`bwt`, v. 0.3: Compute the Burrows-Wheeler Transform

Bernhard Haubold
Max-Planck-Institute for Evolutionary Biology, Plön, Germany

November 6, 2018

1 Introduction

The Burrows-Wheeler Transform (BWT) [3] is an integral part of many compression algorithms, including the widely used program `bzip2` for compacting files. In Bioinformatics the BWT underlies a number of highly memory-efficient tools, including `bwa` [7] and `bowtie` [6].

My program `bwt` demonstrates the encoding and the decoding phase of the BWT. I had two aims when writing it: (i) to demonstrate the BWT when teaching Bioinformatics, and (ii) to explore the properties of the BWT when applied to real-world data sets ranging from Shakespeare's *Hamlet* to bacterial genomes and proteomes.

In the next Section I explain how to get started with `bwt`. This is followed by a tutorial-style exposition of the central ideas behind the BWT as reflected in the features of `bwt`. My approach is heavily indebted to [1], which ought to be consulted for more details on the BWT and its relationship to other string-centered data structures, especially suffix trees and suffix arrays.

2 Getting Started

`bwt` was written in C on a computer running Linux and should work on any standard UNIX system. However, please contact me at `haubold@evolbio.mpg.de` if you have any problems with the program.

- Unpack the program

  ```
  tar -xvzf bwt_XXX.tgz
  ```

  where `XXX` indicates the version.
- Change into the newly created directory

  ```
  cd Bwt_XXX
  ```

  and list its contents

  ```
  ls
  ```
- Generate `bwt`

  ```
  make
  ```
- List its options

  ```
  ./bwt -h
  ```
- `bwt` takes FASTA-formatted input, for example

  ```
  ./bwt Data/mississippi.fasta
  ```

3 Tutorial

The BWT permutes a string in a way that makes it easier to compress. Crucially, this permutation is fully reversible and hence this Tutorial is divided into an encoding and a decoding part.

3.1 Encoding

`bwt` runs in two modes, standard and demo. To learn about the BWT, we start in demo mode and apply it to the favorite word in stringology, `mississippi`:

```
./bwt -D Data/mississippi.fasta
                Rotations:      Sorted Rotations:
                mississippi$    $mississippi
                ississippi$m    i$mississipp
                ssissippi$mi    ippi$mississ
                sissippi$mis    issippi$miss
Text:           issippi$miss    ississippi$m    Transform:
mississippi$    ssippi$missi    mississippi$    ipssm$pissii
                sippi$missis    pi$mississip
                ippi$mississ    ppi$mississi
                ppi$mississi    sippi$missis
                pi$mississip    sissippi$mis
                i$mississipp    ssippi$missi
                $mississippi    ssissippi$mi
```

The output shows how the BWT of a string is obtained: first, all rotations of the input are written down. Next, these rotations are sorted alphabetically. The final transform, `ipssm$pissii`, is the last column of these sorted rotations. The sentinel character `$` at the end of the input will be helpful later on for decoding the transform. But before we get to that, you might wonder how the BWT facilitates compression.

To see this, we need to leave the demo mode and transform a larger text, for example Shakespeare's *Hamlet*:

```
./bwt Data/hamlet.fasta | less
```

Press `d` to go down one page in `less`, and `u` to go up one page; to quit press `q`. As you page through the transform, notice long runs of identical characters. We focus on one such region:

```
./bwt -p 91554-92113 Data/hamlet.fasta
>hamlet - bwt
nnnnnnnngnnnnnnnnnnnnnnnrnnnrennnnnnnnnnnennnnnnnuonnnnnnnnnnnnnnnnnnn
nnnnnnnnnnnnnnnnnnnninnnnnininngnnnnnnnnnrnnnnnnnnnnnnnnannnnnnnnnnnnn
nnnnnnnnnnnnnnnnnnnnnnnnnnnnrnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
nnnnnnnnnennnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
nnnnneninnnnnnnennnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnen
nnnnnnnnnnninnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnennunnnnn
nnnnnnnnnnnnannnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnonnnnnnnninnn
```

This segment illustrates the central point of the BWT: since particular suffixes are more likely to be preceded by the same characters, the BWT tends to generate runs of identical characters. As we will see later, this is not necessarily the case, but happens to be true for natural languages and many other types of discrete sequential data. Runs of identical characters can be compressed and a naïve way to do this would be to replace them by a single character followed by its count:

```
./bwt -r -p 91554-92113 Data/hamlet.fasta
>hamlet - bwt
n8gn15rn3ren11en7uon39in5inin2gn9rn14an41rn50enenin7en52en12in49en2un17
an44on8in3
```

