Efficient Indexing Algorithms for Approximate Pattern Matching in Text

Matthias Petri
School of of CS&IT
RMIT University and NICTA VRL
Melbourne, Victoria, 3000
matthias.petri@rmit.edu.au

J. Shane Culpepper
School of of CS&IT
RMIT University and NICTA VRL
Melbourne, Victoria, 3000
shane.culpepper@rmit.edu.au

ABSTRACT

Approximate pattern matching is an important computational problem with a wide variety of applications in Information Retrieval. Efficient solutions to approximate pattern matching can be applied to natural language keyword queries with spelling mistakes, OCR scanned text incorporated into indexes, language model ranking algorithms based on term proximity, or DNA databases containing sequencing errors. In this paper, we present a novel approach to constructing text indexes capable of efficiently supporting approximate search queries. Our approach relies on a new variant of the Context Bound Burrows-Wheeler Transform (k-BWT), referred to as the Variable Depth Burrows-Wheeler Transform (v-BWT). First, we describe our new algorithm, and show that it is reversible. Next, we show how to use the transform to support efficient text indexing and approximate pattern matching. Lastly, we empirically evaluate the use of the v-BWT for DNA and English text collections, and show a significant improvement in approximate search efficiency over more traditional q-gram based approximate pattern matching algorithms.

Keywords

Burrows-Wheeler Transform, Approximate Pattern Matching

1. INTRODUCTION

Approximate pattern matching is a classic problem in computer science with a wide variety of applications [10, 13]. For example, the role of approximate pattern matching in biological applications has been well documented [8]. Efficient solutions to approximate pattern matching can also be applied in a variety of Information Retrieval applications. Examples where approximate pattern matching can be applied in the IR domain include natural language keyword queries with spelling mistakes [10], OCR scanned text incorporated into indexes [10], language model ranking algorithms based on term proximity [12], or DNA databases containing sequencing errors [11].

ADCS'12, December 5–6, 2012, Otago, Dunedin, New Zealand.

The approximate pattern matching problem can be defined as follows: LOCATE, COUNT, or EXTRACT all occurrences of pattern P of length m in a text T of size n with at most k errors. If n is not large and only a few search queries will be performed, *on-line* algorithms such as `agrep` [26] can perform these operations in time proportional to the length of the text. However, on-line solutions are typically not sufficient for massive document collections, or situations which require a large number of queries to be run on the same collection. In these scenarios, building an index capable of supporting approximate matching queries is desirable. In this paper, we focus on a new approach to indexing and searching text collections allowing errors.

One viable approach to indexing text collections allowing errors is to use a modified self-index to support approximate text matching [23]. Most research in this domain has focused on providing worst case performance guarantees using a suffix array to perform fast substring matches [3]. In contrast, inverted indexes using q-grams can also be used, and generally perform well in practice despite providing no worst case performance guarantees. A q-gram index is simply an inverted index storing positions of all distinct substrings of length q in T. The q-gram index is used as a filtering tool to generate potential positions in T matching P. These positions must then be verified in the text using a variety of different *edit distance* based algorithms [17].

A weakness of traditional q-gram indexes is the use of fixed text segments of length q. If q is small, the inverted files can be very long for common q-grams, degrading performance for many queries. However, if q is large, then the size of the index grows at an unacceptable rate. Recently, Navarro and Salmela [18] show that using variable length q-grams can help find the best tradeoff between the length of the postings lists, and the total number of "grams" that must be indexed. Unfortunately, the approach to finding the substrings to be indexed still requires the creation of a suffix tree, which can dominate the construction time of the index.

1.1 Our Contribution

We present a new variant of the BWT, called the variable depth Burrows-Wheeler transform (v-BWT). We prove that the transform is always reversible, and show how it can be used to create a variable length q-gram partitioning for variable length q-gram indexes without requiring a complete suffix array. We describe how to use the v-BWT to construct a self-index, precluding the need to explicitly represent the v-grams using postings lists. We empirically evaluate the usefulness of our transform by comparing the number of verifications required when performing approximate matching on both DNA and English text. Finally, we describe future work where we intend to apply our new approach to common IR problems.

2. BACKGROUND AND RELATED WORK

We define a text $T[0..n-1]$ of n symbols and a pattern $P[0..m-1]$ of m symbols over an alphabet Σ of size σ. We denote the symbol \$ to be lexicographically smaller than all symbols in Σ. Without loss of generality, we require \$ to be the last symbol in T. We refer to the BWT over T as T^{BWT}, the k-BWT transformed text as $T^{k\text{-BWT}}$ and the variable depth transformed text as $T^{v\text{-BWT}}$. We define $\mathcal{M}$ to be a matrix containing all lexicographically sorted cyclic rotations of T. We similarly define $\mathcal{M}_k$ and $\mathcal{M}_v$ to be the equivalent matrix for the k-BWT and v-BWT.

2.1 Text Transformations

The BWT was originally proposed by Burrows and Wheeler [2] as the first step in a transform based compression system. The transform is used to permute T so symbols with similar context are grouped together. Conceptually, the BWT creates a matrix $\mathcal{M}$ consisting of all rotations of T. The rows in the matrix are then sorted based on the lexicographical ordering. T^{BWT} refers to the last column of $\mathcal{M}$. T^{BWT} is usually more compressible than T. The BWT is reversible in $\mathcal{O}(n)$ time without the need of any additional information using the following steps: (1) Recover the first column F of $\mathcal{M}$ by sorting T^{BWT} in lexicographical order. (2) Compute the mapping between the first F and last column L so $LF(i) = j$ if $F[j] = L[i]$. (3) Using the $LF()$ mapping, we can recover T from T^{BWT} in reverse order as $T[i] = T^{\text{BWT}}[LF(j)]$.

