Full-Text Search on Multi-Byte Encoded Documents

Raymond K. Wong*,† Fengming Shi* Nicole Lam*
*School of Computer Science & Engineering
University of New South Wales
and
†National ICT Australia
Sydney, Australia
wong@cse.unsw.edu.au

ABSTRACT

The Burrows Wheeler transform (BWT) has become popular in text compression, full-text search, XML representation, and DNA sequence matching. It is very efficient to perform a full-text search on BWT encoded text using backward search. This paper aims to study different approaches for applying BWT on multi-byte encoded (e.g. UTF-16) text documents. While previous work has studied BWT on word-based models, and BWT can be applied directly on multi-byte encodings (by treating the document as single-byte coded), there has been no extensive study on how to utilize BWT on multi-byte encoded documents for efficient full-text search. Therefore, in this paper, we propose several ways to efficiently backward search multi-byte text documents. We demonstrate our findings using Chinese text documents. Our experiment results show that our extensions to the standard BWT method offer faster search performance and use less runtime memory.

Categories and Subject Descriptors

I.7 [**Document and Text Processing**]: Miscellaneous; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*

General Terms

Performance

Keywords

Burrows Wheeler transform, multi-byte encodings, full-text search

1. INTRODUCTION

Large character sets (for example, Chinese, Vietnamese, Japanese and Korean) have several thousands of characters. Since an 8 bit byte can only represent 256 code points, we need more space to represent these large character sets.

DocEng'12, September 4–7, 2012, Paris, France.

Wide character encodings, which are fixed width, use correspondingly more bytes to hold the code points of the encoding being used. For example, 4 bytes can represent 4,294,967,296 unique code points. In fact, most Microsoft Windows application programming interfaces, as well as the Java and .Net Framework platforms, require that wide character variables be defined as 16-bit values, and that characters be encoded using UTF-16. Modern Unix-like systems generally require that 32 bit values are encoded using UTF-32. [1]

On the other hand, multi-byte encodings (usually variable length) use a sequence of bytes to represent a code point that cannot be represented in 8 bits. The first use of multi-byte encodings was in fact for the encoding of Chinese, Japanese and Korean, which have large character sets that are well in excess of 256 characters. Multi-byte encodings can be made more space efficient, but at the cost of increased complexity of processing. Furthermore, existing software that can handle 8 bit wide chunks can generally be ported to handle multi-byte encodings with relative ease, since each "chunk" in a multi-byte encoding is typically 8 bits in width.

As more and more information is encoded in these wider encodings, being able to search keywords from texts encoded in these encodings is vital. Recently, the Burrows Wheeler transform (BWT) [2] has become popular in text compression, full-text search [5], indexing and compressing XML documents [3], and DNA sequence matching [10]. In particular, full-text search on BWT encoded documents can be performed very efficiently using backward search [4].

However, to the best of our knowledge, the most recent study on BWT backward search was on ASCII encodings, with some proposals mentioning how they may be extended to multi-byte encoded texts.

In this paper we compare four methods of BWT and backward search for multi-byte encoded texts:

Approach A: This is a straightforward approach that most programmers would take, if space and runtime efficiency were not of concern. In this approach, characters in fixed width, m-byte encodings are treated as m-byte symbols during symbol comparisons and sortings in BWT and backward search. For example, instead of comparing and re-arranging byte by byte during BWT encoding, the texts will be compared and re-arranged in an m-bytes by m-bytes manner.

Approach B: In contrast to Approach A, this approach performs BWT and backward search on multi-byte

[1] From `http://en.wikipedia.org/wiki/Wide_character`

encoded text by treating every character as one byte long. This approach is ideal for UTF-8 encoded documents as UTF-8 is fully compatible with ASCII encodings. However, for most other encodings, an explicit check for word boundaries is needed to guarantee the matches returned from the backward search are correct. In this paper, we propose a concise bit array to mark the beginning byte of each character to resolve correct matches during backward search.

Approach C: BWT encoding and backward search performance is greatly affected by the length of text to be searched. Therefore, in this approach, we propose splitting the given text according to the byte location of each character, and then BWT encode each byte array separately. This will speed up the BWT encoding, and it will also enhance the performance of backward search (as shown in this paper). In this approach, an extra data structure and extra mappings need to be introduced in order to merge the matching bytes together to confirm the final matching characters. We will show that this approach outperforms Approach A and B in most situations.

Approach D: While Approach C is efficient in runtime, its extra data structure for mapping the bytes together can consume lots of memory. In Approach D, when the given character encoding is fixed width, we propose to maintain this mapping information only for every g bytes, so the extra data structure will be g times smaller. However, by doing so, we trade speed for space saving. Extra computations are needed to compute the mappings between the bytes from different byte arrays, since not every byte has its mapping information stored.

Finally, experiments to compare these approaches on different encodings (including UTF-8, UTF-16 and UTF-32) are included in this paper.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 provides an overview of BWT and backward search, and Section 4 provides some background informaton on multi-byte encodings for Chinese characters. Section 5 describes our proposed approaches and our experiment results are shown and discussed in Section 6. Finally, Section 7 concludes the paper.

2. RELATED WORK

The most well known data structures used for full-text indexing are suffix trees [13] and suffix arrays [12]. They support efficient pattern search with a tradeoff of large space usage. Although there have been active improvements on this from the research community, the hidden constants in the space bound are usually very large in practice. Exploiting the relationship between the suffix array and the Burrows Wheeler transform (BWT) [2], Ferragina and Manzini proposed the FM-index [4] that encapsulates the input text and takes the same space used by traditional compressors that store the input text only. Given these parameters, the FM-index still performs full-text search using backward search in a very efficient manner.