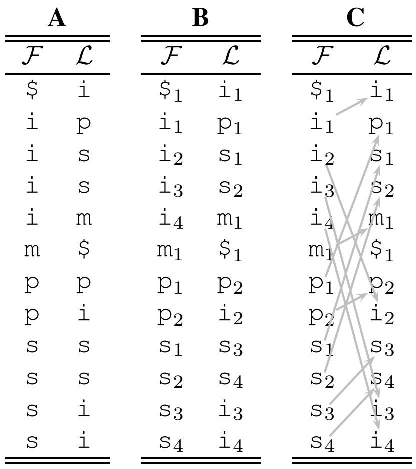

Figure 1: Decoding the Burrows-Wheeler Transform shown in column $\mathcal{L}$. **A**: Pairing the transform with its sorted version. **B**: Counting the occurrences of each character. **C**: Lats-first mapping starting from $\$_1$ in $\mathcal{L}$ to recover the input.

But what feature of *Hamlet* induces this run of `n`? To find out, we print the context of the same segment:

```
./bwt -c -p 91554-92113 Data/hamlet.fasta | less
```

In other words, because `g` is frequently preceded by `n`, the transform groups `ns` by sorting on `g`.

3.2 Decoding

The method for retrieving the original string from its BWT is surprisingly simple: Consider again the original string `mississippi$`, which we have transformed to `ipssm$pissii`. Remember that this is the last column, $\mathcal{L}$, of the sorted matrix of string rotations. We write $\mathcal{L}$ next to the first column of this matrix, $\mathcal{F}$ (Figure 1A). Recall that $\mathcal{F}$ is just the alphabetically ordered input string. The trick is now to count the occurrences of each character in $\mathcal{F}$ and $\mathcal{L}$ such that $\mathtt{c}_i$ is the i-th occurrence of `c` (Figure 1B). To recover the input string, we start at the position of the sentinel character, $\$_1$ in $\mathcal{L}$; the corresponding character in $\mathcal{F}$ is the first character, $\mathtt{m}_1$, of the input string. Next, search for $\mathtt{m}_1$ in $\mathcal{L}$ and look up its partner in $\mathcal{F}$, and so on (Figure 1C). The end result of this single table traversal is $\mathtt{m}_1\mathtt{i}_4\mathtt{s}_4\mathtt{s}_2\mathtt{i}_3\mathtt{s}_3\mathtt{s}_1\mathtt{i}_2\mathtt{p}_2\mathtt{p}_1\mathtt{i}_1\$_1$.

In practice it is not necessary to explicitly construct $\mathcal{F}$. Let T be the text we consider; then three auxiliary arrays suffice to reconstruct $\mathcal{F}$ from $\mathcal{L}$:

- $C[i]$: the number of times the character $T[i]$ has occurred before position i;
- $K[c]$: the total count of character c;
- $M[c]$: the position at which character c first appears in $\mathcal{F}$.

Decoding in demo mode returns all this information (and a bit more):

```
./bwt -d -D Data/mississippiEncoded.fasta
                        K:     M:
P: 123456789012         $ 1    $ 1    P: 123456789012
S: ipssm$pissii         i 4    i 2    S: mississippi$
C: 000100112323         m 1    m 6
                        p 2    p 7
                        s 4    s 9
```

The details of how these arrays are used to reconstruct the input string from $\mathcal{L}$ are explained in [1, chp. 2]. More important than these details is the fact that decoding only takes time proportional to the length of the text. This contrasts with encoding, which is based on sorting n strings of length n. Sorting usually takes time proportional to $n\log(n)$.

To convince ourselves that decoding really reverses the work of encoding, try the following:

```
./bwt Data/hamlet.fasta  | bwt -d > tmp.fasta
diff tmp.fasta Data/hamlet.fasta
1c1
< >hamlet - bwt - decoded
---
> >hamlet
```

Only the header lines differ, but the text is unchanged after one round of encoding and decoding.

