Multiary Wavelet Trees in Practice

Alex Bowe
alex.bowe@rmit.edu.au

Honours Thesis

Supervisor: Simon Puglisi

School of Computer Science and Information Technology
RMIT University
Melbourne, AUSTRALIA

November 2010

Abstract

Self-index data structures for pattern matching based on Suffix Arrays and the Burrows-Wheeler Transform have recently grown popular. The fundamental operation in these self-indexes is the rank query: $rank(i, c)$ requests the number of occurrences of symbol c before position i in a string. The *wavelet tree* is currently the data structure of choice for implementing rank queries. Current wavelet tree implementations encode the Burrows-Wheeler Transform as a hierarchy of binary strings, which they then store as RRR sequences; the RRR structure offers $O(1)$ rank queries and zeroth-order entropy compression for binary strings. A generalisation of the RRR tecnique extends wavelet trees to have higher order encoding, that is, increased branching, in theory making traversal faster. To support the implementation of such a multiary wavelet tree, this thesis investigates the generalisation of RRR to sequences over small alphabets. We also analyse the use of concatenated bitmaps to represent sequences over small alphabets, thus allowing continued use of the binary RRR structure. Our results show that multiary wavelet trees are faster than their binary counterparts, but require large amounts of memory in the case of the generalised RRR. We also show binary RRR on concatenated bitmaps to be an effective, practical alternative.

Keywords: data structures, algorithms, string search, compression, indexes

Contents

1 Introduction

As collections of text grow larger, our need to find information in them and infer patterns and rankings increases, too. Suffix arrays, first described by Manber and Myers [18], allow a variety of complex pattern matching and pattern discovery problems to be performed in optimal time. There are many areas where suffix arrays are likely the most appropriate data structure for the task, including:

- Genome analysis [1, 9];
- Searching for patterns in non-word data such as images, multimedia signals and DNA [5];
- Searching for patterns in oriental languages; as some oriental languages do not have spaces between certain particles, an inverted file would be insufficient;
- Pattern discovery and visualisation using arc diagrams, as proposed by Wattenberg [27].

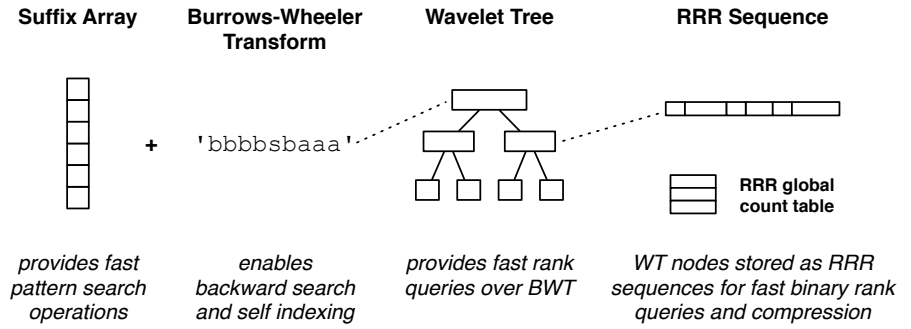

Figure 1: Overview of the key data structures and their relationship with each other. When a Suffix Array and a Burrows-Wheeler Transform (BWT) [3] are combined, they form an FM-Index [7]. FM-Indexes use Backward Searches on the BWT to provide fast pattern matching, counting, and substring extraction operations. Backward Search requires rank operations, which are best implemented using a Wavelet Tree (WT) [12]. A WT encodes a string as a hierarchy of bit vectors, which it uses to answer general rank queries using $\log \sigma$ *binary* rank queries. Binary rank queries can be answered in $O(1)$ time when the bit vector is stored as a RRR sequence [25], which utilise a global table of pre-calculated ranks. The RRR data structure also offers compression of the Wavelet Tree. (we defer a detailed discussion of RRR to Section 3.2)

Due to its performance in these important applications, the improvement of suffix arrays has been the focus of intensive research over the past 20 years. *Self-indexing* is one of these improvements. Self indexes support fast pattern counting (reporting the number of occurrences of a pattern in a text), fast pattern matching (reporting the positions of each pattern occurrence) and extraction of arbitrary substrings of the original text, including the original text

itself. These operations in some sense allow us to replace the original text with a self index.

One such self index is known as the *'FM-Index'*, proposed by Ferragina and Manzini [7]. An FM-Index utilises a sparse Suffix Array and the *Burrows-Wheeler Transform* (BWT) [3] (the two leftmost diagrams in Figure 1) of the original text, and supports fast pattern matching operations by using the so-called *'backward search'* method.

Backward searching with a BWT requires a fundamental operation called a *'rank query'*, which counts the number of occurrences of a given symbol up to a given position in the string.

While a naive implementation of the rank query operation might inspect every symbol of the BWT, and do so in $O(N)$ time where N is the length of the BWT, it is possible to improve this to $O(\log \sigma)$ time where σ is the size of the alphabet. To do this, we construct a Wavelet Tree [12] over the BWT (the third diagram in Figure 1). A Wavelet Tree is constructed by encoding the string as a hierarchy of bit vectors. These bit vectors are then used to answer rank queries by traversing and performing *binary* rank queries[1]. A more detailed description of Wavelet Trees appears in Section 3.

Binary rank queries can be performed in $O(1)$ time when the bit vectors are stored in a RRR sequence, as proposed by Raman, Raman and Rao [25]. RRR also offers compression of the Wavelet Tree.

The key motivation behind this project arises from the increasing number of papers, such as Ferragina et al. 2007 [8], Yu et al. 2009 [29], and Barbay et al. 2010 [2], which utilise *Multiary* Wavelet Trees as a theoretical tool, that is, Wavelet Trees with a branching factor greater than two. However no known implementations of Multiary Wavelet Trees exist. This thesis aims to address this need, and bring theory closer to practice. It was expected that increasing arity would improve the time performance of self-indexes which use a BWT.

1.1 Our Contribution

The contribution of this paper is the experimental analysis of two types of Multiary Wavelet Tree. The first of these, to our knowledge, is the first implementation of the Generalised RRR structure, which is used to support rank queries on small alphabets, as first suggested by Ferragina et al. [8].

We also propose a new data structure, a Multiary Wavelet Tree that utilises Binary RRR using concatenated bitmaps to represent sequences on small alphabets as bit vectors.

We discover that although Multiary Wavelet Trees are faster, Generalised RRR requires significant memory for the supporting global table in its current form, which, depending on the size of the text, may make these Multiary Wavelet Trees impractical. We show that our new data structure, which continues to use Binary RRR, is a practical way to make rank queries faster. Additionally, we observe that the BWT gives rise to a sparse class list as the arity increases.

1.2 Roadmap

The rest of the thesis is organised as follows. Section 2 begins with some notation, followed by background on the problem domain. We then discuss how

[1] Binary rank queries are also known as *popcounts* in other literature.

each data structure is built and used together.

Section 3 provides a description of Binary Wavelet Trees and their use of the RRR structure, followed by a discussion in Section 4 of how our Multiary Wavelet Trees are designed, including some implementation details of interest.

In Section 5 we describe how we measured the time and space performance for each Multiary Wavelet Tree variation, and provide a rationale of our test dataset. Our results are presented in Section 6, accompanied by a discussion of the apparent trends.

The conclusion follows in Section 7. We discuss the practicality of each Multiary Wavelet Tree variation, and in Section 8 we consider how their performance might be improved.

2 Preliminaries

Figure 2: Array representation of 'mississippi' string.

Throughout this paper we represent the string we are searching in as S of length $|S| = N$, and the pattern we are searching for as P. $S[i]$ represents the symbol located at position i in S, and $S[i..j]$ represents the substring of S beginning at position i and ending at j inclusive. Strings are one-based, so in Figure 2 $S[1]$ = 'm', $S[3]$ = 's', and $S[1..3]$ = 'mis'.

The i^{th} *suffix* is thus defined as $S[i..N]$, so the 1^{st} suffix in Figure 2 is $S[1..12]$ = 'mississipi$', and the 5^{th} suffix is $S[5..12]$ = 'issippi$'. The i^{th} prefix is defined as $S[1..i]$, so the 5^{th} prefix in Figure 2 is $S[1..5]$ = 'missi'.

The *log* operation is base 2 unless otherwise stated.

2.1 Background

In 1970 Knuth, Morris and Pratt (KMP) discovered an algorithm to match patterns in time proportional to the length of the text [14, 20]. If the text is large, then KMP becomes ineffective for ranking and pattern discovery. Moreover, KMP is only useful for exact matches.

One alternative to KMP for document ranking is the use of an inverted index, but they must work with keywords and are thus inappropriate for many applications, such as searches on certain oriental languages, and other strings that do not have a clear definition of keywords (MIDI, for example). Suffix arrays are also more efficient than inverted files for searching phrases or partial patterns [13, 19]. This was originally possible with the suffix tree [20, 28] (a precursor to the suffix array), although suffix trees require three to five times as much space [18].

2.2 Suffix Arrays

In its simplest form, a suffix array can be constructed for a string $S[1..N]$ like so:

1. Construct an array of pointers to all suffixes $S[1..N]$, $S[2..N]$, ..., $S[N..N]$.
2. Sort these pointers by the lexicographical (i.e. alphabetical) ordering of their associated suffixes.

Figure 2 shows an example string, 'mississippi'. The construction of the corresponding suffix array is shown in Figure 3. Construction of suffix arrays is now a well studied problem [24].

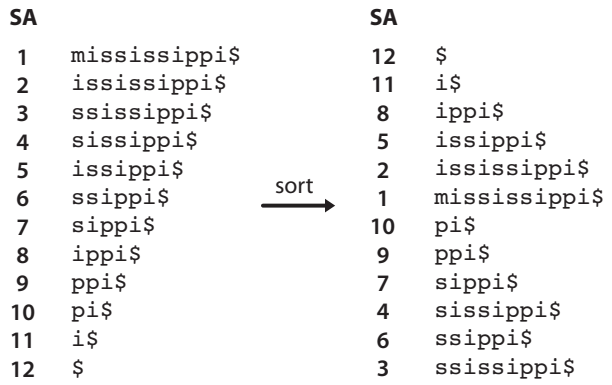

Figure 3: Construction of Suffix Array for 'mississippi'.

BWT	SA	
i	12	$
p	11	i$
s	8	ippi$
s	5	issippi$
m	2	ississippi$
$	1	mississippi$
p	10	pi$
i	9	ppi$
s	7	sippi$
s	4	sissippi$
i	6	ssippi$
i	3	ssissippi$

Figure 4: Suffix Array and Burrows-Wheeler Transform for 'mississippi' string.

2.3 Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) is a string of length N defined by the suffix array SA and the original text S. In particular, $BWT[i] = S[SA[i] - 1]$, and $BWT[1] =$ '\$', that is, the i^{th} symbol of the BWT is the symbol prior to the i^{th} suffix in the Suffix Array SA. See Figure 4.

As proposed by Ferragina and Manzini, when a BWT is stored alongside a Suffix Array, it is known as a *FM-Index* [7], which supports backwards search.

2.4 Rank Query

Munro [21] describes how to do rank queries on binary strings in $O(1)$ time using $o(n)$ bits of extra space. Much of the early work on rank queries focussed on binary strings.

A rank query on the string S is defined as $rank_S(i, c) = n$, with n being the number of times symbol c appears in the range $S[1, i]$. This paper omits the subscript when the string we are querying is clear from the context. For example in Figure 5, $rank(9, s) = 3$. If $i \leq 0$ then $rank(i, c) = 0$.

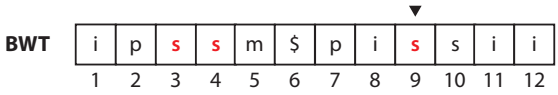

Figure 5: Rank query $rank(9, s) = 3$ on the Burrows-Wheeler Transform of 'mississippi'.

2.5 Backward Search

Since any pattern P in S is a prefix of a suffix, and because the suffixes are lexicographically ordered, all occurrences of a search pattern P lie in a contiguous portion of the Suffix Array. In earlier implementations, the range that this pattern lies on would be located by using successive binary searches. Backward search utilises the BWT in a series of paired rank queries, improving the query performance considerably [4, 6, 8, 10, 16, 17, 19, 22]

Backward search issues $|P|$ pairs of rank queries, where $|P|$ denotes the length of the pattern. The paired rank queries are:

$$s' = C[P[i]] + rank(s-1, P[i]) + 1$$
$$e' = C[P[i]] + rank(e, P[i])$$

Where s denotes the start of the range, initially at $s = 1$, and e is the end of the range, initially $e = N$.

In Figure 7 there is a column F, which contains the first symbol for each suffix. Note that the F column is not stored as we store it encoded as C instead.

C is an array[2] containing the count of all symbols in Σ which sort lexicographically before $P[i]$, where Σ is the alphabet from our original string S, as in Figure 6.

In the first iteration we query the final character of the pattern, so $i = |P|$. For each iteration, we decrement i until $i = 1$. This maintains the invariant that $SA[s..e]$ contains all the suffixes of which $P[i..|P|]$ is a prefix, and hence all locations of $P[i..|P|]$ in S. This is illustrated in Figure 8 through to Figure 10. If at any stage $e < s$, then the pattern does not exist in our original string.

An example is given in Figure 7 through to Figure 10, where the pattern *'iss'* is searched for in the string 'mississippi', starting with $i = 3$, $P[3] =$'s'. The working for each rank query is shown below each figure. We represent the current symbol as c to avoid confusion between 's' and s and s'.

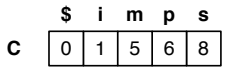

Figure 6: Table C of number of occurrences in F of each symbol which sorts alphabetically before the displayed symbol.

[2]Note that we are indexing C by a symbol $P[i]$, so this may be implemented with a suitable hash function.

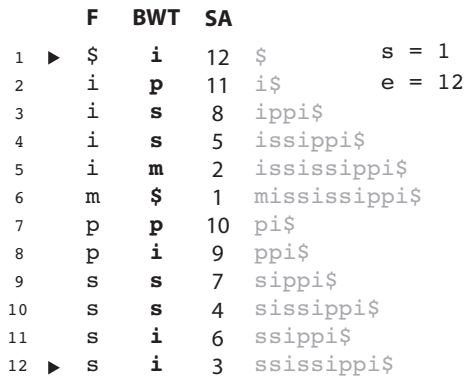

Figure 7: First stage of backwards search for 'iss' on 'mississippi' string - before any rank queries have been made.

1. Starting from $s = 1$ and $e = 12$ as in Figure 7, and $c = P[i]$ ='s' where $i = 3$, we make our first two rank queries:

$$s' = C[c] + rank(0, c) + 1 = 8 + 0 + 1 = 9$$

$$e' = C[c] + rank(12, c) = 8 + 4 = 12$$

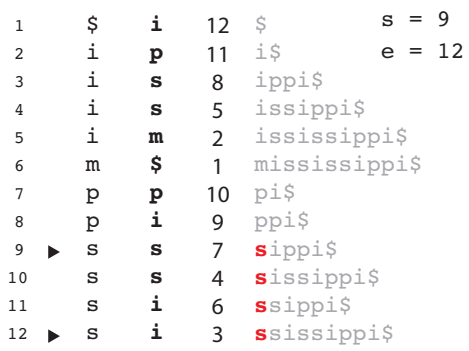

Figure 8: Second stage of backwards search for 'iss' on 'mississippi' string. All the occurrences of 's' lie in $SA[9..12]$.

2. From $s = 9$ and $e = 11$ as in Figure 8, and $c = P[i] =$'s' where $i = 2$, our next two rank queries are:

$$s'' = C[c] + rank(8, c) + 1 = 8 + 2 + 1 = 11$$

$$e'' = C[c] + rank(12, c) = 8 + 4 = 12$$

		F	BWT	SA		
1		$	i	12	$	s = 11
2		i	p	11	i$	e = 12
3		i	s	8	ippi$	
4		i	s	5	issippi$	
5		i	m	2	ississippi$	
6		m	$	1	mississippi$	
7		p	p	10	pi$	
8		p	i	9	ppi$	
9		s	s	7	sippi$	
10		s	s	4	sissippi$	
11	▶	s	i	6	**ss**ippi$	
12	▶	s	i	3	**ss**issippi$	

Figure 9: Third stage of backwards search for 'iss' on 'mississippi' string. All the occurrences of 'ss' lie in $SA[11..12]$.

3. From $s = 11$ and $e = 12$ as in Figure 9, and $c = P[i] =$'i' where $i = 1$, our final two rank queries are:

$$s''' = C[c] + rank(10, c) + 1 = 1 + 2 + 1 = 4$$

$$e''' = C[c] + rank(12, c) = 1 + 4 = 5$$

		F	BWT	SA		
1		$	i	12	$	s = 4
2		i	p	11	i$	e = 5
3		i	s	8	ippi$	
4	▶	i	s	5	**iss**ippi$	
5	▶	i	m	2	**iss**issippi$	
6		m	$	1	mississippi$	
7		p	p	10	pi$	
8		p	i	9	ppi$	
9		s	s	7	sippi$	
10		s	s	4	sissippi$	
11		s	i	6	ssippi$	
12		s	i	3	ssissippi$	

Figure 10: Fourth and final stage of backwards search for 'iss' on 'mississippi' string. All the occurrences of 'iss' lie in $SA[4..5]$.

3 Binary Wavelet Trees and RRR

This section describes Binary Wavelet Trees, which provide fast rank queries over strings with an alphabet size larger than 2, and the RRR structure, which is used for fast *binary* rank queries and compression.

3.1 Binary Wavelet Trees

One of the most effective data structures for answering rank queries is the Wavelet Tree [4, 6, 8, 12, 16].