Sadakane [15] proposed a modified, reversible transform, namely `bwu`, where the sorting of the rows of the cyclic shifts matrix is done by ignoring the case of alphabetic symbols. He called this technique 'unification' and showed how to extend it to multi-byte character codes such as the Japanese EUC code [16].

Although word based indexes have been used widely in information retrieval [17] in the last 20 years (e.g., PAT trees and PAT arrays [6], word-based suffix trees have only been introduced recently [1]. The basic idea of word based indexes is to store just the text positions corresponding to word beginnings. Assume a document of size n with k words. [1] states that word suffix trees can be built with $O(k)$ working space in $O(n)$ expected running time. Recently, [7] improved this performance to $O(k)$ working space, in $O(n)$ running time, in the worst case.

Sparse string matching is similar to, but a more general problem than, word-based string matching. In sparse string matching, the set of points to be indexed is arbitrary, i.e., the indexed set of points are not necessarily the word boundaries. A special case where the indexed positions are evenly spaced was considered in [9]. In [9], algorithms on evenly spaced sparse suffix trees, i.e., representing every kth suffix of the text, were improved by using 'dual suffix trees'.

[18] stated that 16-bit Asian language texts are difficult to compress using conventional 8-bit sampling text compression schemes, and suggested a better scheme using lexicon dictionaries to first convert the input strings to 16-bit tokens. In this scheme, infrequent words that are not registered in the dictionary are encoded on a character by character basis. For similar reasons, most efficient word-based compression schemes either assume, employ or self-generate a dictionary for transforming the input strings into tokens before performing BWT [8]. This process uses the same concept as the unification technique in [15]. Most other efficient compression schemes for Chinese languages are mainly dictionary-based (e.g., [14])

The transformation of input strings into tokens that are more efficient for compression has proved to be very effective. However, these extra mappings increase the complexity and resource requirements (such as memory, CPU processing and possibly storage space), when our primary aim is for full-text search. Rather than proposing a new transform that may have different properties compared to the well-studied BWT, or utilizing word based indexes that may suffer from performance and space drawbacks as mentioned above, our proposal utilizes ordinary BWT. Hence, our proposal can be optimized by following any previous work or proposal on BWT optimization and its more efficient variants or auxiliary data structures (e.g. wavelet trees in RLFM [11]). Furthermore, we also study the use of byte-oriented BWT on multi-byte languages to avoid the word segmentation problem (which may be subject to the false positive problem as discussed later in this paper).

3. OVERVIEW OF BWT

3.1 BWT Basics

Let $T[1,n]$ be a text over a finite alphabet Σ. BWT permutes T, based on a reversible transformation, into a new string L that is easier to compress. Specifically, it consists of three steps:

1. append to the end of T a special character \$ smaller than any other character;

2. form a logical matrix M whose rows are the cyclic shifts of T\$ sorted in lexicographic order;

3. construct the transformed text L by taking the last column of M.

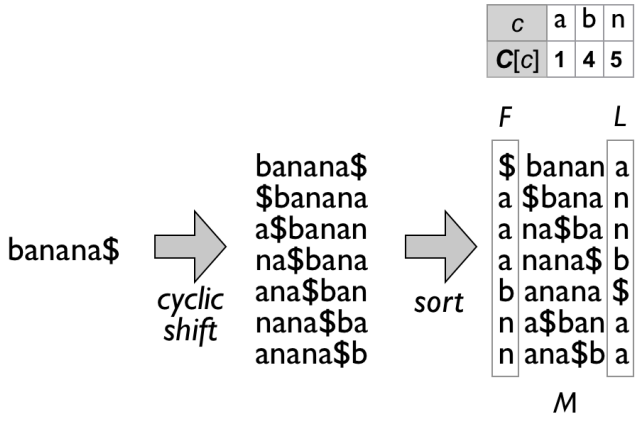

Figure 1: A BWT encoding process.

This BWT encoding process is illustrated in Figure 1 using the example string `banana`. Note that in particular, the first column of M, namely F, can be obtained by lexicographically sorting the characters of T\$ or L. The transformed text L usually contains long runs of identical characters, which can be compressed efficiently. For the scope of this paper, we will not go into detail on data compression using BWT. We will, however, focus on full-text search using BWT.

3.2 Backward Search

There is a bijective correspondence between the rows of M and the suffixes of T (and hence a strong relationship between L and the suffix array of T). This observation leads to the FM-index approach [4, 5]. For simplicity, FM-index focuses on two search operations, i.e., counting the number of matches in T and retrieving the positions of the matches. These operations are realised by the get_rows algorithm as shown in Algorithm 1. The algorithm exploits the properties of M: that all suffixes of T prefixed by a pattern $P[1, p]$ occupy a contiguous set of rows of M, and that this set of rows has start position $first$ and end position $last$, where $M[first]$ is the lexicographically smallest element of M that has prefix P. The value $(last - first + 1)$ is the total number of pattern occurrences.

