Lecture 20: Burrow-Wheeler Transform Continued

Sam McCauley

November 22, 2021

Williams College

Admin

- Office hours as normal today and tomorrow
- Then break!

Plan going forward

- Finish up BWT today
- After break: suffix arrays! (Two lectures, one assignment.)
- Then Van Emde boas trees lecture and finally a review class.

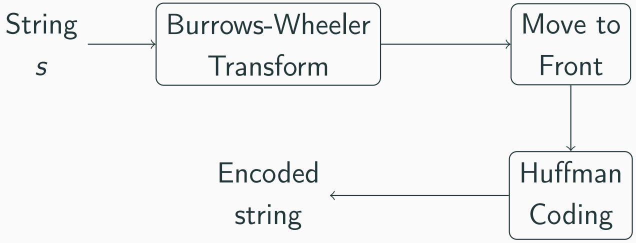
Recap

Compression: Our goals

- Take a string s, map it to a string $m = c(s)$ (using a compression function c)
- There exists a decryption function d, such that for all s, $d(c(s)) = s$
    - *Lossless* compression: we want to be able to recover the exact string
- Goal: make the string smaller. Want $|m| \leq |s|$.

Our strategy

- Use Burrows-Wheeler transform to make characters with similar "contexts" appear close together
- Use Move-to-Front coding to make close-together characters into frequent characters
- Use Huffman coding to map characters to bits; common characters will be the cheapest to encode



Move-to-front transform

Transform a string s into a string $MTF(s)$:

- Keep an list L of all possible characters. Start with L just keeping the characters in some arbitrary order.
    - For these examples: $L = \{a, b, c, \ldots, y, z\}$
    - In general, L encodes all 255 possible `char` values. Start with $L[i] = i$.
- Start with empty s'. For each $i = 1$ to $|s|$:
    - If $s[i]$ is the jth character in L, append j to s
    - Move j to the front of L.
- Return s' as $MTF(s)$ when done

Move-to-front transform: decode

Transform a string $s' = MTF(s)$ into the original string s:

- Goal: recover L at each time step used when encoding
- Start with same L
- Start with empty s. For each $i = 1$ to $|s'|$:
    - If $s'[i] = j$, then write $L[j]$ to s
    - Move j to the front of L.
- Let's decode the board examples.

Burrows-Wheeler Transform

Recall: goals

- Make characters with similar contexts (i.e. u generally coming after q) into close-together characters
- Needs to be reversible
- Would like it to be fast

BWT: Looking at the context of a character

b	a	n	a	n	a	$
$	b	a	n	a	n	a
a	$	b	a	n	a	n
n	a	$	b	a	n	a
a	n	a	$	b	a	n
n	a	n	a	$	b	a
a	n	a	n	a	$	b

- Take all n *circular suffixes* of the string (wrap around from beginning)
- The "context" of each character is the $n - 1$ characters following it

BWT: Looking at the context of a character

b	a	n	a	n	a	$
$	b	a	n	a	n	a
a	$	b	a	n	a	n
n	a	$	b	a	n	a
a	n	a	$	b	a	n
n	a	n	a	$	b	a
a	n	a	n	a	$	b

- Key idea: since we have all circular suffixes, the context of the *last* character in a row is the first $n - 1$ characters in the row
- Let's sort the contexts (sort the rows)

The Burrows-Wheeler Transform

\$	b	a	n	a	n	a
a	\$	b	a	n	a	n
a	n	a	\$	b	a	n
a	n	a	n	a	\$	b
b	a	n	a	n	a	\$
n	a	\$	b	a	n	a
n	a	n	a	\$	b	a

- First, sort the suffixes lexicographically
- Take the *last character* of each suffix
- This is the BWT of the string
- BWT(banana) = annb\$aa

Still to show: reversible and efficient

Let's start with reversible

- On the board: let's say we have a BWT transformed string; the result is e$elplepa
- What do we know based on how the BWT works? Can I recover ANYTHING about the original string? Can I recover anything about the original table?
- Result:
- appellee$

Reversing the BWT

- If we sort the BWT-transformed string, we obtain the first column of the table
- This gives us *all pairs* of characters. If we sort THOSE, the second character of the result gives the second column of the table
- So on until the table is recovered
- Our string: row ending with $

