CS 262 Lecture 4 Notes

The Burrows-Wheeler Transform

Scribe: Robbie Ostrow

January 14, 2016

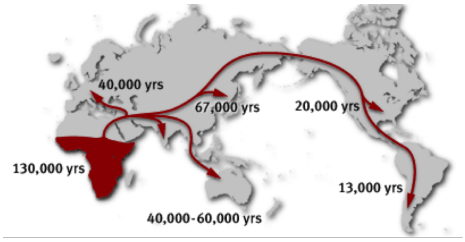
---

1 Human Genome Resequencing

Resequencing is the act of sequencing of an organism's DNA with the aid of a reference genome. This differs from **De Novo Assembly**, which is the method of sequencing some genome without a reference.

Humans have a **polymorphism rate** of $\approx 1/1000$. This is to say that the probability that some arbitrary base pair differs between two humans is approximately 1/1000. The human polymorphism rate is very low, in part because the human population has been very small and concentrated for most of its history. Until about 40,000 years ago, humans were a tribe of 1000 to 10000 individuals in Sub-Saharan Africa (Figure 1). This small, interbreeding population had little genetic variation, and so today the polymorphism rate remains low.

Figure 1: A summary of historical human migration.



Other organisms, especially those with large populations, tend to have much higher polymorphism rates.

Heterozygosity is a measure of the genetic diversity of a population: the probability of a mutation during some generation divided by the probability of any "event" during that generation.

Let μ be the probability of a mutation, and let N be the population size.

Then heterozygosity H can be expressed as

$$H = \frac{4N\mu}{1 + 4N\mu}$$

For most of human history, $\mu \approx 10^{-8}, N \approx 10^4$. $N \gg \mu$ so $H \approx 10^{-4}$.

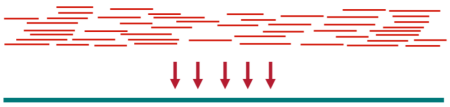
This formulation of heterozygosity assumes an entirely random method of mating – it assumes that arbitrary chromosomes randomly "mate" with one another. Any modification to this assumption that reduces randomness (sexual preference, location, etc.) makes the effective population size smaller and thus the heterozygosity even lower.

2 DNA Sequencing

We are working to solve the problem: given some DNA, find the complete sequence over the alphabet $\Sigma = \{A, C, G, T\}$.

Unfortunately, no machine can sequence an entire genome at once. Instead, we have the capability to sequence about 150 letters at a time. Modern techniques cut many times at random and attempt to reconstruct the genome from these small segments, as in Figure 2.

Figure 2: Assembling many small segments into an accurate whole.



Coverage is a measure of how much of a genome is expected to have been read. Let N be the number of reads, L the length of each read, and G the length of the genomic segment. Then, coverage C is defined as

$$C = \frac{NL}{G}$$

The higher the coverage, the less likely there is some base pair that has not been read. But how much coverage is enough?

The **Lander-Waterman** model assumes a uniform distribution of reads. Under this assumption,

$$P(\text{base pair } i \text{ is not covered}) = e^{-C}$$

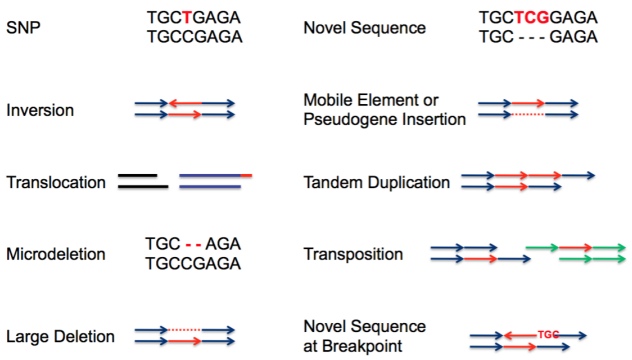
It follows that the expected number of missing base pairs $E[\text{uncovered}] = Ne^{-C}$.

