The Burrows-Wheeler Transform with applications to bioinformatics

Anselm Blumer
9 March, 2020
ablumer@cs.tufts.edu

Introduction to the Burrows-Wheeler Transform

- The Burrows-Wheeler Transform (BWT) was developed in the early 1990s by Michael Burrows at the Digital Equipment Corporation Systems Research Center in Palo Alto based on earlier work by David J. Wheeler

- Burrows used it as part of a data compression system but it has also been applied to sequence alignment in genomics

-

-

Introduction to the Burrows-Wheeler Transform

- The Burrows-Wheeler Transform (BWT) was developed in the early 1990s by Michael Burrows at the Digital Equipment Corporation Systems Research Center in Palo Alto based on earlier work by David J. Wheeler
- Burrows used it as part of a data compression system but it has also been applied to sequence alignment in genomics
- The input string can be specified using an end-of-string character ($) or as a (string, length) pair. These slides use the former method.
-

Introduction to the Burrows-Wheeler Transform

- The Burrows-Wheeler Transform (BWT) was developed in the early 1990s by Michael Burrows at the Digital Equipment Corporation Systems Research Center in Palo Alto based on earlier work by David J. Wheeler

- Burrows used it as part of a data compression system but it has also been applied to sequence alignment in genomics

- The input string can be specified using an end-of-string character ($) or as a (string, length) pair. These slides use the former method.

- The BWT transforms the input string by permuting its characters in such a way that the inverse transform can recover the original string without using any information aside from the permuted string (including the new location of the $)

High-level description of the Burrows-Wheeler Transform

- The BWT is easy to describe at a high level
- The inverse transform is a bit more complicated
-
-

High-level description of the Burrows-Wheeler Transform

- The BWT is easy to describe at a high level
- The inverse transform is a bit more complicated
- Both are based on two operations, sorting and left-rotations (moving the first character of the string to the end) or right-rotations
-

High-level description of the Burrows-Wheeler Transform

- The BWT is easy to describe at a high level
- The inverse transform is a bit more complicated
- Both are based on two operations, sorting and left-rotations (moving the first character of the string to the end) or right-rotations
- Efficient implementations require more sophisticated data structures and algorithms

High-level pseudocode for the Burrows-Wheeler Transform

```
• BWT( inString ) returns outString
    tempString[0] = inString
    for i = 1  to  inString.Length - 1
        tempString[i] = LeftRotate( tempString[i-1] )
    sort the tempString array lexicographically
    outString = the last characters of each tempString
```

High-level pseudocode for the inverse Burrows-Wheeler Transform

- iBWT(inString) returns outString

```
iBWT( inString ) returns outString
   n = inString.length
   allocate X[1..n,1..n], an array of $ characters
   for i = 1  to  inString.Length
        insert inString into the first column of X
        sort the rows of X lexicographically
        right-rotate the rows of X
   outString = the row that ends in $
```

Example of the Burrows-Wheeler Transform

- Starting from the input string $X_0 = X = a_0a_1a_2...a_{n-2}\$$, create n-1 new strings by successive left rotations, so $X_1 = a_1a_2...a_{n-2}\$a_0$, $X_2 = a_2...a_{n-2}\$a_0a_1$ …
- For example, X_0 = AATA\$, X_1 = ATA\$A, X_2 = TA\$AA, X_3 = A\$AAT, X_4 = \$AATA,
-
-
-

Example of the Burrows-Wheeler Transform

- Starting from the input string $X_0 = X = a_0a_1a_2...a_{n-2}\$$, create n-1 new strings by successive left rotations, so $X_1 = a_1a_2...a_{n-2}\$a_0$, $X_2 = a_2...a_{n-2}\$a_0a_1$ …
- For example, X_0 = AATA\$, X_1 = ATA\$A, X_2 = TA\$AA, X_3 = A\$AAT, X_4 = \$AATA,
- Sort these n strings lexicographically, but keep track of their names:
  X_4 = \$AATA, X_3 = A\$AAT, X_0 = AATA\$, X_1 = ATA\$A, X_2 = TA\$AA
-
-

Example of the Burrows-Wheeler Transform

- Starting from the input string $X_0 = X = a_0a_1a_2\ldots a_{n-2}\$$, create n-1 new strings by successive left rotations, so $X_1 = a_1a_2\ldots a_{n-2}\$a_0$, $X_2 = a_2\ldots a_{n-2}\$a_0a_1$ …

- For example, X_0 = AATA\$, X_1 = ATA\$A, X_2 = TA\$AA, X_3 = A\$AAT, X_4 = \$AATA,

