3

Coders for the Burrows-Wheeler Transform

Like most transforms, the Burrows-Wheeler Transform does not change the size of the file that has been transformed, but merely rearranges it so that it will be easier to represent it compactly. It then needs to be coded using a second phase which we will refer to as the "Local to Global Transform" (LGT). Figure 3.1 shows a section of the transformed text for Shakespeare's "Hamlet", which reveals the kind of regularities that the BWT exposes. These characters are ones that appear before the context `nd`; initially the `nd` is followed by a space, and hence `a` is very common, but then the character is followed by `ndeed`, where the `i` becomes common, and the last few characters precede `nder`.

Clearly the text in Figure 3.1 contains a lot of patterns, and therefore will be easy to compress. Many sophisticated techniques have been proposed to exploit the regularities of the BWT transformed text, and yet it has emerged that one of the simplest approaches (RleAc, based on run-length encoding and an order-zero arithmetic coder) gives the best compression and is also very fast compared with more complicated methods. We will begin this section by looking at this simple coder, but later we will also review various other approaches that have been proposed, including Burrows and Wheeler's original "Move to Front" (MTF) list, inversion frequencies, distance coding, frequency counting methods, wavelet trees, and alternative permutations. We will also consider the effect of the block size on compression performance.

3.1 Entropy coding

Before we look in detail at how the structure of a BWT string can be exploited for compression, we will briefly review some fundamentals of how symbols can be converted to bits based on an estimated probability distribution for how likely each symbol is. This process is often referred to as *entropy coding*, because the aim is to represent symbols in as few bits as possible, and the limit of this is dictated by the *entropy*. The BWT-based systems that we will

```
AaaaaaAaaaAaAaAaAeAeiuaaAaoaaiAauaaauiaaaaaieaeeaoeuueauiiiAaaua
aaaaaaaaoaaaaiaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaAAaAAaAAAAaAaa
AuaeaaaaaaaaaaaauaaaaaeaaiaaauueuaeaaaieaaaaaaaaaeAAaaaaeaeaaaaaaa
eaaAaaaAaaaaAaaiaaaaaeeiaaaaaaAiaaAaAaiAaAaaaaaaAaaaaaAaaaAAAaAA
aaAAaaaAaaaaoaaaaaAaaAAaAAaaAaaaaaAaaaaaaaaaaaAAaaaaaaAaAaaaaaaa
aaaaaaaaaaaaaaaiiaAaaAAaaaaeaaaaAaAaaaaaAaaaAAaaaaaaAAAaaaaAaAa
aaaaaaAaaaaaaaaaaaaaaAaAaauaAAaAaaaAAaAAaaAaaiaaAAAaaaaAAAaaaaa
aaaaaaaaaaaaaaaaaaaaAAAAAaaAAauaAauaaaaaaaaaaAaaaieiiiieeAaaaaa
eeAaiaaoAaAaaaaAAaAoooaiaAaaiAAaaAaAAAAaaaaaaaiiiaeeeaaeAiuiaaaa
aaAaaaaaaaaAAoaaaAAAAaAAAAAAAAAaaAiaaaAaaaaAAaaaaAAaaaAaaAaaaoaaAu
aaieaauaeaaAAaaaaaaaaaaAaaaaaaaaaaAaaaaaeaaaAaaaaaaAaaauAAaaaaaaa
aAaaAAAAaaAaAaaaaaAAiiiiiiaieaiaiaaiiAeeaaAaaeaaaaaAaaaaAaoaaiia
aaaaaaaAaaaaaaaAaAaaaaaaAaaaAaaaAaaAiaaaaaAaAoaaaaaaaaaaaaaaoaaaa
AaaaaaaaaAAAaiaaaaaaaAaiaiaAaaAuAaAaaaAaaaaaaaaaaaaaaaAAaaaaaaAaa
aaaaaAaaaaaaaAaaaAaaAAaaAAaaAAAaAaAaaAaiiiiaaaaauAaaaaaaaaaaaAoae
auaaaaaaaaaaaaieaaaeAAaaAAaAAAaaaaaaAAaAAaAaAAAaaAaaaeaaAAaAaaaAA
aAAAAaeaoaoaaeaaaauaaaaAiAAAuaaaaaaaaaaaaaaaaaauaaaaaeiaAaaAAAaaa
AAAaaaaaaAaaaaAAaAAuaaAAaAaAeAaaaaaAaaaaaAaaaaAAaaAaAAaaaaaaaAAA
aauuAaAaAAaAAueaaaaaaAaAaaaeAAaAaaaaAaeaaAeeaaaeaaeaaueieaaiaeea
aaeeaaaaueiaaeaaaeeeeeeeeeeeauoueueeeeaeeueuuuaeiiiiiiIiiIIIiiiiI
IiIiiiiiiiiiIIeieaeoooouuuueaeeoauaeeeaoUUiaeaeooueaoeaeeuauuuu
```

Fig. 3.1. Some of the transformed text generated from Shakespeare's "Hamlet"

be looking at often use entropy coding as their final stage, so it is good to understand the methods that the BWT processing is preparing data for.

In entropy coding, the representation of a symbol is based on some estimated probability of that symbol occurring. The next symbol to be coded will be drawn from a probability distribution that is typically estimated based on previous observations. For example, if the character "e" has occurred in 20 out of the last 100 characters, we might estimate that the probability of the next character being an "e" is 20%.

