DNA SUBSEQUENCING USING BURROW-WHEELER ALIGNMENT

Rohan C
PES University
PES1UG20CS345
Bangalore -74 India
chandrashekar.rohans@gmail.com

***Abstract*—**

Motivation: The enormous number of short reads generated by the new DNA sequencing technologies call for the development of fast and accurate read alignment programs. Several such algorithms and programs have been developed for such use However, many of them do not support gapped alignment for single end reads, which makes it unsuitable for alignment of longer reads where indels may occur frequently. The speed of sequencing is also a concern when the alignment is scaled up to the resequencing of hundreds of individuals.

Results: Here, Burrows-Wheeler Alignment tool (BWA) has been implemented. A new read alignment package that is based on backward search with Burrows–Wheeler Transform (BWT), to efficiently align short sequencing reads against a large reference sequence such as the human genome, allowing mismatches and gaps. BWA supports space reads as well. However, these features are beyond the scope of the project, and we will be implementing a straight-forward subsequence matching algorithm.

I. INTRODUCTION

Many of the current sequence searching algorithms involve hash table technique. This is a unique and useful technique because it allows for very fast matches on short search strings. However, these pre-processing techniques become very tedious when the string is over a million bytes long and must be read repeatedly for short matching sequences. Furthermore, it also must allow for some error margin to be defined as DNA matching is not always 100% accurate. Such limitations must be kept in mind while designing an algorithm because even a few changes in the **indels** can mean the difference between Cancer and Alzheimer's.

II. PREFIX TRIE AND SUFFIX ARRAY

The prefix trie for string X is a tree where each edge is labeled with a symbol and the string concatenation of the edge symbols on the path from a leaf to the two numbers in a node give the SA interval of the string represented by the node (see Section 2.3). The dashed line shows the route of the brute-force search for a query string 'LOL', allowing at most one mismatch. Edge labels in squares mark the mismatches to the query in searching. The only hit is the bold node [1,1] which represents string 'GOL'. The root gives a unique prefix of X. On the prefix trie, the string concatenation of the edge symbols from a node to the root gives a unique substring of X, called the string represented by the node. Note that the prefix trie of X is identical to the suffix trie of reverse of X and therefore suffix trie theories can also be applied to prefix trie.

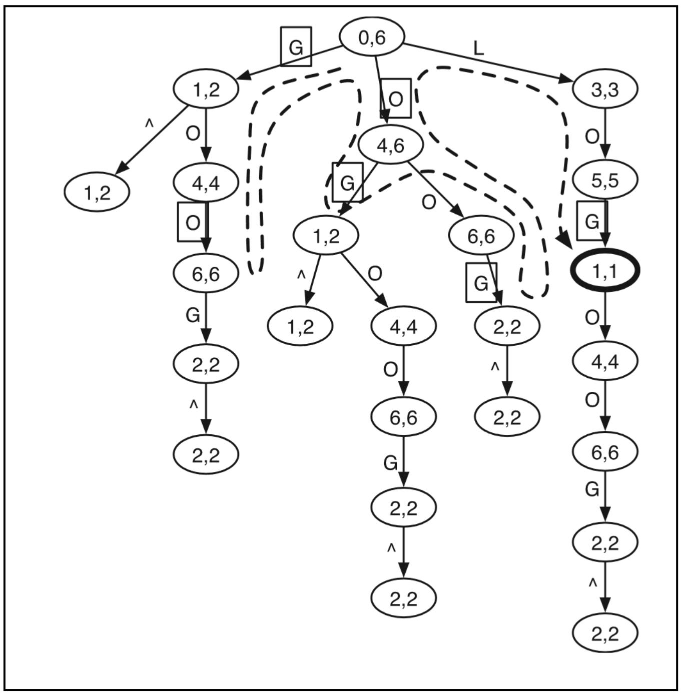


Fig. 1. Example of a suffix trie design for 'GOOGOL' ^, marks the start.