Algorithm 1 Backward search algorithm for FM-index [4]

Algorithm get_rows($P[1, p]$)

```
1   i = p, c = P[p], first = C[c] + 1, last = C[c + 1];
2   while ((first ≤ last) and (i ≥ 2)) do
3       c = P[i − 1];
4       first = C[c] + Occ(c, first − 1) + 1;
5       last = C[c] + Occ(c, last);
6       i = i − 1;
7   if (last < first) then return "not found"
    else return (first, last).
```

In Algorithm 1, get_rows finds the set of rows prefixed by pattern $P[1, p]$. $C[1...|\Sigma|]$ is an array where $C[c]$ returns the number of occurrences in T of the characters {\$, 1, ..., c - 1}. Note that $C[c] + 1$ is the position of the first occurrence of c in F, if any. Function $\mathsf{Occ}(c, k)$ counts the number of occurrences of character c in the string prefix $L[1, k]$. It can be implemented with constant runtime cost by an auxiliary data structure using the extra $O(\frac{n}{\log n} \log\log n)$ space, as shown in [4]. Hence, get_rows will take $O(p)$ time, in the worst case. However, for wide character encodings such as UTF-32, if we apply BWT using 32-bit as a unit, the auxiliary data structure to compute Occ can be very large (i.e., requires a lot of memory to run efficiently).

4. MULTI-BYTE CHARACTER ENCODINGS

Without loss of generality, we use Chinese characters as illustrating examples through out this paper to explain how our proposed methods work. Our proposal would work for other multi-byte languages, such as many Asian and European languages.

There are two kinds of Chinese characters: simplified and traditional. GB2312 is the registered internet name for a key official character set of the People's Republic of China, used for simplified Chinese characters. EUC-CN is often used as the character encoding (i.e. for external storage) in programs that deal with GB2312, thus maintaining compatibility with ASCII. Two bytes are used to represent every character not found in ASCII. The value of the first byte is from 0xA1-0xF7 (161-247), while the value of the second byte is from 0xA1-0xFE (161-254). Big5 is a Chinese character encoding method used in Taiwan, Hong Kong, and Macau for traditional Chinese characters. It is a double-byte character set with the first byte being a lead byte with a value from 0x81 to 0xfe (or 0xa1 to 0xf9 for non-user-defined characters, while the value of the second byte is within 0x40 to 0x7e or 0xa1 to 0xfe.

UTF (especially UTF-8, UTF-16 and UTF-32) encodings are also becoming popular in many Chinese websites and software applications. UTF-8 is a variable-width encoding that can represent every character in the Unicode character set and is backward compatibility with ASCII. The first 128 code points (used to represent US-ASCII) need one byte. The next 1,920 code points need two bytes to encode. This includes Latin letters with diacritics and characters from the Greek, Cyrillic, Coptic, Armenian, Hebrew, Arabic, and so on. Three bytes are needed for characters in the rest of the Basic Multilingual Plane (which contains virtually all characters in common use). Four bytes are needed for characters in the other planes of Unicode, which include less common CJK characters and various historic scripts and mathematical symbols. UTF-16 is capable of encoding 1,112,064 numbers (called code points) in the Unicode code space from 0 to 0x10FFFF. It produces a variable-length result of either two bytes or four bytes per code point. UTF-32 uses exactly 32 bits per Unicode code point while all other Unicode transformation formats use variable-length encodings. Therefore, compared to variable length encodings, UTF-32's Unicode code points are directly indexable.

Consider an example of the following five Chinese characters '上下左右中'. Their corresponding byte encodings for

Table 1: Example of some Chinese characters in different encodings

	上	下	左	右	中
GB2312	0xC9CF	0xCFC2	0xD7F3	0xD3D2	0xD6D0
BIG5	0xA457	0xA455	0xA5AA	0xA56B	0xA4A4
UTF-8	0xE4B88A	0xE4B88B	0xE5B7A6	0xE58FB3	0xE4B8AD
UTF-16	0x0A4E	0x0B4E	0xE65D	0xF353	0x2D4E
UTF-32	0x0A4E0000	0x0B4E0000	0xE65D0000	0xF3530000	0x2D4E0000

GB2312 (EUC-CN), BIG5, UTF-8, UTF-16 and UTF-32 are shown in Table 1.

5. BWT ON MULTI-BYTE ENCODED TEXTS

Before we discuss BWT on multi-byte text documents, we note that this application is straightforward when the encoding is UTF-8, since UTF-8 is fully compatible with ASCII encodings. However, characters 0x0800 through 0xFFFF use three bytes in UTF-8, but only two in UTF-16. As a result, text in Chinese (and Japanese or Hindi etc.) could take more space in UTF-8 than in UTF-16, as shown in the examples in 1 and later in the experiment section, i.e., Table 2. In addition, in the case of Chinese character sets, the most popular encodings to date are still GB2312 and BIG5, and neither of them can be handled correctly by software applications that are designed for ASCII encodings. Therefore, we believe that it is significant and important to consider different ways of processing multi-byte text documents that are not UTF-8.

5.1 Approach A: Multi-byte, fixed width approach

In the context of using BWT for full-text search on multi-byte encoded texts, we consider each multi-byte encoded character as a word of m bytes, i.e., a fixed width, m byte atomic unit. We then apply all the BWT and backward search data structures and algorithms from the literature without any extension, except that all the comparisons and sortings are done by assuming every symbol is m bytes long instead of one byte. This is the most straightforward approach if one wants to apply BWT backward search on multi-byte encodings (except for UTF-8 since it is ASCII compatible).

5.2 Approach B: Single-byte approach

Approach B treats a multi-byte text document as single-byte encoded document to apply BWT. This has many advantages. For example, all standard BWT and backward search algorithms can be applied without modifications or transformations (such as the unification process proposed in [15]). Hence the usual space and runtime costs / analysis apply. Furthermore, optimization techniques for BWT and backward search, e.g., wavelet trees [11], can also be used. This is especially the case for all files encoded using UTF-8.