Instead of *Hamlet*, we can also subject the genome sequence of the standard laboratory strain of *Escherichia coli* to BWT:

```
./bwt Data/k12genome.fasta | less
```

A cursory glance at the transform reveals no obvious long runs of identical nucleotides.—The sequence looks like a "normal" genome. We can quantify this impression of low repetitiveness by computing the index of repetitiveness, I_r, using the program `ir` [4], which is freely available from my home page under the link "Software".

```
./bwt Data/k12genome.fasta | ir
# Len I_r
4639676 0.0010
```

In fact, the I_r for the transform is much lower than for the original genome:

```
ir < Data/k12genome.fasta
# Len I_r
4639675 0.0587
```

In contrast, the original *Hamlet* has a lower I_r than its transform:

```
ir < Data/hamlet.fasta
# Len I_r
176686 -0.0174

./bwt Data/hamlet.fasta | ir
# Len I_r
176687 0.0430
```

This would indicate that in natural languages a given suffix determines much more strongly the preceding character than in genome sequences.

What about the *E. coli* proteome? We begin by converting it into a single string:

```
echo '>k12proteome' > proteome.fasta
grep -v '^>' Data/k12proteome.fasta >> proteome.fasta
```

The repetitiveness of this string is substantial:

```
ir < proteome.fasta
# Len I_r
1297495 0.3479
```

However, as with the genome, the BWT *reduces* repetitiveness:

```
./bwt proteome.fasta | ir
# Len I_r
1297496 -0.0488
```

We can compare this to the I_r of the randomized proteome using `randomizeSeq`, which is available from my homepage under "Software→bioBox":

```
randomizeSeq proteome.fasta | ir
# Len I_r
1297495 -0.0496
```

The I_r of the randomized proteome is almost identical to that of the transform. This illustrates that the dependence between consecutive amino acids is so weak that the BWT effectively shuffles a proteome. The independence of adjacent amino acids was already observed in insulin, the very first protein sequenced. At the time, the free association between amino acids lead to the rejection of Crick's commaless genetic code [5]. The essential incompressibility of protein sequences has repeatedly been investigated since then [2]. So it is less the compression aspect that motivates application of the BWT in Bioinformatics than the fact that the BWT is a space-efficient replacement of enhanced suffix arrays, which in turn are space efficient representations of suffix trees [1].

4 Listings

4.1 The Driver Program, `bwt.c`

```
/***** bwt.c ****************************************
 * Description: Program for exploring the Burrows-
 *    Wheeler Transform.
 * Reference: D. Adjeroh, T. Bell, and A.
 *    Mukkherjee (2008). The Burrows-Wheeler Trans-
 *    form; Data Compression, Suffix Arrays, and
 *    Pattern Matching. Springer.
 * Author: Bernhard Haubold, haubold@evolbio.mpg.de
 * Date: Fri Mar  2 08:18:05 2012
 ***************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include "interface.h"
#include "eprintf.h"
#include "sequenceData.h"

void scanFile(int fd, Args *args);
void decode(Args *args, Sequence *seq);
void encode(Args *args, Sequence *seq);

int main(int argc, char *argv[]){
  int i;
  char *version;
  Args *args;
  int fd;

  version = "0.3";
  setprogname2("bwt");
  args = getArgs(argc, argv);
  if(args->h || args->e)
    printUsage(version);
  if(args->v)
    printSplash(version);
  if(args->numInputFiles == 0){
    fd = 0;
    scanFile(fd, args);
  }else{
    for(i=0;i<args->numInputFiles;i++){
      fd = open(args->inputFiles[i],0);
      scanFile(fd, args);
      close(fd);
    }
  }
  free(args);
  free(progname());
  return 0;
}

void scanFile(int fd, Args *args){
  Sequence *seq;

  seq = readFasta(fd,args->s);
  if(args->d)
    decode(args, seq);
  else
    encode(args, seq);
  freeSequence(seq);
}
```