One of the main problems of constructing the BWT is that individual row comparisons in $\mathcal{M}$, or the equivalent suffix sorting comparisons in a suffix array construction algorithm can be computationally expensive. Two suffixes are compared by iterating over T starting from each respective position. In the worst case, each suffix comparison can take $\mathcal{O}(n)$ time. To alleviate this problem, Schindler [24] and Yokoo [27] independently proposed a bounded version of the BWT, the k-BWT. The k-BWT compares each row/suffix up to a depth of k symbols while stable sorting the equal rows based on initial text positions. This guarantees that each suffix comparison can be done in $\mathcal{O}(k)$ time. However, this implies that two suffixes are considered equal if they share the same k-prefix in $\mathcal{M}$. All rows in $\mathcal{M}$ sharing the same k-prefix are grouped together in a context group. Within a context group, rows are sorted based on the corresponding position in T. This implies that the suffix array positions in each context group are in monotonically increasing order.

To recover T from $T^{k\text{-BWT}}$, the boundaries of the context groups in BWT are required as LF() only returns the correct result if the BWT is fully sorted lexicographically. Definition 1 defines a bitvector D_k describing the context group boundaries:

DEFINITION 1. *For any $0 \leq k < n$, let $D_k[0, n-1]$ be a bitvector, such that $D_k[0] = 1$ and, for $1 \leq i < n$,*

$$D_k[i] = \begin{cases} 0 & \text{if } \mathcal{M}_k[i][0,k-1] = \mathcal{M}_k[i-1][0,k-1] \\ 1 & \text{if } \mathcal{M}_k[i][0,k-1] \neq \mathcal{M}_k[i-1][0,k-1] \end{cases}$$

LF() is still guaranteed to jump to the correct context group in $\mathcal{M}$ corresponding to the previous symbol in T as the individual k-groups are still sorted lexicographically [21]. Recall that with a context group, the rows are sorted based on the initial position in T. As T is recovered in reverse sequential order, during the recovery of T, each k-group is processed in reverse sequential order. Fortunately, the context group boundaries (D_k) can be recovered from $T^{k\text{-BWT}}$ in $\mathcal{O}(n)$ time [20]. To recover T from $T^{k\text{-BWT}}$, the k-group boundaries are recovered first. The $LF()$ mapping is then used to jump between context groups, while using D_k to process each individual context group in reverse sequential order. Interestingly, although the additional cost of recovering the context group is asymptotically worse than in the full BWT, T can be recovered faster from $T^{k\text{-BWT}}$ than from T^{BWT} due to the sequential access of each context group. The sequential access results in a significant cache effect not present in the random jumps induced by the BWT [4].

2.2 Self-Indexing

The BWT has been used in many compression systems to increase the compressibility of the text. Additionally, the BWT is the core of many compressed indexing schemes as there exists a duality between the BWT and the suffix array: $SA[i] = T^{bwt}[i]-1$. The duality between T^{BWT} and the suffix array over T allows searching in T using only compressed representation of T^{BWT} [5]. This type of text index is usually referred to as a self-index as T is not required to perform search. Self-indexes support the following operations efficiently:

COUNT(P, m):	Return the number of occurrences of P in T.
LOCATE(P, m):	Return all occurrences of pattern P in T.
EXTRACT(i, j):	Extract $T[i..j]$ from the self-index.

Self-indexes typically provide this functionality by allowing the following basic operations over T^{BWT}:

ACCESS(T^{BWT}, i):	Return $T^{\text{BWT}}[i]$.
RANK(T^{BWT}, i, c):	Return the number of times symbol c occurs in $T^{\text{BWT}}[0..i-1]$.
SELECT(T^{BWT}, i, c):	Return the position of the i-th occurrence of symbol c in T^{BWT}.

To support these operations, a wavelet tree [7] is built over T^{BWT} which supports all operations in $\mathcal{O}(\log \sigma)$ time. Wavelet trees over T^{BWT} take roughly the space of the compressed representation of T [6]. For an overview of wavelet trees refer to Navarro [14]. The main component of all operations in self-indexes is *backward search*, where all rows in $\mathcal{M}$ prefixed by P can be found in $\mathcal{O}(m \log \sigma)$ time by processing the pattern backwards by calculating LF() $2m$ times in $\mathcal{O}(\log \sigma)$ time as shown in Equation 1.

$$\text{LF}(i) = \text{LF}(i, c) = C[c] + \text{RANK}(T^{\text{BWT}}, i, c) \quad (1)$$

where c is the symbol $T^{\text{BWT}}[i]$, and $C[c]$ stores the number of symbols in T^{BWT} smaller than c. For a more detailed overview of self-indexes refer to Navarro and Mäkinen [16] or Ferragina et al. [6]. Similar techniques are used to allow searching in $T^{k\text{-BWT}}$ [21].

3. THE VARIABLE DEPTH TRANSFORM

The k-BWT sorts each suffix up to a fixed depth of k. Figure 1 shows the context size distribution for sorting depth $k = 2$ to 8. Note that as the sorting depth increases, the number of small contexts increase. However, even at a depth of 8, many large contexts groups remain which correspond to substrings in T that have a length of 8. Instead of sorting all rows deeper, we sort only context groups above a threshold v.

The k-BWT specifies the sorting depth k until the rows in $\mathcal{M}_k$ are sorted. The incomplete sorting of $\mathcal{M}_k$ results in rows in $\mathcal{M}_k$ being grouped together in context groups. Each row in an individual context group shares the same k symbol prefix. Instead of defining the sorting depth k, we define the maximum context group size v in $\mathcal{M}_v$ allowed. We continue to sort context groups with more than

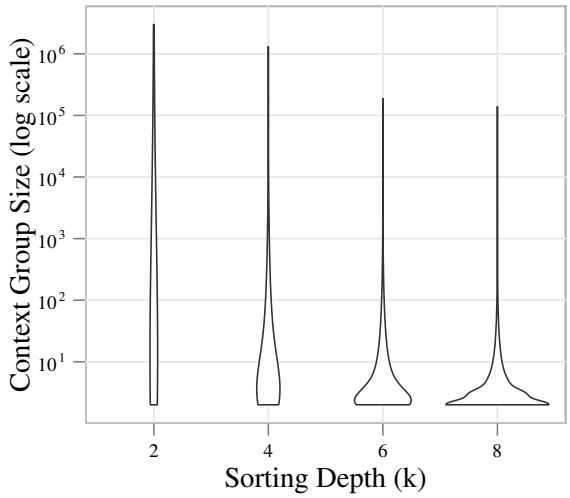

Figure 1: Context group size (logarithmic) distribution for different sorting depth k of the k-BWT for a 100 MB English text file.