However, for most multi-byte encodings, if they are treated as single-byte encoded for performing BWT, the results found by a backward search should be filtered (by discarding those results that do not occur at word boundaries). Otherwise the search results may contain false positives.

For example, consider the Chinese character '空' that has a UTF-16 value of 0x7A7A; and the Chinese character '中' that has a BIG5 value of 0xA4A4. If we do not filter results that do not occur at word boundaries, searching the for character '空' using backward search on a single-byte based BWT of the UTF-16 encoded text '空空' will result in 3 matches instead of 2. Note that this problem may occur even for multi-byte characters with different byte values. For example, consider the text '的了牧' in GB2312 with byte values 0xB5C4 0xC1CB 0xC4C1, respectively. Backward search for '牧' (i.e., 0xC4C1) on this text will have 2 matches but in fact there is only one occurrence of '牧'.

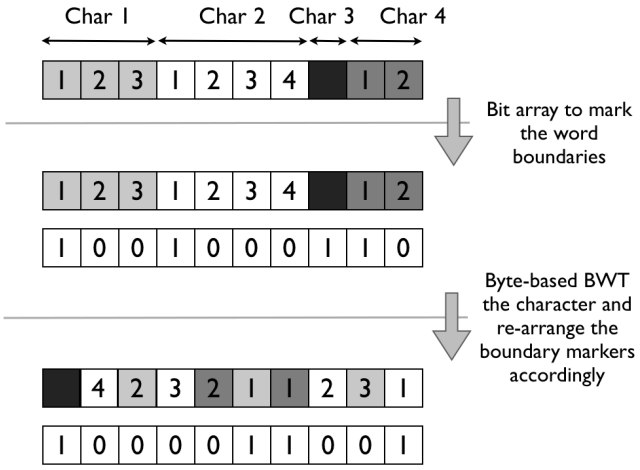

Figure 2: Byte based BWT encoding of multi-byte characters.

Discarding these false positives can be computationally costly. In addition, putting every byte into a single BWT index will result in unnecessary space occupancy of the index, waste cache / memory space and hence, slow down the backward search.

To minimize the space overhead of marking word boundaries, we use a bit array as illustrated in Figure 2. In Figure 2, we denote the byte position of each byte of a character by 1, 2, 3 and 4 respectively. After byte-based BWT transformation, these bytes will be relocated according to BWT. We will need to keep track of the relocations of all first bytes and relocate the boundary markers accordingly. Using the example above, two occurrences of 0xC4C1 on 0xB5C4 0xC1CB 0xC4C1 will be found by backward search. The byte 0xC4 of the first occurrence is not a boundary marker and hence will be discarded.

5.3 Approach C: Multi-byte characters split into byte arrays

To avoid explicit word boundary detection and to reject

false positives at an early stage during the search process, we propose Approach C – constructing byte arrays based on the byte location of each character in the text, and performing BWT on each byte array separately. For example, consider a multi-byte encoded text that encodes characters using one to four bytes as shown in Figure 3. In Figure 3, the bytes of

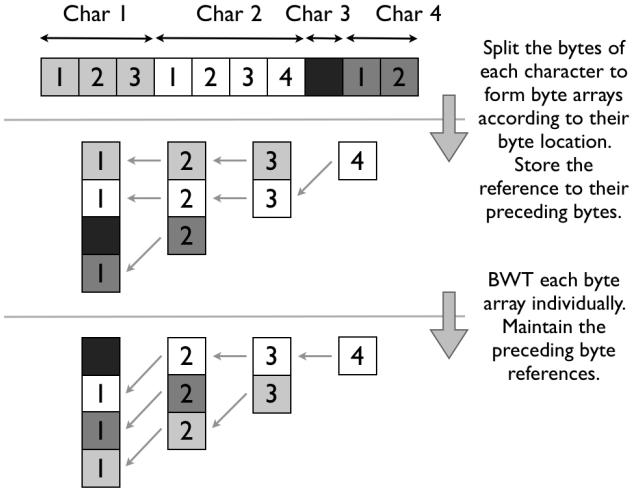

Figure 3: BWT encoding of byte arrays split from multi-byte characters.

characters: Char 1, Char 2, Char 3 and Char 4, in the given string are split according to the byte location of each character. i.e., all the first bytes of each character are grouped together in one array, and all the second bytes are grouped into another array, and so on. Same as Figure 2, in Figure 3, we denote the byte position of each byte of a character by 1, 2, 3 and 4 respectively (and we use different shading colors for different characters). Therefore, after the bytes are split according to their byte positions, all first bytes (all 1's) will be grouped into an array, ordered according to the order of their corresponding characters (i.e., the order of the shading colors are preserved). Similarly for all second, third and fourth bytes respectively. After that, when BWT is applied to each array, order of the shading colors will change.

When a byte (except the first byte) is being split from a character, reference to its preceding byte (represented as an array index to another array) is stored. The references between two arrays are stored as an array of preceding byte references, with its length equal to the shorter of the two arrays. Next, each byte array is BWT transformed separately, while references to the preceding bytes are maintained.

In practice, it is much faster to BWT encode several shorter single-byte arrays individually than to BWT encode all of them together as one array. Furthermore, the runtime memory requirement for efficient backward search is also lower for several smaller BWT transformed arrays than for the combined array, due to the larger data structure required for the Occ function when the BWT array is longer.