Shannon (1948) showed that, on average, the optimal representation for a symbol with probability p would use $-\log_2 p$ bits. There are two general approaches to generating the bits given a probability distribution: Huffman coding, and arithmetic coding. Huffman coding can be computed very quickly, but the number of bits used for a particular symbol are whole numbers, and hence won't necessarily be equal to the optimal size of $-\log_2 p$. This is a problem especially when p is close to one; in this case the optimal number of bits approaches zero, but Huffman coding must use at least one bit to represent each symbol. Arithmetic coding overcomes this problem by effectively overlapping the bit representations of successive characters, so one bit in the output might correspond to more than one symbol. Although arithmetic coding is optimal, it is usually an order of magnitude slower than Huffman coding, and hence should be used only in situations where the probability distribution will cause poor behavior in a Huffman coder.

The task of coding using Huffman or arithmetic coding is well understood, and the further reading section at the end of this chapter gives a number of references that explain these techniques in detail. Many BWT-based systems rely on these entropy coders to produce their output. Generally Huffman coding is used if speed is important and the loss of compression is tolerable; otherwise arithmetic coding is used to get the best possible compression. The main property that we need to understand about arithmetic coding is that if it codes a symbol with an estimated probability of p, then the number of bits used will be arbitrarily close to the optimum value of $-\log_2 p$ bits, and thus we can estimate the size of a compressed file without even performing the coding.

Estimating the probability distributions for an entropy coder can be done adaptively or non-adaptively. Non-adaptive systems use the same probability estimates (and therefore the same code) for the entire coding, whereas adaptive systems allow the probability distributions to change from character to character, effectively "learning" during the encoding. Non-adaptive methods would typically scan the entire sequence to be encoded, estimate probabilities, create a code table, and proceed to code the entire sequence using the code table. Adaptive systems don't need to scan the sequence in advance; they generally start with simple assumptions (such as all characters being equally likely), and then change the probabilities as coding proceeds. The decoder is able to adapt in synchronization as long as the adaptation occurs *after* each symbol is encoded, since the symbol can be decoded using the current codes, and the decoder can update the statistics after each character is decoded.

For example, Figure 3.2 shows an adaptive estimation of probabilities for coding the word `mississippi` assuming an alphabet of $|\Sigma| = 4$ characters. The first four columns count how many times each character has been observed so far, although they are initialized to 1 to avoid having zero probabilities. The first row shows that with the initial counts all being equal, the first `m` in the string is given a probability of $\frac{1}{4}$, and is therefore coded in 2 bits, which is what we would expect for an alphabet of four characters if no compression was being attempted. The next line shows the count of `m` incremented, as both the encoder and decoder have observed that one `m` has occurred, and therefore it might be more likely than other characters. Thus the next code, which happens to be for an `i`, gives it a less than even chance of $\frac{1}{5}$ (its count is 1, and the total for all characters is 5). Using Shannon's formula we find that it should be coded in 2.32 bits. The system then "learns" from this, and increases the count of `i` by 1, ready to code the next character. Notice that the second `s` is coded in 1.8 bits, and the fourth one in just 1.32 bits, as the adaptation has given `s` the highest probability.

This example is too short to show the effectiveness of adaptation, but generally after initial poor performance where the probability estimates are very crude, an adaptive system will settle down to represent the statistics of the text it is modeling. The model used in this example is also very simple; it assumes that each character occurs independently of the others. While this is

Character counts				Probability	$-\log_2 p$
i	m	p	s		
1	1	1	1	$p(\mathtt{m}) = \frac{1}{4}$	2 bits
1	2	1	1	$p(\mathtt{i}) = \frac{1}{5}$	2.32 bits
2	2	1	1	$p(\mathtt{s}) = \frac{1}{6}$	2.58 bits
2	2	1	2	$p(\mathtt{s}) = \frac{2}{7}$	1.8 bits
2	2	1	3	$p(\mathtt{i}) = \frac{2}{8}$	2 bits
3	2	1	3	$p(\mathtt{s}) = \frac{3}{9}$	1.58 bits
3	2	1	4	$p(\mathtt{s}) = \frac{4}{10}$	1.32 bits
3	2	1	5	$p(\mathtt{i}) = \frac{3}{11}$	1.87 bits
4	2	1	5	$p(\mathtt{p}) = \frac{1}{12}$	3.58 bits
4	2	2	5	$p(\mathtt{p}) = \frac{2}{13}$	2.7 bits
4	2	3	5	$p(\mathtt{i}) = \frac{4}{14}$	1.8 bits

Fig. 3.2. Adaptive order zero probability estimation for coding the word `mississippi`

useful (especially for BWT applications), adaptive systems like this generally take account of prior characters when making probability estimates.

In the example some arbitrary choices were made which affect the rate of adaptation. The initial counts need not have been set to 1; a higher value could have been used to make the system take longer to make significant changes in the probabilities, and conversely, instead of incrementing by 1 each time, a higher value could be used to increase the effect of each character arriving. For example, if we were incrementing by 10, then the first `m` would cause the probability on the next character being an `m` to be estimated at $\frac{11}{14}$ and the other three characters at $\frac{1}{14}$. This kind of aggressive increment (combined with an aging scheme) is useful for BWT coding because of the strong clustering of similar characters in the input, such as in the example in Figure 3.1.