- Sort these n strings lexicographically, but keep track of their names:
  X_4 = \$AATA, X_3 = A\$AAT, X_0 = AATA\$, X_1 = ATA\$A, X_2 = TA\$AA

- The transformed Burrows-Wheeler string is formed by concatenating the last characters of the sorted strings: B = AT\$AA

-

Example of the Burrows-Wheeler Transform

- Starting from the input string X_0 = X = $a_0a_1a_2 \ldots a_{n-2}\$$, create n-1 new strings by successive left rotations, so $X_1 = a_1a_2 \ldots a_{n-2}\$a_0$, $X_2 = a_2 \ldots a_{n-2}\$a_0a_1$ …

- For example, X_0 = AATA\$, X_1 = ATA\$A, X_2 = TA\$AA, X_3 = A\$AAT, X_4 = \$AATA,

- Sort these n strings lexicographically, but keep track of their names:
  X_4 = \$AATA, X_3 = A\$AAT, X_0 = AATA\$, X_1 = ATA\$A, X_2 = TA\$AA

- The transformed Burrows-Wheeler string is formed by concatenating the last characters of the sorted strings: B = AT\$AA

- The *suffix array* (to be used later) is the array of names (indices) of the rotated strings in sorted order: S = [4,3,0,1,2]

Pictorial example of the Burrows-Wheeler Transform

- Input string: X = X_0 = AATA$ with length *n* = 5

- Array containing rotated strings and names:

A	A	T	A	$	X_0
A	T	A	$	A	X_1
T	A	$	A	A	X_2
A	$	A	A	T	X_3
$	A	A	T	A	X_4

- Array of sorted strings:

$	A	A	T	A	X_4
A	$	A	A	T	X_3
A	A	T	A	$	X_0
A	T	A	$	A	X_1
T	A	$	A	A	X_2

- Output string: B = AT$AA (from second-to last column of array)

- Suffix array: S = [4,3,0,1,2] (from subscripts in the last column)

Example of the Inverse Burrows-Wheeler Transform

- Given the output of the Burrows-Wheeler transform B = AT$AA it is possible to reconstruct the input

- B gives the last characters of the sorted strings, so
  $X_?$ = - - - - A, $X_?$ = - - - - T, $X_?$ = - - - - \$, $X_?$ = - - - - A, $X_?$ = - - - - A

-

-

-

-

Example of the Inverse Burrows-Wheeler Transform

- Given the output of the Burrows-Wheeler transform B = AT$AA it is possible to reconstruct the input

- B gives the last characters of the sorted strings, so
  $X_?$ = - - - - A, $X_?$ = - - - - T, $X_?$ = - - - - \$, $X_?$ = - - - - A, $X_?$ = - - - - A

- The strings are in sorted order, so the first characters must be   \$ A A A T :
  $X_?$ = \$ - - - A, $X_?$ = A - - - T, $X_?$ = A - - - \$, $X_?$ = A - - - A, $X_?$ = T - - - A

-

-

-

Example of the Inverse Burrows-Wheeler Transform

- Given the output of the Burrows-Wheeler transform B = AT$AA it is possible to reconstruct the input

- B gives the last characters of the sorted strings, so
  $X_?$ = - - - - A, $X_?$ = - - - - T, $X_?$ = - - - - \$, $X_?$ = - - - - A, $X_?$ = - - - - A

- The strings are in sorted order, so the first characters must be   \$ A A A T :
  $X_?$ = \$ - - - A, $X_?$ = A - - - T, $X_?$ = A - - - \$, $X_?$ = A - - - A, $X_?$ = T - - - A

- The strings were all formed from left rotations of the input string, so right-rotating the first string above shows that the \$ must have been preceded by an A

•

•

Example of the Inverse Burrows-Wheeler Transform

- Given the output of the Burrows-Wheeler transform B = AT\$AA it is possible to reconstruct the input
- B gives the last characters of the sorted strings, so
  $X_?$ = - - - - A, $X_?$ = - - - - T, $X_?$ = - - - - \$, $X_?$ = - - - - A, $X_?$ = - - - - A
- The strings are in sorted order, so the first characters must be \$ A A A T :
  $X_?$ = \$ - - - A, $X_?$ = A - - - T, $X_?$ = A - - - \$, $X_?$ = A - - - A, $X_?$ = T - - - A
- The strings were all formed from left rotations of the input string, so right-rotating the first string above shows that the \$ must have been preceded by an A
- Right-rotating the next three strings shows that the A's were preceded by T, \$, and A
  and right-rotating the last string shows that the T was preceded by an A:
  $X_?$ = \$ - - - A, $X_?$ = A - - A T, $X_?$ = A - - A \$, $X_?$ = A - - - A, $X_?$ = T - - - A
-