During backward search, bytes from the search string are split according to their byte locations (within each character) and are backward searched accordingly from their corresponding byte arrays. Next, matches from these arrays will be merged using their preceding byte references. This merging process is very efficient for fixed width character encodings. For example, for 32-bit fixed width encodings, we can use the matches from the array containing the fourth byte to construct the final match. This is done by starting each match from the fourth byte array and following its preceding byte reference to find all its preceding bytes (i.e. the third byte, followed by the second byte and finally the first byte). The byte array matches that are not included in any of these assembled characters will be discarded. For variable length encodings, the efficiency of this merging depends greatly on the distribution of the character lengths and query selectivity (i.e., number of matches).

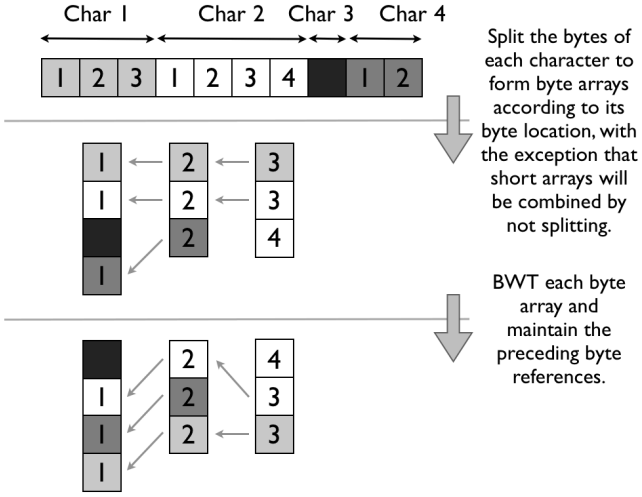

Figure 4: BWT encoding of byte arrays with short arrays merged.

To reduce the overhead of merging the matches from different byte arrays for variable length encodings, we propose not to split characters at byte location p, where most characters of a given document are encoded using less than p bytes. This proposal is shown in Figure 4. As shown in Figure 4, except for the first byte, a reference to the preceding byte will only be stored and maintained if the preceding byte is located in another array.

For example, suppose that most of the characters of a given UTF-8 document are English alphabets, with a small proportion of characters that are Chinese (and each of them is 3 bytes long). We can split the document into 2 arrays, with one of them containing all the first bytes and the other containing the rest of the bytes, and then BWT transform them separately.

Furthermore, many characters in documents encoded with wide character encodings (especially those larger than 16 bits) have trailing zeros (i.e., the last byte or last few bytes of these characters have the value zero). For example, in UTF-32, most characters can be encoded in two bytes and only extended character sets (and those characters are not very usually used) may need to use all four bytes to encode. Hence, we propose an extension to Figure 3 by trimming all trailing zeros, which would further reduce the space and runtime costs. This extension is illustrated in Figure 5. Note that different from the previous figures, the numbers in the byte arrays denote the actual byte values of the characters instead of byte positions, and 0s are used to represent the byte value 0s.

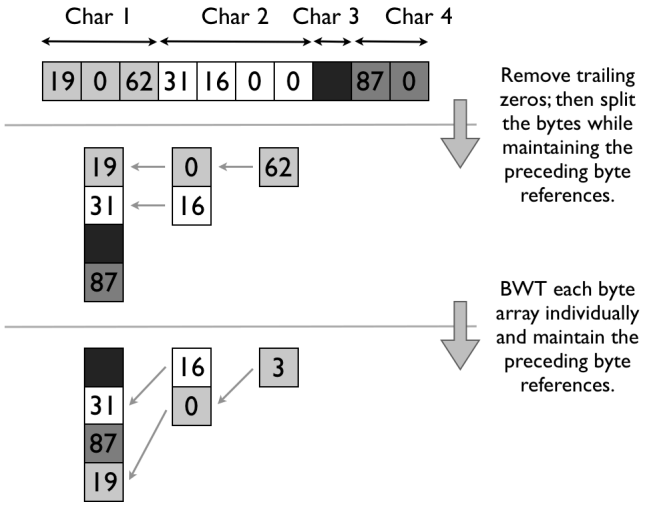

Figure 5: BWT encoding of byte arrays with trailing zeros trimmed.

5.4 Approach D: Location mappings with gaps

Given the string T, which contains n bytes and c characters. Each character is encoded by at most e bytes, and we use $T[i]$ to denote the i-th byte of T. Using Approach C (i.e. assuming that there are no trailing zeros and we split all the bytes), T will be split into T_1, T_2, ... T_e, where T_i contains the i-th byte of all characters in T. Note that $|T_1| \geq |T_2| \geq ... \geq |T_e|$. Since we need to maintain references to the preceding bytes, we will have the array J_2, J_3, ..., J_e where J_j maps the bytes from T_j to T_{j-1}. Hence, $|J_2| = |T_2|$, ..., $|J_e| = |T_e|$. Therefore, the memory overhead for merging the matches is the worst when $|T_1| = |T_2| = ... = |T_e|$, i.e., for fixed width encodings.

To address this problem for fixed width encodings, we propose Approach D, which leaves a gap g between any two references such that each array J_j will be g times smaller. i.e., we only maintain the mapping to the preceding byte every g bytes in the T_j (same as every g character in T).

During backward search on each byte array L_j, where L_j = BWT(T_j), we continue to BWT decode L_j for each match, even after we complete the backward search process, until a byte that contains a mapping to its preceding character is found. Next, we merge the matches from different byte arrays using these mappings. In summary, although we need to spend (on average) an extra $g/2$ number of BWT decoding iterations for each match from each byte array, we save g times the amount of space to handling byte merging.

6. EXPERIMENTS

Experiments are performed in the machine with Core i5-430M (2.26GHz), 2GB memory and 7200rpm harddisk. It is running Ubuntu 11.10 and experiments are implemented in Java (1.6.0_31).

