Genome Assembly

officialprofile

2021-08-31

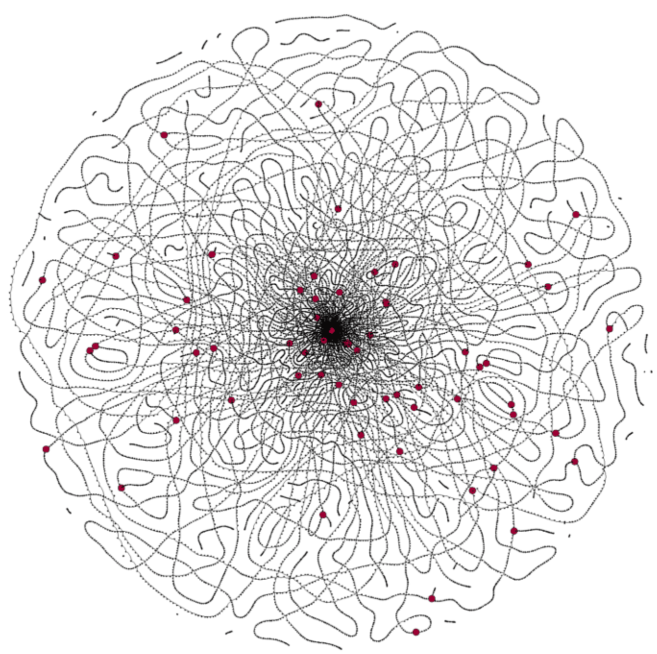
Contents

Chapter 1

Preface

This mini textbook describes selected algorithms that play a vital role in the *de novo* genome assembly or in some related areas. The premise of this book is to construct these algorithms from the very bottom along with brief explanation of their gists. Naturally, the applications are included as well.

By default the code is written in R, but python and shell can appear at some point as well (not very likely though).



A HiFi De Bruijn graph for a pile of reads from Drosophila genome sequencing. Each dot represents a k-mer (k=23), the edges denote neighboring k-mers. The larger red dots mark the head of heterozygous bubbles. Source: pacb.com.

1.1 Prerequisites

It is assumed that the reader:

1. Has a basic understanding of genetics.
2. Has some experience with programming in R (is familiar with pipe syntax, etc.)
3. Had some contact with higher mathematics, e.g. statistics, graph theory.

Throughout the book the following libraries are being used and it is assumed that the reader has them loaded.

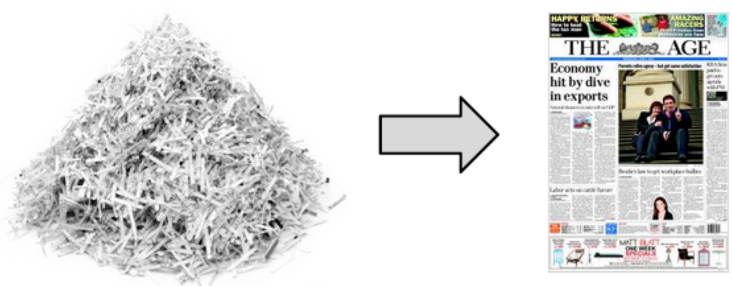
```
library(stringr)
library(dplyr)
```

Chapter 2

Introduction

Genome assembly has been regarded as one of the most important and the most challenging problems in bioinformatics, at least since the late 80's when the Human Genome Project was announced. Although genome tend to be almost inconceivably long, the algorithmic part of the problem arises not mainly because of its length. The core of this problem is located deeper.

Usually, at this point an example of shredded newspaper is being provided. Namely, de novo assembly is very similar to reconstruction of the original document from a set of its unordered fragments.



As one can imagine the problem is difficult, but not because the newspaper has twenty pages instead of just one. The length of the genome, although far from being irrelevant, is more of a second-tier issue. Reader will perhaps come up with the very similar conclusion as the succeeding problems will be uncovered.

One should also underline the fact that transcriptome assembly is not the same as the genome assembly. Read coverage of the latter is relatively uniform, whereas transcriptome can be differentially expressed and therefore frequency of its reads can vary a lot. In genome assembly non-uniform abundance of reads simply indicates existence of repetitions. In the case of RNA it is much less straightforward. Methods that overcome this issue exist but are beyond of the scope of this textbook.

