

CSE 584A Class 12

Jeremy Buhler

February 26, 2018

1 Space and Time Results So Far for BWT-based Search

Last time, we saw that we could obtain $\Theta(m)$ search using the Burrows-Wheeler Transform of a text T .

- Search time is $\Theta(m)$ to check whether pattern P occurs in text T , and to count the number of instances if it does.
- This part uses only the C and $\text{occ}_B(*, i)$ tables for T .
- To actually enumerate suffixes that start with P , we need to enumerate a contiguous range of entries from the suffix array A for T .
- If we have A around, this takes time proportional to the number of entries.
- Hence, given C , occ_B , and A , we have an algorithm that is $\Theta(m + \# \text{ occurrences})$ in time.
- Space usage (assuming most efficient possible number storage) is as follows:
 - $|\Sigma| \log n$ bits for C
 - $|\Sigma|n \log n$ bits for occ_B
 - $n \log n$ bits for A
- This is still quite a lot of space – in practice, $4n$ to $8n$ bytes for suffix array, and (for DNA) $16n$ to $32n$ bytes for occ_B .

Yikes! Can we improve on these space bounds? Yes, at the cost of slightly slower execution.

2 Compressing occ_B

Here's a trick to compress occ_B by an arbitrary constant factor k .

- Suppose we know $\text{occ}_B(a, i)$, and we want to compute $\text{occ}_B(a, i + \ell)$.
- If we have the BWT B handy, we can simply count the number of times x that a occurs between $i + 1$ and ℓ , in time $\Theta(\ell)$.
- Then we have that

$$\text{occ}_B(a, i + \ell) = \text{occ}_B(a, i) + x.$$

- Suppose we *sample* the occ_B table, storing $\text{occ}_B(a, i)$ only for i such that

$$i = 0 \pmod{k}.$$

- If we keep B around in memory, we can still compute any entry in time $\Theta(k)$.
- Space cost is now $n \log |\Sigma| + \frac{|\Sigma|n \log n}{k}$ bits (cost for B plus cost for sampled occ_B).
- *Example*: set $k = 128$. Then for DNA, we use only $n/4$, plus $n/8$ to $n/4$, bytes.

Sneaky, but space is now dominated by cost of keeping suffix array A around in memory – $4n$ to $8n$ bytes.

3 Compressing A – the Compressed Suffix Array

We can actually sample the suffix array A as well! The result is the Compressed Suffix Array (CSA), invented by Grossi and Vitter (2000).

- We need our trusty rprev function, which (recall) is computable in constant time from occ_B .
- Here’s a useful identity from Grossi and Vitter:

$$A[i] = A[\text{rprev}(i)] + 1$$

- **Pf:**

$$\begin{aligned} A[\text{rprev}(i)] + 1 &= A[R[A[i] - 1]] + 1 \\ &= A[i] - 1 + 1 \\ &= A[i]. \end{aligned}$$

- **Cor:** $A[i] = A[\text{rprev}^j(i)] + j$.
- Now suppose that for some k we sample A by storing *only* those $A[q]$ for which $A[q] = 1 \pmod{k}$.
- To compute an arbitrary $A[i]$, we successively compute values $\text{rprev}^j(i)$ until $A[\text{rprev}^j(i)]$ is in our sample, then return $A[\text{rprev}^j(i)] + j$.
- Space cost for sampled A is now $\frac{n \log n}{k}$ bits.
- In practice, if $k = 32$, space is $n/8$ to $n/4$ bytes.

Speed? How many times might we have to iterate rprev to reach a value in our sample?

- Observe that by our corollary, $\text{rprev}^j(i) = R[A[i] - j]$.
- In words, $\text{rprev}^j(i)$ is the rank in A of the suffix that begins j characters before $A[i]$.
- Each additional application of rprev therefore moves us back one character in the text T .

- Our choice of stored sample guarantees that we store entries of A for every k th suffix of the text.
- Hence, we will reach a suffix in our sample after at most k applications of rprev , which take total time $\Theta(k)$.

A few more details: how do we know when we've reached a suffix that is represented in our A sample?

- We need to remember for which i we've stored $A[i]$.
- Pattern is regular in T but could be arbitrary in A .
- Can mark stored A values using a bit vector V of size n .
- Set $V[i] = 1$ iff we've stored $A[i]$.
- Each time we compute $q = \text{rprev}^j(i)$, check if $V[q] = 1$; when it is, we know $A[q]$ and so can immediately get $A[i]$.
- OK, we know when $A[q]$ is in our sample, but how do we store all the sampled A values so that we can look up any one of them quickly?
- *DO NOT* use a hash table – the space multiplier for such a table is typically $10\times$ or more the number of stored values!
- *Idea*: use a dense array A^* that stores these values in the order of their occurrence in A .
- The p th value in the sample occurs at $A^*[p]$.
- Equivalently, if $A[i]$ is in the sample, it occurs at $A^*[\text{occ}_V(1, i)]$.
- I.e., if $V[i] = 1$, and V has q 1-bits at or before i , then $A[i]$ is the q th value in the sampled array A^* .
- To compute occ_V in constant time and little space, we can use the same sampling trick as we did for occ_B !
- For some k' , store $\text{occ}_V(1, i)$ for $i = 0 \pmod{k}'$.
- V plus the extra stored values enable arbitrary occ_V lookup in time $\Theta(k')$.
- The compressed occ_V takes $n/k' \log n + n$ bits.
- Hence, our total space cost for sampling A is now $(n/k + n/k') \log n + n$ bits.
- In practice, if $k = 32$ and $k' = 128$, we need about $n/4 + n/32$ to $3n/8 + n/16$ bytes.