6.1 File size

All four approaches described in the previous section are considered in our experiments. For the purposes of this paper, we assume that the BWT encoding and byte location information (where appropriate) has already been written out to file, and hence are not included in our timings below. Table 2 shows the file sizes for each approach.

Approach A treats every symbol as fixed length (and therefore all symbols are treated as 2 bytes or 4 bytes at a time, for UTF-16 and UTF-32 respectively, in any sortings and comparisons for BWT), i.e., n is smaller (and $|\Sigma|$ is larger when compared to Approach B). Note that UTF-16 is either two bytes or four bytes per code point, but all the UTF-16 datasets used in our experiments are two byte encoded. Approach B treats every symbol as one byte so that all comparisons and sortings are done in a byte by byte manner, resulting in a relatively larger n and smaller $|\Sigma|$. In Approach C, we split the bytes of each symbol according to their positions and BWT them separately. References to the preceding bytes are stored and maintained to assemble the characters back during backward search. To reduce the size of these additional position mappings, we only keep the mappings of every g character in the file for Approach D. In Approach D, we vary the size of this 'gap' (g) from 10 to 30 as shown in Table 2 (labelled D_{10}, D_{20} and D_{30} respectively). For Approach C and D, the file size is calculated by adding up the size of all the files, i.e., the files for the BWT arrays and the files for the preceding byte references. We have tested our scheme using >10 popular Chinese ebooks in our experiments. For conciseness, we have selected three of them, ranging from 800KB to 10MB when in UTF-32, to present in this section. To easily identify these files in our discussion, we associate each file with an ID, for example, F1 for the ebook '地缘看世界.txt'.

In Table 2, Approach C is large due to the additional location mapping files, and as shown in Approach D, the size of these files decreases when we increase the gap (g) for the location mappings. To further optimize these numbers, traditional compression techniques such as differential compression could have been applied to the location mappings.

6.2 Memory usage and loading time

Table 3: Runtime memory required (in MB) for each approach.

ID	Encodings	A	B	C	D_{10}	D_{20}	D_{30}
F1	UTF-8	-	8	8	-	-	-
	UTF-16	68	15	11	13	11	11
	UTF-32	68	28	13	14	13	13
F2	UTF-8	-	38	23	-	-	-
	UTF-16	355	49	32	36	32	32
	UTF-32	360	94	42	48	44	43
F3	UTF-8	-	135	83	-	-	-
	UTF-16	>500	161	112	142	124	119
	UTF-32	>500	346	128	156	135	129

We recorded the memory usage (Table 3) and loading time (Table 4) required for each approach.

The loading time includes the time to read the files, allocate the runtime memory, time to construct data structures (e.g., the bucket structures for Occ [4]) and time to construct auxiliary data structures (if any). Therefore, without any surprise, the loading time is closely related to the runtime memory usage.

Overall, Approach D has the best loading time and memory usage, while Approach C is also very close. They are

Table 2: File size (in bytes) for each approach.

ID	Filename	Encodings	Original	A	B	C	D_{10}	D_{20}	D_{30}
F1	地缘看世界.txt	UTF-8	609,282	-	609,282	1,409,814	-	-	-
		UTF-16	418,676	418,676	471,011	1,256,028	586,148	502,412	474,500
		UTF-32	837,352	837,352	942,021	1,256,028	586,148	502,412	474,500
F2	刘亚洲文集.txt	UTF-8	1,862,281	-	1,862,281	4,282,681	-	-	-
		UTF-16	1,304,646	1,304,646	1,467,727	3,913,938	1,826,510	1,565,582	1,478,606
		UTF-32	2,609,292	2,609,292	2,935,454	3,913,938	1,826,510	1,565,582	1,478,606
F3	中国通史.txt	UTF-8	7,353,094	-	7,353,094	17,043,818	-	-	-
		UTF-16	5,018,258	5,018,258	5,645,541	15,054,726	7,025,546	6,021,898	5,687,346
		UTF-32	10,036,484	10,036,484	11,291,045	15,054,726	7,025,546	6,021,898	5,687,346

better than Approach A and Approach B because of their shorter BWT arrays (instead of combining these shorter arrays into one larger BWT encoded array). Hence the space and runtime costs for constructing the data structures for BWT backward search are relatively cheaper. Compared to Approach C, the location mappings for Approach D are much smaller due to the gaps. However, the performance of Approach C is still very close to Approach D, since both approaches do not need to build complicated auxiliary data structures in addition to the mapping arrays themselves.

Table 4: Loading time (in ms) for each approach.

ID	Encod.	A	B	C	D_{10}	D_{20}	D_{30}
F1	UTF-8	-	322	505	-	-	-
	UTF-16	3799	780	566	473	488	458
	UTF-32	3728	1449	700	593	492	507
F2	UTF-8	-	887	1216	-	-	-
	UTF-16	21871	2430	1567	1108	1042	1004
	UTF-32	30546	4430	1780	1326	1237	1154
F3	UTF-8	-	3112	3861	-	-	-
	UTF-16	>60000	4870	5173	3628	3330	3199
	UTF-32	>60000	17409	5557	4095	3651	3725

6.3 Full-text search performance

In order to test the full-text search performance for each approach, we have designed three groups of search queries by varying the length of the search pattern and query selectivity. For selectivity, we tested for queries with no matches, with exactly one match, and with many matches. These queries, labelled with ID Q1...Q9, are listed in Table 5.