Reversing the BWT

- If we sort the BWT-transformed string, we obtain the first column of the table

a
n
n
b
$
a
a

Reversing the BWT

- If we sort the BWT-transformed string, we obtain the first column of the table

\$	a
a	n
a	n
a	b
b	\$
n	a
n	a

Reversing the BWT

- If we sort the BWT-transformed string, we obtain the first column of the table
- This gives us *all pairs* of characters. If we sort THOSE, the second character of the result gives the second column of the table

\$	b	a
a	\$	n
a	n	b
a	n	n
b	a	\$
n	a	a
n	a	a

Reversing the BWT

- If we sort the BWT-transformed string, we obtain the first column of the table
- This gives us *all pairs* of characters. If we sort THOSE, the second character of the result gives the second column of the table
- So on until the table is recovered
- Our string: row ending with \$

\$	b	a	n	a	n	a
a	\$	b	a	n	a	n
a	n	a	\$	b	a	n
a	n	a	n	a	\$	b
b	a	n	a	n	a	\$
n	a	\$	b	a	n	a
n	a	n	a	\$	b	a

Efficiency

How much time and space does *encoding* take for a string of length n? First, encoding:

- Filling out the table: $O(n^2)$ time and space.
- Sorting the table?
    - $O(n)$ time to compare two items
    - $O(n \log n)$ comparisons
    - Total: $O(n^2 \log n)$ time.

Efficiency

How much time and space does this method take now for a string of length n? Now, decoding:

- Recover one column at a time
- To recover a column: sort (last column) appended to current columns we have
    - $O(n)$ time to compare two items
    - $O(n \log n)$ comparisons
    - $O(n^2 \log n)$ time per column
    - $O(n^3 \log n)$ time overall

This is terrible. But there's a ton of redundancy here. Can we do better?

Efficient BWT Encoding

- Using a clever method, can get much faster time BWT encoding
- Need another data structure: suffix array

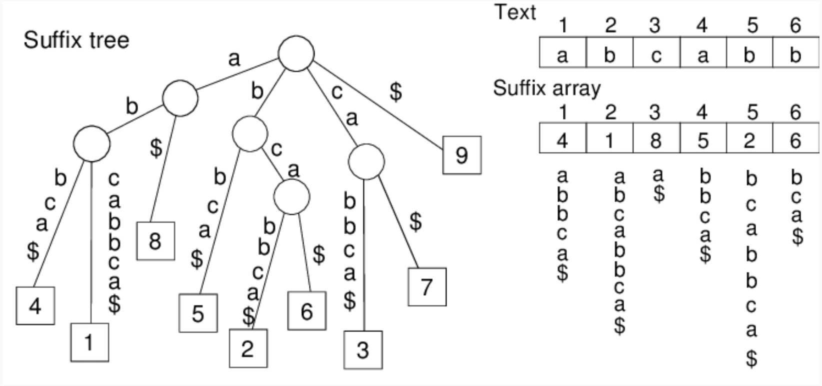
Suffix Array

Any string of length n has a suffix array A of n indices:

- $A[i]$ contains the index of the ith suffix of s in sorted order.
- Example: suffix array for banana\$ is:
- 6 5 3 1 0 4 2
- Very similar to what we want for BWT. (We'll talk about that in a second). How fast do you think one can compute this?
- Answer: can do this in $O(n)$ time for constant-size alphabet. (*Faster* than sorting.)
- We'll talk about how to do this after Thanksgiving

Suffix Array Applications

- *Far* wider use than just creating BWT
- In short: a suffix array is compressed, but allows trie-like operations



Text

1	2	3	4	5	6
a	b	c	a	b	b

Suffix array

1	2	3	4	5	6
4	1	8	5	2	6
abbca$	abcabbca$	a$	bbca$	bcabbca$	bca$

Suffix Array to BWT

- Let's say you are given the suffix array for the string.
- How can you get the BWT?
- Let's do it one character at a time.
- The ith character of the string is the *last column character* corresponding to the ith suffix in sorted order
- So: $\mathrm{BWT}[i] = s[j]$, where $j = SA[i] - 1$. (Watch out for negative indices)
- Linear time method to calculate the BWT!
- Any practical problems with this methdology?
    - Very cache-inefficient if our string is large enough for that to be an issue

Where we are

- If you're given a suffix array (you are), can calculate the BWT in a simple linear scan. No extra space (beyond the original string and the suffix array)
- Now: can we reverse the BWT quickly as well?
- Let's fill out the BWT in reverse order. What characters can we fill in?
- Key observation: let's say we just wrote a character in the last column. We want to find the character before that in the original string. If we can find that character in the first column, we know the next character to write (as it's the corresponding last-column-character).