Example of the Inverse Burrows-Wheeler Transform

- Given the output of the Burrows-Wheeler transform B = AT$AA it is possible to reconstruct the input
- B gives the last characters of the sorted strings, so
  $X_?$ = - - - - A, $X_?$ = - - - - T, $X_?$ = - - - - \$, $X_?$ = - - - - A, $X_?$ = - - - - A
- The strings are in sorted order, so the first characters must be   \$ A A A T :
  $X_?$ = \$ - - - A, $X_?$ = A - - - T, $X_?$ = A - - - \$, $X_?$ = A - - - A, $X_?$ = T - - - A
- The strings were all formed from left rotations of the input string, so right-rotating the first string above shows that the \$ must have been preceded by an A
- Right-rotating the next three strings shows that the A's were preceded by T, \$, and A
  and right-rotating the last string shows that the T was preceded by an A:
  $X_?$ = \$ - - - A, $X_?$ = A - - A T, $X_?$ = A - - A \$, $X_?$ = A - - - A, $X_?$ = T - - - A
- The original string was either AATA\$ or ATAA\$, but the second of these is impossible, since A\$ATA comes before AA\$AT in sorted order, contradicting the A - - A T

The Burrows-Wheeler Transform and substring matching

- Since every substring of X occurs as a prefix of at least one of the rotated strings, the Burrows-Wheeler Transform algorithm can be used as part of a string matching algorithm

- The BWT algorithm produces the suffix array, giving the sorted order of the rotated strings. In other words, S[*i*] is the start position of the *i*th smallest rotated string

- If W is any substring of X, it will occur as a prefix of successive rotated strings, so define:

  $R_{\min}(W)$ = the smallest index *i* where W is a prefix of $X_{S[i]}$
  $R_{\max}(W)$ = the largest index *i* where W is a prefix of $X_{S[i]}$

- Thus the set of all occurrences of W in X is given by $\{ S[k] \mid R_{\min}(W) \leq k \leq R_{\max}(W) \}$

A recursive calculation for R_{min} and R_{max}

- $R_{min}()$ and $R_{max}()$ can be calculated recursively, as follows
- $R_{min}(\text{empty string}) = 0 \quad R_{max}(\text{empty string}) = n - 1$
- If a is any character, then

  $R_{min}(a\mathrm{W}) = \mathrm{C}(a) + \mathrm{Occ}(a, R_{min}(\mathrm{W}) - 1) + 1$
  $R_{max}(a\mathrm{W}) = \mathrm{C}(a) + \mathrm{Occ}(a, R_{max}(\mathrm{W}))$

  where C(a) is the number of positions in X occupied by characters that come lexicographically before a, and Occ(a, i) is the number of occurrences of a in positions 0 through i of the BWT output string

- Rationale: C(a) counts the number of strings that come before any string beginning with a
  Occ(a, R_{max}(W)) counts the number of strings beginning aW
  Occ(a, R_{min}(W) - 1) counts the number of strings beginning with a that come before any string beginning with aW

P Ferragina and G Manzini, Opportunistic data structures with applications (FOCS 2000)

Inexact search

- The search procedure outlined on the previous slides finds all *exact* occurrences of query string W in target string X, but applications to sequence alignment require the ability for *inexact* matches

- There are three ways a match can be inexact:
  mismatch: a character in W corresponds to a different character in X
  insertion: a character would need to be inserted into X to match W
  deletion: a character would need to be deleted from X to match W

- Examples using X = ACCTCGG
  W = CAT matches the second position of X with one mismatch
  W = CAT matches positions 1 through 5 of X with three mismatches
  W = CAT matches the third position of X with one insertion
  W = CAT matches positions 1 through 5 of X with two insertions
  W = TGG matches the fourth position of X with one deletion
  W = CAT doesn’t match any position of X with any number of deletions

Inexact search with mismatches

```
inexactM( X, W, i, z, k, l) returns set of intervals
    i: position in query string, initially right end
    z: number of mismatches allowed
    [k, l]:  [$R_{min}$, $R_{max}$] interval
if (i < 0) return { [k, l] }
setVal = empty;
ksave = k; lsave = l;
for each b in {A, C, G, T}
    k = ksave; l = lsave;
    k = C(b) + Occ(b, k-1) + 1;
    l = C(b) + Occ(b, l)
    if (k <= l) then
        if (b = W[i]) setVal = setVal ∪ inexactM(X, W, i-1, z, k, l)
        else setVal = setVal ∪ inexactM(X, W, i-1, z-1, k, l)
return setVal
```

Speeding up the search using a lower bound

- The search algorithm on the previous slide is correct, but not efficient as it explores all possible mismatches. Parts of the search tree can be pruned off using a lower bound array, D, where D(i) is a bound on the number of mismatches possible at the ith position of W when searching in X.

