Burrows–Wheeler transform

The **Burrows–Wheeler transform** (**BWT**, also called **block-sorting compression**) rearranges a character string into runs of similar characters. This is useful for compression, since it tends to be easy to compress a string that has runs of repeated characters by techniques such as move-to-front transform and run-length encoding. More importantly, the transformation is **reversible**, without needing to store any additional data. The BWT is thus a "free" method of improving the efficiency of text compression algorithms, costing only some extra computation.

1 Description

The Burrows–Wheeler transform is an algorithm used in data compression techniques such as bzip2. It was invented by Michael Burrows and David Wheeler in 1994 while working at DEC Systems Research Center in Palo Alto, California.[1] It is based on a previously unpublished transformation discovered by Wheeler in 1983.

When a character string is transformed by the BWT, the transformation permutes the order of the characters. If the original string had several substrings that occurred often, then the transformed string will have several places where a single character is repeated multiple times in a row.

For example:

The output is easier to compress because it has many repeated characters. In fact, in the transformed string, there are a total of eight runs of identical characters: XX, II, XX, SS, PP, .., II, and III, which together make 17 out of the 44 characters in it.

2 Example

The transform is done by sorting all rotations of the text into lexicographic order, by which we mean that the 8 rotations appear in the second column in a different order, in that the 8 rows have been sorted into lexicographical order. We then take as output the last column and the number k = 7 of the row that the non rotated row ends up in. For example, the text "^BANANA|" is transformed into "BNN^AA|A" through these steps (the red | character indicates the 'EOF' pointer):

The following pseudocode gives a simple (though inefficient) way to calculate the BWT and its inverse. It assumes that the input string s contains a special character 'EOF' which is the last character, occurs nowhere else in the text, and is ignored during sorting.

function BWT (*string* s) create a table, rows are all possible rotations of s sort rows alphabetically return (last column of the table) **function** inverseBWT (*string* s) create empty table **repeat** length(s) **times** // first insert creates first column insert s as a column of table before first column of the table sort rows of the table alphabetically return (row that ends with the 'EOF' character)

3 Explanation

To understand why this creates more-easily-compressible data, consider transforming a long English text frequently containing the word "the". Sorting the rotations of this text will group rotations starting with "he " together, and the last character of that rotation (which is also the character before the "he ") will usually be "t", so the result of the transform would contain a number of "t" characters along with the perhaps less-common exceptions (such as if it contains "Brahe ") mixed in. So it can be seen that the success of this transform depends upon one value having a high probability of occurring before a sequence, so that in general it needs fairly long samples (a few kilobytes at least) of appropriate data (such as text).

The remarkable thing about the BWT is not that it generates a more easily encoded output—an ordinary sort would do that—but that it is *reversible*, allowing the original document to be re-generated from the last column data.

The inverse can be understood this way. Take the final table in the BWT algorithm, and erase all but the last column. Given only this information, you can easily reconstruct the first column. The last column tells you all the characters in the text, so just sort these characters alphabetically to get the first column. Then, the first and last columns (of each row) together give you all *pairs* of successive characters in the document, where pairs are taken cyclically so that the last and first character form a pair. Sorting the list of pairs gives the first *and second* columns. Continuing in this manner, you can reconstruct the entire list. Then, the row with the "end of file" character at the end is the original text. Reversing the example above is done like this:

4 Optimization

A number of optimizations can make these algorithms run more efficiently without changing the output. There is no need to represent the table in either the encoder or decoder. In the encoder, each row of the table can be represented by a single pointer into the strings, and the sort performed using the indices. Some care must be taken to ensure that the sort does not exhibit bad worst-case behavior: Standard library sort functions are unlikely to be appropriate. In the decoder, there is also no need to store the table, and in fact no sort is needed at all. In time proportional to the alphabet size and string length, the decoded string may be generated one character at a time from right to left. A “character” in the algorithm can be a byte, or a bit, or any other convenient size.

One may also make the observation that mathematically, the encoded string can be computed as a simple modification of the suffix array, and suffix arrays can be computed with linear time and memory.

There is no need to have an actual 'EOF' character. Instead, a pointer can be used that remembers where in a string the 'EOF' would be if it existed. In this approach, the output of the BWT must include both the transformed string, and the final value of the pointer. That means the BWT does expand its input slightly. The inverse transform then shrinks it back down to the original size: it is given a string and a pointer, and returns just a string.

A complete description of the algorithms can be found in Burrows and Wheeler’s paper, or in a number of online sources.

5 Bijective variant

When a bijective variant of the Burrows–Wheeler transform is performed on "^BANANA”, you get ANNBAA^ without the need for a special character for the end of the string. This forces one to increase character space by one, or to have a separate field with a numerical value for an offset. Either of these features makes data compression more difficult. When dealing with short files, the savings are great percentage-wise.

The bijective transform is done by sorting all rotations of the Lyndon words. In comparing two strings of unequal length, one can compare the infinite periodic repetitions of each of these in lexicographic order and take the last column of the base-rotated Lyndon word. For example, the text "^BANANA|" is transformed into “ANNBAA^|" through these steps (the red | character indicates the EOF pointer) in the original string. The EOF character is unneeded in the bijective transform, so it is dropped during the transform and re-added to its proper place in the file.

The string is broken into Lyndon words so the words in the sequence are decreasing using the comparison method above. "^BANANA” becomes (^) (B) (AN) (AN) (A), but Lyndon words are combined into (^) (B) (ANAN) (A).

The above may be viewed as four cycles
^ = (^)(^)... = ^^^^^...
B = (B)(B)... = BBBB...
ANAN = (ANAN)(ANAN)... = ANANANAN...
A = (A)(A).. = AAAAA..
or 5 cycles WHERE ANAN broken into 2
AN = (AN) (AN) ... = ANANANAN
AN = (AN) (AN) ... = ANANANAN

If a cycle is N character it will be repeated N times:

(^)
(B)
(ANAN)
(A)

or

(^)
(B)
(AN)
(AN)
(A)

to get the ^BANANA

Since any rotation of the input string will lead to the same transformed string, the BWT cannot be inverted without adding an EOF marker to the input or, augmenting the output with information such as an index, making it possible to identify the input string from all its rotations.

There is a bijective version of the transform, by which the transformed string uniquely identifies the original. In this version, every string has a unique inverse of the same length.[2][3]

The fastest versions are linear in time and space.

The bijective transform is computed by factoring the input into a non-increasing sequence of Lyndon words; such a factorization exists in the Chen–Fox–Lyndon theorem,[4] and may be found in linear time.[5] The algorithm sorts the rotations of all the words; as in the Burrows–Wheeler transform, this produces a sorted sequence of *n* strings. The transformed string is then obtained by picking the final character of each string in this sorted list.

For example, applying the bijective transform gives:

The bijective transform includes eight runs of identical characters. These runs are, in order: XX, II, XX, PP, .., EE, .., and IIII.

In total, 18 characters are used in these runs.

6 Dynamic Burrows–Wheeler transform

When a text is edited, its Burrows-Wheeler transform will change. Salson *et al.*[6] propose an algorithm that deduces the Burrows–Wheeler transform of an edited text from that of the original text, doing a limited number of local reorderings in the original Burrows–Wheeler transform, which can be faster than constructing the Burrows–Wheeler transform of the edited text directly.

7 Sample implementation

This Python implementation sacrifices speed for simplicity: the program is short, but takes more than the linear time that would be desired in a practical implementation.

Using the null character as the end of file marker, and using s[i:] + s[:i] to construct the ith rotation of s, the forward transform takes the last character of each of the sorted rows:

def bwt(s): """Apply Burrows-Wheeler transform to input string.""" assert "\0” not in s, “Input string cannot contain null character ('\\0')" s += "\0” # Add end of file marker table = sorted(s[i:] + s[:i] for i in range(len(s))) # Table of rotations of string last_column = [row[−1:] for row in table] # Last characters of each row return "".join(last_column) # Convert list of characters into string

The inverse transform repeatedly inserts r as the left column of the table and sorts the table. After the whole table is built, it returns the row that ends with null, minus the null.

def ibwt(r): """Apply inverse Burrows-Wheeler transform.""" table = [""] * len(r) # Make empty table for i in range(len(r)): table = sorted(r[i] + table[i] for i in range(len(r))) # Add a column of r s = [row for row in table if row.endswith("\0”)][0] # Find the correct row (ending in "\0”) return s.rstrip("\0”) # Get rid of trailing null character

Here is another, more efficient method for the inverse transform. Although more complex, it increases the speed greatly when decoding lengthy strings.

def ibwt(r, *args): """Inverse Burrows-Wheeler transform. args is the original index \ if it was not indicated by a null byte.""" firstCol = "".join(sorted(r)) count = [0]*256 byteStart = [−1]*256 output = [""] * len(r) shortcut = [None]*len(r) #Generates shortcut lists for i in range(len(r)): shortcutIndex = ord(r[i]) shortcut[i] = count[shortcutIndex] count[shortcutIndex] += 1 shortcutIndex = ord(firstCol[i]) if byteStart[shortcutIndex] == −1: byteStart[shortcutIndex] = i localIndex = (r.index("\x00”) if not args else args[0]) for i in range(len(r)): #takes the next index indicated by the transformation vector nextByte = r[localIndex] output [len(r)-i-1] = nextByte shortcutIndex = ord(nextByte) #assigns localIndex to the next index in the transformation vector localIndex = byteStart[shortcutIndex] + shortcut[localIndex] return "".join(output).rstrip("\x00”)