For example, under Lander-Waterman, $C = 10$ results in one gapped region per million nucleotides. In practice, the expected coverage is less than C, because reads are not truly uniformly random. Generally, $C = 30$ is the minimum acceptable for rigorous work. In cancer genomics, which requires an even higher

degree of certainty – looking for a needle in a haystack – researchers usually use $C = 50$ to $C = 100$. That's a lot of data to store. In fact, a single Illumina machine can generate up to 40 terabases per week. As such, we need a faster algorithm than Smith-Watterman to find alignments.

We need fast, similar alignments to detect genomic variation that can be created in several different ways (See Figure 3).

Figure 3: Human genome variation



3 Burrows-Wheeler Transform (BWT)

Note: these notes only discuss the algorithm that finds exact alignments using the BWT. However, inexact matching can be implemented using a similar technique.

The modern fast read aligners BWT, Bowtie, and SOAP are all based on the Burrows-Wheeler transform. This technique allows us to solve a problem that seems intractable:

Problem. *Given strings S and w, where $|S| \gg |w|$, find all occurrences of w in S in $O(|w|)$ time.*

3.1 Defining the BWT

First, we define the Burrows-Wheeler transform.

Definition 1. BWT(X) is a permutation of the letters of X and a stop character \$ constructed as follows:

1. X' = X + \$, where \$ is lexicographically smaller than any letter in the language.
2. Construct the BWT matrix of X to be the list of all possible distinct rotations of X'.

3. Sort the BWT matrix lexicographically.

4. BWT(X) is the last letter of each item in the sorted matrix.

Example 1. We construct BWT(BANANA) using the steps outlined in the definition.

Step 1: append a $:

X' = BANANA$

Step 2: make BWT matrix

BANANA$
$BANANA
A$BANAN
NA$BANA
ANA$BAN
NANA$BA
ANANA$B

Step 3: lexicographically sort

$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA

Step 4: take last letters

$BANAN**A**
A$BANA**N**
ANA$BA**N**
ANANA$**B**
BANANA**$**
NA$BAN**A**
NANA$B**A**

Thus,

BWT(BANANA) = ANNB$AA

3.2 Suffix Arrays

Key idea: Every substring is the prefix of some suffix.

Note that all possible suffixes are sorted in the BWT matrix.

Sorted suffixes in BWT matrix

$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA

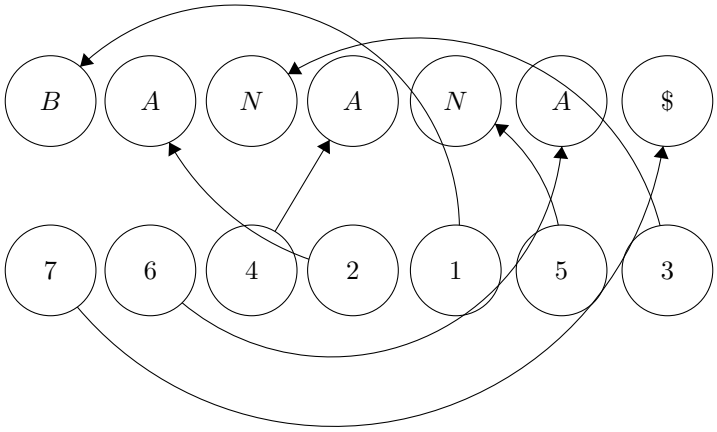
Definition 2. Define suffix array S:

$S(i) = j$, where $X_j \dots X_n$ is the i^{th} suffix lexicographically.

We can construct BWT(X) from the suffix array S. At each position, take the letter to the left of the one pointed to by S.

Example 2. Consider again the word BANANA.

By the definition of the suffix array, we can construct S:



So $S = [7, 6, 4, 2, 1, 5, 3]$.

Now, to construct the BWT, we look at each value of S minus one. (The first index rotates and becomes seven):

$$BWT(X) = [X_6, X_5, X_3, X_1, X_7, X_4, X_2] = [A, N, N, B, \$, A, A]$$

Which is the same as what we derived above from the definition.