v rows until the resulting context groups contain at most v rows. This implies that different parts of $\mathcal{M}_v$ will be sorted to different depths as not all k-grams in T occur equally often. Figure 2 shows an example of the v-BWT. Context groups '`p`' and '`$`' are sorted up to a depth 1. Context groups '`ay`' and '`yay`' are sorted to depth 2 and 3 respectively.

i	D_v	LF	**F**											L
0	1	11	**$**	y	a	y	a	y	a	p	y	a	y	a
1	1	10	**a**	**$**	y	a	y	a	y	a	p	y	a	y
2	1	5	**a**	**p**	y	a	y	a	$	y	a	y	a	y
3	1	1	**a**	**y**	a	y	a	p	y	a	y	a	$	y
4	0	3	**a**	**y**	a	p	y	a	y	a	$	y	a	y
5	0	8	**a**	**y**	a	$	y	a	y	a	y	a	p	y
6	1	6	**p**	y	a	y	a	$	y	a	y	a	y	a
7	1	9	**y**	**a**	**$**	y	a	y	a	y	a	p	y	a
8	1	4	**y**	**a**	**p**	y	a	y	a	$	y	a	y	a
9	1	0	**y**	**a**	**y**	a	y	a	p	y	a	y	a	$
10	0	2	**y**	**a**	**y**	a	p	y	a	y	a	$	y	a
11	0	7	**y**	**a**	**y**	a	$	y	a	y	a	y	a	p

Figure 2: v-BWT for T =`yayayapyaya$` including LF mapping and the context-group vector for threshold $s = 3$. The different sorting depths are boldfaced.

The bitvector D_v describing the context group boundaries is now defined as shown in Definition 2. $D_v[i]$ is 0 if $D_{v-1}[i]$ is 0 and the size of the context group containing row i in the previous sorting stage (d^i_{v-1}) was smaller or equal to v or if the v-prefix of row i is equal to row $i-1$. $D_v[i]$ is 1 otherwise.

Algorithm 1 v-BWT Forward Transform of text T with threshold v

```
VBWT (T[0...n-1], v)
Initialize SA[0...n-1]
Count symbols to create B_1
for each context group D_1[i...j] do
    RADIXSORT (T, SA, D_1[i...j], v, 2)
end for
for i ← 0 to n-1 do
    if SA[i] = 0 then
        T^{v-BWT}[i] ← T[n-1]
    else
        T^{v-BWT}[i] ← T[SA[i]-1]
    end if
end for
return T^{v-BWT}
```

```
FUNCTION RADIXSORT (T, SA, D[i...j], v, k)
if j - i + 1 ≤ v or k ≥ k_max then
    return
end if
COUNTSORT symbols in SA[i...j]
for all symbols ∈ SA[i...j] do
    Mark start of new context group in D
    RADIXSORT (T, SA, D[i...j], v, k+1)
end for
```

DEFINITION 2. *For any $1 \leq v < n$, let d^i_{v-1} be the size of the context group containing row i after sorting step $v-1$. Let $D_v[0, n-1]$ be a bitvector, such that $D_v[0] = 1$ and, for $1 \leq i < n$,*

$$D_v[i] = \begin{cases} 0 & \text{if } D_{v-1}[i] = 0 \text{ and } d^i_{v-1} \leq s \\ 0 & \text{if } D_{v-1}[i] = 0 \text{ and } d^i_{v-1} > s \text{ and} \\ & \mathcal{M}_v[i][0, v-1] = \mathcal{M}_v[i-1][0, v-1] \\ 1 & \text{otherwise} \end{cases}$$

The forward transformation of the v-BWT is outlined in Algorithm 1. We recursively perform radixsort for each of the context groups until the context group size is less then our defined threshold v. The algorithm returns the context group vector D_v, as well as the suffix array (SA) sorted up to variable sorting depth. The duality between the BWT and suffix arrays is used to create $T^{v\text{-BWT}}$ as $T^{v\text{-BWT}}[i] = T[\text{SA}[i]-1]$.

In order to bound worst-case sorting time, additional parameters are necessary. Let k_{min} to be the minimum sorting depth for all context groups and let k_{max} to be the maximum sorting depth. This guarantees a worst case runtime complexity of $\mathcal{O}(k_{max} n)$. In practice small values for the parameter k allow the bounded sorting depth transform to perform faster than full suffix array construction algorithms [4].

3.1 Transform Reversal

The k-BWT can be reversed using the bit vector D_k marking the beginning of the context boundaries as contexts are required to be processed in reverse sequential order. The LF() mapping is only guaranteed to "jump" into the correct context group [21]. Similarly, D_v can be used to reverse the v-BWT even though not all columns are sorted to the same depth k.

LEMMA 1. *The text T can be recovered from the permutation $T^{v\text{-BWT}}$ using the context group boundaries D_v and the* LF() *mapping.*

Proof. Equation 1 counts the number of occurrences of $c = T^{\text{BWT}}[i]$ in $T^{\text{BWT}}[0, i]$. If i represents the last row of a context group, the set of rows in $\mathcal{M}[0, i]$ is identical to the rows in $\mathcal{M}_v[0, i]$ as the context groups are lexicographically sorted. Therefore the number of occurrences of any c in $T^{v\text{-BWT}}[0, i]$ is the same as in $T^{\text{BWT}}[0, i]$. The different sorting depths k' and k'' do not affect the lexicographical order of different contexts as the sorting depth can only affect the order within a context group. So, $j = \text{LF}(i)$ maps correctly between two context groups $d^i_{k'}$ and $d^j_{k''}$ despite being sorted to different depths k' and k''. As $\text{LF}(i)$ must map to the correct preceding context group as in the k-BWT, using D_v we can process each context group in reverse sequential order to recover T. ■

To recover T from $T^{v\text{-BWT}}$ no additional information is required as the context boundaries D_v can be recovered from $T^{v\text{-BWT}}$ in $\mathcal{O}(k_{max}n)$ time, where k_{max} is the maximum sorting depth of any context group in $T^{v\text{-BWT}}$.

LEMMA 2. *D_k can be recovered directly from $T^{v\text{-BWT}}$ using no additional information.*

Proof. Recall that context information is not needed to restore the first k columns of $\mathcal{M}_k$. Instead of recovering $\mathcal{M}_v$ to a depth of k, we can recover based on the number of rows, s, with an identical prefix $\mathcal{M}_v[1...j]$. Let t be the the maximum number of rows in $\mathcal{M}_v$ that have the same prefix $\mathcal{M}_v[1...j]$ when sorted to a depth of j. If the number of rows t with the same prefix exceeds s at the current sorting depth j, this context group must be sorted up to depth $j + 1$. We continue recovering $\mathcal{M}_v$ in the current context group until $t \leq s$. The final context group recovered is D_v. ■

We now give an example of how to recover D_v from $T^{v\text{-BWT}}$. First, we recover F by sorting $L = T^{v\text{-BWT}}$ and initialize D_1 to the symbol boundaries. We also keep track of the $F \rightarrow L$ column mapping:

D_1	1	1	0	0	0	0	1	1	0	0	0	0
F	\$	a	a	a	a	a	p	y	y	y	y	y
L	a	y	y	y	y	y	a	a	a	\$	a	p
FL_1	9	0	6	7	8	10	11	1	2	3	4	5

Next, for all context groups larger or equal $s = 3$, we recover the next column in $\mathcal{M}$ using the initial FL_1 mapping. We update D_2 to include the new context boundaries and use the initial FL_1 mapping to create $FL_2[i] = FL_1[FL_1[i]]$ for context groups larger than v.

D_2	1	1	1	1	0	0	1	1	0	0	0	0
F	\$	a	a	a	a	a	p	y	y	y	y	y
		\$	p	y	y	y		a	a	a	a	a
L	a	y	y	y	y	y	a	a	a	\$	a	p
FL_2								0	6	7	8	10

Using FL_2 we recover the next column for context groups larger than v in a similar manner:

D_3	1	1	1	1	0	0	1	1	1	1	0	0
F	\$	a	a	a	a	a	p	y	y	y	y	y
		\$	p	y	y	y		a	a	a	a	a
								\$	p	y	y	y
L	a	y	y	y	y	y	a	a	a	\$	a	p

We now have D_v as the size of all of the context groups less than or equal to v, and can therefore be used to recover T from $T^{v\text{-BWT}}$.

4. VARIABLE LENGTH Q-GRAM INDEX

A q-gram is a contiguous sequence of symbols in a text T: $T[i..\ell]$. A q-gram index uses all q-grams in T to support approximate pattern matching over the text [15]. Traditional q-gram indexes are based on inverted files. For each distinct q-gram q_i in T, a list of positions of all occurrences of q_i are stored. These list can be d-gapped and compressed to reduce space. Individual inverted files are accessed through the vocabulary, which can be represented using a data structure such as a trie [18]. In large text collections, q-gram indexes have a few serious limitations. First, the number of distinct q-grams in T can grow exponentially with the size of q in the worst case. Second, certain q-grams tend to occur much more frequently than others.

Navarro and Salmela [18] propose a variable length q-gram index, where each variable length q-gram is required to have a uniform number of occurrences, and no q-gram occurs more than s times. The index is prefix-free, so no "selected" q-gram can be a prefix of any other q-gram in the index. To create the index, Navarro and Salmela first construct a suffix tree over T in $\mathcal{O}(n)$ time. Next the suffix tree is traversed in depth first order in $\mathcal{O}(n)$ time to retrieve the vocabulary of the index by pruning the suffix tree at nodes whose subtree contains at most s leaf nodes corresponding to suffix positions in T. Next, the position lists are sorted in increasing order in $\mathcal{O}(n \log \sigma)$ time and compressed in $\mathcal{O}(n)$ time. The total cost of constructing the index is therefore $\mathcal{O}(n \log \sigma + n \log s)$.

The v-BWT can significantly simplify the construction of a variable q-gram index. First, we create $T^{v\text{-BWT}}$ of T with threshold v. In the process the following components of the q-gram index can be created. The suffix tree partitioning of Navarro and Salmela [18] can be accomplished using D_v since each context group contains at most v rows. The postings lists can be obtained implicitly via SA_v, the suffix array used to sort T. Within each context group, the suffix array positions correspond to the entries in the postings list in the q-gram index. These lists are already sorted and do not require the $\mathcal{O}(n \log \sigma)$ sort described by Navarro and Salmela. In fact, we perform this step implicitly *while* creating the partitioning.

4.1 Representing the Vocabulary

Traditional q-gram indexes consist of two main components. The vocabulary stored as a trie, and a compressed inverted file for each distinct indexed q-gram containing all occurrences of the q-gram in T. To perform an approximate pattern search, a pattern is split up into $k + 1$ substrings. Next, for each substring the inverted list is loaded by querying the vocabulary. Previously we showed how to obtain a variable q-gram partitioning using the v-BWT. Here we show how we can replace the vocabulary of a variable q-gram index with a wavelet tree over $T^{v\text{-BWT}}$.

The v-BWT can be used to obtain a variable length q-gram partitioning equivalent to the index proposed by Navarro and Salmela [18]. Instead of using a trie to store the vocabulary, we can instead perform a backwards search using a compressed wavelet tree over $T^{v\text{-BWT}}$.

