Lecture 19: Burrow-Wheeler Transform

Sam McCauley

November 22, 2021

Williams College

Admin

- All mini-midterms done! Remaining assignments are back to fun collaboration
    - And `C`! (Did you miss it?)

Assignment 7

- No assignment 7
- Plan instead: let's talk about how this algorithm works today
- On Monday we'll code it up "together"
- We'll have one more assignment after Thanksgiving

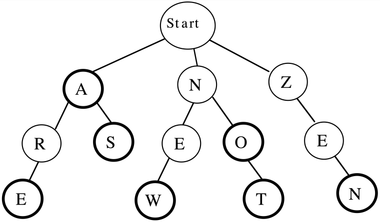
Reflections on course so far

- Part 1: Time vs space
- Part 2: Randomization
- Part 3: LP, ILP, MIP

Part 4: Strings and Trees

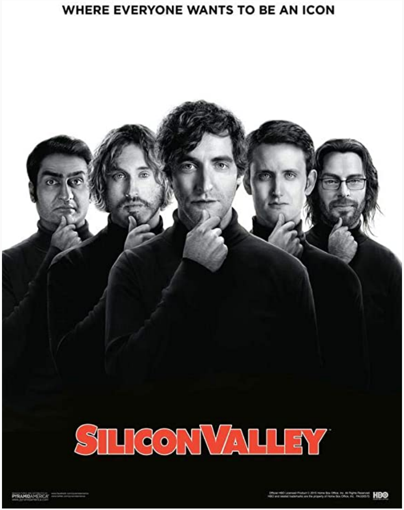
What is this part of the course?

- Previously in this course we've looked at how to *solve problems*
- This section: more about how to *handle data*
- Focus on strings—tons of applications; lots of really cool algorithms research
- Also learn some new aspects of trees



The "Lexicon lab" in 136

Today's topic: compression



- Take data and make it smaller
- Important! (Though sometimes overstated...)
- Nice self-contained topic to start before Thanksgiving

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 $|m| \leq |s|$.

Bad news: lossless compression is not possible

Proof sketch (we show not even possible in expectation):

- Let's say we want to compress all binary strings of length ℓ.
- Each of the 2^ℓ strings of length ℓ must be mapped to some other string
  - A given compressed string m can only be mapped to by one string (otherwise we don't know which of the original strings to recover
- Only one compressed string of length 0, two of length 1, . . . ,
- In general: only $2^{k+1} - 1$ compressed strings of length $\leq k$.
- For all $k < \ell$, must have $2^\ell - 2^{k+1} + 1$ strings that are of length $> k$ when compressed

Bad news: lossless compression is not possible

Proof sketch (contd.):

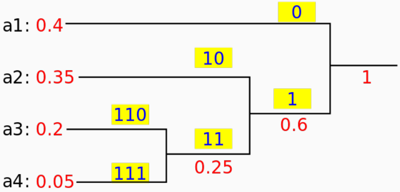
- For all $k < \ell$, must have $\geq 2^\ell - 2^{k+1} + 1$ strings that are of length $> k$ when compressed
- Expected length $= \sum_{i=1}^{2^\ell}$ length of ith string $\cdot \frac{1}{2^\ell}$
- $= \frac{1}{2^\ell} \sum_{i=1}^{2^\ell} \sum_{k=0}^{\infty} [1 \text{ if string } i \text{ has length } \geq k]$
- $\geq \frac{1}{2^\ell} \sum_{k=0}^{\ell-1} \sum_{i=1}^{2^\ell} [1 \text{ if string } i \text{ has length } > k]$
- $\geq \frac{1}{2^\ell} \sum_{k=0}^{\ell-1} 2^\ell - 2^{k+1} + 1 \geq \ell - \sum_{k=0}^{\ell-1} 2^{k+1-\ell}$
- $= \ell - \sum_{i=0}^{\ell-1} \frac{1}{2^i} \geq \ell - 2.$

So: can only save $O(1)$ bits in expectation.

What does this mean?

- Our methods can't guarantee that strings get smaller
- What we'll do instead:
  - Methods that often help on strings we care about!
- We probably don't want to compress an arbitrary string like `afoiewjfiowefjweoifjawepgvheufgahegieg`
  - (Which is good because we can't!)
- Instead, we want to compress strings that look like English text, or DNA, or something like that.
- Goal: compression methods that work well on text we care about

First try: Huffman Coding



- Assign a sequence of bits to each character
- More frequent characters get longer sequences of bits
- Prefix-free: allows us to greedily decode
- Simple, linear-time method to calculate the optimal Huffman code
- Does this help us compress? When?

