Lecture 20: Burrows-Wheeler Transform

Sam McCauley

November 22, 2024

Williams College

Admin

- Algorithms talk today on Bellman Ford improvement!
- BWT went fast so I’ll probably do Van Emde Boas trees briefly on Monday (last lecture!)
- Office Hours by appointment from now on (I encourage you to do this!)
- Any questions?

BWT Game Plan

To compress a string s:

1. Use BWT to obtain a string $s_b = BWT(s)$. s_b has the property that common subsequences of s correspond to nearby characters of s_b
2. Use MTF to obtain a string $s_m = MTF(s_b)$. s_m has the property that nearby characters of s_b (and therefore common subsequences of s) correspond to common characters in s_m
3. Use Huffman coding on s_m to obtain a final compressed string s_h. Common characters in s_m require few bits to output.

All of the above is reversible, so this is a method for lossless compression.

Let's assume for simplicity that our alphabet has size at most n (easy to generalize).

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 rotated strings lexicographically
- Take the *last character* of each rotation
- This is the BWT of the string
- BWT(`banana`) = `annb$aa`

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 rotated strings
    - $O(n \log n)$ 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
    - This means $O(n^2 \log n)$ 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`
- How does this help us create the BWT?
- Plan: First, we'll assume we have a suffix array and use it to help us calculate the BWT. Then we'll discuss how to efficiently construct it.

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.
- Key insight: the suffixes and the rotated strings are in the same order (because of the \$ character)
- The ith character of the string is the *last column character* corresponding to the ith suffix in sorted order
- The suffix array points to the *first column character*. Since the strings were rotated, just need to go back one character
- So: $\text{BWT}[i] = s[j]$, where $j = SA[i] - 1$. (Watch out for negative indices)
- Linear time method to calculate the BWT!

Where we are

- If you're given a suffix array, can calculate the BWT in a simple linear scan. No extra space (beyond the original string and the suffix array)
- Now: can we decode the BWT quickly as well?
- Let's fill out the BWT in *reverse order*. What characters can we fill in?
    - Last character must be \$ (easy)
    - Previous character is the first character in the row containing \$
- 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

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

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 *i*th character in the *first* column of the BWT table
    - Its previous character is the *i*th character in the *last* column—in other words, the *i*th 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 see how well it works

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

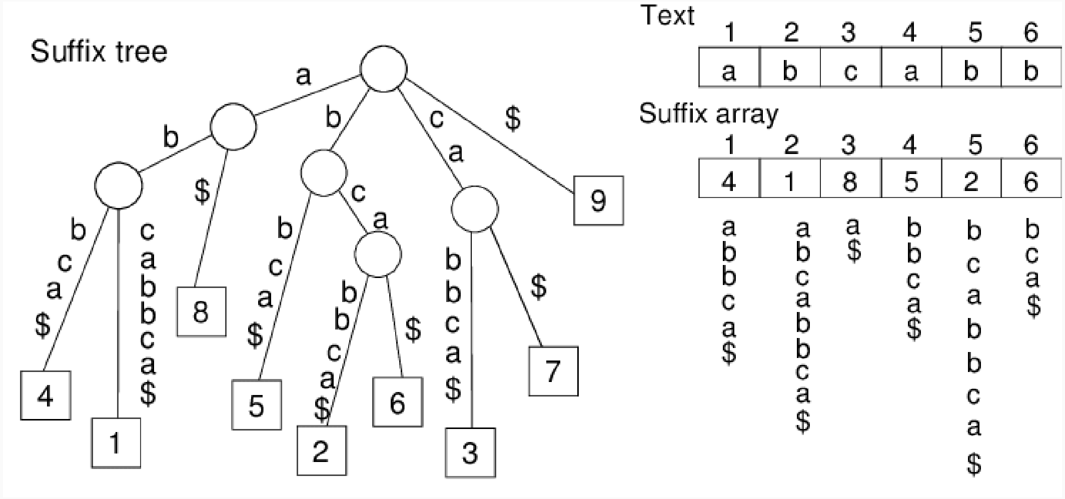
Suffix Array Construction

One piece left

- To build the BWT we need the suffix array
- Recall: array of length n. ith entry holds the *index* of the ith suffix in sorted order
- Example: suffix array for `banana$` is:
- `6 5 3 1 0 4 2`
- How can we construct this?
- Let's assume that our alphabet has size at most n for simplicity (easy to generalize).

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 Construction

- Naive approach: use merge sort (or quick sort, heap sort, whatever) on the suffixes
- Problem: comparing two suffixes may take $\Theta(n)$ time!
- Sorting n suffixes requires $\Theta(n \log n)$ comparisons. Each comparison takes $\Theta(n)$ time, leading to $\Theta(n^2 \log n)$ time overall
- (This method really is quite slow if a lot of your suffixes begin in similar ways.)

Suffix Arrays

- Our goal: $O(n \log n)$ total time
- *Many* algorithms are known that construct a suffix array; I think this one is a good combination of being understandable and efficient

Let’s start sorting the suffixes

- How fast can we sort the first character of all of the suffixes?

- Answer: $O(n)$ time. (Can just count the occurrences of each character.)

Sorting the first two characters

- First: write down all of the two-character prefixes of all suffixes (all two-character pairs in the string)
- Then: sort them using radix sort.
    - We saw radix sort last week. It's $O(wn)$ time if all elements consist of at most w characters, and all characters are in $\{1, \ldots, n\}$.
- Still $O(n)$ time!

Sorting the first four characters?

- Let's say we've sorted all pairs of characters
- How can we use that to sort the first four characters?
- (Yes we can do it in $O(1)$ using radix sort since 4 is a constant. But we're going to keep doubling until we get to n, and we want all rounds to be efficient.)

Sorting the first four characters?

- Idea: Each set of 4 characters consists of 2 pairs of characters. There are at most n of them
- Let's sort all pairs of characters, and then assign each of them a number in $\{1, \ldots, n\}$ based on their sorted order
- Now, for each suffix, replace the first four characters with a pair of numbers; sort these pairs using radix sort
- Let's look at an example

Extending Further

- Now we've sorted all suffixes by their first four characters
- Replace each character by a number in $\{1, \ldots, n\}$, corresponding to its first-four-characters number in sorted order
- Now want to sort by first 8 characters: concatenate the i and $i + 4$th number; then sort these numbers using radix sort and renumber $\{1, \ldots, n\}$
- Keep doing this until entirety of the string is sorted
- Let's do this for `CACATACACAGACACAC$`
- Correct answer:
    - `17 15 13 11 5 7 1 9 3 16 14 12 6 0 8 2 10 4`

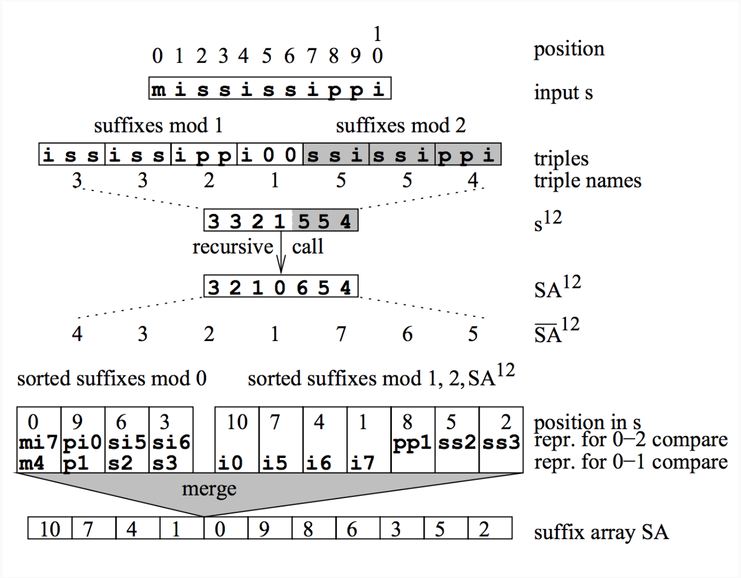
Suffix Array Construction Algorithm

- Sort suffixes by first two characters, then four, then eight, so on until sorting by all $\leq n$ characters
- In each round, use the previous sorted round to create “pairs” of characters
- Sort the pairs, then replace by numbers in order; now we’re ready to recurse

Suffix Array Construction

- This sorts the suffixes in $O(n \log n)$ time; called the “prefix-doubling approach”
- Is this good/bad? Can we do better? What is known?
- Can find the suffix array in $O(n)$ time
- Two ways to do this:
    - Using suffix links, and
    - using recursion

Linear-time Algorithm Diagram



(There's a reason why we're not going to use this algorithm. But, let's go over the

Recursive Suffix Array Construction: Basic Idea [Farach-Colton 97]

- Let’s sort the odd-numbered suffixes recursively. (The base case is when there’s just one odd-numbered suffix, which is trivial to sort.)
- Then: we can assume (recursively) that the odd-numbered suffixes are sorted.
- Use the ordering of the odd-numbered suffixes to build the suffix tree for the even-numbered suffixes

Recursive Suffix Array Construction: Combining

- On the one hand, this seems impossible: the even-numbered suffixes all start with entirely different characters than the odd-numbered suffixes, so the ordering is, right off the bat, 100% different.
- On the other hand, this first character is the *only* difference: once we sort the even-numbered suffixes by their direct character, the odd-numbered suffixes determine the rest of their ordering.
- It's possible in $O(n)$ time with some bookkeeping!

SA-IS Algorithm

- An extremely practical $O(n)$ time suffix array construction algorithm (far faster than version we did)
- I posted a writeup on the website
- Incredibly short, clean, and unintuitive

Suffix Array Uses

Pattern matching

- One of the most ubiquitous string problems involves searching for occurrences of one string in another.
- In particular, let's say you have a large text T. You want to preprocess T (using at most $O(|T|)$ space) so that for any pattern P, you can quickly determine if P is a substring of T.
- Let's say that $|T| = n$ and $|P| = m$. How fast can we solve this?
- First: compute the suffix array on T
- Then: binary search for P in the suffix array

Pattern Matching Analysis

- Binary search requires $O(\log n)$ comparisons
- Each comparison takes $O(m)$ time (may need to walk through whole pattern)
- Gives $O(m \log n)$ time overall
- Can be improved to $O(m)$ time—can search for all occurrences of a pattern in the time it takes to read the pattern!

Other Suffix Array Uses

- Already: Can count the # of occurrences of P in $O(m \log n)$ total time (how?)
    - Also improvable to $O(m)$
- List all locations of P in T
- Useful for calculating BWT
- Find the longest common substring between two strings S_1 and S_2 in $O(S_1 + S_2)$ time
- Find the longest palindrome in S in $O(|S|)$ time
- With more work: searching for P in T with errors