LEMMA 3. *Backwards search for any substring p_i can be performed in $T^{v\text{-BWT}}$ as long as the number of matching rows, $[sp, ep]$ in $\mathcal{M}_v$ are $\geq v$.*

Proof. Petri et al. [21] show that performing backwards search for a pattern up to length k works correctly in $T^{k\text{-BWT}}$ as each context is guaranteed to be sorted up to depth k. Therefore, performing $k - 1$ backwards probes is guaranteed to return the correct range of rows, sp, ep, in $\mathcal{M}_k$ for any p_i of length k. Similarly, every context group

d_k^i corresponding to a prefix $\mathcal{M}_v[0..j]$ is sorted if there are more than v rows in $\mathcal{M}_v$ prefixed by $\mathcal{M}_v[0..j]$ in $T^{v\text{-BWT}}$. Therefore, backwards search is guaranteed to result in the correct sp, ep in $\mathcal{M}_v$ if $ep - sp + 1 \geq v$. ■

So, we use a wavelet tree over $T^{v\text{-BWT}}$ to determine ranges in $\mathcal{M}_v$ which correspond to substrings p_i of P. The size of the range corresponds to the number of occurrences of p_i in T. For patterns with less than v occurrences, the range in $\mathcal{M}_v$ is not guaranteed to be continuous, so the $i-1$ context must be used instead. When this happens, all occurrences are still found, but the number of verifications is not guaranteed to be minimal.

4.2 Optimal Pattern Partitioning

To search for a pattern P with at most k errors, a q-gram index performs a *filtering* step whereby a string A is split into $k+1$ substrings $a_1 \ldots a_{k+1}$. For A to occur in a string B with at most k errors, at least one substring a_i must appear in B [19]. A q-gram index is used to find all *candidate* positions of P in T by partitioning P into $k+1$ substrings $p_1 \ldots p_{k+1}$ and retrieving the positions in T for all p_i. Navarro and Baeza-Yates [15] provide a dynamic programming algorithm which calculates the optimal partitioning of P into $k+1$ pieces to minimize the number of candidates. In the second step, a standard *edit distance* algorithm is then used to verify all candidates [17].

We now show how to use the optimal pattern partitioning algorithm proposed by Navarro and Baeza-Yates [15] and later used by Navarro and Salmela [18] to enable approximate searching using a wavelet tree over $T^{v\text{-BWT}}$. The key intuition of Navarro and Baeza-Yates's algorithm is to compute all m^2 possible substrings $P[i-j]$ and the resulting candidate list lengths in a matrix $R[i,j]$ of size $\mathcal{O}(m^2)$. Dynamic programming is then used to to retrieve the optimal partitioning by processing R in $\mathcal{O}(m^2 k)$ time [15, 18]. Using the backwards search (`BWS`) procedure, we compute $R[i,j]$:

$$R[i,j] = \begin{cases} |\langle sp, ep\rangle| & \text{if } \texttt{BWS}(P[i-j]) = |\langle sp, ep\rangle| \geq v \\ \infty & \text{otherwise} \end{cases}$$

Where $\langle sp, ep\rangle$ is the range in the suffix array prefixed by P. This range is only guaranteed to be continuous if $|\langle sp, ep\rangle| \geq v$, as within a context group rows are not lexicographically sorted. All substrings for which we cannot determine $\langle sp, ep\rangle$ are set to infinity in our calculations, thus making sure they are not included in the final partitioning of P into $p_{ij}, ..p_{j+1,l}, ...p_{m-1}$. For each substring we retrieve the corresponding $\langle sp, ep\rangle$ ranges in order to determine the parts of the suffix array containing the candidate positions.

4.3 Storing Postings Lists

Traditionally, the vocabulary contains pointers (file offsets) at which the individual postings list for the indexed strings (q-grams) are stored. As we are using a wavelet tree to store the vocabulary, we choose a different representation to store postings lists. Recall that within a context group in $T^{v\text{-BWT}}$, all corresponding suffix array positions are in ascending text order. We can therefore store a compressed version SA'_v of SA_v which d-gaps and compresses all offsets in a single context group in the same manner as is often used in postings lists for inverted indexes.

Unfortunately, the ranges $\langle sp, ep\rangle$ in SA_v cannot be used to find the corresponding position in SA'_v. So, we store an additional bitvector D'_v that maps context groups in SA_v to the corresponding starting positions in SA'_v. First we calculate the distance of sp to the corresponding context group start in SA_v using $\ell = \text{RANK}(D_v, sp, 1)$ and $t = \text{SELECT}(D_v, \ell, 1)$. Next we map the context group into the compressed representation SA'_v using $sp' = \text{SELECT}(D'_v, t, 1)$. Starting from sp' we skip the first $sp - \ell$ encoded numbers and then retrieve the next $ep - sp + 1$ encoded positions of $\langle sp, ep\rangle$. Note that $\langle sp, ep\rangle$ might span multiple smaller context groups which must each contain separately compressed d-gap lists.

The vocabulary and all auxiliary information needed to perform optimal partitioning can be stored using $H_{k_{min}}(T)$ space – the cost of storing a wavelet tree over $T^{v\text{-BWT}}$ with a minimal sorting depth of k_{min}. The text positions in SA'_v use variable byte coding which uses up to 30% more space than bit-compressed inverted lists, but allows for faster decoding time [25]. We further store H_0 compressed representations of D_v and D'_v [22].

5. EXPERIMENTS

5.1 Experimental Setup

In our experiments we use two datasets. We use the first 1 GB of a genome sequence created by concatenating the "Soft-masked" assembly sequence of the human genome (hg19/GRCH37) and the Dec. 2008 assembly of the cat genome (catChrV17e) in FASTA format. We remove all comment/section separators and replaced them with a separator token to fix the alphabet size. We call this data set DNA. Our second data set was generated from from the 2009 Clueweb web crawl available at `http://lemurproject.org/clueweb09.php/`. The first 64 WARC files in the directory `Clueweb09/disk1/Clueweb09_English_1/enwp00/` were concatenated together and null bytes in the text were replaced with 0xFF-bytes. The first 1 GB were used in our experiments which we denote as WEB.

We use a server with 2× Intel Xeon E5640 Processors with a 12 MB L3 cache, 144 GB of DDR3 DRAM running Ubuntu Linux version 12.04. The `g++` compiler version 4.6.3 with the basic compile option `-O3 -DNDEBUG -funroll-loop` was used. For basic succinct data structures, we use the succinct data structure library (`sdsl`) available at `http://github.com/simongog/sdsl/`. For suffix array construction we use the `libdivsufsort` library available at `http://code.google.com/p/libdivsufsort/`. In our v-BWT transform implementation, we used the cache efficient radixsort implementation proposed by Kärkkäinen and Rantala [9]. To compare our wavelet tree based vocabulary, we use a Hu-Tucker front-coding based vocabulary proposed by Brisaboa et al. [1].

