Burrows-Wheeler Transform and FM-Index

CS 7800/4810 Lecture Notes

1 Burrows-Wheeler Transform [BW, 1994]

The Burrows-Wheeler Transform (BWT) is a **reversible permutation** of the characters of a string, originally used for compression.

1.1 Construction

Given a string T, append a special end-of-string character \$, then:

1. Generate all rotations of T
2. Sort the rotations lexicographically to form the **Burrows-Wheeler Matrix** (BWM)
3. The BWT is the last column of the BWM

Example: $T =$ `abaaba$`

All Rotations		**BWM (sorted)**	**L**
`abaaba$`		`$abaaba`	`a`
`baaba$a`		`a$abaab`	`b`
`aaba$ab`	$\xrightarrow{\text{sort}}$	`aaba$ab`	`b`
`aba$aba`		`aba$aba`	`a`
`ba$abaa`		`abaaba$`	`$`
`a$abaab`		`ba$abaa`	`a`
`$abaaba`		`baaba$a`	`a`

Thus, BWT(T) = `abba$aa`.

1.2 Key Questions

- How is it useful for compression?
- How is it reversible?
- How is it an index?

2 Why BWT is Useful for Compression

Key insight: BWT brings like characters together in runs.

$\Rightarrow$ We can use **run-length encoding** to compress BWT(T).

3 Connection to Suffix Arrays

The BWM bears a strong resemblance to the **suffix array**.

	BWM(T)		**SA**(T)
0	`$abaaba`	6	`$`
1	`a$abaab`	5	`a$`
2	`aaba$ab`	2	`aaba$`
3	`aba$aba`	3	`aba$`
4	`abaaba$`	0	`abaaba$`
5	`ba$abaa`	4	`ba$`
6	`baaba$a`	1	`baaba$`

Observation: The sort order is the same whether rows are rotations or suffixes.

3.1 Alternate Way of Constructing BWT(T)

$$\text{BWT}[i] = \begin{cases} T[\text{SA}[i]-1] & \text{if SA}[i] > 0 \\ \$ & \text{if SA}[i] = 0 \end{cases}$$

Intuition: "BWT = characters just to the left of the suffixes in the suffix array."

4 Reversing the BWT

4.1 T-Ranking

Given each character in T, assign a **rank** equal to the number of times the character occurred previously in T.

$$\underset{0}{a}\,\underset{0}{b}\,\underset{1}{a}\,\underset{2}{a}\,\underset{1}{b}\,\underset{3}{a}\,\$ \quad \Rightarrow \quad a_0\, b_0\, a_1\, a_2\, b_1\, a_3\, \$$$

Note: Ranks are not explicitly stored; they are just for illustration.

4.2 First and Last Columns (F and L)

Consider the first column (F) and last column (L) of the BWM with T-ranking:

Row	**F**	**L**
0	\$	a_3
1	a_0	b_0
2	a_1	b_1
3	a_2	a_1
4	a_3	\$
5	b_0	a_2
6	b_1	a_0

4.3 LF-Mapping

Key Property: The i-th occurrence of a character c in L and the i-th occurrence of c in F correspond to the **same occurrence** in T (i.e., they have the same rank).

Equivalently: However we rank occurrences of c, the ranks appear in the same order in F and L.

4.4 B-Ranking

We would like a different ranking so that for a given character, ranks are in **ascending order** as we look down the F/L columns.

Row	**F**	**L**
0	\$	a_0
1	a_0	b_0
2	a_1	b_1
3	a_2	a_1
4	a_3	\$
5	b_0	a_2
6	b_1	a_3

Observation: F has a very simple structure:

- One \$
- A block of a's with ascending ranks
- A block of b's with ascending ranks

Example query: Which BWM row begins with b_1?

- Skip row starting with \$ (1 row)
- Skip rows starting with a (4 rows)
- Skip row starting with b_0 (1 row)
- **Answer: Row 6**

4.5 BWT Reversal Algorithm

Reverse BWT(T) starting at the right-hand side of T and moving left:

1. Start in the first row (F must have \$). L contains the character just prior to \$, which is a_0.
2. LF-mapping says a_0 in L is the same occurrence as the first a in F.
3. Jump to the row beginning with a_0. L contains the character just prior to a_0: b_0.
4. Repeat:
   - $b_0 \rightarrow a_2$
   - $a_2 \rightarrow a_1$

- $a_1 \to b_1$
- $b_1 \to a_3$
- $a_3 \to \$$ (DONE)

Collecting characters: $T = a_3 b_1 a_1 a_2 b_0 a_0 \$ =$ `abaaba$` (original string).

5 FM-Index [Ferragina-Manzini, FOCS 2000]

The FM-Index is an index combining the BWT with a few small auxiliary data structures.

5.1 Core Structure

The core of the index consists of F and L from the BWM:

- **F** can be represented very simply (1 integer per alphabet symbol)
- **L** is compressible

$\Rightarrow$ Potentially very space economical.

5.2 Space Comparison for Human Genome (3 Gbases)

Data Structure	**Size**
Suffix Tree	~ 47 GB
Suffix Array	~ 12 GB
FM-Index	< 1.5 GB

6 FM-Index: Querying

Although BWM is related to the suffix array, we cannot query it the same way.

Note: Binary search isn't possible directly.