Chapter 3

Burrows-Wheeler transform

The Burrows-Wheeler transform is one of the most effective lossless text compression method available. It provides a reversible transformation for text that makes it easier to compress. Of course, one may wonder what text compression has to do with genome assembly. As a matter of fact these two issues are closely related. But we should to be more precise here - text compression is closely related to pattern matching which in turn is crucial for the genome assembly. In a broad sense compression algorithms look for patterns and try to remove repetitions. We want to take advantage of this feature, especially because repetitive patterns tend to be very abundant in genomic sequences.

It is worth mentioning that the Burrows-Wheeler transform is also closely related to suffix trees and suffix arrays, which are commonly used within pattern matching. This relationship will be studied later but perhaps reader should already keep the trivia in mind. (Donald Adjeroh, 2008)

3.1 Introduction

The Burrows-Wheeler transform method is often referred to as “block sorting”, because it takes a block of text and permutes it. By permuting a block of text we mean rearranging the order of its symbols. Once again, we should be more precise here because Burrows-Wheeler transform performes a specific type of permutation, namely *circural shift permutation*: all of the characters are moved one position to the left, and first character moves to the last position.

3.2 Burrows-Wheeler matrix

Consider the following sequence:

```
sequence <- 'GATTACA'
```

In order to create the Burrows-Wheeler matrix, from which the transform itself can be obtained, for the given string we at first add the dollar sign $ at the end of the sequence.

```
sequence  <- str_c(sequence, '$')
```

Afterwards we perform a series of circular shift permutations.

```
sequences <- c(sequence)
n         <- nchar(sequence)

for (i in 1:(n-1)){
  sequence <- str_c(str_sub(sequence, 2, n),
                    str_sub(sequence, 1, 1))

  sequences <- c(sequences, sequence)
}

cat(sequences, sep = '\n')
#> GATTACA$
#> ATTACA$G
#> TTACA$GA
#> TACA$GAT
#> ACA$GATT
#> CA$GATTA
#> A$GATTAC
#> $GATTACA
```

Then we sort these sequences with the assumption that the dollar sign precedes lexicographically every other symbol.

```
sequences <- sort(sequences)
cat(sequences, sep = '\n')
#> $GATTACA
#> A$GATTAC
#> ACA$GATT
#> ATTACA$G
#> CA$GATTA
#> GATTACA$
#> TACA$GAT
#> TTACA$GA
```

For our convenience let's split these permutations into vectors of single characters.

```
bw.matrix           <- data.frame(matrix(, n, n))
colnames(bw.matrix) <- 1:n

for (i in 1:n){
  bw.matrix[i, ] <- strsplit(sequences[i], split = '')[[1]]
}

knitr::kable(bw.matrix)
```

1	2	3	4	5	6	7	8
$	G	A	T	T	A	C	A
A	$	G	A	T	T	A	C
A	C	A	$	G	A	T	T
A	T	T	A	C	A	$	G
C	A	$	G	A	T	T	A
G	A	T	T	A	C	A	$
T	A	C	A	$	G	A	T
T	T	A	C	A	$	G	A

Thus we have created the **Burrows-Wheeler matrix**. Sequence in the last column is called the **Burrows-Wheeler transform**.

```
transform <- paste(bw.matrix[,n], collapse = '')

cat('The Burrows-Wheeler transform of',
    sequence, 'is', transform)
#> The Burrows-Wheeler transform of $GATTACA is ACTGA$TA
```

3.3 Inverse transform

As we said at the very beginning the transform is reversible. Having only the transformed sequence we are going to reconstruct the Burrows-Wheeler matrix and initial sequence itself.

Firstly let's sort the characters of the transformed sequence.

```
first.sequence <- strsplit(transform, split = '')[[1]] %>% sort
paste(first.sequence, collapse = '')
#> [1] "$AAACGTT"
```

Note that this string is equivalent to the first column of the Burrrows-Wheeler transform.

```r
bw.inverse           <- data.frame(matrix(, n, 2))
colnames(bw.inverse) <- c(n, 1)

bw.inverse[, 1] <- strsplit(transform, split = '')[[1]]
bw.inverse[ ,2] <- first.sequence

knitr::kable(bw.inverse)
```