8 BWT in bioinformatics

The advent of next-generation sequencing (NGS) techniques at the end of the 2000 decade has led to another application of the Burrows–Wheeler transformation. In NGS, DNA is fragmented into small pieces, of which the first few bases are sequenced, yielding several millions of “reads”, each 30 to 500 base pairs (“DNA characters”) long. In many experiments, e.g., in ChIP-Seq, the task is now to align these reads to a reference genome, i.e., to the known, nearly complete sequence of the organism in question (which may be up to several billion base pairs long). A number of alignment programs, specialized for this task, were published, which initially relied on hashing (e.g., Eland, SOAP,[7] or Maq[8]). In an effort to reduce the memory requirement for sequence alignment, several alignment programs were developed (Bowtie,[9] BWA,[10] and SOAP2[11]) that use the Burrows–Wheeler transform.

9 References

[1] Burrows, Michael; Wheeler, David J. (1994), *A block sorting lossless data compression algorithm*, Technical Report 124, Digital Equipment Corporation

[2] Gil, J.; Scott, D. A. (2009), *A bijective string sorting transform* (PDF)

[3] Kufleitner, Manfred (2009), “On bijective variants of the Burrows-Wheeler transform”, in Holub, Jan; Žďárek, Jan, *Prague Stringology Conference*, pp. 65–69, arXiv:0908.0239.

[4] • Lothaire, M. (1997), *Combinatorics on words*, Encyclopedia of Mathematics and Its Applications **17**, Perrin, D.; Reutenauer, C.; Berstel, J.; Pin, J. E.; Pirillo, G.; Foata, D.; Sakarovitch, J.; Simon, I.; Schützenberger, M. P.; Choffrut, C.; Cori, R.; Lyndon, Roger; Rota, Gian-Carlo. Foreword by Roger Lyndon (2nd ed.), Cambridge University Press, p. 67, ISBN 0-521-59924-5, Zbl 0874.20040

[5] Duval, Jean-Pierre (1983), “Factorizing words over an ordered alphabet”, *Journal of Algorithms* **4** (4): 363–381, doi:10.1016/0196-6774(83)90017-2, ISSN 0196-6774, Zbl 0532.68061.

[6] Salson M, Lecroq T, Léonard M and Mouchard L (2009). "A Four-Stage Algorithm for Updating a Burrows–Wheeler Transform". *Theoretical Computer Science* **410** (43): 4350–4359. doi:10.1016/j.tcs.2009.07.016.

[7] Li R et al. (2008). "SOAP: short oligonucleotide alignment program". *Bioinformatics* **24** (5): 713–714. doi:10.1093/bioinformatics/btn025. PMID 18227114.

[8] Li H, Ruan J, Durbin R (2008-08-19). "Mapping short DNA sequencing reads and calling variants using mapping quality scores". *Genome Research* **18** (11): 1851–1858. doi:10.1101/gr.078212.108. PMC 2577856. PMID 18714091.

[9] Langmead B, Trapnell C, Pop M, Salzberg SL (2009). "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome". *Genome Biology* **10** (3): R25. doi:10.1186/gb-2009-10-3-r25. PMC 2690996. PMID 19261174.

[10] Li H, Durbin R (2009). "Fast and accurate short read alignment with Burrows–Wheeler Transform". *Bioinformatics* **25** (14): 1754–1760. doi:10.1093/bioinformatics/btp324. PMC 2705234. PMID 19451168.

[11] Li R et al. (2009). "SOAP2: an improved ultrafast tool for short read alignment". *Bioinformatics* **25** (15): 1966–1967. doi:10.1093/bioinformatics/btp336. PMID 19497933.

10 External links

- Compression comparison of BWT based file compressors
- Article by Mark Nelson on the BWT
- A Bijective String-Sorting Transform, by Gil and Scott
- Yuta's openbwt-v1.5.zip contains source code for various BWT routines including BWTS for bijective version
- On Bijective Variants of the Burrows–Wheeler Transform, by Kufleitner
- Blog post and project page for an open-source compression program and library based on the Burrows–Wheeler algorithm

11 Text and image sources, contributors, and licenses

11.1 Text

11.2 Images

11.3 Content license