BWT and FM-index

©kuko 26.11.2023

1 Introduction

- suffix tree (ST) – even if we are careful and try to use as little memory as possible, it takes about 10–20B/character
- suffix array (SA) needs 1 int/character; for texts up to 4 billion chars, we can use a 32-bit ints, which is 4B/character (if we use the LCP array – another 12B/character) + the text itself
- take for example the human genome, which is a string of about 3 billion chars from the alphabet `A`, `C`, `G`, `T`
- the string itself therefore takes about 3GB (if we use 1B/character), or 750MB if we use packed representation with 2bits/character
- the suffix tree will occupy 30–60GB or more and the suffix array about 12GB (+the string itself +0.75GB)
- and that's just the memory of the resulting structure, where we don't count the memory used temporarily during construction
- when processing larger inputs, we will be limited by the RAM size
- in this lecture we will show how to achieve an even more memory-efficient solution
- we will even show that we can compress(!!!) the input text in such a way that we still allow fast search
- the resulting structure, FM-index, is based on Burrows-Wheeler transformations and rotations of the string T

2 Burrows-Wheeler transformation

- consider all rotations of the string T and sort them lexicographically
- i.e., consider a BW $n \times n$ matrix M
- T^{bwt} is a string consisting of the last symbol in each row – the last column of the matrix
- sorting all rotations is similar to sorting all suffixes, so it is not surprising that that there is a simple relationship between T^{bwt} and the suffix field:
  - $T^{\mathrm{bwt}}[i] = T[SA[i] - 1]$ (where $T[-1] = \$$)
- it follows that we can calculate T^{bwt} in linear time

2.1 Application of BWT in compression

- example: we find the substring "`ATTLE`" in English text; what do you think is the previous letter?
  - most likely `B` (from the words battle, embattle) or `C` (cattle)
  - but there are a few other options: `R` (prattle), `T` (tattle)
- if we sort T rotations, all rotations starting with "`ATTLE`" will be consecutive, so in the last column, there will be a lot of `B`'s and `C`'s and maybe a few `R`'s, `T`'s in this interval;
- symbols with the same context are brought together using BWT
- in general, T^{bwt} will contain intervals with repeated symbols and more generally longer intervals, where there are only a few different symbols – such a string is much easier to compress
- for example, the `bzip2` algorithm consists of several steps:
  - 1. BWT – we get repeated symbols and long sections with a small number of symbols
  - 2. MTF (move-to-front: during encoding, maintain a list of symbols; replace the i-th symbol by number i and at the same time move it to the beginning of the list) – this way, if some symbol repeats often, it tends to stay at the beginning of the list and is encoded as small number; e.g., a run of a single repeated character will be transformed into a run of zeros; and

  a long section containing only d different symbols will be transformed into a long section of small numbers $0..d-1$
  - 3. RLE (run-length encoding) – replace a substring $ccc...c$, a run of symbol c repeated $k\times$ by encoding it as a pair (k, c)
  - 4. finally use the Huffman code to encode individual symbols

2.2 Reverse transformation $T^{\text{bwt}} \to T$

- assume for a moment that all characters in T are different – then sorting the rotations is easy, because it is enough to compare the first symbol
- if we sort the characters $L = T^{\text{bwt}}$, we get the first column F
- note that if we have the first and the last column, we can easily reconstruct T, namely, L_i is the letter that is located before F_i in T
- simply start from the end with \$, find the row where $F_i = \$$; the last character is $c = L_i$; next, find the row where $F_j = c$ and that means the previous character was $c = L_j$ – we continue this way until the beginning of the string
- now let's think about the general case: if the character c is repeated in T, which position in L belongs to which position in F?
- it is not difficult to see that the i-th occurrence of c in L corresponds to the i-th occurrence of c in F:
  - if cx is before cy, then $x < y$ and therefore xc is before yc
  - in other words, all rows starting with c are sorted by the rest of the string – and so are rows with c at the end
- for efficient reverse transformation, we need for a given L_i quickly find the corresponding row F_j
- this is the so-called LF-mapping; it can be easily calculated already during the sorting of L into F – however, we would additionally like to represent the LF-mapping using a small memory
- for this, it is sufficient to know the number of c's in $L[0..i]$ for each character c and for any i (this is the classic $\text{rank}_c(L, i)$ problem) and for a given c, where does the section c's start in F

3 FM-index

3.1 Search

- in the previous section, we described the use of BWT in compression and the inverse transformation; but can we search in T^{bwt} efficiently?
- since BWT is closely related to SA, we could try binary search just like in SA; however, in the resulting FM-index we will not directly remember the whole T and answering what is the j-th character in the ith row will be much slower than in SA – thus the entire binary search would be much slower
- however, there is a better way, using the LF-mapping
- the search for $P = p_0 \dots p_{m-1}$ will proceed backwards, starting from the last character; we will successively search for suffixes $P_{i\dots} = p_i p_{i+1} \dots p_{m-1}$ for $i = m-1, \dots, 0$
- more precisely: since the rows of the imaginary matrix M are sorted lexicographically, rows starting with word w form an interval; so, let $[s_i, e_i)$ be the interval of rows beginning with the suffix $P_{i\dots}$
- if we know $[s_{i+1}, e_{i+1})$, how do we find $[s_i, e_i)$?
- we know the interval of rows that start with $P_{i+1\dots}$; some of these rows end with p_i – these rows correspond to occurrences of $P_{i+1\dots}$ preceded with p_i
- although rows starting with $P_{i+1\dots}$ and ending with p_i may not form one continuous interval, their rotations by 1 character left are rows starting with $P_{i\dots} = p_i P_{i+1\dots}$ and they do form an interval of rows