8	1
A	$
C	A
T	A
G	A
A	C
$	G
T	T
A	T

Also keep in mind that the characters from last and the first column are adjacent. In other words, at this point we have a set of 2-mers.

```r
kmers <- apply(bw.inverse, 1,
               function(x) paste(x, collapse = ''))
kmers
#> [1] "A$" "CA" "TA" "GA" "AC" "$G" "TT" "AT"
```

The reconstruction process strictly relies on the fact that Burrows-Wheeler matrix is sorted lexicographically. This property will allow us to retrieve the remaining columns.

```r
kmers <- sort(kmers)
kmers
#> [1] "$G" "A$" "AC" "AT" "CA" "GA" "TA" "TT"
```

The 2-mers (k-mers in general) that we sorted lexicographically represent first two columns of the Burrows-Wheeler matrix. We can extract last character of each 2-mer in the following way:

```r
sapply(kmers, function(x) str_sub(x, 2, 2),
       simplify = TRUE, USE.NAMES = FALSE)
#> [1] "G" "$" "C" "T" "A" "A" "A" "T"
```

By inserting this set of characters we obtained the second column, and by iterating the proccess of building substrings, sorting them, and retrieving last characters we can fill the whole Burrows-Wheeler matrix.

```r
for (i in 2:(n-1)){
  kmers              <- apply(bw.inverse, 1,
                              function(x) paste(x, collapse = ''))
  kmers              <- sort(kmers)
  bw.inverse[, i+1] <- sapply(kmers, function(x) str_sub(x, i, i),
                               simplify = TRUE, USE.NAMES = FALSE)
  colnames(bw.inverse)[i+1] = i
}
knitr::kable(bw.inverse)
```

8	1	2	3	4	5	6	7
A	$	G	A	T	T	A	C
C	A	$	G	A	T	T	A
T	A	C	A	$	G	A	T
G	A	T	T	A	C	A	$
A	C	A	$	G	A	T	T
$	G	A	T	T	A	C	A
T	T	A	C	A	$	G	A
A	T	T	A	C	A	$	G

Finally we move first column to the very end

```r
bw.inverse[,n+1] <- bw.inverse[, 1]
bw.inverse       <- bw.inverse[,2:(n+1)]
colnames(bw.inverse)[n] = n

knitr::kable(bw.inverse)
```

1	2	3	4	5	6	7	8
$	G	A	T	T	A	C	A
A	$	G	A	T	T	A	C
A	C	A	$	G	A	T	T
A	T	T	A	C	A	$	G
C	A	$	G	A	T	T	A
G	A	T	T	A	C	A	$
T	A	C	A	$	G	A	T
T	T	A	C	A	$	G	A

One can also verify that bw.matrix and bw.inverse are in fact the same.

```r
knitr::kable(bw.inverse == bw.matrix)
```

1	2	3	4	5	6	7	8
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE

Additionally we can encapsulate the Burrows-Wheeler transform in a form of a single function.

```
BWT <- function(sequence){
  sequence  <- str_c(sequence, '$')
  sequences <- c(sequence)
  n         <- nchar(sequence)

  for (i in 1:(n-1)){
    sequence <- str_c(str_sub(sequence, 2, n),
                      str_sub(sequence, 1, 1))
    sequences <- c(sequences, sequence)
  }
  sequences <- sort(sequences)

  bw.matrix           <- data.frame(matrix(, n, n))
  colnames(bw.matrix) <- 1:n

  for (i in 1:n){
    bw.matrix[i, ] <- strsplit(sequences[i], split = '')[[1]]
  }
  return(paste(bw.matrix[,n], collapse = ''))
}
```

```
BWT('GATTACA')
#> [1] "ACTGA$TA"
```

One can verify that this output is equal to result we obtained earlier.

Out of pure curiosity lets check the Burrows-Wheeler transform for a longer sequence.

```
BWT('ATGCTCGTGCCATCATATAGCGCGCGCGCGATCTCTACGCGCG')
#> [1] "GTTTCCG$TCGGGGGAGGGTTGTCCTCCCCCCATCCAAACCAGA"
```

Please note that the input string has no identical characters at adjacent positions, whereas in the transformed sequence such situation appears quite often.

These substrings of identical characters will allow us represent the sequence in a more condensed manner and expediate pattern matching.

Bibliography

Donald Adjeroh, Tim Bell, A. M. (2008). *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer.