CSE 584A Class 11

Jeremy Buhler

February 21, 2018

1 Intro to the Burrows-Wheeler Transform

We're going to introduce a new augmentation for the suffix array.

- The *Burrows-Wheeler transform* of a string T is a string B over the same alphabet as T, such that, if A is the suffix array for T,

$$B[i] = T[A[i] - 1].$$

- (By convention, $T[0]$ is assumed to be \$, the end-of-string marker, for purposes of constructing B.)

- In other words, B is the concatenation of the characters preceding every suffix in A in sorted order.

- B is exactly the same size as T and has exactly as many of each character as T does.

- Clearly, it's trivial to derive B from A and T in time $O(n)$.

- It's equally easy to derive B directly from R, the rank array inverse to A, and T, in the same time (though not in order).

2 What's So Great About the BWT?

There's a fundamental property of the BWT that makes it very useful. It was first described by Ferragina and Manzini (2001).

- We'll show how to compute a function "rprev", defined as follows.

- Let A be the suffix array for string T, and let R be the corresponding inverse permutation of A.
- $\text{rprev}(i) = R[A[i] - 1]$.
- In words: rprev maps i to the location of the suffix $\sigma_{A[i]-1}$, i.e. the suffix beginning one character before $A[i]$, in A.
- **Example**:

What good is rprev?

- **Claim**: $\text{rprev}(R[i+1]) = R[i]$.
- **Pf**:

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

- **Cor**: If we can compute the function rprev, then we can use it to construct the suffix array for T.
- **Pf**: First, observe that $R[n]$ is always 1, since the string is assumed to end with \$, which sorts before all other characters.
- We can derive the rest of the rank array R by repeatedly applying the equation in the claim.
- If we want the suffix array A, we can simply invert R.
- **Cor**: given the function rprev and the BWT B, we can reconstruct the original text T in time $\Theta(n)$.
- **Pf**: Use rprev to get R, and then recover the string T using the equivalence

$$T[i] = B[R[i+1]].$$

Great, but where does this magical rprev thingie come from?

- Let S be a string over alphabet Σ. For each $a \in \Sigma$, define $C[a]$ to be the number of occurrences in S of characters *lexicographically less than* a.
- Moreover, let $\text{occ}_S(a, i)$ be the number of times the character $a \in \Sigma$ occurs in the prefix $S[1..i]$.
- **Thm**: for any text T with corresponding BWT B,

$$\text{rprev}(i) = C[B[i]] + \text{occ}_B(B[i], i).$$

- (**Note**: C may be defined with respect to B or T – the result is the same, since they have the same character counts.)
- **Pf**: To get $\text{rprev}(i)$, we need to know the location in A of the suffix $\sigma_{A[i]-1}$.
- Suppose that
  - $B[i] = a$
  - $\text{occ}_B(a, i) = j$
- Then
  - the character preceding $\sigma_{A[i]}$ is a.
  - $A[i]$ is the jth suffix in A (in lexicographic order) that is preceded by an a.
- Now where does the previous suffix $\sigma_{A[i]-1} = a \cdot \sigma_{A[i]}$ appear in A?
- Surely it lies among suffixes that begin with a, which occur in a contiguous block starting at position $C[a]$ in A.
- Among all these suffixes, we claim that $\sigma_{A[i]-1}$ is the jth in lexicographic order.
- Indeed, given suffixes α and β, if $\alpha < \beta$, then surely $a\alpha < a\beta$.
- Hence, the latter two suffixes are ordered the same as the former in the "a" block.
- Hence, if there are j suffixes $\alpha_1 \ldots \alpha_k$ in A preceded by "a", then $a\alpha_1 \ldots a\alpha_k$ appear in the "a" block in the same order as the originals do in A overall.
- Conclude that

$$\begin{aligned} \text{rprev}(i) &= R[A[i]-1] \\ &= C[a] + \text{occ}_B(a, i) \end{aligned}$$

  as desired.

So, what do we know about the relation of the BWT to the suffix array?

- Given only B, we can preprocess it in time $O(|\Sigma|n)$ both to compute the count array C and to construct a table of size $|\Sigma|n$ sufficient to compute $\text{occ}_B(a, i)$ in constant time for any a and i.
- This gives us the ability to compute $\text{rprev}(i)$ for any i in time $O(1)$.

- Hence, from the BWT alone, we can in *linear time* reconstruct R.
- (In fact, we don't even need to precompute more than C to rebuild R in linear time, because we can progressively compute the $\text{occ}_B(*, i)$ and rprev values in $O(\Sigma)$ working space as we are reconstructing R – details left as exercise.)
- As we saw, R and B are enough to reconstruct the text in additional linear time. If we just want the text, we need not even allocate more than constant space for R.

