8. *Burrows-Wheeler transzformáció*

A BURROWS-WHEELER transzformáció n hosszú szöveget képez le $(n+1)$ hosszú szövegre, visszafordíthatóan. A $T[1..n]$ szöveg transzformációja három lépésben végezhető el: (1) T végére biggyesztünk egy speciális # karaktert, ami minden más szövegbeli szimbólumtol különbözik, és azok előtt van az ábécében; (2) tekintjük az M mátrixot, aminek sorai a T# ciklikus eltoltjait tartalmazzák, lexikografikusan növekvő sorba rendezve; (3) az M mátrix utolsó oszlopa adja T Burrows-Wheeler transzformáltját (bwt). A Burrows-Wheeler transzformáció ekvivalens a *suffix array* (SA) reprezentációval, ami a T# szuffixainak lexikografikus sorrendjét adja meg, az $1..n+1$ startpozíciók permutációjaként.

1	lalialilalo#		12	#lalialilal**o**
2	alialilalo#l		2	alialilalo#**l**
3	lialilalo#la		5	alilalo#lal**i**
4	ialilalo#lal		9	alo#laliali**l**
5	alilalo#lali		4	ialilalo#la**l**
6	lilalo#lalia		7	ilalo#lalia**l**
7	ilalo#lalial	⇒	1	lalialilalo**#**
8	lalo#laliali		8	lalo#lalial**i**
9	alo#lalialil		3	lialilalo#l**a**
10	lo#lalialila		6	lilalo#lali**a**
11	o#lalialilal		10	lo#lalialil**a**
12	#lalialilalo		11	o#lalialila**l**

bwt(lalialilalo#) = olilll#iaaal

SA(lalialilalo#) = 12, 2, 5, 9, 4, 7, 1, 8, 3, 6, 10, 11

A gyakorlatban az M mátrixot nem tároljuk sehol (hiszen az $O(n^2)$ memóriát igényelne), de egy $T[1..n]$ szöveg bwt(T#) Burrows-Wheeler transzformáltját gyorsan és memória-takarékosan ki lehet számítani[1].

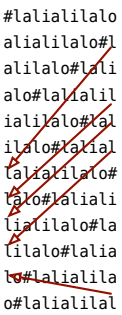
[1] A `bwa` illesztő program [Li & Durbin] például létező *suffix array* implementációt használ, ami SA(T#)-t $\Theta(n)$ időben és $\Theta(n)$ memóriával számolja.

Tétel. Minden szövegre bwt(T#) egyértelműen meghatározza az M konceptuális mátrixot, és így az eredeti T szöveget is.

A BWT egyik előnye, hogy jol tömöríthető akármilyen ábécére (a `bzip` is erre épül), mivel repetitív szöveg transzformáltjában gyakran hosszan ismétlődik ugyanaz a karakter. DNS szekvencia BWT-jét azonban egyszerűen, direktben lehet tárolni 2 biten pozíciónként.

LF mapping

A BWT-vel való kereséshez azt a tulajdonságát használjuk ki, hogy az azonos karakterek sorrendje az utolsó oszlopban megegyezik a nekik megfelelő karakterek sorrendjével az első oszlopban. Azaz, ha LF(i) azt jelöli, hogy az M mátrix i-edik sorának utolsó karaktere az első oszlopban melyik sorban van, akkor minden $M[i, n+1] = M[i', n+1]$ esetén LF(i) > LF(i') akkor és csak akkor ha $i > i'$. Legyen B = bwt(T#). Az *LF-mapping* számolásához kétféle adatot tárolunk:



- $C[a]$ az a karakternél kisebb szimbólumok B-beli számát tartalmazza
- Occ(i, a) az a karakter előfordulását számolja a $B[1..i]$ prefixben

	#	a	i	l	o
C	0	1	4	6	11

Occ	a	i	l	o	#
1	0	0	0	1	0
2	0	0	1	1	0
3	0	1	1	1	0
4	0	1	2	1	0
5	0	1	3	1	0
...					
12	3	2	5	1	1

Az *LF-mapping* számításához ekkor $LF(i) = C[B[i]] + \text{Occ}(i, B[i])$. Ezt a BWT invertálásához is lehet használni: legyen $s \leftarrow 1$ és $T[n] \leftarrow B[1]$, aztán $i \leftarrow n-1, n-2, \ldots, 1$ ciklusban legyen $s \leftarrow LF(s)$ és $T[i] \leftarrow B[s]$.

Keresés

Tegyük fel, hogy a $P[1..p]$ szó előfordulását akarjuk megszámolni a T-ben. Ehhez az *LF-mapping*-et használjuk: egy $i \rightarrow p, p-1, \ldots, 1$ ciklusban a $P[i..p]$ szuffixek előfordulását követjük az M mátrix soraiban.

```
    COUNTOCCURRENCE(P[1..p])
S1  a ← P[p]; i ← p
S2  s ← C[a] + 1; e ← C[a + 1]     //P[p]-t megtaláljuk az s..e sorokban
S3  while s ≤ e && i > 1 do
S4      i ← i − 1; a ← P[i]
S5      s ← C[a] + Occ(s − 1, a) + 1; e ← C[a] + Occ(e, a)
                                   //most P[i..p]-t találjuk az s..e sorokban
S6  return max{e − s + 1, 0}
```

BWA

Tegyük fel, hogy nem megszámolni akarjuk, hanem azt tudni, hogy milyen pozíciókban fordul elő P. A keresés hasonlóan zajlik le, mint az előfordulások számolása, de most a megtalált sorokhoz tartozó $\mathsf{SA}(s)$, $\mathsf{SA}(s+1)$, …, $\mathsf{SA}(e)$ értékeket kell tudni. Ezt akár a BWT tömörített változatával is meg lehet csinálni nagyon hatékonyan. Az általános ábécére működő kompakt adatstruktúrát FM-indexnek hívják, Paolo Ferragina és Giovanni Manzini fejlesztette ki. DNS-re azonban lehet ennél praktikusabb megoldás is. A bwa program [Li & Durbin] a Burrows-Wheeler transzformálton kívül ($2n$ bit) az Occ értékeket közvetlenül tárolja a négy nukleotidra, minden 128. pozícióban ($n/32$ integer változó), és a SA értékeket szintén, minden 32. pozícióban ($n/32$ integer változó). Köztes i pozíciókban az SA értéke kiszámolható az LF-mapping segítségével, Occ pedig a nukleotidok direkt számolásával. Az adatstruktúrát tárolni kell mind a forward mind a reverse szálra. Mindösszesen néhány Gigabyte elég az emberi genom teljes indexeléséhez.

A Burrows-Wheeler alapú illesztés nehézsége, hogy míg pontos előfordulást nagyon gyorsan megtalál, a mismatch-ek és a gap-ek kezelése elég nehézkes. Ehhez a bowtie [Langmead ++] és a bwa program is úgy keres, hogy max. d hibát megenged a keresett szóban, és a hibákkal módosított variánsokra is rákeres.

Hivatkozások

M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report SRC 124, Digital Equipment Corporation, 1994. http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf.

Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science (FOCS)*, pages 390–398, 2000. http://people.unipmn.it/manzini/papers/focs00draft.pdf.

Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10:R25, 2009. DOI: 10.1186/gb-2009-10-3-r25.

Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25:1754–1760, 2009. DOI: 10.1093/bioinformatics/btp324.