4.2 Encoding, `encoding.c`

```
/***** encode.c *************************************
 * Description: Encoding step of Burrows-Wheeler
 *    Transform.
 * Reference: Donald Adjeroh, Timorhy Bell, and
 *    Amar Mukherjee (2008). The Burrows-Wheeler
 *    Transform; Data Compression, Suffix Arrays,
 *    and Pattern Matching. Springer.
 * Author: Bernhard Haubold, haubold@evolbio.mpg.de
 * Date: Mon Mar  5 11:58:31 2012
 **************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "interface.h"
#include "eprintf.h"
#include "sequenceData.h"

char *globalText;
int globalTextLen;


int ringStrCmp(const void *v1, const void *v2);
void printRotations(char *text, int *indexArr, int n);
void printTransform(Args *args, char *text, char *header, int *indexArr,
    int n);
void printContext(Args *args, char *text, char *header, int *indexArr, int
    n);
void printRuns(Args *args, char *text, char *header, int *indexArr, int n);
void encode(Args *args, Sequence *seq){
  int *indexArr;
  int i;

  indexArr = (int *)emalloc(seq->len*sizeof(int));
  for(i=0;i<seq->len;i++)
    indexArr[i] = i;
  globalTextLen = seq->len;
  globalText = seq->seq;
  if(args->D){
    printf("Text:\n");
    printf("%s\n\n",seq->seq);
    printf("Rotations:\n");
    printRotations(globalText, indexArr, globalTextLen);
  }
  qsort(indexArr,globalTextLen,sizeof(int),ringStrCmp);
  if(args->D){
    printf("\nSorted␣Rotations:\n");
    printRotations(globalText, indexArr, globalTextLen);
    printf("\nTransform:\n");
    printTransform(args, globalText, NULL, indexArr, globalTextLen);
  }
  if(!args->D){
    if(!args->c){
      if(args->r)
        printRuns(args, globalText, seq->headers[0], indexArr,
           globalTextLen);
      else
        printTransform(args, globalText, seq->headers[0], indexArr,
           globalTextLen);
    }else
      printContext(args, globalText, seq->headers[0], indexArr,
         globalTextLen);
  }

  free(indexArr);
}

void printContext(Args *args, char *text, char *header, int *indexArr, int
   n){
  int i, j, start, end;

  if(args->p){
    start = args->p[0] > -1 ? args->p[0] : 0;
    end = args->p[1] + 2 < n ? args->p[1] + 2 : n;
  }else{
    start = 0;
    end = n+1;
  }

  for(i=start;i<end-1;i++){
    printf("%d:␣",i+1-start);
    for(j=indexArr[i];j<indexArr[i]+args->l;j++)
```

```
      printf("%c",text[j%n]);
    printf("\n");
  }

}

void printRuns(Args *args, char *text, char *header, int *indexArr, int n){
  int i, j, o, start, end;
  int rowLen = 70;
  char c1, c2;
  char *count;

  count = (char *)emalloc(sizeof(char)*256);
  for(i=0;i<256;i++)
    count[i] = 0;

  if(header)
    printf("%s␣-␣bwt\n",header);
  o = n-1;
  j = 0;
  if(args->p){
    start = args->p[0] > -1 ? args->p[0] : 0;
    end = args->p[1] + 2 < n ? args->p[1] + 2 : n;
  }else{
    start = 0;
    end = n+1;
  }
  c1 = text[(indexArr[start]+o)%n];
  count[(int)c1] = 1;
  for(i=start+1;i<end-1;i++){
    c2 = text[(indexArr[i]+o)%n];
    if(c2 != c1){
      if(count[(int)c1] > 1)
        printf("%c%d",c1,count[(int)c1]);
      else
        printf("%c",c1);
      j += (log(count[(int)c1])/log(10)+1) + 1;
      count[(int)c1] = 0;
      if(j>=rowLen){
        j = 0;
        printf("\n");
      }
    }
    count[(int)c2]++;
    c1 = c2;
  }
  if(count[(int)c1] > 1)
    printf("%c%d",c1,count[(int)c1]);
  else
    printf("%c",c1);
  printf("\n");
  free(count);
}
```

```
void printTransform(Args *args, char *text, char *header, int *indexArr,
   int n){
  int i, j, o, start, end;
  int rowLen = 70;

  if(header)
    printf("%s␣-␣bwt\n",header);
  o = n-1;
  j = 0;
  if(args->p){
    start = args->p[0] > -1 ? args->p[0] : 0;
    end = args->p[1] + 2 < n ? args->p[1] + 2 : n;
  }else{
    start = 0;
    end = n+1;
  }

  for(i=start;i<end-1;i++){
    j++;
    printf("%c",text[(indexArr[i]+o)%n]);
    if(j==rowLen){
      j = 0;
      printf("\n");
    }
  }
  if(j)
    printf("\n");
}

void printRotations(char *text, int *indexArr, int n){
  int i, j;

  for(i=0;i<n;i++){
    for(j=0;j<n;j++)
      printf("%c",text[(indexArr[i]+j)%n]);
    printf("\n");
  }

}

int ringStrCmp(const void *v1, const void *v2){
  int a, b, i;

  a = *(int *)v1;
  b = *(int *)v2;
  i = 0;

  while(globalText[a % globalTextLen] == globalText[b % globalTextLen] && i
      < globalTextLen){
    a++;
    b++;
    i++;
  }
  if(globalText[a] == globalText[b])
    return 0;
  else if(globalText[a] < globalText[b])
    return -1;
  else
    return 1;
}
```