With the prefix trie, testing whether a query W is an exact substring of X is equivalent to finding the node that represents W, which can be done in O(|W|) time by matching each symbol in W to an edge, starting from the root. To allow mismatches, we can exhaustively traverse the trie and match W to each possible path. We will later show how to accelerate this search by using prefix information of W. Figure 1 gives an example of the prefix trie for 'GOOGOL'.

III. BURROW-WHEELER TRANSFORM

A. History

The Burrow-Wheeler Transform was a file compression technique that was discovered in 1994. Initially this algorithm set a benchmark and continues to do so for compression of JPEG and other files.

B. *Uniqueness*

This algorithm has the unique property of lossless compression. This means that at any given state of the transform, we are only one step away from converting it back into the original state. This is the property of BWT that we are going to exploit in order to pre-process the input string.

IV. Method

Let there be an alphabet. Symbol \$ is not present in the alphabet and is lexicographically smaller than all the symbols in . A string X =a0a1...an−1 is always ended with symbol \$ (i.e. an−1=\$) and this symbol only appears at the end. Let X[i]=a_i, i=0,1,...,n−1, be the ith symbol of X, X[i,j] = ai,...,aj substring and X_i =X[i,n−1] a suffix of X. Suffix array S of X is a permutation of the integers 0...n−1 such that S(i) is the start position of the i-th smallest suffix. The BWT of X is defined as B[i]=\$ when S(i)=0 and B[i]=X[S(i)−1] otherwise. We also define the length of string X as |X| and therefore |X|= B|=n. Figure 2 gives an example on how to construct BWT and suffix array.

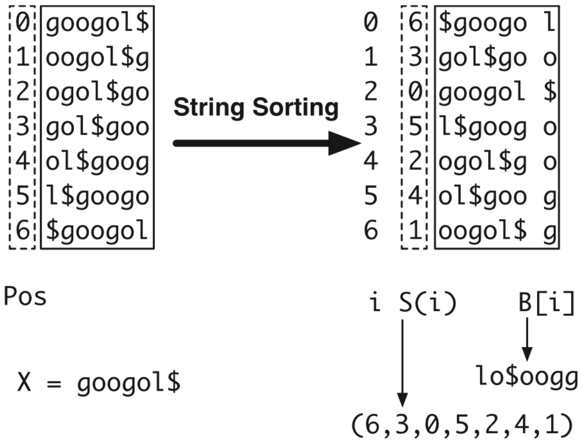


Fig. 2. Constructing suffix array and BWT string for X =`googol$`. String X is circulated to generate seven strings, which are then lexicographically sorted. After sorting, the positions of the first symbols form the suffix array (6,3,0,5,2,4,1) and the concatenation of the last symbols of the circulated strings gives the BWT string `lo$oogg`.

The algorithm shown in Figure 2 is quadratic in time and space. However, this is not necessary. In practice, we usually construct the suffix array first and then generate BWT. Most algorithms for constructing suffix array require at least nlog2n bits of working space, which amounts to 12GB for human genome. Recently, Hon et al. (2007) gave a new algorithm that uses n bits of working space and only requires <1GB memory at peak time for constructing the BWT of human genome. This algorithm is implemented in BWT.

V. Approach and Algorithm

Let C(a) be the number of symbols in X[0,n−2] that are lexicographically smaller than a∈ and O(a,i) the number of occurrences of a in B[0,i]. Ferragina and Manzini (2000) proved that if W is a substring of X:

$$R(aW) = C(a)+O(a,R(W)-1)+1 \quad (3)$$

$$\overline{R}(aW) = C(a)+O(a,\overline{R}(W)) \quad (4)$$

and that R(aW)≤R(aW) if and only if aW is a substring of X. This result makes it possible to test whether W is a substring of X and to count the occurrences of W in O(|W|) time by iteratively calculating R and R from the end of W. This procedure is called backward search.