3.3 Reconstructing X from BWT(X)

Problem. *Given BWT(X), find X in $O(n)$ time.*

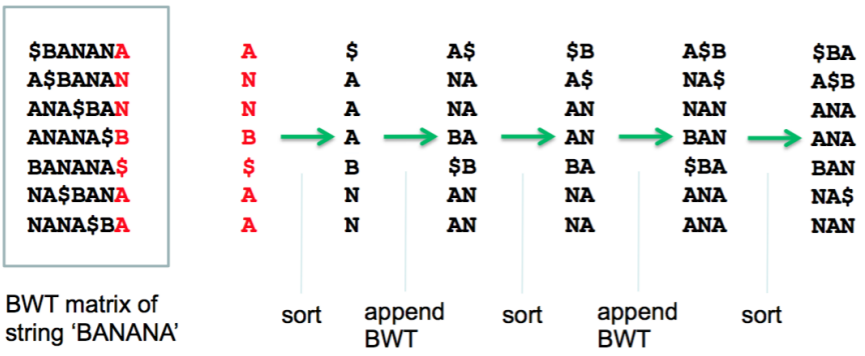
3.3.1 Naive approach

We can reconstruct the original string from its BWT by repeatedly sorting a list of strings then prepending the BWT.

Consider the last column of the sorted BWT matrix of X. Given only this information (which is the BWT(X)), it is easy to reconstruct the first column. Since every column includes every character, and the first column is sorted, we only need to sort the last column to find the first column. Together, the two columns make the first and second columns. Continue using the same technique to find the whole BWT matrix and thus the original input.

Unfortunately, this algorithm runs in $O(n^2 \log n)$ time, where n is the length of the word X, since it has to sort a list of length n, n times in order to reconstruct the whole matrix.

Example 3.



3.3.2 Fast approach

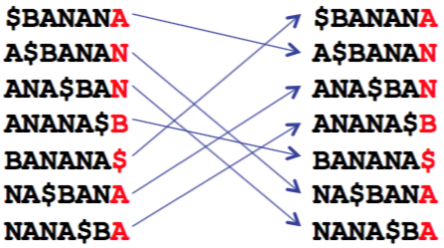
Lemma 1. *The i^{th} occurrence of a character c in the last column is the same text character as the i^{th} occurrence of c in the first column.*

Definition 3. LF() maps the i^{th} occurrence of character c in the last column to the first column. Specifically,

LF(r): Let row r contain the i^{th} occurrence of c in the last column. Then LF(r) = r' where r' is the i^{th} row that starts with c.

Row LF(r) is obtained by rotating row r one position to the right. Therefore, the last character in row LF(r) is the character before the last character in row r.

Example 4. Continuing our use of BANANA, we can compute LF[]:



So,

$$LF[] = [2, 6, 7, 5, 1, 3, 4]$$

Definition 4. Let C(a): number of characters lexicographically smaller than a in the string.

For example, given the word BANANA\$, C(\$) = 0, C(A) = 1, C(B) = 4, and C(N) = 5.

It's easy to see that C can easily be precomputed in $O(n)$ in a single pass of the string. Additionally, storing C only takes $O(a)$ space, where a is the size of the alphabet. (Just store one integer for each letter.)

Let row r end with the i^{th} occurrence of a in the last column. Then

$$LF(r) = C(a) + i$$

These definitions of C and LF allow us to reconstruct X from BWT(X) in $O(n)$ time, using the following algorithm:

```
define RECONSTRUCT(BWT):
    LF = COMPUTE_LF(BWT)
    S := ‘’
    r := 1
    c := BWT[r];
    while c != ‘$’:
        S := cS
        r := LF(r)
        c := BWT[r]
    return S
```

We can compute LF(r) in O(n) time. As discussed above, it is easy to compute C in O(n), and $LF(r) = C(a) + i$, where i represents the occurrence number of a in the last column. i is also simple to precompute with a single pass.