At the other end of the spectrum from adaptive coding, for the probability distributions that arise in some BWT systems (particularly using MTF) often a simple fixed code can be effective, using the same representations regardless of the statistics. This will inevitably give slightly worse compression performance than if the probability distribution is taken into account, but the gain in speed and simplicity may well justify making the approximation. For example, the codes shown in Table 3.1 can be used to represent values where very small numbers are the most common. For the α code, a value of one is represented using just one bit; a value of two using two bits, and so on. It is sometimes referred to as "unary", because it is based on coding in base 1 (each position is worth 1 times as much as the one to the right). The disadvantage of this code is that larger values will require many bits. Because an event with probability p is ideally represented in $-\log_2 p$ bits, the unary code implies that the probability of a zero is $\frac{1}{2}$, of a one is $\frac{1}{4}$, and so on. The α code strongly favors small values, and is not used so often on its own, although, for example, it is used in BZIP2 to select which Huffman table to use for coding, as there are only six tables to choose from, and 1 and 2 are the most common.

A coding that grows in length somewhat slower than this is the γ code, also shown in Table 3.1. The γ code represents a value as its binary number, but because the length of the binary number can vary, a unary code is prefixed to indicate the length of the binary number. Because all of the binary numbers begin with a 1, that bit is omitted. For example, the number 5 has the 3-bit binary representation 101; it is therefore represented as the unary value for 3, followed by the last two digits of its binary representation (01).

value	α	γ	δ
1	1	1	1
2	01	01 0	010 0
3	001	01 1	010 1
4	0001	001 00	011 00
5	00001	001 01	011 01
6	000001	001 10	011 10
7	0000001	001 11	011 11
8	00000001	0001 000	00100 000
9	000000001	0001 001	00100 001

Table 3.1. Two fixed-length codes for the integers

There are many more codes that use this kind of approach; these are often referred to as Elias codes, after the author of a seminal paper on the topic (Elias, 1975). An interesting one is the δ code, which essentially uses a γ code to represent the number of bits in the integer, followed by the binary representation of the integer (again, missing its highest order bit since that will always be a 1). The δ code is a *universal* code, which means that the expected code length is within a constant factor of what would have been assigned by an entropy coder, as long as the ranking of the probabilities is correct.

Codes such as α, γ and δ are fixed, and therefore are completely unresponsive to any probability distribution changes in the input. Their main advantage is that they are very fast, and no code table needs to be stored or transmitted. In contrast, the Huffman and arithmetic entropy coders will use codes that approach optimality for the given probability distribution, although in principle a universal code will also be close to optimal. The further reading section gives references for more information about the implementation and performance of entropy coders and fixed codes, and Section 5.5.2 looks at this kind of code in more detail.

3.2 Run-length and arithmetic coder

One of the simplest ways to code the output of a Burrows-Wheeler Transform is also one of the most effective. In this approach we put the output (such as that shown in Figure 3.1) through a "run-length encoder", which takes advantage of runs of identical characters in a sequence, and then through an arithmetic coder, not unlike the approach shown in Figure 3.2. This method has been labeled RleAc (Ferragina et al., 2006a).

Traditionally these are combined with other techniques (such as the move-to-front list) that exploit the structure of the transformed text, but a somewhat surprising result is that with properly chosen parameters these two components perform very well on their own. Run-length encoding (RLE) features strongly in much of the BWT research, and can be applied at almost any stage (including before the transform). However, most of the compression is achieved in the arithmetic coding, and there is some evidence that even the run-length encoding isn't necessary if the arithmetic coding is done correctly, although RLE can make implementation simpler and faster.

The run-length encoder simply replaces a run of the same character with a code that specifies what the character is, and how long the run is. For example, in the second line of Figure 3.1 the letter `a` appears 36 times in a row, which would be replaced with a code; on the other hand, the single occurrences of a letter `o` are a trivial run of length one, and are not worth replacing. A run-length code generally has three components: a flag to indicate that a run is about to be encoded, the character in the run, and the length of the run. This can be reduced to just two components if it is possible to have a run of length one. In a Burrows-Wheeler transformed file, typically only about 33% of the characters occur in runs of 3 or more, but nevertheless there are some gains to be made as this still represents a significant proportion of the file that occurs in a run.

The key component of the RleAc backend to a BWT system is the arithmetic coding. This uses a simple adaptive model to count the occurrences of each character in the transformed text, adapting the probability distribution after each character is coded. Adaptive systems are particularly useful when compression must start before the entire input is known, or where only one pass through the input is possible. Because BWT-based systems work with a block of text at a time, adaptive coding is not essential for this purpose. However, another feature of adaptive coding is not just the ability to "learn" new symbols, but to "forget" old ones. Normally this is a side-effect of mechanisms to prevent integer overflow in an arithmetic coder; as symbol frequencies are collected, if the total count of symbols is threatening to cause overflow in the calculations, all individual symbol counts are halved. As well as preventing the overflow, the halving means that subsequent symbol occurrences will have twice the weight of ones prior to the halving. This mechanism can be brought into play more aggressively by incrementing counts by more than one, so that the halving will occur more often, and hence even more weighting is given to recent symbols. A similar effect could be achieved by lowering the threshold for halving, although this means that some accuracy may be lost in rounding during the halving process, since counts are stored as integers. Another possibility is instead of halving the counts, they can be divided by a larger number to "age" them faster.

For example, consider the coding of the characters `eeeeaeeueuuuaeiiiiii` (taken from the transformed text in Figure 3.1). Figure 3.3 shows them being coded using a very aggressive policy where their counts are incremented by 10, and divided by 3 when the total count exceeds 50 (different figures would be used in practice, and the table shows only four characters, with all "other" characters lumped together in one count, which also is a simplification for the sake of the example). Near the beginning of the sample text the letter `e` is common. We can see that although at the very start of coding it has an estimated probability of just $\frac{15}{46}$, because its count is incremented by 10 each time it occurs, by the time we get to the fourth `e` it has a probability of $\frac{28}{39}$, and is coded in just 0.48 bits. Notice also that each time the total count exceeds 50 all the counts are divided by 3, and this means that the relative weight of characters not seen recently is even lower. When dividing by 3 it is important that none of the counts are set to zero, which is why the division is not exact. Toward the end where the character `i` becomes common, again it only takes a few occurrences (and especially one scaling) before it dominates the probability distribution.

Using an arithmetic coder like this with very rapid adaptation has been found to be very effective for coding the output of the BWT. Ferragina et al. (2006a), for example, suggest incrementing counts by 64, and halving the counts when they get to 16,383. This means that in the steady state counts are being halved every 128 characters (since the total will be ranging from about 8,192 to 16,383) — only the last 128 characters have their full weight, the 128 before them have half the weight, the ones before them only a quarter, and so on.

In principle even the run-length encoding need not be performed separately from the statistical modeling since it simply captures a particularly predictable sequence of characters. However, by using an RLE stage we effectively have two models that are being switched between: the run-length model which places a high probability on the next character being the same as the last character, and the statistical model which places high probability on the next character being the same as some recent characters.

3.3 Move-to-front lists

Traditionally BWT compressors use a "Move to Front" (MTF) list, which essentially ranks characters based on how recently they have occurred. This is done by keeping a list with one entry for each character in the alphabet, and a character is moved to the front of the list each time it is coded, thereby

Character counts					Total	Probability	$-\log_2 p$
a	e	i	u	other			
12	15	2	1	16	46	$p(\mathtt{e}) = \frac{15}{46}$	1.62 bits
12	25	2	1	16	56		
4	8	1	1	5	19	$p(\mathtt{e}) = \frac{8}{19}$	1.25 bits
4	18	1	1	5	29	$p(\mathtt{e}) = \frac{18}{29}$	0.69 bits
4	28	1	1	5	39	$p(\mathtt{e}) = \frac{28}{39}$	0.48 bits
4	38	1	1	5	49	$p(\mathtt{a}) = \frac{4}{49}$	3.61 bits
14	38	1	1	5	59		
4	12	1	1	1	19	$p(\mathtt{e}) = \frac{12}{19}$	0.66 bits
4	22	1	1	1	29	$p(\mathtt{e}) = \frac{22}{29}$	0.40 bits
4	32	1	1	1	39	$p(\mathtt{u}) = \frac{1}{39}$	5.29 bits
4	32	1	11	1	49	$p(\mathtt{e}) = \frac{32}{49}$	0.61 bits
4	42	1	11	1	59		
1	14	1	3	1	20	$p(\mathtt{u}) = \frac{3}{20}$	2.74 bits
1	14	1	13	1	30	$p(\mathtt{u}) = \frac{13}{30}$	1.21 bits
1	14	1	23	1	40	$p(\mathtt{u}) = \frac{23}{40}$	0.80 bits
1	14	1	33	1	50	$p(\mathtt{a}) = \frac{1}{50}$	5.64 bits
11	14	1	33	1	60		
3	4	1	11	1	20	$p(\mathtt{e}) = \frac{4}{20}$	2.32 bits
3	14	1	11	1	30	$p(\mathtt{i}) = \frac{1}{30}$	4.91 bits
3	14	11	11	1	40	$p(\mathtt{i}) = \frac{11}{40}$	1.86 bits
3	14	21	11	1	50	$p(\mathtt{i}) = \frac{21}{50}$	1.25 bits
3	14	31	11	1	60		
1	4	10	3	1	19	$p(\mathtt{i}) = \frac{10}{19}$	0.93 bits
1	4	20	3	1	29	$p(\mathtt{i}) = \frac{20}{29}$	0.54 bits
1	4	30	3	1	39	$p(\mathtt{i}) = \frac{30}{39}$	0.38 bits

Fig. 3.3. Aggressive adaptive order zero probability estimation for coding the text `eeeeaeeueuuuaeiiiiii`

increasing its rank (and decreasing the corresponding number of bits that will be used to code it next time).