Binary Wavelet Trees encode the BWT (or any string over which we require fast rank queries) as a perfect binary tree of bit vectors, to enable $O(\log \sigma)$ time rank queries, where σ is the size of the alphabet. The tree is defined recursively as follows:

1. Encoding half the alphabet as 0, and half as 1, for example:

$$\Sigma = \{\$, i, m, p, s\}$$

$$enc(\Sigma) = \{0, 0, 0, 1, 1\}$$

2. Group each 0-encoded symbol, $\{\$, i, m\}$, as a sub-tree;

3. Group each 1-encoded symbol, $\{p, s\}$, as a sub-tree;

4. Reapply to each sub-tree recursively until there is only one symbol left.

The encoded binary Wavelet Tree *root node* for the 'mississippi' BWT is shown in Figure 11. For a more detailed example, showing the whole tree, see Figure 12.

i	p	s	s	m	$	p	i	s	s	i	i
0	1	1	1	0	0	1	0	1	1	0	0
1	2	3	4	5	6	7	8	9	10	11	12

Figure 11: Root node of Binary Wavelet Tree encoding for 'mississippi' BWT. Each symbol in the string is assigned a bit (0 or 1) depending on which half of the alphabet it is from.

After the tree is constructed, a rank query on the Wavelet Tree can be done with $\log \sigma$ binary rank queries on the bit vectors. For example, if we wanted to know $rank(6, e)$ in Figure 12, we use the following procedure which is illustrated in Figure 14. We know that $enc(e) = 0$ at this level, so:

1. At the root node, count the number of 0s in the range [1..6], which is given by $rank(6, 0) = 4$. This gives us the index to query in our 0-child.

2. Calculate $rank(4, 1) = 2$, as e is encoded as 1 at this child. We traverse the 1-branch this time, with the next index as 2.

3. $rank(2, 1) = 2$, which we use as the index in the child on the 1-branch, with our next index as 2.

4. $rank(2, 0) = 2$, as e is encoded as 0 here. Since the children at this point are leaf nodes, we return 2 as our result.

Hence the result of $rank(6, e)$ is 2. If we store these nodes in RRR, binary rank queries can be answered in $O(1)$ time.

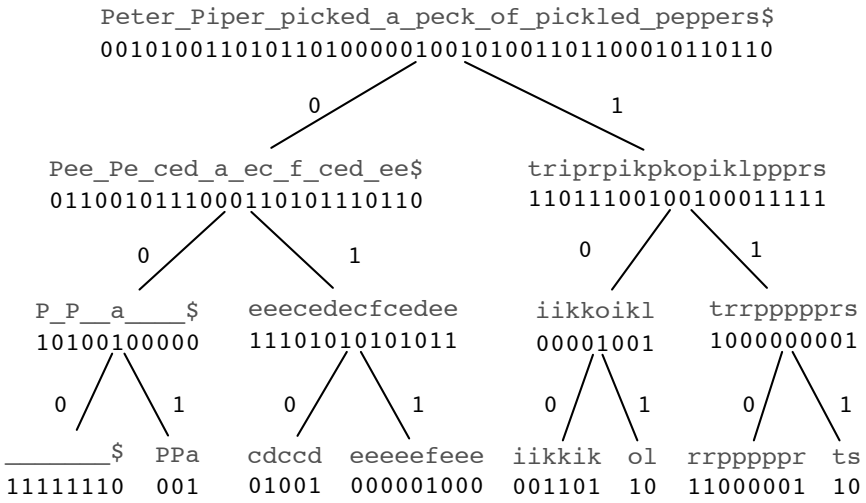

Figure 12: Binary Wavelet Tree for 'Peter Piper...' where spaces are displayed as underscores.

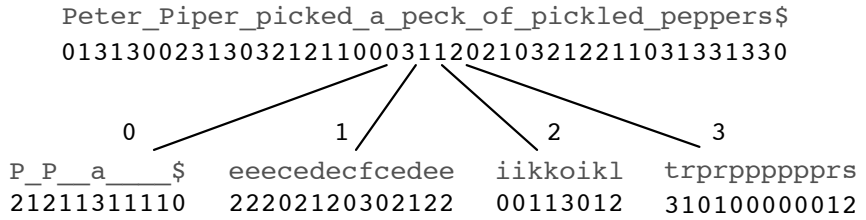

Figure 13: 4-ary Wavelet Tree for 'Peter Piper...' where spaces are displayed as underscores.

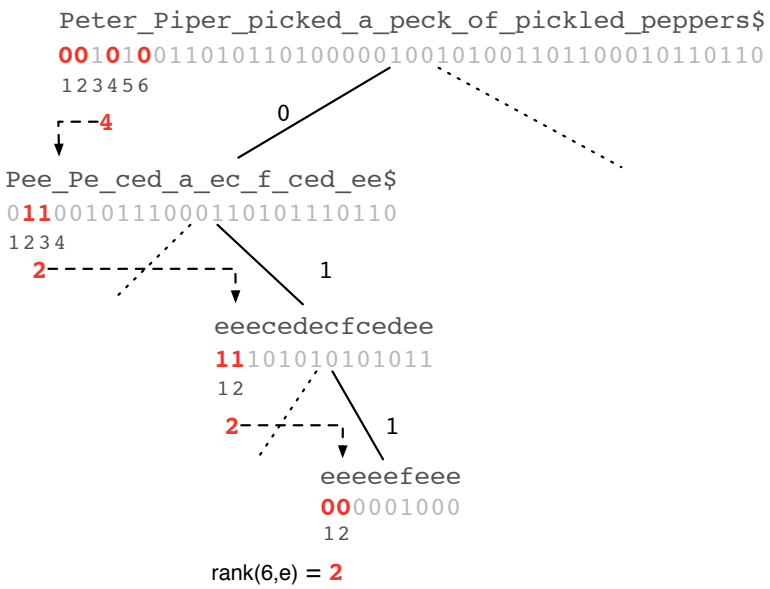

Figure 14: Answering $rank(6, e)$ over the Binary Wavelet Tree for 'Peter Piper...' where spaces are displayed as underscores.

3.2 RRR

RRR was first proposed by Raman et al. [25]. The purpose of RRR is to encode a bit sequence in such a way that supports $O(1)$ time binary rank queries. It also provides implicit (i.e. automatic) compression, requiring $NH_0(S) + o(N)$ where $H_0(S)$ is the *zeroth-order empirical entropy* of S.

Zeroth-order empirical entropy is a lower bound for the average code word size when a symbol is mapped to the same code word irrespective of the context in which they appear. It can be calculated as $H_0(S) = \sum_{i=1}^{\sigma} \frac{N^i}{N} \log \frac{N}{N^i}$, for an alphabet Σ size of σ, and N^i is the number of occurrences of the i^{th} element of Σ in our text $S[1..N]$.

The classical $O(1)$ time implementation of binary rank queries required the N bits of the original bit sequence, plus $O(\frac{N \log \log N}{\log N}) = o(N)$ additional bits [11].

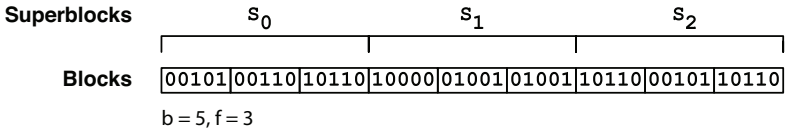

Figure 15: Block division scheme for 'Peter Piper...' Wavelet Tree's root node bit vector.

To construct the RRR we divide the bit vector into several so-called *superblocks*, we then divide these superblocks further, into *blocks* of b bits each, as

in Figure 15. We call the number of blocks in a superblock the *superblock-factor*, f.

For each of these blocks we store a class number c, which in the binary case is the number of 1s in the block. This is used as a lookup key in a table G, which is a table of tables, and will be explained shortly. We also store offset o, which is an index into the table at $G[c]$. Each offset value o tells us precisely which of the possible blocks of class c a block is. See Figure 17.

G is a table having subtables $G[c]$ for each class c. For every possible permutation of c 1-bits, $G[c]$ contains an array of cumulative sums for each position in the block of given offset and class - this is illustrated in Figure 16. It is important to note that the size of o varies, since the number of possible permutations of c bits, and hence entries in $G[c]$, is $\binom{b}{c}$, and can be encoded in $\log \binom{b}{c}$ bits.

The reason for grouping blocks into superblocks is to avoid iterating over each block to answer a rank query; a query requires the sum of the ranks of previous blocks as well, as depicted in See Figure 17. If we store the sum of all block ranks up to a superblock boundary, then a rank query $rank(i, c)$ can be answered like so:

1. Calculate which block our index is in as $i_b = i/b$.
2. Calculate which superblock our block resides in as $i_s = i_b/f$.
3. Set `result` to the sum of previous ranks at i_s boundary (which is precalculated).
4. Using each blocks class-offset pair (c, o) after the boundary at i_s, add the rank for that entire block to `result`.
5. Repeat previous step until we reach i_b. We then add $rank_{i_b}(j, c)$ to our result, where $j = i \mod b$, and is the position we are querying local to i_b. Our final answer is `result`.