The search performance (in microseconds) for Approach B and Approach C on UTF-8 encoded documents are shown in Figure 6. We did not perform these UTF-8 experiments using Approach A and Approach D, since these two approaches would not offer any substantial benefits (i.e., Approach A adds unnecessary space usage and Approach D adds unnecessary computation complexity) as UTF-8 is compatible to ASCII encodings.

The UTF-8 encoded files in our experiments contains characters with length ranging from one byte (e.g., punctuations, numerics) to three bytes (e.g., Chinese characters). Therefore, we use these UTF encoded files on our Approach C with the extension that all the second to fourth bytes (if any) of characters are not split.

With no surprise, Approach B performs really well overall and performs the best when the search pattern is short and the selectivity is low. This is because UTF-8 is fully compatible with ASCII encodings, and hence the no frills BWT and backward search algorithms perform the most efficient without any additional data structures or operations such as checking for word boundaries. Approach C performs well too, due to the fact that all second, third and fourth bytes are BWT transformed and stored in one array. i.e., merging of matches happens between two BWT arrays (instead of three or four arrays). As a result, the extra cost of merging matches is minimized. Furthermore, Approach C performs better than Approach B for queries with long search patterns because the former breaks one long byte array into two. Hence, Approach C has a smaller Occ data structure footprint and better cache locality for the data in the array, i.e., faster backward searching for each byte. Long search patterns will amplify this effect and long search patterns usually result in high selectivity. Therefore, it will reduce the overhead of merging the matches between the two byte arrays. So, for long search patterns such as Q3, Q6 and Q9, Approach C outperforms Approach B. For the rest, Approach B performs better but Approach C is not far behind.

This has shown that our approach for splitting the multibyte encodings into byte arrays and then merging the matches is efficient and works well in practice. In fact, due to its relatively smaller Occ memory requirement as shown in Table 3, Approach C may be more preferrable on devices with limited memory.

Figure 7 and Figure 8 show the backward search performance (also in microseconds) on UTF-16 and UTF-32 encodings, respectively. Approach A is always the worst performer. It assumes 2 bytes as a basic unit for UTF-16 and 4 bytes for UTF-32. So, it will have a large and relatively slower Occ and hence worse backward search performance. Approach C works the best overall as it has smaller and hence faster Occ than Approach A and B. This advantage is highlighted by the queries with long search pattern such as Q3, Q6 and Q9. Furthermore, the cost of merging the matches from byte arrays may be cancelled out by the cost of checking word boundaries needed by Approach B. Note that in order to eliminate the possibility of false positives for Approach B for UTF-16 and UTF-32, word boundary checking is unavoidable. Approach D (with gap size of 10, 20, 30) also performs better than Approach B for the same reasons. However, it performs far worse than Approach B and C due to its more complex computations for byte merging. However, it offers better loading time than Approach

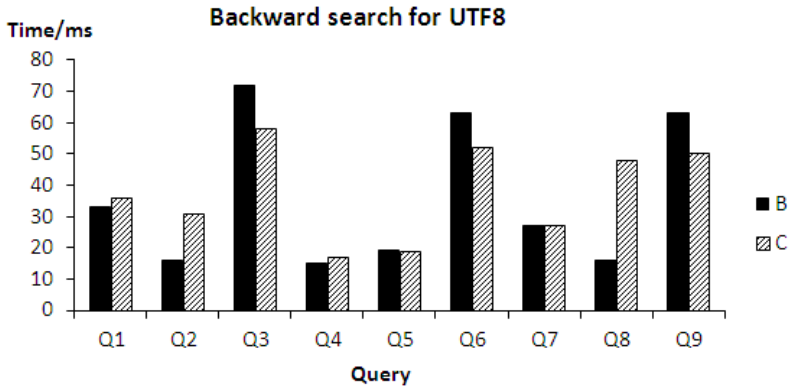

Figure 6: Search performance of different approaches for UTF-8 encodings.

Table 5: The list of search queries

ID	File	Query	#Results
Q1	F1	是中国挑战世界中心的路线图	0
Q2	F1	大中亚地区	22
Q3	F1	看了上面这张图大家就清楚了吧！中国沿海可以分为三个区域	1
Q4	F2	一九七人年参军	0
Q5	F2	金门战役	30
Q6	F2	这些商船是金门部队撤退用的。打平潭岛时，敌人不也派商船来撤兵嘛。	1
Q7	F3	他承史学工作者来信指教	0
Q8	F3	马克思	26
Q9	F3	长城外有一支细石器文化，它的特征是用燧石制成细小而锐利的锋刃	1

B and C, and a significantly smaller file size than Approach C. Note that the performance of Approach C is similar for both the UTF-16 and UTF-32 encodings, as trailing zeros are trimmed (cf. Section 5.3). Similarly for Approach D. As a result, UTF-32 will have a similar byte distribution to UTF-16. This is because, for most Chinese characters, the first two bytes of UTF-32 have the same values as UTF-16's, and the last two bytes of UTF-32 are zero (as demonstrated by the characters in Table 1.

7. CONCLUSIONS

We have discussed different approaches for applying BWT on multi-byte text documents. In particular, we have proposed techniques on efficiently backward search documents in multi-byte encodings. We use public domain Chinese ebooks to validate our findings, which are summarized as follows:

- Treating every character as a fixed width symbol and applying BWT directly on these wide symbols is straightforward but not computationally efficient, both in terms of space and runtime.
- If a document is in UTF-8, it is generally efficient to apply BWT and backward search on it, as if it is in ASCII. However, UTF-8 is not the most popular encoding for Chinese and Chinese characters encoded in UTF-8 consumes more space than in GB2312, BIG5 and UTF-16.
- Our proposed, new Approach C, which splits the bytes of characters according to their byte locations and performing BWT on each byte array separately, out performs other approaches in most cases.
- Further extensions, such as the introduction of gaps (in Approach D) and trimming of trailing zeros, have been proposed to optimize Approach C.