For example, for the Burrows-Wheeler Transform of `mississippi`, we get the text `pssmipissii`. The MTF list that is used for coding this is shown in Figure 3.4, assuming only four characters in the alphabet and starting in alphabetical order. The character at the front (left end) of the list is numbered 0, so the first character to be coded (`p`) has a rank of 2. *After* it has been coded, it is then moved up to rank 0 (the move-to-front step), which in this case happens to be unfortunate because every other character in our small alphabet will be encoded before the next `p`. Next, the `s` is coded as rank 3, and then moved up to rank 0. This time it works out well, because the next character is also an `s`, and is represented by rank 0. The decoder maintains the same list, and after each character is decoded it makes the same updates to the list, so it always has the up-to-date ranking for each character.

MTF list	rank
`im(p)s`	2
`pim(s)`	3
`(s)pim`	0
`spi(m)`	3
`msp(i)`	3
`ims(p)`	3
`p(i)ms`	1
`ipm(s)`	3
`(s)ipm`	0
`s(i)pm`	1
`(i)spm`	0

Fig. 3.4. The MTF ranks for the characters in the BWT transformed text $L =$ `pssmipissii`, assuming that the initial list is `imps`

In practice the list would typically have 256 entries, one for each possible byte value, and for text such as the example in Figure 3.1 we can see that most of the time we will be dealing with characters with very low ranks, with the occasional high value when a new character is encountered for the first time. Figure 3.3 shows the MTF ranks for a segment of this BWT transformed Hamlet text; for such a text most of the codes are very low values (often 0) because of the clustering of characters. One advantage of the ranks being so low is that linear searches can be sufficient to locate characters in the MTF list if the search starts at rank zero, since most searches will succeed in the first few comparisons.

```
a e e a o e u u e a u i i i A a a u a a a a a a a a o a a a a i a
2 1 0 1 5 2 4 0 1 3 2 4 0 0 5 3 0 3 1 0 0 0 0 0 0 0 5 1 0 0 0 4 1

a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

a A A a A A a A A A A a A a a A u a e a a a a a a a a a a u a a a
0 4 0 1 1 0 1 1 0 0 0 1 1 1 0 1 4 2 5 1 0 0 0 0 0 0 0 0 0 2 1 0 0
```

Fig. 3.5. MTF ranks for a series of characters from Hamlet that occur before the sort key `nd`

Like the Burrows-Wheeler Transform, the MTF stage is really just another transform, since an input text of n characters will still give us n ranks to encode. However, the ranks are particularly easy to encode because the low numbers are so common. The zero rank is particularly common — typically around half of the ranks are zero — and runs of zeros occur frequently since they correspond to runs of the same character, which is common in a BWT transformed file. For example, in the BWT transformed Hamlet file 76% of the zero ranks occur in runs of three or more, and only 14% occur in isolation. The longest run of 0 ranks is 575 (this occurs because of the character `o` being common before `u`). In contrast, the longest run of ranks of 1 is 10, which happens to be caused by alternating "s" and space characters occurring before "how". This is more due to chance than any inherent structure in the text. Because of these kinds of patterns, the MTF ranks are often run-length encoded before any further processing is done.

Eventually (whether or not run-length encoding is used), the ranks are represented using an entropy coder — usually arithmetic coding or Huffman coding. In principle arithmetic coding is best when such high probability symbols need to be represented, but combined with run-length encoding of the output, a Huffman code can be made to work satisfactorily (and in fact this is what is used by BZIP2). Probabilities are estimated using a simple order-0 model; that is, there seems to be little relationship between consecutive MTF ranks apart from the runs of zero ranks. Even a fixed code such as the γ code described in Section 3.1 produces satisfactory results for compressing the output of the MTF, and may be preferred if speed and simplicity are preferred to getting the best possible compression.

The MTF approach is very fast to promote new symbols as they are encountered; in fact, just one occurrence will instantly have a character designated as the most likely to occur next. This is a good assumption much of the time, but the BWT-transformed text will also contain a number of characters that occur in isolation, and promoting these so quickly will result in a higher cost of coding subsequent characters while the one-off character slowly works its way back down the MTF list. A number of variations to MTF have been proposed, such as the "sticky MTF", which checks how often the symbol at the head of the MTF list has been used before demoting it; if it has only been used once then it is moved some distance down the list, whereas if it has occurred twice in succession, it stays at the top of the list and will be demoted slowly in the future.

Recent research has established that the MTF stage may not be needed to compress transformed texts effectively, but instead run-length encoding followed by a fast-adapting entropy coder are sufficient to get excellent compression. This makes sense given the predominance of zero ranks in the MTF output; a zero rank simply means that the previous character has been repeated, and runs of zeros correspond to runs of the repeated character. Hence an appropriately tuned run-length encoder can be expected to pick up this structure.

3.4 Frequency counting methods

Frequency counting methods improve on MTF by basing the ranking of symbols on their frequencies. The simplest approach, to simply give the highest rank to the symbol with the highest frequency, would not be very effective since it would take too long to adapt to favoring symbols that have become popular at the expense of previously popular ones.

The Weighted Frequency Count (WFC) addresses this by defining a function based on symbol frequencies and also the distance to the last occurrence of each symbol within a sliding window, with higher weights being given to more recent occurrences of a symbol. While this approach does give good compression, it is very slow because of the computation involved.

An alternative is the Incremental Frequency Count (IFC) which approximates the WFC by keeping count of character occurrences as they are observed, giving more weight to recent occurrences. This makes computation faster, but at a small cost in compression performance.