With the superblock we also store an initial address for the variable-length offset values. After finding the first offset address in a superblock, we calculate the next offset address in bits according to the blocks class c as $\lceil \log \binom{b}{c} \rceil$ bits, which we add to the current address. See Figure 17, which shows what is calculated for each superblock[3].

It is possible to support Multiary Wavelet Trees using RRR with a more extensive class allocation, which we will discuss in Section 4.4.

[3]We omit the first superblock, since the first offset address is easy to find, and the sum of ranks before the first superblock is always 0.

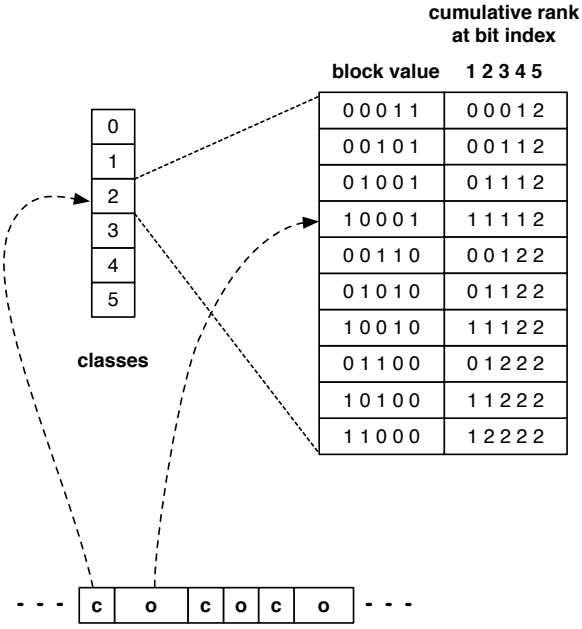

Figure 16: Binary RRR Count Table, with example lookup for class $c = 2$ and offset $o = 3$ in a RRR sequence.

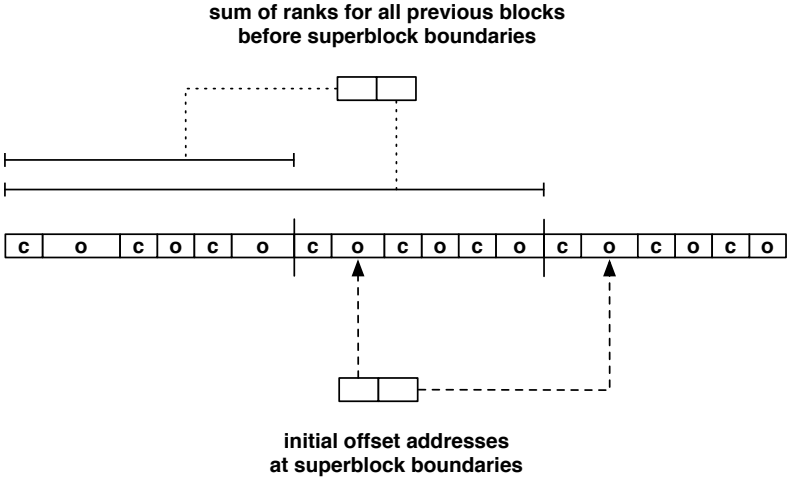

Figure 17: RRR Sequence divided into three superblocks. For each superblock boundary, a sum of previous ranks is stored, as well as the address of the first offset value. These allow us to reduce the amount of iteration required to answer a rank query.

4 Multiary Wavelet Trees and Generalised RRR

This section discusses the design of Multiary Wavelet Trees, and three approaches to support rank queries on the nodes.

4.1 Multiary Wavelet Trees

Multiary Wavelet Trees are analogous to their binary counterparts, although now we encode each node recursively like so:

1. Encoding one A^{th} of the alphabet as 0, the next A^{th} as 1, the next N^{th} as 2 and so on until $A-1$. For example, with the 'Peter Piper...' string:

$$\Sigma = \{\$, \_, P, a, c, d, e, f, i, k, l, o, p, r, s, t\}$$

$$enc(\Sigma) = \{0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3\}$$

2. Group each 0-encoded symbol as a sub-tree
3. Group each 1-encoded symbol as a sub-tree
4. Group each 2-encoded symbol as a sub-tree, and so on until $A-1$
5. Reapply to each sub-tree recursively until the amount of symbols is less than or equal to $A-1$.

See Figure 13 for a Multiary Wavelet Tree constructed over the 'Peter Piper...' string.

We can no longer use binary rank queries, and hence Binary RRR, the same way as we do with a Binary Wavelet Tree. We discuss three alternative approaches in the following sections.

4.2 Variation 1 : Uncompressed

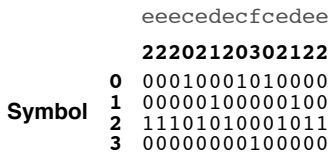

Figure 18: Concatenated bitmap binary encoding of multiary Wavelet Tree Node representing 'eeecedecfcedee' from the 'Peter Piper...' string.

It is possible to represent each encoded symbol c, where c is an element of $0, 1, ..., A-1$ and A is the arity, using A bitmaps. First we construct the bitmaps for each symbol, as in Figure 18, then we concatenate these bitmaps and store them as one bit-vector. A rank query then involves a ranged binary rank query on $N[cL, cL+i]$ at each node N, where L is the length of the node string before concatenation, and i is the position.

We use A bits per symbol, when they could be represented in $\log A$ bits, but this allows us to utilise binary RRR, which we do in the *'Multi-Binary RRR'* Wavelet Tree discussed in Section 4.3.

4.3 Variation 2 : Multi-Binary RRR

Like the uncompressed version, bitmaps are created for each symbol and concatenated, but the bit-vector is stored in a binary RRR sequence. A query then becomes two binary rank queries[4];

$$rank(c * L - 1, 1) \tag{1}$$
$$rank(c * L + i, 1) \tag{2}$$

Where c is the symbol we are querying at position i, $c > 0$, and L is the original length before concatenation. If $c = 0$ then we say the result of the first binary rank query is 0. The final result of $rank(i, c)$ is calculated as the second binary rank query minus the first.

This variation means that we will not need to store a bigger G table to accommodate the additional classes when increasing the arity, when compared with a generalised RRR structure, but does not offer the same sequence compression as the concatenated bitmaps take more bits than required.

Our third variation stores the symbols (without binary encoding) in a generalised RRR structure.

4.4 Generalised RRR

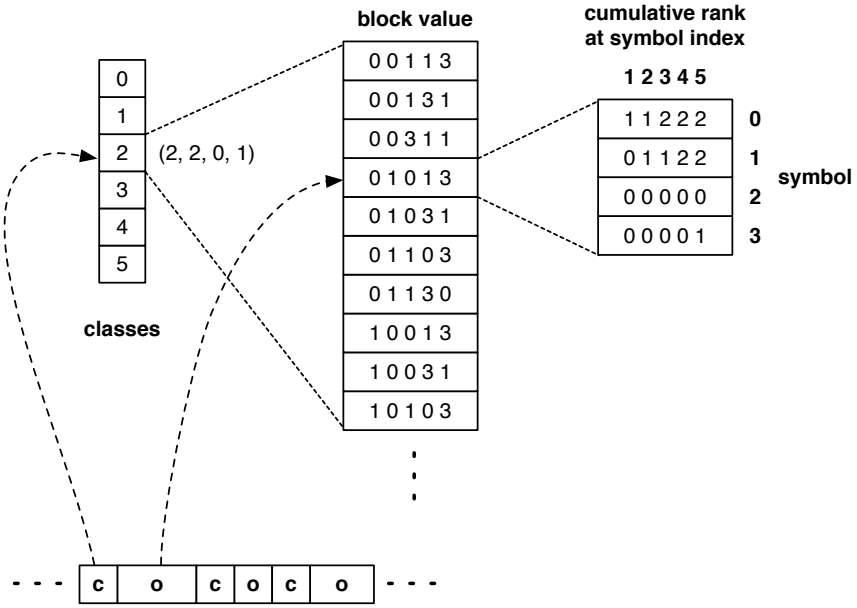

Figure 19: 4-ary RRR Count Table, with example lookup for class $c = 2$, which represents $(2, 2, 0, 1)$, and offset $o = 3$ in a RRR sequence.

[4]In our implementation we pre-calculate the first binary rank queries for each symbol and store it with the node.

The purpose of the Generalised RRR structure is to provide $O(1)$ time rank queries and compression of a sequence of small integers (as opposed to binary integers) on the range $[0..A-1]$ for arity A. This requires these differences:

- Rather than simply being the number of 1-bits, classes are now considered to be a tuple of $(N^0, N^1, ..., N^{A-1})$ for a given block, where N^0 is the number of 0s, N^1 is the number of 1s, and so on. The RRR sequence still stores classes as a unique integer, though.
- Rather than having $\binom{b}{N^1}$ permutations per class for blocksize b, there are now $\binom{b}{N^0,N^1,...,N^{A-1}} = \prod_{i=1}^{A} \binom{b-\sum_{j=1}^{i-1} N^j}{N^i}$ different permutations. Each offset value o therefore requires $\lceil \log \binom{b}{N^0,N^1,...,N^{A-1}} \rceil$ bits. The number of different permutations grows rapidly as arity increases, so our implementation only stored the classes and offsets that we encountered in an attempt to make use of sparsity we can expect in the BWT.
- The G table must also store cumulative ranks for each symbol, per permutation. See Figure 19.
- Each superblock now has A rank sums of each previous block - a rank sum for each symbol.