5.2 Forward Transform Performance

Now we evaluate the runtime efficiency of our new transform and compare the forward transform with the k-BWT and the full BWT. We use the the suffix sorting algorithm implemented in `libdivsufsort` to construct the full BWT efficiently. Table 1 shows the runtime performance to create $T^{v\text{-BWT}}$, $T^{k\text{-BWT}}$ and T^{BWT} respectively for both test files.

	Time [sec]							
	k-BWT			v-BWT				BWT
	3	5	9	5	50	500	5000	
DNA	63	121	253	289	224	191	138	283
WEB	89	145	258	312	262	235	209	213

Table 1: Construction time (in seconds) of v-BWT, k-BWT, and the full BWT using `divsufsort`

The bounded transforms perform better for DNA than for WEB compared to the full BWT. For DNA, constructing the k-BWT is faster than constructing the full BWT. The v-BWT can be constructed more efficiently for sorting depths up to 5. Note that for $v = 5$, the v-BWT is "almost" identical to the full BWT, and only contexts up to size 5 remain. For $v = 50$ to 5000 the v-BWT can be constructed even more efficiently. The WEB data set can be constructed 40% faster with the full BWT compared to DNA. Induced suffix sorting reduces the number of suffix comparisons required to construct the suffix array. Therefore, the number of suffix comparisons needed is the limiting factor. Longer text comparisons have to be performed to determine the order of two suffix positions. The bounded transforms also perform slower for WEB. For $v = 5$, the variable transform is 40% slower than the induces suffix sorting method. As the sorting depth decreases, the v-BWT again outperforms the full BWT.

Overall the v-BWT can be constructed efficiently. However, we have not attempted to apply induced suffix sorting techniques commonly used during suffix array construction to speed up the construction process. This could potentially speed up the construction process significantly but remains future work.

5.3 Variable q-gram Index Construction

We now compare the construction time of our variable q-gram based index to the suffix tree method of Navarro and Salmela [18]. As described by Navarro and Salmela, we first construct a compressed suffix tree using `libsdsl`. Next we perform a depth first search traversal to determine the highest nodes in the suffix tree that have at most $v = 50$ children. The ranges in the suffix array corresponding to the marked nodes are recorded. Lastly, the individual ranges in the suffix array are sorted. We compare this approach to constructing an equivalent index using our v-BWT for $v = 50$. The different steps required in addition to the time required to build an index for threshold $v = 50$ for DNA are shown in Table 2. Note that the table further lists the cost to construct the vocabulary and compress the individual postings lists.

	Time [sec]	
Step	suffix tree	v-BWT
construct CST	736	-
suffix tree traversal	405	-
sort suffix array	453	-
create v-BWT	-	224
build vocabulary	24	
vbyte compress postings lists	30	
Total	1648	278

Table 2: Construction cost comparison of the method by Navarro and Salmela and the v-BWT for $v = 50$ on the DNA data set.

As expected, the construction of the suffix tree is the main bottleneck in the method of Navarro and Salmela. In fact, traversing the suffix tree to determine the different ranges in the suffix array for the approach is more expensive than creating the entire index using the v-BWT transform. Sorting each range in the suffix array in the suffix tree method is also computationally expensive, and unnecessary when using v-BWT. Overall, the v-BWT index can be constructed 5 times faster than the best known v-gram approach.

5.4 Variable q-gram Verifications

All q-gram based approximate pattern matching approaches use filtering to reduce verification costs. Potential matching candidates must still be verified using an *edit distance* algorithm. The goal of the filter is to minimize the number of verifications required to perform approximate search. We now evaluate the number of verifications required by each indexing approach. First, we perform 1000 approximate pattern searches for pattern lengths 20 to 50 using different error levels. The patterns were randomly sampled from each data set. Figure 3 shows the number of candidate positions which must be verified after pattern partitioning is performed. For this experiment, we only compare $k = 5$ and $v = 50$ using a wavelet tree as the vocabulary for both approaches. For DNA, the number of positions requiring verification tend to be higher than for WEB as the data is more uniform, and the alphabet size is smaller. The v-BWT always outperforms classical k-BWT partitioning. The variance in the WEB data set is higher than for DNA, while DNA generally requires more verifications using the k-BWT based approach. The v-BWT approach outperforms the k-BWT approach for the DNA data set by several orders of magnitude except for patterns of length 20 with error rates of 3 and 4. This implies that P has to be split into 4 and 5 substrings respectively. As the sorting depth for the k-BWT is 5, we conjecture that the substrings being evaluated with the v-BWT are rarely longer than in the k-BWT.

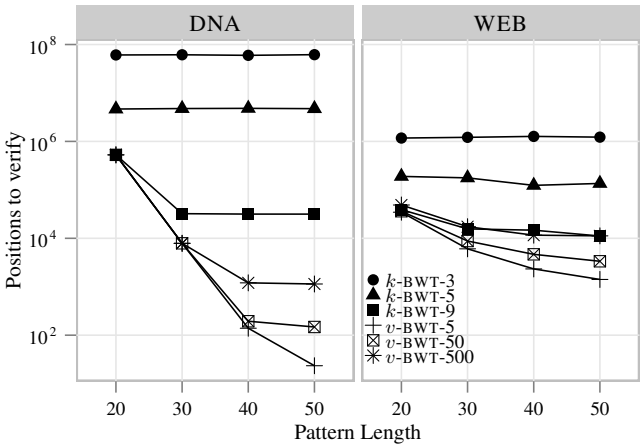

Figure 4: Number of verifications required for k-BWT for variable $k = 3, 5, 9$ and $v = 5, 50, 500$ for 2 errors for DNA and WEB data sets.