It is important to note that Equations (3) and (4) actually realize the top down traversal on the prefix trie of X given that we can calculate the SA interval of a child node in constant time if we know the interval of its parent. In this sense, backward search is equivalent to exact string matching on the prefix trie, but without explicitly putting the trie in the memory.

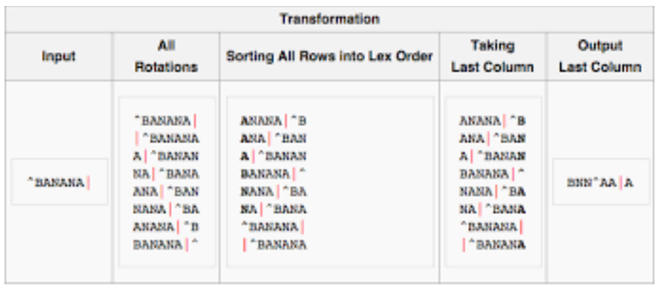
```
Pre calculation:
Calculate BWT string B for reference string X
Calculate array C(·) and O(·,·) from B
Calculate BWT string B for the reverse reference
Calculate array O(·,·) from B

Procedures:
Inexact Search (W, z) CalculateD(W)
  return InexRecur(W,|W|−1,z,1,|X|−1)

CalculateD(W) k←1
  l←|X|−1 z←0
                                   for
    $\leftarrow C(W[i])+O'(W[i],l)$    i=0
    >                              to
              |W|−1 do k
    $\leftarrow C(W[i])+O'(W[i],k-1)+1$l
    if k        l
      then k←1
      l←|X|−1
      z←z+1
              D(i)←z
```

```
*           I ←I ∪ InexRecur(W,i−1,z−1,k,l) for each
            b∈{A,C,G,T} do
            k←C(b)+O(b,k−1)+1 l←C(b)+O(b,l)
                if k≤l then
*           I ←I ∪ InexRecur(W,i,z−1,k,l)
                if b=W[i] then
                    I ←I ∪ InexRecur(W,i−1,z,k,l)
                else
            I ←I ∪ InexRecur(W,i−1,z−1,k,l) return I
```

Fig 3. Algorithm for inexact search of SA intervals of substrings that match *W*. Reference *X* is $ terminated, while *W* is A/C/G/T terminated. Procedure InexactSearch(W,z) returns the SA intervals of substrings that match *W* with no more than *z* differences (mismatches or gaps); InexRecur(W,i,z,k,l) recursively calculates the SA intervals of substrings that match $W[0,i]$ with no more than *z* differences on the condition that suffix W_{i+1} matches interval $[k,l]$. Lines started with asterisk are for insertions to and deletions from *X*, respectively. $D(i)$ is the lower bound of the number of differences in string $W[0,i]$.



Transformation				
Input	All Rotations	Sorting All Rows into Lex Order	Taking Last Column	Output Last Column
^BANANA|	^BANANA| |^BANANA A|^BANAN NA|^BANA ANA|^BAN NANA|^BA ANANA|^B BANANA|^	ANANA|^B ANA|^BAN A|^BANAN BANANA|^ NANA|^BA NA|^BANA ^BANANA| |^BANANA	ANANA|^B ANA|^BAN A|^BANAN BANANA|^ NANA|^BA NA|^BANA ^BANANA| |^BANANA	BNN^AA|A

Fig 4. An example showing how the Burrow-Wheeler Transform will work on the word BANANA.