Quickly inverting the BWT

Lemma 1

Consider a character c in the last column of the BWT table. The order of all occurrences of c in the last column is the same as the order of all occurrences of c in the first column of the table.

Proof: Let's say the ith instance of c is followed by a (circular) suffix s_i in the original string s. Then row i of the BWT table consists of s_i concatenated with c. Therefore, the order of all instances of c in the *last column* of the table is exactly the same as the lexicographic order of the s_i.

Quickly inverting the BWT

Lemma 2

Consider a character c in the last column of the BWT table. The order of all occurrences of c in the last column is the same as the order of all occurrences of c in the first column of the table.

Proof (contd):

Now let's look at the *first column*. Since the first row is sorted lexicographically, all instances of c in the first column are adjacent rows in the BWT table. Furthermore, the row beginning with the ith instance of c consists of c concatenated with s_i. But then, the order of the instances is the same as the lexicographic order of the s_i.

So the orders are the same!

Diagram of proof

$	b	a	n	a	n	a
a	$	b	a	n	a	n
a	n	a	$	b	a	n
a	n	a	n	a	$	b
b	a	n	a	n	a	$
n	a	$	b	a	n	a
n	a	n	a	$	b	a

Diagram of proof

\$	b	a	n	a	n	a
a	\$	b	a	n	a	n
a	n	a	\$	b	a	n
a	n	a	n	a	\$	b
b	a	n	a	n	a	\$
n	a	\$	b	a	n	a
n	a	n	a	\$	b	a

Quickly inverting the BWT

Let's start deducing the original string from back to front. Let's use the example do$oodwg.

- What's the last character? What's the second to last character?
- Idea: keep a pointer to the index in the BWT we just wrote. How can we use that to deduce the next index we're writing?
- Rephrase: how can we deduce the next character we're writing? As: how can we deduce what row it is in in the first column ?
    - Let's say we just wrote the ith character in the *first* column of the BWT table
    - Its previous character is the ith character in the *last* column—in other words, the ith character in the BWT

Quickly Inverting the BWT

- When we write a character, goal is: find out where it was in the first column. If we get that we're done
- Idea:
- We just wrote the jth character in BWT; let's say it's character c
- Let's say there are ℓ occurrences of c earlier in the BWT
- Then we're looking for the ℓth c in the first column
- Example: invert e\$elplepa using this method.

Data structures for inversion

How can we quickly answer: “I’m at index j of the BWT; I see character c. How many instances of c are there at indices j and earlier in the BWT?”

- Precompute with a linear scan!
- Keep track of how many of each character seen so far. Write the value for each character of the BWT. Call this array $rank$.

Data structures for inversion (contd.)

How can we quickly answer: “In the first column of the BWT table, in what row does the *i*th occurrence of character *c* occur?”

- The *c*s are all clustered together in the first row. Enough to tell where the grouping begins.
- For each character *c*, keep track of how many characters before *c* (in lexicographic order) occur in the entire string *s*
- Linear time preprocessing: first, get counts of each character in the string. Then, sum successive entries to get the count of all entries before the character. Call this array *C*

Algorithm for Inversion

First, write \$ in the last slot. Find the index of \$ in the BWT; call this `index`. Then, do the following for $n-1$ iterations:

- Find the row r in the BWT whose first column contains the character c we just wrote:
    - Calculate how many instances of this character occur earlier in the BWT; this is $rank[index]$
    - Find the location where we begin writing character c in the first column: this is $C[c]$
    - Therefore, we are looking for $r = rank[index] + C[c]$.
- Prepend BWT[r] to s.
- Update index = r

Our final compression approach

- First, BWT the string using the above (causes characters that appear in similar contexts to be grouped together)
- Then use MTF on the result (causes characters that are grouped together to result in a large number of low-value characters)
- Then use Huffman coding on the result of that (characters that appear often can be written with a very small number of bits).

Let's code this up!

- First let's look at the code
- Then, write a method to perform BWT encoding and decoding
- How well does it help with compression?
    - Test on `chromosome1.txt` from MiniMidterm 2
    - Then test on `proust.txt` from Assignment 4
    - What happens if we don't BWT?
    - Why is this? Print table.

Practical considerations

- This is still pretty slow: why?
- Cache-inefficient! Each encode/decode step requires random array access. On large enough strings this is an L3 miss for EACH characterwe encode/decode
- How can we avoid this?
- In practice: break into decent-sized blocks that fit into L3 cache! BWT each individually