Next, we show how the number of verifications varies with different sorting parameters. We choose only small k values as the number of potential dictionary entries can, in the worst case, grow exponentially as k increases. Similarly, we choose the parameter v to have similar construction costs as our chosen k values. Figure 4 shows the *mean* number of verifications required for 1000 approximate pattern searches for patterns of length 20 to 50 for variable transform parameters. The number of verifications required using the standard fixed q-gram k-BWT approach decreases as k increases due to the fact that longer substrings can be matched. For $k = 9$, performance is similar to that of the v-BWT for patterns of length 20. Generally, for all k the k-BWT approach requires more verifications. The average number of verifications required stays roughly constant for the fixed q-gram approach whereas the mean number of verifications decreases using the variable length q-gram approach as the length of the pattern increases. As the pattern length increases, our approach can match longer variable length q-grams during the optimal partitioning phase. Longer q-grams occur less frequently. Therefore, the number of verifications required decreases.

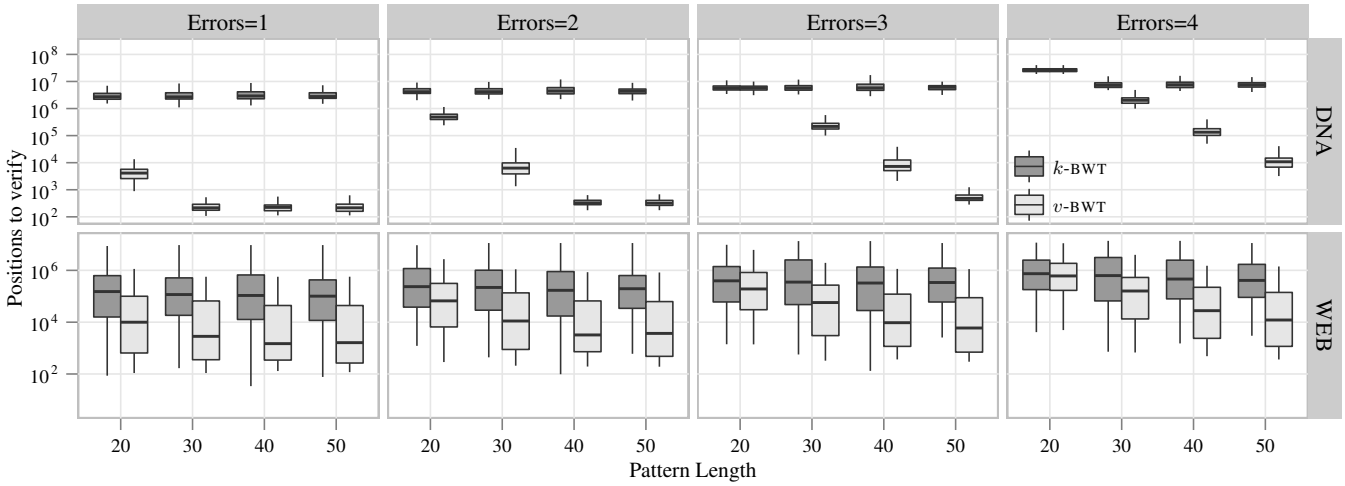

Figure 3: Number of verifications required for k-BWT with $k = 5$ and v-BWT with $v = 50$ for the DNA and WEB data sets.

5.5 Variable q-gram Vocabularies

Last, we evaluate the performance of a wavelet tree based vocabulary by comparing the performance to a Hu-Tucker front coding (HTFC) vocabulary as proposed by Brisaboa et al. [1]. We compare the running time required by each vocabulary type to execute the optimal partitioning algorithm with the space required to store the vocabulary.

We use the HTFC dictionary as follows: Insert the first row of each context group in M_k and M_v into the HTFC vocabulary. For the k-BWT based approach, insert the suffix with length k. For the v-BWT, insert the suffix that maximizes the longest common prefix (LCP) of the adjacent context groups. For example, given the context group corresponding to `bba` with adjacent contexts groups `ba` and `bbbb`, insert `bba`. Since the sorting depth for each context group is not stored, the unique prefix representing the context group is calculated during insertion into the vocabulary. Inside the HTFC vocabulary, each string is inserted into a block of size B and compressed using front coding and Hu-Tucker coding. During the query phase, binary search is performed over the first entries of each block to find the candidate block the search string must occur in. Next, the block is sequentially decompressed until the string is found, or the next uncompressed block entry is larger than the search string. The overall performance of the vocabulary depends on the block size B (which determines the number of sequential decompression steps), and the number of blocks in the vocabulary (which depends on B and the number of strings inserted). We modify the original HTFC of Brisaboa et al. [1] to support prefix search by returning the first and last entry in the vocabulary for which a given search string is a prefix. On a successful search for a pattern P, the vocabulary returns to numbers i, j corresponding to the positions of the first and last entry in the vocabulary prefixed by P. We then use D_v, the bitvector describing the context group boundaries, to calculate the number of times P occurs in T by determining the i-th and j-th one bit in D_v: $sp = \text{SELECT}(D_v, i, 1)$ and $ep = \text{SELECT}(D_v, j + 1, 1) - 1$. We can then determine the number of occurrences of P in T as $occ = ep - sp + 1$.

For the wavelet tree vocabulary we use three different Huffman shaped wavelet trees. The first wavelet tree uses uncompressed bitvectors (`wt-bv`). The second (`wt-15`) and third (`wt-63`) use H_0 compressed vectors proposed by Raman et al. [22]. By using compressed bitvectors and a Huffman shaped wavelet tree the cost of storing $T^{v\text{-BWT}}$ is roughly equal to the size of the compressed representation of T. We determine the number of occurrences of a pattern P by performing backwards search as described previously. Figure 5 shows the time-space trade-offs for DNA and the v-BWT with $v = 5, 50, 500$. We show the mean time in seconds per partitioning step compared to the vocabulary size in MB.

For $v = 500$ the HTFC vocabulary outperforms both wavelet tree dictionaries using compressed bitvectors (`wt-15` and `wt-63`) while the wavelet tree using uncompressed bitvectors is much faster, but also uses much more space. This can be explained as follows. For $v = 500$, the number of context groups is small. Therefore, not many strings are inserted into the HTFC while the wavelet tree always contains all rows in M_v. Searching in the HTFC vocabulary depends on the number of strings in the vocabulary. Therefore, for $v = 500$ with block sizes $B = 5, 50$, the HTFC vocabulary is both smaller and roughly as fast as `wt-15` and `wt-63`.