5 Experiments

We will detail below what our expected findings were, and our method, including the data selected for the experiments and why it was chosen.

5.1 Hypotheses

The experiments were designed to test the following hypotheses:

- Since BWTs have many runs of the same symbol, RRR classes and offsets will not be equally distributed.
- Increasing the arity of a Wavelet Tree will make it shallower and hence reduce the amount of nodes visited per query, resulting in faster queries.
- There is a practical limit to the order of the arity increase for Generalized RRR, since the RRR count table will increase in size. In particular there is tension between the size of the RRR count table and the size of the class / offset sequences.

5.2 Method

For each Multiary Wavelet Tree variant[5] (using Generalised RRR, Multi-binary RRR, and uncompressed concatenated bitmaps to provide sequence ranking at each node), we generated 1000 random rank queries $rank(c, i)$. For three runs the mean query time was recorded, and the minimum result was taken as our result, as it is the time least influenced by external factors (e.g. Operating System swapping). The above experiment was repeated as we doubled the arity.

The size of the Wavelet Tree (including the RRR encoded sequences at each node) was recorded. The size of the RRR count tables was recorded for the cases which used a variant of RRR. For the Generalized RRR, we recorded how many unique class and offset values were encountered, and calculated the percentage of total possible classes and offsets these were[6].

We used Francisco Claude's SPIRE 2008 implementation of binary RRR[7] implementation as a base line for comparison [4]. We used the same default block-size and super-block factor as Claude, which were 15 and 32 respectively.

The data set used for testing is described below[8] in Section 5.3 with some statistics in Table 1.

The experiments were run on an otherwise idle Mac OS X Snow Leopard with a 2.4 GHz Intel Core 2 Duo processor, and 4GB 1067 MHz DDR3 RAM.

[5] All source code is available at http://github.com/alexbowe/multiary-wt.

[6] All raw and processed data is available at http://github.com/alexbowe/multiary-wt, and the graphs are available at http://github.com/alexbowe/wavelet-paper/tree/thesis/.

[7] Claude's compressed data structures library is available at http://libcds.recoded.cl

[8] The BWT files for each dataset are available at http://bwt-corpus.s3.amazonaws.com/list.html, and are reconstructible using the scripts at http://github.com/alexbowe/bwt-corpus.

5.3 Test Data

Since a prime use of Wavelet Trees is to provide faster rank queries over BWTs (as part of an FM-Index), we constructed BWT strings over a selection of texts taken from the *Pizza&Chili* Corpus website[9]. This is the standard corpus used when developing compressed self indexes, and has been collected by Paolo Ferragina and Gonzalo Navarro, two prominent contributors of suffix array research.

The corpus consists of source code for the Linux kernel and GNU C Compiler (GCC), protein sequences, DNA, English texts from Project Gutenberg[10], and XML-formatted bibliographies from several major Computer Science journals. These are considered to be representative of the sort of texts a suffix array may be used for. In the case of the English corpus, we also took a mapping of each unique word to an integer, allowing us to test word-based indexing.

Three data sizes were used to test the scalability: 25MB, 50MB and 75MB. Importantly, these data sizes are much bigger than the available CPU cache, but will not take large amounts of time for experimentation. The length and alphabet size of these files are described in Table 1.

Data	25 MB		50 MB		75 MB	
	σ	length	σ	length	σ	length
xml	96	26 214 400	98	52 428 804	98	78 643 208
dna	14	26 214 400	19	52 428 804	21	78 643 208
proteins	25	26 214 400	29	52 428 804	33	78 643 208
sources	116	26 214 400	229	52 428 804	229	78 643 208
english	154	26 214 400	179	52 428 804	188	78 643 208
words	83 083	5 969 593	115 754	11 860 943	164 757	17 668 587

Table 1: Text lengths and alphabet sizes for each test file.

[9] http://pizzachili.dcc.uchile.cl
[10] http://www.gutenberg.org

6 Results

Note that measurements for the Generalized RRR with arity 16 are missing, since the experiments took too long due to paging. This supports our third hypothesis, but there may be ways to overcome this, as detailed in Section 8.

Also Note that the Uncompressed Wavelet Tree timings have been plotted on a different graph due to the difference in scale.

Arity	Max Total Classes	Max Total Permutations
2	16	32 768
4	816	1 073 741 824
8	17 0544	35 184 372 088 832

Table 2: Maximum Total Classes and Offsets possible with a blocksize of 15 for arity values 2, 4, and 8.

From Figure 20 we can see how the number of unique classes and block permutations increases depending on file size and arity. Table 2 shows the maximum of each of these values, and Figure 21 plots the number of unique classes and block permutations as a percentage of the values in Table 2.

Figure 21 indicates that in all data sets, around 100 percent of the classes and block permutations were encountered for arity 2. In most data sets, all classes were encountered for arity 4 as well, with the exception being DNA, which encountered around 30 percent.

Figure 20 shows that the proteins data set had significantly more unique block permutations than any other data set, while the words data set is the only one to have encountered all classes for arity 8. This is because protein data is essentially random, so its BWT will not contain long runs of the same symbol, reducing the skew of its classes.

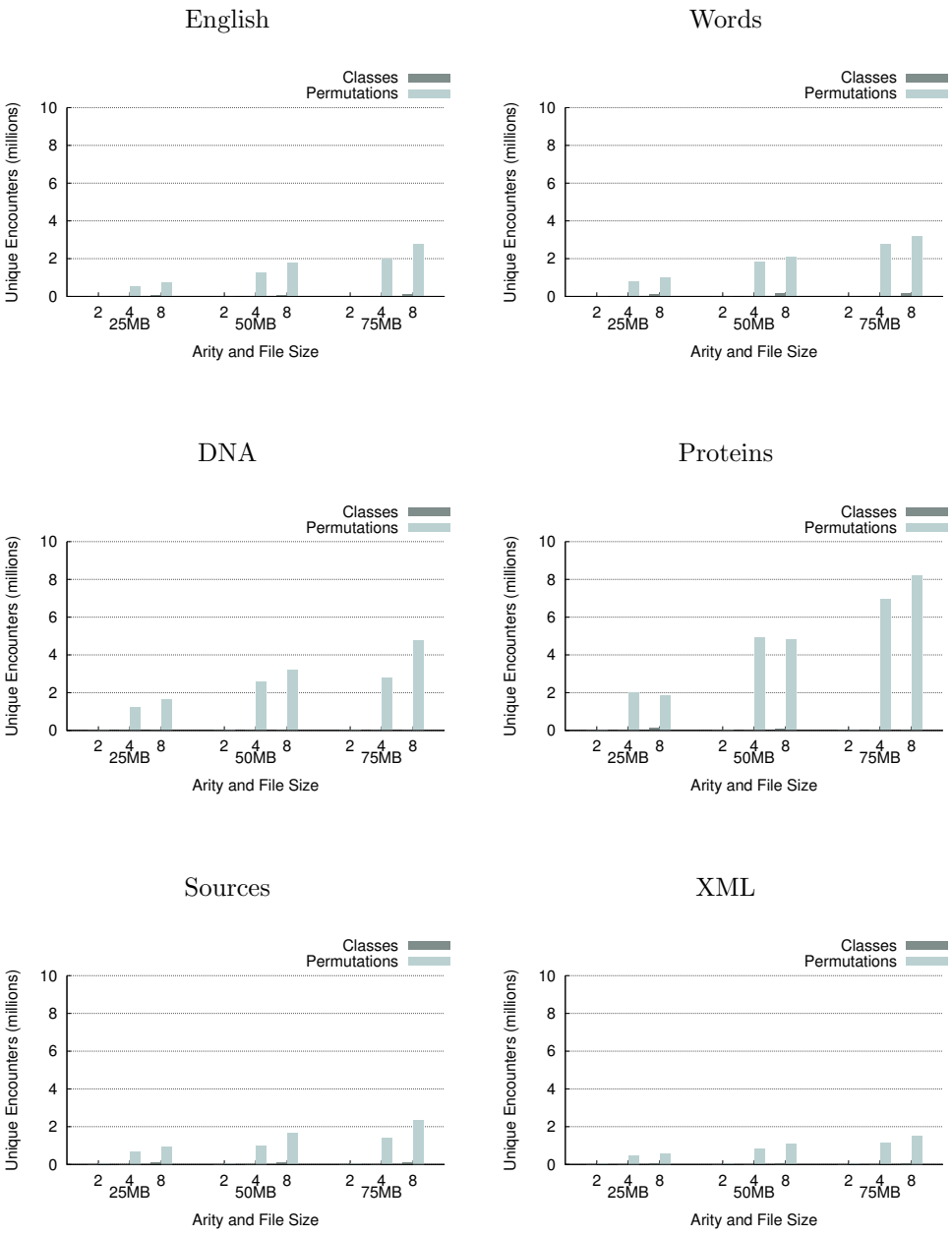

Figure 20: Number of unique classes and block permutations for each data file. The vertical axis represents the total number of classes and block permutations that were witnessed for the given file size and arity when using the Multiary Wavelet Tree with Generalised RRR.