- Conceptually, D can be calculated by the following inefficient procedure:

-
```
calculateD( W, X ) returns array of int
    z = 0; j = 0;
    for i=0 to W.length()-1
        if (W[j, i] is not a substring of X) then
            z = z+1;  j = i+1;
        D[i] = z
```

- This can be made more efficient by replacing the search implied by the “if” statement with a Burrows-Wheeler interval calculation for the reverse of X

Rationale for the lower bound

- D(0) is either 0 or 1. A 1 indicates that the character W[0] does not occur in X, so W cannot occur in X with zero modifications.

- D(1) is either 0, 1, or 2. A 2 indicates that W[0] does not occur in X and neither does the substring W[1]W[0] (since calculateD() was called using the reverse of X). This means that at least two modifications are required to match W with a substring of X. A 1 indicates that either W[0] does not occur or W[1]W[0] does not occur, so at least one modification is required.

- Generalizing this, if z is less than D(i) at a call to inexactM() then there is no chance for a match, so this branch of the search can be pruned.

Inexact search with mismatches, insertions, and deletions, using the lower bound

```
inexactM( X, W, i, z, k, l) returns set of intervals
    i: position in query string, initially right end
    z: number of mismatches allowed
    [k, l]:  $[R_{min}, R_{max}]$ interval
$\underline{\text{if (z < D(i) ) return;}}$
if (i < 0) return { [k, l] }
setVal = $\underline{\text{inexactM( X, W, i-1, z-1, k, l)}}$;
ksave = k; lsave = l;
for each b in {A, C, G, T}
    k = ksave; l = lsave;
    k = C(b) + Occ(b, k-1) + 1;
    l = C(b) + Occ(b, l)
    if (k <= l) then
        $\underline{\text{setVal = setVal } \cup \text{ inexactM(X, W, i, z-1, k, l)}}$
        if (b = W[i]) setVal = setVal ∪ inexactM(X, W, i-1, z, k, l)
        else setVal = setVal ∪ inexactM(X, W, i-1, z-1, k, l)
return setVal
```

Efficient calculation of the lower bound

- The following procedure gives a more efficient calculation of the lower bound array D Note that it calls OccRev(), which is defined in terms of the Burrows-Wheeler transformation for the reverse of X

- calculateD(W, X) returns array of int

```
calculateD( W, X ) returns array of int
    z = 0; k = 0; l = X.length()-1;
    for i=0 to W.length()-1
        k = C( W[i] ) + OccRev( W[i], k-1 ) + 1;
        l = C( W[i] ) + OccRev( W[i], l );
        if (k > l) then
            z = z+1; k = 0;  l = X.length()-1;
        D[i] = z
```

References

- M Burrows and DJ Wheeler (1994) A block-sorting lossless data compression algorithm, *Technical report 124*, Palo Alto, CA, Digital Equipment Corporation

- Heng Li and Richard Durbin (2009) Fast and accurate short read alignment with Burrows-Wheeler transform, *Bioinformatics*, 25 (14) pp. 1754-60.

- Heng Li and Richard Durbin (2010) Fast and accurate long read alignment with Burrows-Wheeler transform, *Bioinformatics*, 26 (5) pp. 589-95.

- Dan Gusfield (1997) *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press

- Wikipedia

- There's a nice tutorial at http://blog.thegrandlocus.com/2016/07/a-tutorial-on-burrows-wheeler-indexing-methods

Notation

- $\sum$ - a lexicographically ordered alphabet (*e.g.* {A, C, G, T})
- $\$ \notin \sum$ - end-of-string symbol, lexicographically before all symbols in $\sum$
- $X = a_0a_1a_2...a_{n-2}\$$ - input string with end-of-string symbol
- $X[i] = a_i$ - the symbol at position *i* of the input string
- $X[i, j] = a_i\ a_{i+1}\ ...\ a_j$ - the substring from positions *i* through *j*
- $X_i = X[i, n-1]$ - the *i*th suffix of the input string