Example 5. For BANANA, we have:

	A	N	N	B	$	A	A	
C()	1	5	5	4	0	1	1	C() copied for convenience
index i	1	1	2	1	1	2	3	indicating this is i-th occurrence of ‘c’
LF()	2	6	7	5	1	3	4	LF() = C() + i

Using this table, we can walk through the RECONSTRUCT algorithm.

```
On input BWT(BANANA) = ANNB$AA:
START
  Loop 0: S = ‘’, r = 1, c = A
  Loop 1: S = A, r = 2, c = N
  Loop 2: S = NA, r = 6, c = A
  Loop 3: S = ANA, r = 3, c = N
  Loop 4: S = NANA, r = 7, c = A
  Loop 5: S = ANANA, r = 4, c = B
  Loop 6: S = BANANA, r = 5, c = $
  Return BANANA
END
```

3.4 Searching for a substring

After all these definitions and funny manipulations of data, we're finally ready to present an algorithm that solves the main problem: finding a substring w in a large string S in $O(|w|)$ time (with some preprocessing).

This is incredibly useful in resequencing, as small DNA samples need to be compared against a reference genome millions of times to find the new genome. The reference genome only has to be preprocessed once, then any searches can be done in $O(|w|)$ time.

Definition 5. Let L(W) be the lowest index in the BWT matrix where W is a prefix.

Similarly, let U(W) be the highest index in the BWT matrix where W is a prefix.

For example, L(NA) = 6 and U(NA) = 7:

L(NA), U(NA) example

$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA

Lemma 2.
Let i be the number of a's up to L(W)-1 in BWT(X).
Let j be the number of a's up to U(W) in BWT(X).
Then L(aW) = C(a) + i + 1 and U(aW) = C(a) + j
(Prove as exercise)

For example,
$L(ANA) = C(A) + i + 1 = 1 + (\#$'A's up to (L(NA) - 1)) $+ 1 = 1 + 1 + 1 = 3$
$U(ANA) = 1 + (\#$ 'A's up to U(NA)) $= 1 + 3 = 4$

Putting it all together, this lemma leads to an algorithm to find the first and last indices of some prefix in the BWT matrix. Knowing these indices, we know exactly where the substring occurs in the original string.

```
Let LFC(r, a) = C(a) + i, where i = # of a's up to r in BWT
define ExactMatch(W[1...k]):
    a := W[k]
    low := C(a) + 1
    high := C(a + 1) // a + 1: lexicographically next char
    i := k - 1
    while (low <= high && i >= 1):
```

```
        a = W[i]
        low = LFC(low - 1, a) + 1
        high = LFC(high, a)
        i := i - 1
    return (low, high)
```

Note that every step can run in constant time (given preprocessing), and i is decremented every iteration of the loop. Thus, `ExactMatch` runs in $O(|W|)$ time.

Example 6. Our final example shows how the `ExactMatch` algorithm would run when searching for the substring ANA in the string BANANA.

```
On input ANA:
START
   Loop 0: a = A, low = C(A) = 1, high = C(B) = 4, i = 2
   Loop 1: a = N, low = LFC(0, N) + 1 = 6, high = LFC(4, N) = 7, i = 1
   Loop 2: a = A, low = LFC(5, A) + 1 = 3, high = LFC(7, A) = 4, i = 0
   Return (3, 4)
END
```

3.5 Notes on the BWT

- Despite all our talk of the BWT matrix, it is merely a tool for analysis. This matrix is never actually stored, as doing so would take $O(n^2)$ space.
- The BWT follows immediately from the suffix array. Suffix array construction is possible in $O(n)$, but in practice most algorithms are $O(n \log n)$.
- We can reconstruct X from BWT(X) in time O(n).
- We can search for all exact occurrences of W in time $O(|W|)$.
- BWT(X) is usually easier to compress than X, which is very important when dealing with terabytes of data. (In fact, the BWT is used some in modern data compression algorithms like bzip2.)

4 Next time

- Inexact matching using a similar technique
- Discuss index construction and memory consumption in more depth
- Heuristics