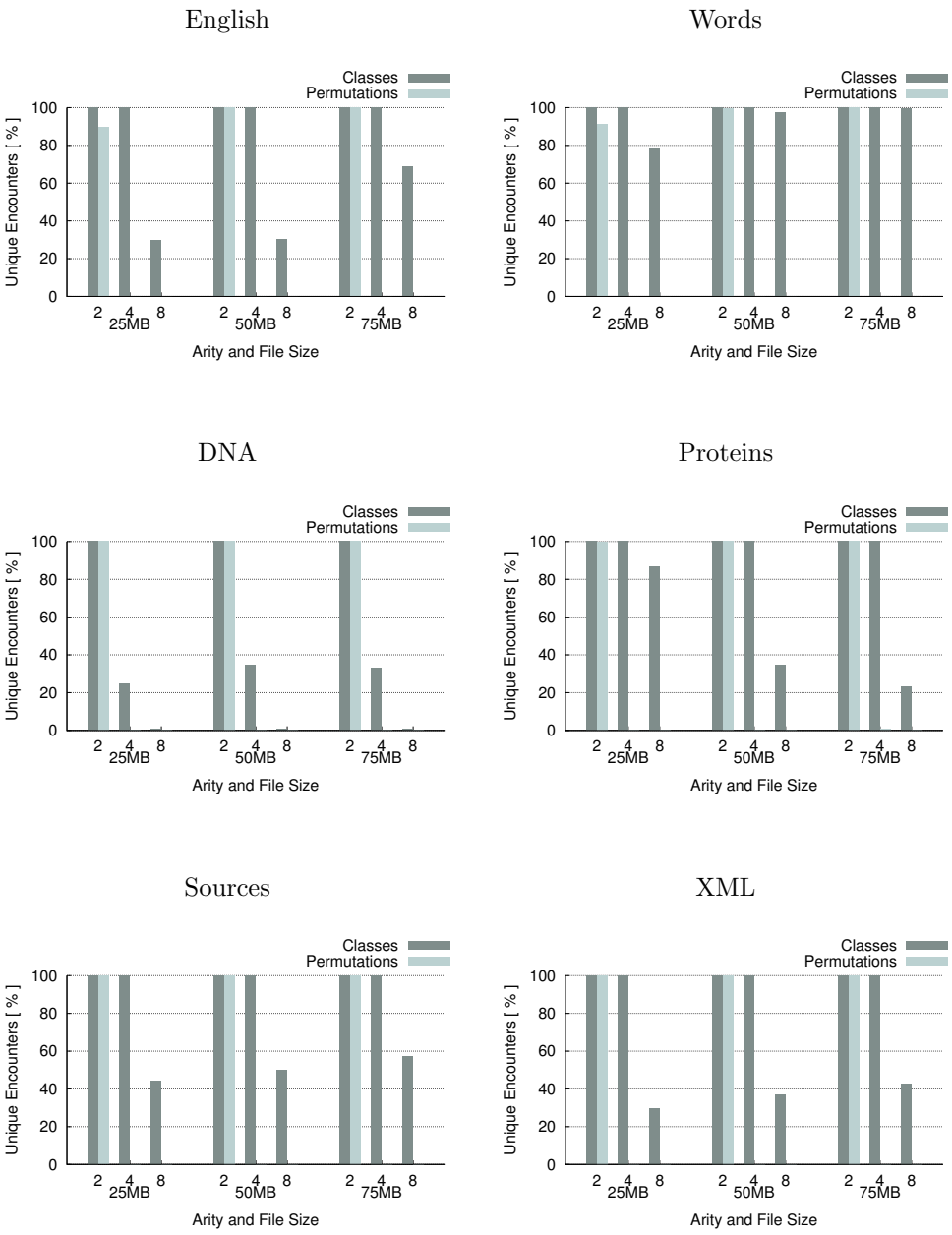

Figure 21: Sparsity measurements for each data file. The vertical axis represents the percentage of *total possible* classes and block permutations that were witnessed for the given file size and arity when using the Multiary Wavelet Tree with Generalised RRR. Note that some bars are too small to see.

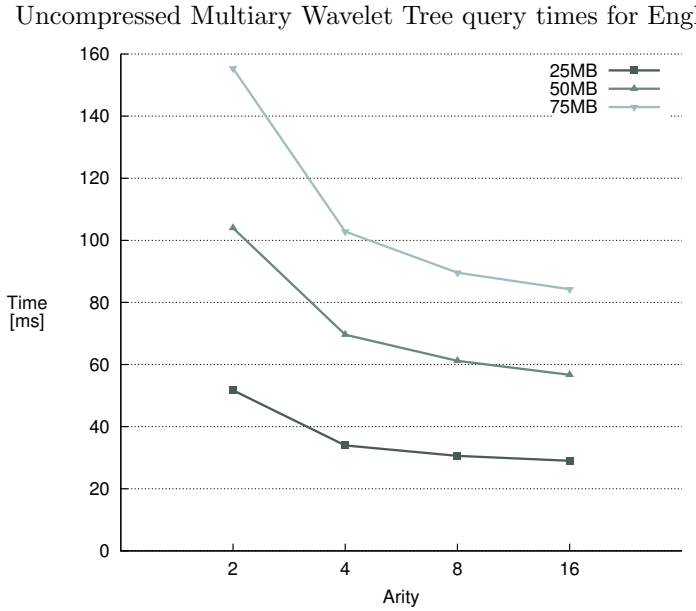

Figure 22: Query times for Uncompressed Multiary Wavelet Tree of increasing arity for each *English* file.

From Figure 22 we can see how increasing the arity affects querying an uncompressed Wavelet Tree. It is slower for increasing file size because it is calculating rank queries without the assistance of RRR.

From Figures 23 and 24 we are able to see that increasing the file size does not significantly affect the time performance of the Wavelet Trees which utilise RRR.

The Generalised RRR is slower than Claude's. We suspect that this is due to our use of pointers, as required to create a sparse table, whereas Claude's avoids dereferencing and may make better use of cache. The trend is similar to the other Multiary Wavelet Trees, though.

The Multi-Binary RRR Wavelet Tree is faster than Claude's when the arity is increased. It is slower at arity 2 possibly due to optimisations in Claude's Wavelet Tree code, or because we need to do an extra calculation to work out the binary rank of all previous symbols (see Section 4.3).

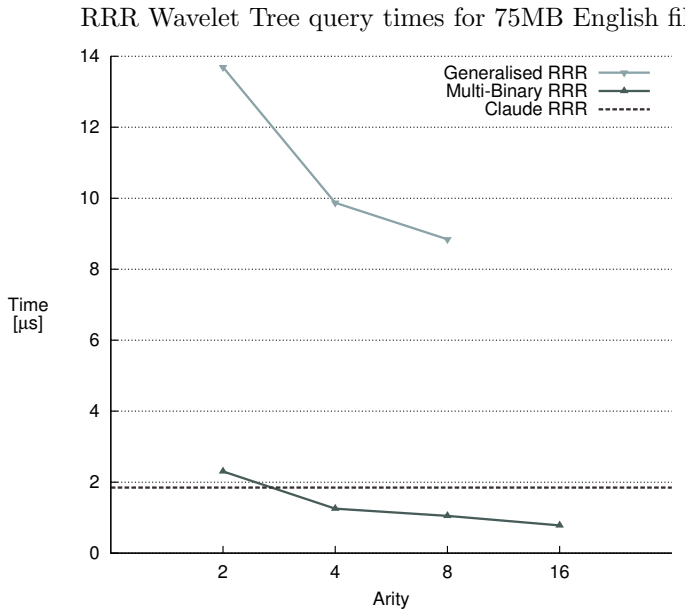

Figure 23: Query times for RRR Wavelet Trees of increasing arity for the 75MB *English* file.

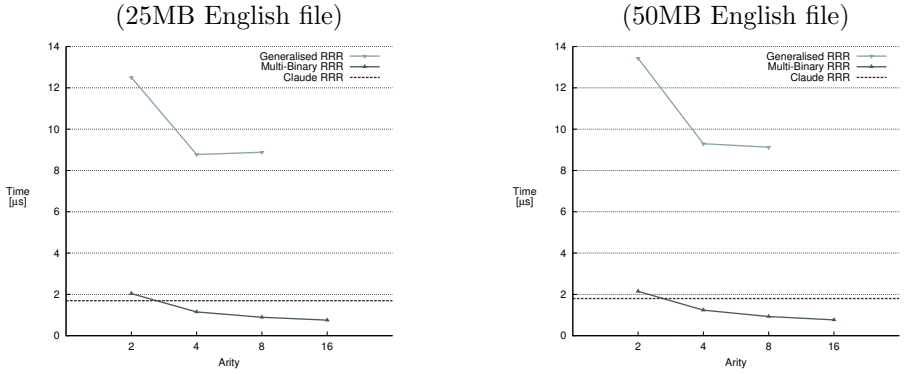

Figure 24: Query times for RRR Wavelet Trees of increasing arity for the 25MB and 50MB *English* files.

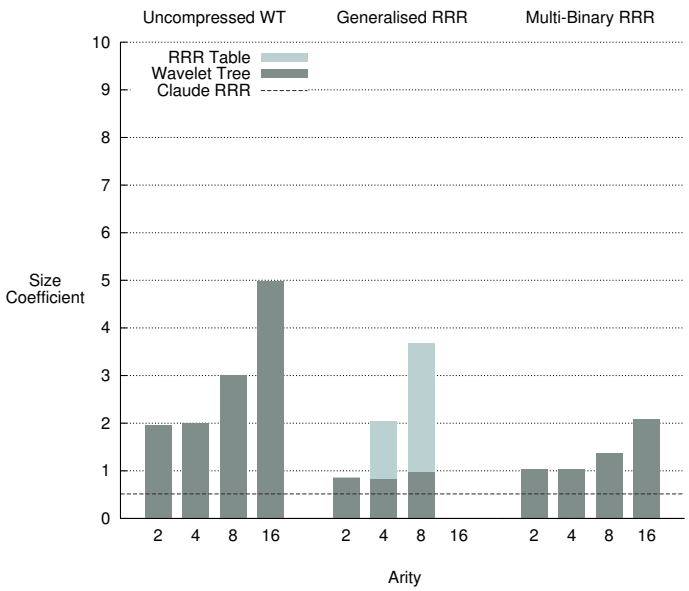

Figure 25: Memory consumption for Wavelet Trees of increasing arity for the 75MB *English* file. The memory required for each data structure is the size coefficient multiplied by the original file size. The bar stacked on top is the space for the supporting RRR count table. The bar underneath is the space for the shape of the Wavelet Tree (which had negligible overhead) and each of its nodes, as RRR sequences or not. The Uncompressed Wavelet Tree is the only one which does not have a RRR count table.

Figure 25 shows the memory consumption of the structures for each English file. As the smaller sized files have similar size coefficients, we only show the graphs for the 75MB files.

Note that even though the Generalised RRR Wavelet Tree size (which includes the RRR sequences it stores) is smaller than the original text, and smaller than the Multi-Binary RRR Wavelet Tree, the size to contain the supporting RRR count structure is large.

The Proteins files have less classes (Figure 21) but more permutations (Figure 20) than the Words files. This may be the cause of the large table size for the Generalised RRR Wavelet Tree in Figure 27 compared with that of the Words file in Figure 26.

In all cases, the Generalised RRR Wavelet Tree size is less than the original text size. It is important to note that the RRR count table can be shared among Wavelet Trees, but this may cause it to grow to its full amount, depending on the distribution of classes for the type of text we are indexing, as more classes and permutations may be encountered.

The tradeoff between memory and query time for the English files can be seen in Figure 28. All other test data follows a similar trend, so the graphs have been omitted[11], with the exception of Words (Figure 29), Proteins (Figure 30)

[11]The other graphs are available at http://github.com/alexbowe/honours-thesis/.

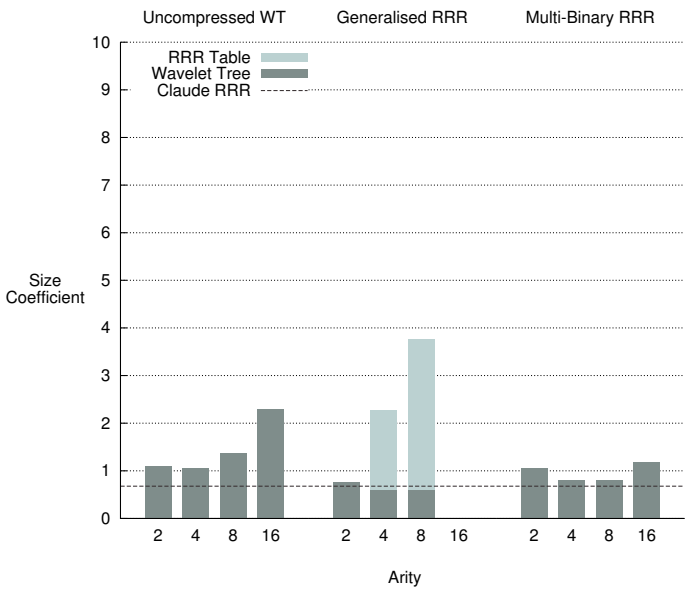

Figure 26: Memory consumption for Wavelet Trees of increasing arity for the 75MB *Words* file. The memory required for each data structure is the size coefficient multiplied by the original file size. The bar stacked on top is the space for the supporting RRR count table. The bar underneath is the space for the shape of the Wavelet Tree (which had negligible overhead) and each of its nodes, as RRR sequences or not. The Uncompressed Wavelet Tree is the only one which does not have a RRR count table.

and DNA.

In Figure 29 The Generalised RRR Wavelet Trees queries slow down for arity 8. This may be due to fragmentation of the RRR count table memory; as the Wavelet Tree becomes less shallow as its arity increases, the benefit of cache may also diminish, since the RRR count table will become larger and possibly dispersed throughout memory. This would not affect the other RRR Wavelet Trees as their RRR count tables are contiguous and small, making the better use of cache.

Figure 30 shows the query time increasing slightly for arity 4, then decreasing again for arity 8. This may be due to the small alphabet size of the Proteins file, meaning the Wavelet Tree would not get much shallower, and as above the performance may be dominated by cache. This trend was seen in DNA as well, which also has a small alphabet.

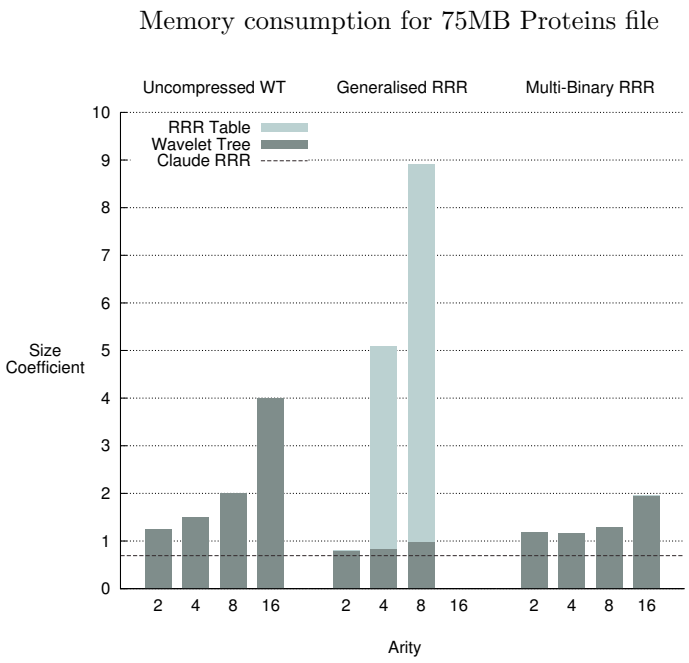

Figure 27: Memory consumption for Wavelet Trees of increasing arity for the 75MB *Proteins* file. The memory required for each data structure is the size coefficient multiplied by the original file size. The bar stacked on top is the space for the supporting RRR count table. The bar underneath is the space for the shape of the Wavelet Tree (which had negligible overhead) and each of its nodes, as RRR sequences or not. The Uncompressed Wavelet Tree is the only one which does not have a RRR count table.

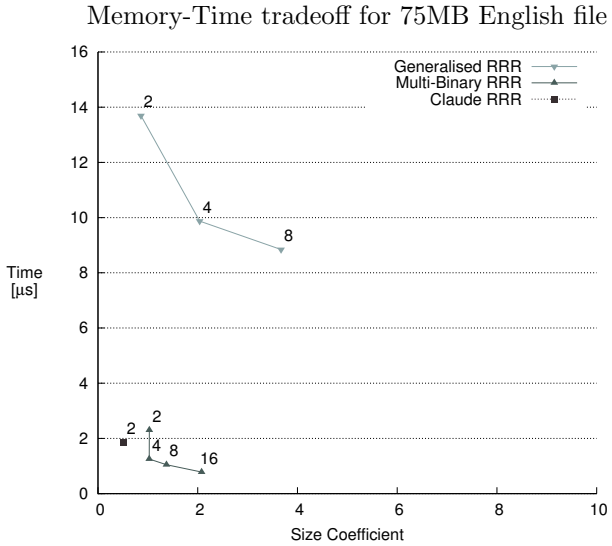

Figure 28: Memory-Time tradeoff for RRR Wavelet Trees of increasing arity for the 75MB *English* file. The memory required for each data structure is the size coefficient multiplied by the original file size.

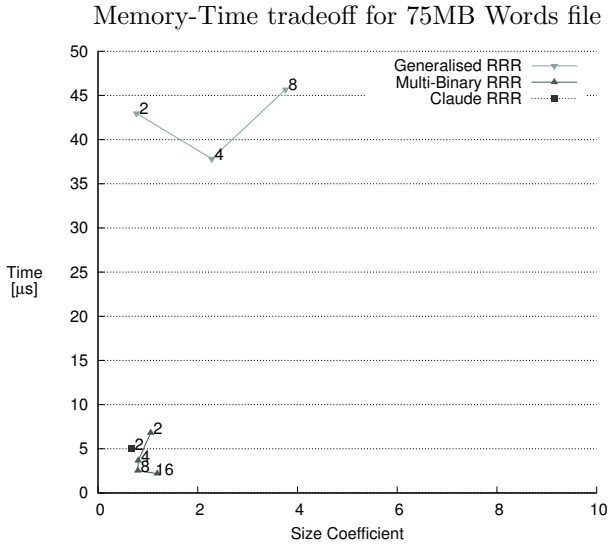

Figure 29: Memory-Time tradeoff for RRR Wavelet Trees of increasing arity for the 75MB *Words* file. The memory required for each data structure is the size coefficient multiplied by the original file size.

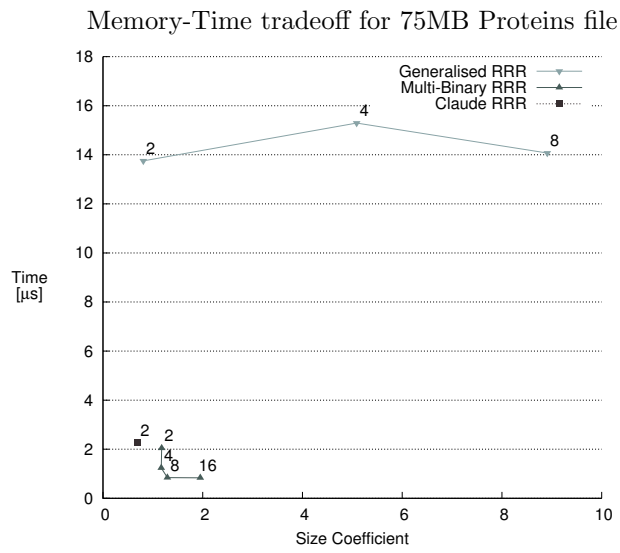

Figure 30: Memory-Time tradeoff for RRR Wavelet Trees of increasing arity for the 75MB *Proteins* file. The memory required for each data structure is the size coefficient multiplied by the original file size.

7 Conclusion

From our observations we have discovered that there is sparsity in the classes and block permutations encountered in our test data, and it can have a significant effect on the supporting table size, in the case of Generalised RRR. However, making use of this sparsity dynamically requires pointers, and may be the cause of some time overhead and cache misses while querying.

Since the RRR count table is shared among all nodes and even all Wavelet Trees of the same or smaller arity and blocksize, it may be the case that when documents are significantly large, or we are indexing a large collection of documents, the overhead of the RRR count table becomes negligible. For such documents, a distributed approach may be required; we may store the RRR table on multiple computers, or increase the arity so that the table fits in one central server for querying by the computers that host the Wavelet Trees. It is likely such a configuration may need the extra speed while querying, since it is typical of a search engine to host indexes over many servers and calculate many queries per second.

For single documents, or small collections of documents, the RRR count table expands too rapidly to make increasing the arity worthwhile. We discuss a possible method to curb this growth in Section 8.

However, we have shown that there are simple ways to implement Multiary Wavelet Trees using rank structures for binary alphabets. In our 'Multi-Binary RRR' Wavelet Tree rank queries become faster, while the Wavelet Tree nodes did not grow too large. The Multi-Binary RRR Wavelet Tree was sometimes larger than the original text, however.

8 Future Work

There are several promising avenues for future work which this thesis has helped reveal:

1. Investigate if there is any way to make the count table for Generalized RRR smaller. One promising idea is to only store base counts which can generate all cyclic permutations of a block. For example, if one block is $b_1 = [0, 1, 2, 2, 3]$ and another block is $b_2 = [1, 2, 2, 3, 0]$ (that is, a left cyclic shift of one position applied to b_1), then there may be a way to calculate the counts for b_2 from b_1, and no longer store the counts for b_2.

2. Similar to above, another option is to share count table entries among different blocks which have similar positioning but for different symbols. For example the block $b_1 = [0, 1, 2, 2, 3]$ has the same count table entry for $c = 2$ as the block $b_2 = [0, 2, 1, 1, 0]$ does for $c = 1$.

3. Reduce the use of pointers when constructing a sparse RRR table. This may be done by completing a full scan of the text before constructing the table, and tracking how many unique blocks there are, then allocating them contiguously.

4. Search for a good tradeoff between arity and block size for the Generalised RRR; the Generalised RRR Table will grow smaller if a smaller block size is used, but the sequences then become bigger as they have more class

and offset values (which require less bits, but perhaps not significantly so). This may not be preferable since the table can be shared among many sequences. The analysis of Ferragina et al. [8] suggests blocksize and arity should be related, in particular blocksize $b = \lfloor \frac{1}{2} \log_A N \rfloor$, that is, b decreases slightly as arity A increases.

5. Implement and investigate a Multiary Huffman-Shaped Wavelet Tree (see Mäkinen's work for details on Huffman-Shaped Wavelet Trees [15]). This may overcome the RRR count table memory consumption while still reducing the tree depth.

6. Real world query patterns may show that there are certain blocks which are queried more frequently, so we may only need to keep the RRR count table entries for these blocks in memory at all times. The other RRR count table sections may be stored on disk and loaded into memory on the occasion that they are queried.

7. Distribute the RRR count table among nodes in a cluster, allowing it to be held in memory as restricted by the cluster as a whole, not a single computer. Then, increasing arity would be an issue of how many nodes are on the cluster. There has been recent research on distributed compressed suffix arrays by Russo et al. [26].

8. Investigate Multiary Wavelet Trees which use the bitmap concatenation technique, but encode each node using other binary sequence structures which answer rank queries, such as the rank index by Okanohara and Sadakane [23].

9 Acknowledgements

I thank my supervisor Simon Puglisi for his patience and guidance, Juha Kärkkäinen (University of Helsinki) for helping me to understand RRR, and Francisco Claude (University of Waterloo) for his advice and explanation of his code. I also thank my brothers Nikolas and James Bowe for their programming advice and motivation. Finally, I thank RMIT University for providing me with a scholarship to complete this paper.

References

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] J. Barbay, F. Claude, and G. Navarro. Compact rich-functional binary relation representations. In A. López-Ortiz, editor, *LATIN 2010: Theoretical Informatics*, volume 6034 of *Lecture Notes in Computer Science*, pages 170–183. Springer, 2010.

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

[4] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187. Springer, 2008.

[5] J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In M. de Berg and U. Meyer, editors, *Proceedings of the 18th European Symposium on Algorithms (ESA 2010)*, to appear 2010.

[6] P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207(8):849–866, 2009.

[7] P. Ferragina and G. Manzini. Opportunistic data structures with applications. *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[8] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):20, 2007.

[9] P. Flicek and E. Birney. Sense from sequence reads: methods for alignment and assembly. *Nature Methods*, 6(11s):S6–S12, October 2009.

[10] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 368–373, New York, NY, USA, 2006. ACM.

[11] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata.

[12] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850. Society for Industrial and Applied Mathematics, 2003.

[13] W.-K. Hon, R. Shah, and J. Vitter. Space-efficient framework for top-k string retrieval problems. In *Foundations of Computer Science, 2009. FOCS '09. 50th Annual IEEE Symposium on*, pages 713 –722, October 2009.

[14] D. Knuth, J. Morris Jr, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323—350, 1977.

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

[16] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 214–226. Springer, 2007.

[17] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.

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

[19] M. Marín and G. Navarro. Distributed query processing using suffix arrays. In *String Processing and Information Retrieval*, volume 2857 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2003.

[20] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[21] J. Munro. Tables. In *Foundations of Software Technology and Theoretical Computer Science*, pages 37–42. Springer, 1996.

[22] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[23] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. *Arxiv Computing Research Repository*, abs/cs/0610001, 2006.

[24] S. J. Puglisi, W. F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):1–31, 2007.

[25] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.

[26] L. Russo, G. Navarro, and A. Oliveira. Parallel and distributed compressed indexes. In A. Amir and L. Parida, editors, *Combinatorial Pattern Matching*, volume 6129 of *Lecture Notes in Computer Science*, pages 348–360. Springer, 2010.

[27] M. Wattenberg. Arc diagrams: Visualizing structure in strings. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 110–116, Washington, DC, USA, 2002. IEEE Computer Society.

[28] P. Weiner. Linear pattern matching algorithms. In *SWAT '73: Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.

[29] C.-C. Yu, W.-K. Hon, and B.-F. Wang. Efficient data structures for the orthogonal range successor problem. In *Computing and Combinatorics*, volume 5609 of *Lecture Notes in Computer Science*, pages 96–105. Springer, 2009.