6.1 Query Algorithm

To search for pattern P:

1. Look for the range of rows of BWM(T) with P as a prefix
2. Start with P's shortest suffix (last character)
3. Extend to successively longer suffixes until the range becomes empty or we have exhausted P

Example: $P =$ `aba`

Row	**F**	**L**
0	\$	a_0
1	a_0	b_0
2	a_1	b_1
3	a_2	a_1
4	a_3	\$
5	b_0	a_2
6	b_1	a_3

- Rows that begin with `a`: rows 1–4
- Rows that begin with `ba`: rows 5–6
- Rows that begin with `aba`: Look at L column for rows 5–6, check for a's ⇒ **Found the pattern!**

Note: Unlike with a suffix array, we don't immediately know where the matches are in T.

7 FM-Index: Lingering Issues

1. If we scan characters in the last column, that can be very slow: $O(m)$
2. Storing ranks takes too much space
3. Need a way to find where matches occur in T

7.1 Solution to Issue 1: Tally/Checkpoint Tables

Question: Is there an $O(1)$ way to determine which b's precede the a's in our range?
Idea: Pre-calculate the count of each character in L up to every row.

F	**L**	**Tally**	
		a	b
\$	a	1	0
a	b	1	1
a	b	1	2
a	a	2	2
a	\$	2	2
b	a	3	2
b	a	4	2

This gives $O(1)$ time but requires $m \times |\Sigma|$ integers.
Idea: Sparsify the tally (use checkpoints).

7.2 Solution to Issue 2: Checkpoints for Ranks

With checkpoints, we greatly reduce the number of integers needed for ranks.
But it's still $O(m)$ space — there's literature on how to improve this space bound.

7.3 Solution to Issue 3: Sampled Suffix Array

Idea: If the suffix array were part of the index, we could simply look up the offsets.
But SA requires $O(|T|)$ integers.
Idea: Store some, but not all, entries of the suffix array. [**Sparsify**]

8 FM-Index: Small Memory Footprint

8.1 Components of the FM-Index

- **First column (F):** $\sim |\Sigma|$ integers
- **Last column (L):** m characters
- **SA sample:** $m \cdot a$ integers, where a is the fraction of rows kept
- **Checkpoints:** $m \times |\Sigma| \cdot b$ integers, where b is the fraction of rows checkpointed

8.2 Example: Human Genome

DNA alphabet: 2 bits/nucleotide

Parameters: T = Human genome (3 Bi), $a = \frac{1}{32}$, $b = \frac{1}{128}$

Component	**Size**
First column	16 bytes
Last column	2 bits $\times$ 3Bi = 750 MB
SA sample	$\frac{3\text{Bi}\times 4\text{B}}{32} \approx 400$ MB
Checkpoints	$\frac{3\text{Bi}\times 4\text{B}}{128} \approx 100$ MB
Total	< 1.5 GB

9 Application: Infini-gram LLM

Goal: A web-scale search engine for natural language.

9.1 Motivation

Consider the RedPajama dataset: 1.4 trillion tokens.

- A 5-gram count table requires 28 TB
- A fixed n-gram count table discards rich context: the context is fixed up to n

Key insight: We don't need to store counts explicitly — use a suffix array to support unbounded n-gram queries on 5 trillion tokens.

9.2 Suffix Array Approach

- Concatenate all documents in corpus into one giant byte array
- Build a suffix array over this concatenation
- To count n-grams: do binary search to find the range of suffixes starting with that n-gram
- Count = [End − Start] of the range

Performance

- N-gram counting: 20 milliseconds (regardless of n)
- Construction: 2 days, 128 cores
- Space: 10 TB (about 7 bytes/token, i.e., 3.5× overhead)

9.3 The ∞-gram Language Model

Traditional N-gram LM

A traditional n-gram LM estimates the probability of the next token given the previous $n-1$ tokens:

$$P(x_n \mid x_1, \ldots, x_{n-1}) = \frac{\text{count}(x_1, \ldots, x_n)}{\text{count}(x_1, \ldots, x_{n-1})}$$

Problem: Fixed n discards useful context.

∞-gram: Backoff Technique

The ∞-gram LM can support arbitrarily long contexts:

- Start with the full context and back off to shorter contexts until you get a non-zero count
- No fixed upper bound on context length

Results

- 47% accuracy on predicting next token
- Reduces neural LM perplexity by up to 73% when interpolated

9.4 Interpolation with Neural LMs

The ∞-gram produces 0 probabilities for unseen n-grams, leading to infinite perplexity. Solution: interpolate with neural LMs:

$$P_{\text{combined}}(x \mid \text{context}) = \lambda \cdot P_{\infty\text{-gram}} + (1-\lambda) \cdot P_{\text{neural}}$$

9.5 Infini-gram Mini: FM-Index Variant

- Storage overhead drops to only 44% of the text size
- Suffix arrays have 350% overhead in comparison
- Can index 83 TB of Common Crawl data

Tradeoffs

- FM-Index is slower: you need a sampled SA to identify the positions in the original text
- FM-Index does not store original text in a contiguous block, requiring random reads
- They use infini-gram mini for contamination detection

9.6 Major Applications

1. Memorization analysis
2. Reducing hallucination
3. Speculative decoding
4. Data contamination detection
5. Attribution