I. CHALLENGE – IMPLEMENTING THE CODE

One of the main challenges of implementing this algorithm was understanding the core function that the transform will perform. This was crucial in implementing the function in python language. The conversion of the string into its suffix array was relatively easier. But checking for the matching was a major difficulty. The code snippet is as given below:

```
        for x in range(len(query)):
            newChar = query[-x-1]
            newI = self.C[newChar] + self.OCC(newChar,i-1) + 1
            newJ = self.C[newChar] + self.OCC(newChar,j)
            i = newI
            j = newJ
        matches = self.SA[i:j+1]
        return matches
```

II. RESULT

Table 2. Evaluation on real data

Program	Time (h)	Conf (%)	Paired (%)
Bowtie	5 2	84.4	96.3
BWA	.4 0	88.9	98.8
MAQ	94 9	86.1	98.7
SOAP2	3 4	88.3	97.5

Here we can clearly see the superiority of BWA algorithm compared to other contemporary algorithms that are used today.

For short read alignment against the human reference genome, BWA is an order of magnitude faster than MAQ while achieving similar alignment accuracy. It supports gapped alignment for single end reads, which is increasingly important when reads get longer and tend to contain indels.

In comparison to speed, memory and the number of mapped reads, alignment accuracy is much harder to evaluate on real data as we do not know the ground truth.

Although in theory BWA works with arbitrarily long reads, its performance is degraded on long reads especially when the sequencing error rate is high. Furthermore, BWA always requires the full read to be aligned, from the first base to the last one (i.e., global with respect to reads), but longer reads are more likely to be interrupted by structural variations or misassembles in the reference genome, which will fail BWA. For long reads, a possibly better solution would be to divide the read into multiple short fragments, align the fragments separately with the algorithm described above and then join the partial alignments to get the full alignment of the read.

REFERENCES

[1] Burrows,M. and Wheeler,D.J. (1994) A block-sorting lossless data compression algorithm. *Technical report 124*, Palo Alto, CA, Digital Equipment Corporation.

[2] Campagna,D. *et al.* (2009) PASS: a program to align short sequences. *Bioinformatics*, **25**, 967–968.

[3] Eaves,H.L. and Gao,Y. (2009) MOM: maximum oligonucleotide mapping.

[4] *Bioinformatics*, **25**, 969–970.

[5] Ferragina,P. and Manzini,G. (2000) Opportunistic data structures with applications. In *Proceedings of the 41st Symposium on Foundations of Computer Science (FOCS 2000)*, IEEE Computer Society, pp. 390–398.

[6] Grossi,R. and Vitter,J.S. (2000) Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings on 32nd Annual ACM Symposium on Theory of Computing (STOC 2000)*, ACM, pp. 397–406.

[7] Hon,W.-K. *et al.* (2007) A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, **48**, 23–36.

[8] Jiang,H. and Wong,W.H. (2008) SeqMap: mapping massive amount of oligonucleotides to the genome. *Bioinformatics*, **24**, 2395–2396.

[9] Jung Kim,Y. *et al.* (2009) ProbeMatch: a tool for aligning oligonucleotide sequences. *Bioinformatics*, **25**, 1424–1425.

[10] Lam,T.W. *et al.* (2008) Compressed indexing and local alignment of DNA.

[11] *Bioinformatics*, **24**, 791–797.

[12] Langmead,B. *et al.* (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, **10**, R25.

[13] Lin,H. *et al.* (2008)ZOOM! Zillions of oligos mapped. *Bioinformatics*, **24**, 2431–2437.

[14] Lippert,R.A. (2005) Space-efficient whole genome comparisons with Burrows-Wheeler transforms. *J. Comput. Biol.*, **12**, 407–415.

[15] Li,H. *et al.* (2008a) Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Res.*, **18**, 1851–1858.

[16] Li,R. *et al.* (2008b) SOAP: short oligonucleotide alignment program. *Bioinformatics*, **24**, 713–714.

[17] Malhis,N. *et al.* (2009) Slider–maximum use of probability information for alignment of short sequence reads and SNP detection. *Bioinformatics*, **25**, 6–13.

[18] Schatz,M. (2009) Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, **25**, 1363–1369.

[19] Smith,A.D. *et al.* (2008) Using quality scores and longer reads improves accuracy of Solexa read mapping. *BMC Bioinformatics*, **9**, 128.