Huffman Coding

Standard	$Cost(C)$	$Entropy(H)$	$(H - C)$
English	4.152941	4.129020	0.023921
French	4.026765	3.988209	0.038557
German	4.132139	4.096078	0.036062
Spanish	4.041112	4.013136	0.027977

(From "A Comparison of Human codes across languages" by L. Buratto

- Useful when some characters are much more common than others
- English text? Yes! Other languages? Also yes. DNA? ...kind of.
- What compression opportunities in (say) language text is this missing out on?

Key observation for compressing much of text

- Characters are NOT independent!
- *u* after *q* is *extremely* frequent in English. But Huffman codes alone can't capture this.
- DNA (and some kinds of text) may have long sequences of the same letter.
- What can we do about this?
  - Could look at encoding *pairs* of characters. (Treat every pair of consecutive characters as a single character.)
  - Or, could use a fancier method. (Run length encoding? Keep track of common substrings? Some adaptive combination of both?)

Two methods for lossless compression

- Tailor-made methods (like Lempel-Ziv and variants)
- Interesting methods, used frequently in practice
- Tons of research into making these methods efficient, effective

- Other option: try to make Huffman coding work
- How well can this do?

First attempt: Move-to-front transform

(We'll be using this for our compression on Monday)

- Goal: preprocess the string so that long runs (and long close-to-runs) of the same character can be encoded more efficiently.
- Must be invertible (so that we can decode later)
- Does:
  - Improve performance when the same character is close to other occurrences of the same character
  - Perform well when one character is repeated a lot
- Does NOT:
  - Take advantage of relationships between different successive characters
  - Example: u always coming after q is no advantage at all

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
- Let's do a couple examples on the board.

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.

Move-to-front discussion

- Move to front transforms sequences of *nearby* characters into *common* characters
- Plan: to encode a string s, we first calculate $MTF(s)$, and do Huffman coding on that
- To decode, first Huffman decode the string. This gives us $MTF(s)$. Use the above method to recover s
- Can greatly improve Huffman coding performance if characters are close togehter
- In the worst case: does nothing at all. (Could even make performance a good amount worse—when?)

Burrows-Wheeler Transform

Where we are

- Very technical method to take advantage of common substrings/correlations between sequences characters
- OR, move-to-front and Huffman to take advantage of consecutive characters
- What we'd like: a simple, reversible preprocessing method that makes common subsequences into common characters.
- We have MTF: so turning common subsequences into *nearby* characters is enough

Burrows-Wheeler Transform (BWT)

- Invented around 1995
- Turns common subsequences into sequences of nearby characters
- (This is a super weird thing to be able to do. We'll look at a few examples to try to get some intuition about it.)
- Reversible!

BWT Game Plan

To compress a string s:

- 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
- 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
- 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.

- Believe it or not: this method *outperforms* fancier state of the art compression methods in some circumstances
- This is exactly what `bzip2` does.

What does Burrows-Wheeler Transform do?

Let's talk about performing BWT on a string s of length n. Let's assume that s ends with a special character \$ (this will be helpful for us)

- Goal: take the *context* of each character into account
- How many other characters should we look at? 1? 2?
- Silly point: we'll do best if we consider the entire $n-1$ characters surrounding each character
- What does it even mean to take the $n-1$-character context of a string into account?

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

What does this give us?

a $ b a n a n

- For each character of the string: we look at all characters that follow it
- What can we glean from the characters after a given character?
- If a substring appears a lot, it will result in a lot of similar (how?) sequences of n characters
- Example: in English text, almost every q will be followed by a u.
- In banana, almost every a is followed by an n; every n is followed by an a
- Recall: group characters with similar contexts together. So let's sort the characters using the $n - 1$ characters that follow them

First, an observation

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

- The context of a character (the $n - 1$ characters following it) are the contents of the row that the character ends
- So: let's look at the *last* column of this table

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

OK What's going on here?

- This is efficient!?
- This is reversible!?

What we *do* have:

- Characters will wind up next to each other if they are followed by lexicographically similar ($n - 1$-character) strings
- So: if all qs are followed by a u, then EVERY q will wind up in the portion of the BWT corresponding to suffixes beginning with u. Unclear how good this is. . .

One board example

- What is the BWT of dogwood?
- Hopefully we got: do$oodwg
- More interesting example: what if we take the BWT of the first line of `chromosome1.txt`

Next class

- Show that the BWT is:
    - Reversible
    - Efficient