3.5 Inversion Frequencies (IF)

The inversion frequencies method is based on the distance between occurrences of symbols in the BWT-transformed text. It relates to an inverted index, which contains a list for each symbol indicating where it occurs in a text.

For example, our sample transformed text `pssmipissii` could be represented by the inverted index shown in Figure 3.6, which gives the number of characters preceding the occurrence of each character named. In the example, “p” occurs after 0 and 5 characters.

```
i: 4, 6, 9, 10
m: 3
p: 0, 5
s: 1, 2, 7, 8
```

Fig. 3.6. Inverted index of the sample transformed text `pssmipissii`

This table can be represented more efficiently by storing the gap since the last occurrence of a character, rather than its absolute position. Furthermore, if the output is being reconstructed by going through the symbols in lexical order, it is only necessary to store the number of intervening symbols that are lexically greater than the current one, as shown in Figure 3.7. From the information in Figure 3.7 the original text is reconstructed by decoding the character `i` first, then `m` and so on. The numbers are used to skip only unfilled slots in the decoded text; the last line is not strictly needed, since the character `s` is simply placed in any unfilled slots.

The inversion frequencies method is comparable in performance to MTF-based methods, performing better on some files and worse on others.

```
i: 4, 1, 2, 0
m: 3
p: 0, 2
s: 0, 0, 0, 0
```

Fig. 3.7. Inversion frequencies representation of the transformed text

3.6 Distance coding

Distance coding has a lot in common with the inversion frequencies method, but is based on encoding the start of each *run* of characters in the transformed text, where a run is a maximal consecutive repetition of the same character, which might be as short as length 1. The end of the run need not be coded, since it will be marked by the start of a run of another character. To seed the decoding, we need to know where the first occurrence of each character is in the text. Subsequent runs are represented as the distance from the previous run of the same character. The distance between two characters is simply the difference between their positions in the array, so two adjacent characters are a distance of 1 apart[1]. A distance of 0 is used to mark the end of the entries for each character.

A simplistic version of distance coding is shown in Figure 3.8, representing the text `pssmipissii`. For example, the first `i` first occurs at character 5 in the text, and then "runs" of `i` occur 2 characters later (a single `i`) and then 3 characters after that (the last two `i`'s in the string). We can detect the end of the first of these "runs" (for example, we know there is a single `i` because the entry for `p` shows that there is a `p` at position 6). The end of each row of distances is marked with a 1.

Character	First occurrence	Distance to next run
i	5	2, 3, 0
m	4	0
p	1	5, 0
s	2	5, 0

Fig. 3.8. Distance coding for the BWT output `pssmipissii`

In addition to this main idea that the ends of runs of a character can be encoded implicitly, there are two other observations that can be used to reduce the amount of information transmitted by this simplified distance coding:

[1] A distance of 1 cannot occur in the coding as described so far, since run-lengths are maximal and therefore adjacent runs are not possible, but later we shall see that it is useful.

1. If we know the length of the file to be decoded, the final 0 at the end of each list is sometimes redundant because it will sometimes be the case that there is no more space in the output for any more runs, hence the decoder already knows that there will be no more occurrences of that character.
2. Some of the characters being skipped by the distance code are known to the decoder, and they need not be included in the distance. This enables us to reduce the value representing the distance.

For example, in Figure 3.8 the first occurrence column tells us that the text is of the form `ps•mi••••••` (where a dot is an unknown character). The first distance for `p` is 5 (putting the second `p` at position 6 in the text), but based on the second rule above we only need to record the distance as 2 (i.e. it is the second *unknown* character). Once that `p` is decoded we have the text `ps•mip•••••`, and we know that the only possible character for the first unknown gap is `s`, yielding `pssmip•••••` without using any extra information. The distance from `i` to its next occurrence is 2, but since the `p` that must be skipped is already known, we can simply record a distance of 1 (the next `i` is in the next unknown position).

Thus relatively few numbers need be recorded to reconstruct the text, particularly if there are long runs and a limited vocabulary. These numbers are then coded using an entropy coder, yielding good compression results.

3.7 Wavelet trees

Another approach proposed for coding a BWT-transformed text is using *wavelet trees*. Figure 3.9 shows a wavelet tree for our sample transformed text, `pssmipissii`. The leaves of the tree correspond to the characters that occur in the string to be represented. The root represents the transformed text, and partitions its characters into two groups — those down the left branch and those down the right. This partitioning is represented by the string of bits associated with the node; a 0 means that the corresponding character is in the left branch, and a 1 means that it is in the right branch. The wavelet tree is stored using a binary-heap-like structure; that is, it is a complete tree, with nodes being filled from left to right, top to bottom.

Coding is achieved by applying run-length encoding to the bit patterns for each node; the decoder can reconstruct the original text given those patterns. The run-lengths are coded using the γ code described above in Section 3.1.

Using wavelet trees can be faster than an MTF list, and they have some nice theoretical properties as well as providing a potentially useful structure for searching the text. However, because a simpler coding is used, the compression performance will be slightly less than the best methods, and with MTF, they are likely to be outperformed all round by the simple fast-adapting system described in Section 3.2.