However, as the sorting requirements of each context group is increased, the wavelet tree becomes more competitive in space usage. For $v = 50$, more strings are inserted into the HTFC vocabulary, requiring more space. Still, the compressed wavelet trees are roughly twice as large as the HTFC based vocabulary. When $v = 5$, the space required for the HTFC vocabulary is larger than the wavelet tree. The wavelet tree using uncompressed bitvectors now uses less space while allowing much faster search. As the sorting depth is increased, the wavelet tree approach becomes more compelling. Moreover, increasing the sorting depth also decreases the number of verifications required. Therefore, using a wavelet tree based vocabulary can be both space efficient and support efficient filtering-based approximate pattern matching.

6. CONCLUSION AND FUTURE WORK

We have presented a new context based sort transformation: the v-BWT. We show how the transform differs from previous context sorting transforms. In addition, we show that the v-BWT can be used to create text indexes which can be used for approximate pattern matching. Our experimental evaluation shows that the transform can be used to construct variable length q-gram indexes five times faster than previous methods. We show that the number of verifications that have to be performed using a variable q-gram index are less than traditional fixed q-gram based indexes. We further show that using wavelet trees over the transform output can be used as the vocabulary component in the approximate index.

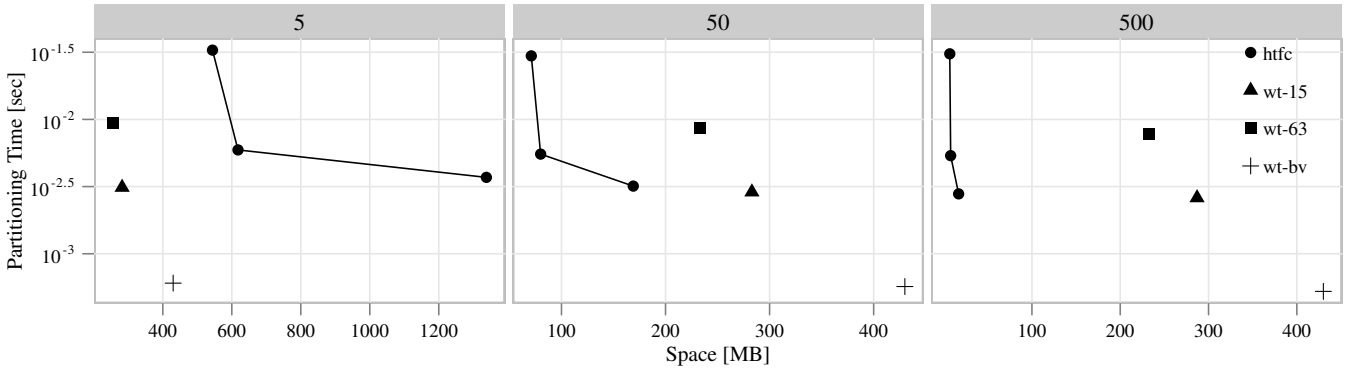

Figure 5: Time and Space trade-offs during optimal pattern partitioning for HTFC ($B = 5, 50, 500$) and wavelet tree based dictionaries for DNA using v-BWT and $v = 5, 50, 500$.

Future work includes: Adopting fast, induced suffix sorting based BWT construction algorithms to construct $T^{v\text{-BWT}}$; Using the transform to construct suffix arrays on disk; and exploring the viability of variable length q-gram indexes for a wide variety of common IR and Bioinformatics search problems.

Acknowledgement. This work was supported in part by the Australian Research Council and by NICTA. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] N. R. Brisaboa, R. Cánovas, F. Claude, M. A. Martínez-Prieto, and G. Navarro. Compressed string dictionaries. In *SEA*, pages 136–147, 2011.

[2] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, May 1994.

[3] H. L. Chan, T. W. Lam, W. K. Sung, S. L. Tam, and S. S. Wong. Compressed indexes for approximate string matching. *Algorithmica*, 58(2):263–281, 2010.

[4] J. S. Culpepper, M. Petri, and S. J. Puglisi. Revisiting bounded context block-sorting transformations. *Software Practice and Experience*, 42 (8):1037–1054, August 2012.

[5] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398, 2000.

[6] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: from theory to practice. *Journal of Experimental Algorithmics*, 13:1.12–1.31, 2009.

[7] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850, 2003.

[8] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, New York, New York, USA, 1997.

[9] J. Kärkkäinen and T. Rantala. Engineering radix sort for strings. In *SPIRE*, pages 3–14, 2008.

[10] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.

[11] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[12] D. Metzler and W. B. Croft. A markov random field model for term dependencies. In *SIGIR*, pages 472–479, 2005.

[13] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[14] G. Navarro. Wavelet trees for all. In *CPM*, pages 2–26, 2012.

[15] G. Navarro and R. A. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electron. J.*, 1(2), 1998.

[16] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[17] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages.

[18] G. Navarro and L. Salmela. Indexing variable length substrings for exact and approximate matching. In *SPIRE*, pages 214–221, 2009.

[19] G. Navarro, R. A. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Eng. Bull.*, 24 (4):19–27, 2001.

[20] G. Nong, S. Zhang, and W. H. Chan. Computing inverse ST in linear complexity. In *CPM*, pages 178–190, 2008.

[21] M. Petri, G. Navarro, J. S. Culpepper, and S. J. Puglisi. Backwards search in context bound text transformations. In *CCP*, pages 82–91, 2011.

[22] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA*, pages 233–242, 2002.

[23] L. Russo, G. Navarro, A. Oliveira, and P. Morales. Approximate string matching with compressed indexes. *Algorithms*, 2(3):1105–1136, 2009.

[24] M. Schindler. A fast block-sorting algorithm for lossless data compression. In J. A. Storer and M. Cohn, editors, *DCC*, page 469, Los Alamitos, California, March 1997. IEEE Computer Society Press.

[25] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *SIGIR*, pages 222–229, 2002.

[26] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[27] H. Yokoo. Notes on block-sorting data compression. *Electronics and Communications in Japan, Part 3*, 82(6):18–25, 1999.