Is this approach practical?

- The BWA program of Li and Durbin is used for short read alignment to a genome or other reference DNA source.
- They use BWT-based search as we've described.

- Their occ_B array is compressed just as described.
- They use a simpler approach to compressing A : just store $A[i]$ for $i \bmod k = 1$.
- We don't need a bit vector to determine if $A[i]$ is in our sample; we can just compute $i \bmod k$.
- However, the number of rprev calls necessary to find an A value in our sample is not *a priori* bounded.

How do you build a CSA, anyway?

- Trivially, we could first build the suffix array, then compress it.
- But this isn't terribly useful if we don't have enough memory to hold A in the first place!
- We can build up a CSA from the BWT.
- As we've seen, you can use the BWT to incrementally compute R , the inverse suffix array.
- To get the set of A entries needed for the bounded approach, sample R every k iterations. This gives ranks for suffixes spaced equally in the text.
- Sort the resulting pairs $(j, R[j])$ by $R[j]$ to obtain the indices of these suffixes in lexicographic order.
- To get the set of A entries needed for the unbounded approach, sample R whenever its value is 1 modulo k . This gives ranks for suffixes spaced equally in A .
- We can preallocate a sparse version of A . If we want to include the j th suffix of the text, set $A[R[j]] = j$.

But how do you get the BWT (or equivalently, the CSA) without computing A or R first?

- See practical $\Theta(n \log n)$ -time, limited-space construction in BWT_GEN program (part of BWA, BWT_SW, and others).
- This program implements the algorithm in Hon, Lam, Sadakane, and Sung, "Constructing Compressed Suffix Arrays with Large Alphabets," in *Proc. ISAAC 2003*, LNCS 2906, 240-249, 2003.
- See https://link.springer.com/chapter/10.1007/978-3-540-24587-2_26

4 Reducing Alphabet Sensitivity

The scheme I've described works great for searching DNA, but what about other kinds of text?

- *One big drawback*: space cost of occ_B is proportional to alphabet size $|\Sigma|$.
- If we're dealing with protein, we need five times more space than for DNA.

- If we're dealing with English text (case-sensitive, with punctuation), we need 20 times more space!
- (Let's not even discuss Chinese...)

Can we reduce the “alphabet sensitivity” of this data structure?

- Yes! We can reduce the storage cost to $\Theta(\log |\Sigma| n \log n)$ bits.
- Need a slightly fancy data structure called a *wavelet tree*.
- Given an alphabet Σ , create a (balanced) binary tree τ with $|\Sigma|$ leaves, and label each leaf of τ with a character in Σ .
- For any node v of τ , let $\ell(v)$ be the set of characters at leaves below v .
- At every internal node v , we create a bit string b_v defined as follows:
 - b_v contains a bit for each occurrence of a character from $\ell(v)$ in the text. i th bit corresponds to i th such occurrence.
 - Let v_ℓ, v_r be left and right children of v .
 -
$$b_v(i) = \begin{cases} 1 & \text{if } i\text{th occurrence of char from } \ell(v) \text{ is in } \ell(v_r) \\ 0 & \text{otherwise (i.e. } i\text{th occurrence is in } \ell(v_\ell)). \end{cases}$$
- (In particular, at the root of τ , the bit string marks each text character as being in the left or right side of the tree.)
- We will augment each bit string to allow the following operations:
 - $\text{nOnes}(S, i)$ – number of ones in $S[1..i]$
 - $\text{nZeroes}(S, i)$ – number of zeroes in $S[1..i]$ ($= i - \text{nOnes}(S, i)$)

How do we implement the occ function over a wavelet tree?

- Suppose we want to compute $\text{occ}_S(c, i)$.
- Trace the path from the root of τ to the leaf labeled with c .
- We will maintain at each node v the quantity $\text{occ}_S(v, i)$, the number of characters in $\ell(v)$ in $S[1..i]$.
- Trivially, at the root r of τ , we have $\text{occ}_S(r, i) = i$.
- In general, suppose we know $\text{occ}_S(v, i)$.
- If c is in $\ell(v_\ell)$, then

$$\text{occ}_S(v_\ell, i) = \text{occ}_S(v, i) - \text{nOnes}(b_v, \text{occ}_S(v, i)).$$

- (Note that the first $\text{occ}_S(v, i)$ posns in b_v describe the chars from $\ell(v)$ in $S[1..i]$. So we're just counting how many of these posns are zero.)

- Similarly, if c is in $\ell(v_r)$, then

$$\text{occ}_S(v_r, i) = \text{occ}_S(v, i) - \text{nZeroes}(b_v, \text{occ}_S(v, i)).$$

Example:

- How much does a wavelet tree cost?
- If τ is balanced, then depth is $\Theta(\log |\Sigma|)$.
- Hence, computing occ requires $\Theta(\log |\Sigma|)$ steps.
- Each step requires computing nOnes or nZeroes on a bit string. If we augment the string with a count of ones every k positions, this takes time $\Theta(k)$.
- Hence, time is $\Theta(k \log |\Sigma|)$.
- What about space?
- Observe that, at each level of the tree, the total length of all bit strings is exactly n , the length of S .
- (Indeed, every character is represented in the leaf set for one of the nodes at a given level.)
- Hence, total length of all bit strings in the tree is $\Theta(n \log |\Sigma|)$.
- If we store a count (requiring $\log n$ bits) every k positions of each bit-string, total space in bits for wavelet tree is

$$\Theta\left(\left(1 + \frac{\log n}{k}\right) n \log |\Sigma|\right).$$

- For reasonably large k , this space compares favorably to $\Theta(n|\Sigma|)$ for the naive representation of character counts, when the alphabet is big.