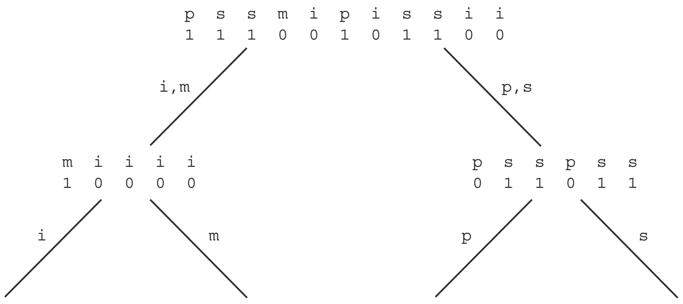

Fig. 3.9. A wavelet tree for the string `pssmipissii`

3.8 Other permutations

The Burrows-Wheeler Transform appears to be the only possible basis of a reversible permutation of the text that can be used for compression; the only other generally reversible transformations are trivial ones that don't usually help with compression performance (such as swapping every second character with its successor). Some variations of the basic BWT are possible. For example, it is possible to limit the size of the substrings being sorted to some constant k characters (typically $k = 2$ to 4). This will inevitably mean that some identical contexts occur, in which case the original order of the substrings is used to resolve the ordering (that is, a stable sort is used). The main advantage of limiting the context size is that it can speed up processing, especially if a very small context is used, since the entire sort can be done using a radix sort. It also avoids the degenerate case where the original text contains many runs of the same character, which can slow down sorting significantly. Contexts of about four characters are enough to capture most of the dependencies in text, and so the loss of compression is very small (typically well under 5% if $k = 4$). A trivial case occurs when $k = 0$, in which case no re-ordering occurs, and the transformed text is the same as the original. Of course, compression performance will be poor in this case. This type of permutation is discussed in more detail in Section 6.1.

A method related to sorting with limited contexts is used in a technique called "Compression boosting", which partitions the BWT transformed text into subsequences of characters that are from related contexts. The value of this can be seen in Figure 3.10, where the sort key initially starts with `tz` and then has a transition to starting with `u␣` (a `u` followed by a space). The `tz` context is primarily preceded by `n` and `i`, whereas `u␣` is preceded by `o`. Compression boosting partitions the text so that the bias being used for one context can be dropped when the next one starts, avoiding having to take some time to "learn" that a new set of characters is more likely. An important feature of compression boosting is that it finds the *optimal* partition for a given entropy coder in linear time.

F	*L*
`tz and Guildenstern.] Ros. Wh` ...	n
`tz and Guildenstern, who go o` ...	n
`tz: And I beseech you instant` ...	n
`tz, Courtier. Guildenstern, C` ...	n
`tz go to’t. Ham. Why, man, th` ...	n
`tz! Good lads, how do ye both` ...	n
`tz, Guildenstern, and Attenda` ...	n
`tz, Guildenstern, and others.` ...	n
`tz, Guildenstern, and some At` ...	n
`tz, Guildenstern, &c.] Ham. G` ...	n
`tz.] How now! what hath befal` ...	n
`tzers? let them guard the doo` ...	i
—	
`u a daughter? Pol. I have, my` ...	o
`u a groaning to take off my e` ...	o
`u a more horrid hent: When he` ...	o
`u a place. [Danish march. A f` ...	o
`u a spirit of health or gobli` ...	o
`u a wholesome answer; my wit’` ...	o
`u advise me? Laer. I am lost ` ...	o
`u again to bed; Pinch wanton ` ...	o
`u all, If you have hitherto c` ...	o
`u all without. Danes. No, let` ...	o
`u alone. Mar. Look with what ` ...	o
`u amble, and you lisp, and ni` ...	o

Fig. 3.10. A variation of BWT where the sorting uses only the first two characters to determine sort order

An efficient algorithm for performing the partitioning for compression boosting is available using suffix trees to determine where the partitions should fall. Compression boosting can be applied to various other local-to-global techniques to improve them, but it is still out-performed by a simple arithmetic coder that is set up to adapt quickly because the fast adaptation soon reduces the significance of the change of context.

Variations on the BWT, such as using limited-length sort keys as above, or using English words as the alphabet, are discussed in detail in Chapter 6.

3.9 Block size

Because a BWT-based compression system must have access to a whole text before it can be encoded, a text which is too large to be processed in memory must be broken up into blocks that can be accommodated. These blocks are then compressed independently[2].

So far we have put aside the issue of the size of the blocks to be used. In general, these blocks should be as large as possible to maximize the opportunity to capture patterns in the text, but will be limited by available memory. Typically the amount of memory required is about 5 to 8 times the block size (in bytes), with 1 Mbyte being a reasonable size for blocks (and therefore about 8 Mbyte being used for encoding). An important factor is that the access to memory for BWT is quite random, and implementations should consider how this will interact with caching, since the random access could lead to very slow memory retrieval if the cache cannot store the entire block. Thus the limit in memory size may be more related to cache sizes than the total RAM in a computer.

Burrows and Wheeler's original paper evaluated the effect of block size on compression performance, and found that gains of a few percent were still being made by quadrupling the block size from 16 Mbytes to 64 Mbytes. And of course, gains will be made only if the file being compressed is larger than the block size, and if the material at the start of the file is similar to that much later on.