Overall, we believe that our findings are useful and valuable for readers who want to perform search (especially full-text search) on texts with multi-byte encodings.

For those who are interested in improving our findings and extending the experiments to other languages or encodings, we have placed the source code at `http://dbx.cse.unsw.edu.au` under an open source license.

Our ongoing work includes applying compression techniques to the mapping structures between the BWT arrays in Approach C and Approach D. We are also interested in extending the concept of this paper into compressed BWT structures such as RLFM [11].

Acknowledgements

The authors would like to thank the reviewers for their very useful comments on improving the paper.

8. REFERENCES

[1] A. Andersson, N. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.

[2] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, Palo Alto, CA, 1994.

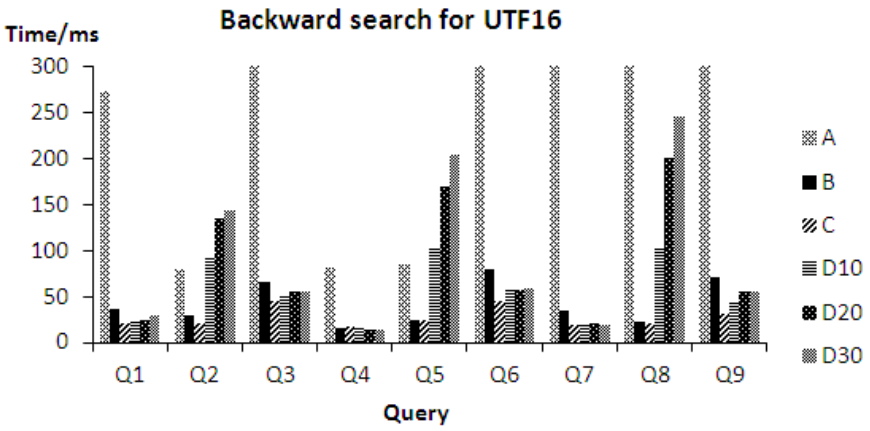

Figure 7: Search performance of different approaches for UTF-16 encodings.

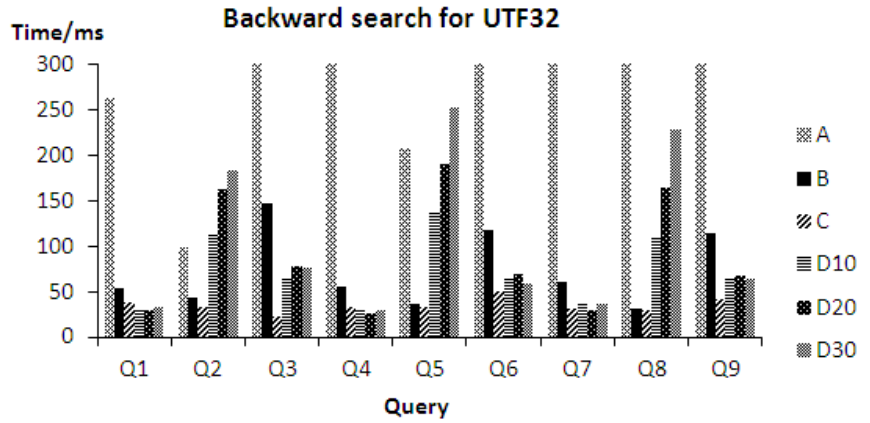

Figure 8: Search performance of different approaches for UTF-32 encodings.

[3] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching XML data via two zips. In *WWW 2006*, Edinburgh, Scotland, 2006. ACM.

[4] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 390–398, Washington, DC, USA, 2000. IEEE Computer Society.

[5] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, July 2005.

[6] W. B. Frakes and R. A. Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992.

[7] S. Inenaga and M. Takeda. On-line linear-time construction of word suffix trees. In *in Proc. 17th Ann. Symp. on Combinatorial Pattern Matching (CPM'06*, pages 60–71. Springer-Verlag, 2006.

[8] R. Y. K. Isal, A. Moffat, and A. C. Ngai. Enhanced word-based block-sorting text compression. In *ACSC '02 Proceedings of the twenty-fifth Australasian conference on Computer Science*. ACS, 2002.

[9] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In J.-Y. Cai and C. Wong, editors, *Computing and Combinatorics*, volume 1090 of *Lecture Notes in Computer Science*, pages 219–230. Springer Berlin / Heidelberg, 1996.

[10] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[11] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. of Computing*, 12(1):40–66, Mar. 2005.

[12] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Computing*, 22(5):935–948, 1993.

[13] E. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[14] G. H. Ong and S. Y. Huang. A data compression scheme for Chinese text files using Huffman coding and a two-level dictionary. *Information Sciences*, 84(1-2):85–99, 1995.

[15] K. Sadakane. A modified Burrows-Wheeler transformation for case-insensitive search with application to suffix array compression. In *DCC: Data*

Compression Conference. IEEE Computer Society, 1999.

[16] K. Sadakane. *Unifying Text Search and Compression: Suffix Sorting, Block Sorting and Suffix Arrays*. PhD thesis, The University of Tokyo, 2000.

[17] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, CA, 1999.

[18] S. Yoshida, T. Morihara, H. Yahagi, and N. Satoh. Application of a word-based text compression method to Japanese and Chinese texts. In *Data Compression Conference, DCC '99*. IEEE, 1999.