- we know that $[s_i, e_i)$ will be a subinterval of rows that start with p_i ($[F[p_i], F[p_i+1])$) + recall from the previous section that if $cx < cy$, then $x < y$ and therefore $xc < yc$, i.e. all rows starting with a given character are in the same order as all rows ending with that character; therefore we can split the interval $[F[p_i], F[p_i+1])$ into 3 parts:
  - rows $[F[p_i], s_i)$ are the rows that start with $p_i x$ where $x < P_{i+1\ldots}$,
  - rows $[s_i, e_i)$ start with $P_{i\ldots}$ (we are looking for this interval), and
  - rows $[e_i, F[p_i+1])$ are rows that start with $p_i y$, where $y > P_{i+1\ldots}$
- it is enough to find out $\mathrm{rank}_{p_i}(L, s_{i+1})$ – how many times the character p_i occurs before s_{i+1}, i.e. number of occurrences of $p_i x$, where $x < P_{i+1\ldots}$, and $\mathrm{rank}_{p_i}(L, e_{i+1})$ – number of occurrences of $p_i x$ for $x \leq P_{i+1\ldots}$ ($|x| = m-i-1$)
- $[s_{m-1}, e_{m-1}) \leftarrow [F[p_{m-1}], F[p_{m-1}+1])$
- $[s_i, e_i) \leftarrow [F[p_i] + \mathrm{rank}_{p_i}(L, s_{i+1}-1), F[p_i] + \mathrm{rank}_{p_i}(L, e_{i+1}))$
- at the end we get the interval $[s_0, e_0)$ – rows starting with P

- that's the idea; now let's think about some "details":
- given a row in M – how do we find the position in the text T?
- this was the information stored in the suffix array – $SA[k]$ = position of the k-th smallest suffix/rotation in T – however, we don't want to remember the entire suffix array (that takes too much space)
- solution: we will remember a subset of SA – for example, only $SA[k]$ values divisible by s
- if we want to find out the value of $SA[k]$, which is not stored, we will use the LF-mapping to move to the previous rotations until after at most $< s$ steps, we encounter a value which is stored in SA (the result then is $SA[k'] + \#$steps back we took)
- how do we represent $L = T^{\mathrm{bwt}}$ so that we can quickly find $\mathrm{rank}_c(L, i)$?
- the easiest way is to precompute values $\mathrm{rank}_c(L, i)$ for every character c, but only for positions which are multiples of some constant b; to answer the queries, we do 1 lookup + count the rest linearly; so again, we are doing a time/space trade-off; for small alphabets and large enough b, this is fine, we will show better solutions in the following lectures – this is a large topic
- can we recover the i-th character of T, or a substring $T_{i\ldots j}$ even without remembering T? we showed how to recover T from T^{bwt} in $O(n)$; however, can we recover random small parts without doing the whole inverse BWT?
- yes: we just need to be able to quickly find for a position in T what is the corresponding row in M
- solution: we need to store the sample of SA in such away that we can quickly answer both $SA[i] = ?$ and $SA[?] = i$
- when we want to extract $T_{i\ldots j}$, we first "round" j up to the nearest value divisible by s, we find the corresponding row in M and using the LF-mapping, we gradually decode T up to the ith position

4 Summary

- the resulting structure will consist of:
  - F (first column of M) – $|\Sigma|$ integers
  - $L = T^{\mathrm{bwt}}$ (last column of M) – n characters (for now – but these can be compressed)
  - data structure for $\mathrm{rank}_c(L, i)$ – $n|\Sigma|/b$ integers (for now – we will show better solutions)
  - a subset of SA – $2n/s$ integers
- how much memory does it take in practice? take our example with DNA, where $|\Sigma| = 4$; let's choose $s = 64$ and $b = 128$ (i.e., calculating rank or finding a value in a sample of SA still takes constant time, but we may have to scan up to 64–128 values):
  - $|F|$=16B – this is nothing
  - $|L|$=750MB – the original string (use the packed encoding)
  - $|SA|$=12GB$\times$2/64=375MB,
  - $|\mathrm{rank}|$=12GB$\times$4/128=375MB

- – that's only 1.5GB in total, or 2× the size of the original string(!)

- note that, unlike ST and SA, we do not need to remember the original string T (we can reconstruct it from $L = T^{\mathrm{bwt}}$)
- so starting with a 30–60GB suffix tree, through 12GB suffix array, we achieved 1.5GB FM-index – that's 20–40× less memory – and we haven't even compressed L yet and we just used a very simple solution for rank
- with compression and other improvements, we can achieve a DS that occupies 30–50% of space of the original string, and we can still support efficient search