4.3 Decoding, `decoding.c`

```
/***** decode.c ***********************************
 * Description: BWT-decode.
 * Reference: Donald Adjeroh, Timorhy Bell, and
 *    Amar Mukherjee (2008). The Burrows-Wheeler
 *    Transform; Data Compression, Suffix Arrays,
 *    and Pattern Matching. Springer, p. 26.
 * Author: Bernhard Haubold, haubold@evolbio.mpg.de
 * Date: Mon Mar  5 12:22:11 2012
 **************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "eprintf.h"
#include "sequenceData.h"
#include "interface.h"

void decode(Args *args, Sequence *seq){
  int i, j, n, sum, dictSize, start, end;
  /* let F be the first and L the last column in the sorted array of
      rotations */
  int *K;  /* K[i]: count of character seq[i] */
  int *C;  /* C[i]: occurrences of character seq[i] before position i in L
      */
  int *M;  /* M[i]: F[M[i]]: first time character seq[i] occurs in F */
  char *Q; /* output string */

  dictSize = 256;
  n = seq->len - 1;
  K = (int *)emalloc(sizeof(int)*dictSize);
  M = (int *)emalloc(sizeof(int)*dictSize);
  C = (int *)emalloc(sizeof(int)*n);
  Q = (char *)emalloc(sizeof(char)*n);
  /* initialize K */
  for(i=0;i<dictSize;i++)
    K[i] = 0;
  /* count characters in K */
  /* record number of previous appearances of character in C */
  for(i=0;i<n;i++){
    C[i] = K[(int)seq->seq[i]];
    K[(int)seq->seq[i]]++;
  }
  /* first occurrence of character in F */
  sum = 0;
  for(i=0;i<dictSize;i++){
    M[i] = sum;
    sum += K[i];
  }
  /* look for starting character */
  for(i=0;i<n;i++)
    if(seq->seq[i] == BORDER){
      break;
    }
  for(j=n-1;j>-1;j--){
    Q[j] = seq->seq[i];
    i = C[i] + M[(int)seq->seq[i]];
  }
  sum = 0;
  if(args->D){
    printf("P:␣");
    for(i=0;i<n;i++)
      printf("%d",(i+1)%10);
    printf("\nS:␣");
    for(i=0;i<n;i++)
      printf("%c",seq->seq[i]);
    printf("\nC:␣");
    for(i=0;i<n;i++)
      printf("%d",C[i]);
    printf("\nK:\n");
    for(i=0;i<dictSize;i++)
      if(K[i]>0)
        printf("%c\t%d\n",i,K[i]);
    printf("M:\n");
    for(i=0;i<dictSize;i++)
      if(K[i]>0)
        printf("%c\t%d\n",i,M[i]+1);
    printf("P:␣");
    for(i=0;i<n;i++)
      printf("%d",(i+1)%10);
    printf("\nS:␣");
  }else{
    printf("%s␣-␣decoded\n",seq->headers[0]);
  }
  if(args->p){
    start = args->p[0] > -1 ? args->p[0] : 0;
    end = args->p[1] + 2 < n ? args->p[1] + 2 : n;
  }else{
    start = 0;
    end = n;
  }
  sum = 0;
  for(i=start;i<end-1;i++){
    printf("%c",Q[i]);
    sum++;
    if(sum == 70){
      printf("\n");
      sum = 0;
    }
  }
  if(args->D){
    sum++;
    printf("%c",Q[i]);
  }
  if(sum)
    printf("\n");

  free(C);
  free(K);
  free(M);
  free(Q);
}
```

5 Change Log

- Version 0.2 (7th March 2012)
  - First released version.
- Version 0.3 (6th November 2018)
  - Fixed bug in `interface.c`

References

[1] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform:: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 2008.

[2] D. Adjeroh and Fei Nan. On compressibility of protein sequences. In *Data Compression Conference, 2006. DCC 2006. Proceedings*, pages 10 pp. –434, march 2006.

[3] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[4] B. Haubold and T. Wiehe. How repetitive are genomes? *BMC Bioinformatics*, 7:541, 2006.

[5] H. F. Judson. *The Eighth Day of Creation*. Penguin Books, London, 1979/1995.

[6] B. Langmead, C Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10:R25, 2009.

[7] H. Li and R. Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25:1754–1760, 2009.