005</td>
<td>0.0005</td>
<td>0.0005</td>
</tr>
<tr>
<td>3</td>
<td>0.0008</td>
<td>0.0008</td>
<td>0.0008</td>
</tr>
<tr>
<td>4</td>
<td>0.0013</td>
<td>0.0021</td>
<td>0.0015</td>
</tr>
<tr>
<td>5</td>
<td>0.0018</td>
<td>0.0045</td>
<td>0.0033</td>
</tr>
<tr>
<td>6</td>
<td>0.0023</td>
<td>0.0084</td>
<td>0.0068</td>
</tr>
<tr>
<td>7</td>
<td>0.0028</td>
<td>0.0118</td>
<td>0.0093</td>
</tr>
<tr>
<td>8</td>
<td>0.0033</td>
<td>0.0168</td>
<td>0.0108</td>
</tr>
<tr>
<td>9</td>
<td>0.0049</td>
<td>0.0237</td>
<td>0.0192</td>
</tr>
<tr>
<td>10</td>
<td>0.0054</td>
<td>0.0295</td>
<td>0.0201</td>
</tr>
<tr>
<td>11</td>
<td>0.0069</td>
<td>0.0358</td>
<td>0.0244</td>
</tr>
<tr>
<td>15</td>
<td>0.0118</td>
<td>0.0556</td>
<td>0.0427</td>
</tr>
<tr>
<td>20</td>
<td>0.0167</td>
<td>0.0884</td>
<td>0.0582</td>
</tr>
<tr>
<td>35</td>
<td>0.111</td>
<td>0.921</td>
<td>0.111</td>
</tr>
</tbody>
</table>

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 3 (left column).

For \( k \)-approximate matches, we tested the proposed method with two popular approximate pattern matching algorithms: AGREP [Wu and Manber, 1992], and NRGREP [Navarro, 2001b]. 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 3, 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 [Adjeroh et al., 2002], while the verification phase is performed using Ukkonen’s DFA [Ukkonen, 1985]. 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 2.

The \( k \)-approximate matching results in Figure 3 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 \( m <= 11 \) and \( k <= 3 \), 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, \( r = 1, 2 \) 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 4 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 \( k = 1 \) and 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.

8 Conclusion

Although the performance of the BWT has made it an important addition to the long list of text compression algorithms, very little has been reported in searching directly on text compressed with the BWT. 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.

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 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. Preliminary results shows that our search aware modified MTF approach have a better compression ratio using DMC and further MTF than conventional MTF, although slightly worse than the optimized MTF implementation in Bzip2.

References


Figure 3: Results for k-mismatches (left column) and k-approximate match (right column)
Figure 4: Total search time including decompression overhead