The BZIP2 program is a general-purpose compression system based on the Burrows-Wheeler Transform. It has a maximum block size of 900 kilobytes, which is larger than many of the files that are likely to be compressed, yet allows the implementation to work with a small footprint in the memory of a standard computer. At this size gains in the order of 1% are made by increasing the block size by about 10%, so larger blocks may well give a little compression gain, but is likely to be at the expense of severe speed deterioration because of cache misses, and may never be noticed since files of that size are more likely to be audio or video rather than text, which are typically compressed with a lossy compression method. Lossless compression would be needed for genomic data, but this kind of file generally does not benefit significantly from larger block sizes.

3.10 Further reading

Entropy coders (especially Huffman and arithmetic coding) are discussed at length in standard texts on lossless compression (see the Further reading section of Chapter 1). They are a particular focus of the book by Moffat and Turpin (2002), and the implementation of arithmetic coding is discussed in Moffat et al. (1995) and Moffat et al. (1998). Some BWT-based systems propose using "range coding", which has similar performance to arithmetic coding, but avoids some potential issues with patents on some versions of

[2] This is why BWT-based compression systems are sometimes referred to as "Block-sorting compression".

arithmetic coding; for examples see Foschini et al. (2004) and Ferragina et al. (2006a). Run-length encoding in the context of the BWT is discussed in detail by Deorowicz (2002) and Fenwick (2007).

The classic reference for fixed codes such as the α, γ and δ codes is Elias (1975), which includes universal codes that will be within a constant factor of the entropy even though the probability distribution is not specified. A detailed description of the use of such codes in compression is given in Witten et al. (1999) and Sayood (2003). Fenwick (2002a, 2003c) provides an extensive investigation of the practical use of integer codes with the BWT, and a related code used by Wheeler called the "1/2" code (Fenwick, 1996a, 2007).

The observation that a simple run-length coder with arithmetic coding outperforms many other techniques for compressing a BWT transformed text was made by Ferragina, Giancarlo and Manzini in their 2006 paper (Ferragina et al., 2006a). This paper introduces RleAc, and compares it empirically with a number of other coders, including "range coding", which is very similar to arithmetic coding. The idea of using a fast adapting arithmetic coder is not a new one; Fenwick (1996a) reported using an arithmetic coder with an increment of 16 and a limit of 8192.

The MTF method was proposed in Burrows and Wheeler's original paper (Burrows and Wheeler, 1994); seminal work on MTF was done by Bentley et al. (1986) and Elias (1987). The proportion of zero ranks after MTF has been applied to a BWT transformed string was reported by Fenwick (1996b) to be about 63%. This paper proposes an approximate entropy coder described as a "unary" model, which takes advantage of the probability distribution of ranks but is simpler and therefore faster than arithmetic or Huffman coding. Balkenhol and Kurtz (2000) explore a number of variations for compressing the transformed text.

Variations on the MTF list which avoid the rapid promotion of one-off symbols were first suggested in Burrows and Wheeler's original paper (Burrows and Wheeler, 1994), but were not evaluated. Subsequently a number of variations have been published, including "Sticky MTF" by Fenwick (2002a, 2003c), Schindler's system (Schindler, 1997a,b), which does not promote a symbol to rank 0 unless it has been seen twice, Balkenhol et al. (1999), where symbols are first promoted to rank 1 before rank 0, and Chapin (2000), which switches between two update algorithms. Wirth (2001) compares the performance of MTF and related approaches on BWT output. Bachrach and El-Yaniv (1997) provide a detailed empirical evaluation of the performance of the MTF and its variations on general text.

The idea of distance coding originated in a posting to the Usenet group `comp.compression` by Edgar Binder in 2000. It was described and evaluated in Deorowicz (2002), and in Gagie and Manzini (2007). Inversion frequencies (IF) as an alternative to MTF are introduced by Arnavut and Magliveras (1997a) and analyzed by Ferragina et al. (2006b). Improvements for the IF method are noted by Abel (2007b), including changing the order in which characters are stored in the table to achieve slight compression improvements. Coding

BWT-transformed text without ranks is explored by Wirth (2001) and Wirth and Moffat (2001).

The Weighted Frequency Count (WFC) was introduced by Deorowicz (2002) and variations to it are explored by Abel (2007b). A comparison of related methods is given by Abel (2007a). A comparison of move-to-front, distance coding, and inversion frequencies can be found in Gagie and Manzini (2007). The Incremental Frequency Count (IFC) is due to Jürgen Abel (Abel, 2005, 2007a).

"Wavelet trees" are introduced by Grossi et al. (2003) and applied to the BWT in Foschini et al. (2004). Further analysis and application to the BWT is given in Ferragina et al. (2006b).

The observation that limited-length contexts can be used for the BWT sort was published by Schindler (1997a), and is discussed further in Section 6.1. Partitioning the transformed text for "compression boosting" was described in Ferragina and Manzini (2004), Ferragina et al. (2005a), and Ferragina et al. (2006a). Deorowicz describes a system that does not explicitly partition the BWT text according to context, but infers it from the transformed file through a process called "context exhumation" (Deorowicz, 2005). Other work on the Burrows-Wheeler permutation is described by Arnavut (2002) and Crochemore et al. (2005).

A special edition of *Theoretical Computer Science* in November 2007 featured a number of papers about the BWT that are relevant to the material in this chapter.