We conclude that *in linear time*, we can use the BWT of a text to reconstruct its suffix array (or the inverse) and the text itself.

3 Using the BWT in Search

So, where are we so far?

- The BWT is a nifty way to store a suffix array "offline" for later use, since one can easily reconstruct the array from it.
- In fact, the BWT is even better than you think for storage!
- Suffixes that occur at the corresponding locations in two instances of a repeat are typically preceded with the same character.
- Hence, because the suffix array groups suffixes from the instances of a repeat, the BWT tends to have many copies of the same character occurring near each other.
- Simple data compression can exploit this regularity; see the original tech report of Burrows and Wheeler (1994) for details.
- The result is called *block-sorting compression*, and it's the basis of the popular `bzip` compressor.
- Indeed, BWT was initially viewed as a reversible permutation of the text (as we proved) to improve its compressibility.

We'll now show how the BWT can also be used to implement pattern matching.

- Suppose we have the BWT for a text T, processed to enable constant-time rprev computations as above.
- Suppose we know that for a given pattern string P, the occurrences of P in T lie at the beginnings of suffixes $A[i..j]$.
- Where are the occurrences (if any) of the string aP, for $a \in \Sigma$?
- Let $A[\ell_a]$ and $A[r_a]$ be the first and last occurrences of P in A that are preceded by character a.
- (Clearly, $i \le \ell_a \le r_a \le j$, if ℓ_a and r_a exist.)
- **Claim**: The occurrences aP in T are precisely the beginnings of suffixes $A[\text{rprev}(\ell_a)..\text{rprev}(r_a)]$.

- **Pf**: All occurrences of aP form a contiguous set of suffixes in A.
- Since $A[\ell_a]$ is the first occurrence of P in A to be preceded by a, all other suffixes starting with P and preceded by a are lexicographically greater than it.
- Hence, all other suffixes starting with aP must occur after $A[\text{rprev}(\ell_a)]$ in A.
- A similar argument shows that all other suffixes starting with aP must occur before $A[\text{rprev}(r_a)]$.

How does this lead to an efficient pattern-matching algorithm?

- Given a pattern $P[1..m]$, here's a method to locate the contiguous range of suffixes in A starting with P.
- Algorithm will iterate *backwards* over progressively longer suffixes of P.
- Let $[i_k, j_k]$ be the range of strings in A beginning with $P[k..m]$.
- Initially, the empty suffix of P occurs at the start of *all* suffixes, so $[i_{m+1}, j_{m+1}] = [1, n]$.
- Now determine the first and last occurrences of $P[m]$ in $B[i_{m+1}..j_{m+1}]$. Let i'_m and j'_m be the positions of these occurrences.
- By our Claim, we have that $P[m..m]$ occurs at the beginnings of suffixes
$$[i_m, j_m] = [\text{rprev}(i'_m), \text{rprev}(j'_m)].$$
- Repeat the above for $k = m - 1$, then $m - 2$, and so on until we either match the entire pattern (success) or no suffix in $A[i_k, j_k]$ is preceded by $P[k - 1]$ (failure).
- **Problem**: we may spend $O(n)$ time searching for each i'_k and j'_k in the BWT!

We need one more trick to get a fast algorithm.

- Suppose again that we have a range $A[i..j]$ of matches to P, and we want to find occurrences of aP.
- Let ℓ_a be the first occurrence of P in $A[i..j]$ preceded by a.
- Then all occurrences of P at $A[i..\ell_a - 1]$ are preceded some character other than a.
- Conclude that $\text{occ}_B(a, \ell_a) = \text{occ}_B(a, i - 1) + 1$.
- By a similar argument, we have that $\text{occ}_B(a, r_a) = \text{occ}_B(a, j)$.
- The above equalities and our theorem on how to compute rprev imply that
$$\begin{aligned}\text{rprev}(\ell_a) &= C[a] + \text{occ}_B(a, i - 1) + 1\\ \text{rprev}(r_a) &= C[a] + \text{occ}_B(a, j).\end{aligned}$$
- Hence, we don't actually need to locate the first and last suffixes in a range preceded by a given a!

- In conclusion, the following algorithm finds all occurrences of P in T.

```
FIND(P[1..m])
  i ← 1
  j ← n

  k ← m
  while k > 0 do
    i ← C[P[k]] + occ_B(P[k], i − 1) + 1
    j ← C[P[k]] + occ_B(P[k], j)
    if i > j                                  ▷ no preceding P[k]'s in i..j
      return not found
    else
      k − −
  return starting positions in T of all suffixes in A[i..j]
```

What about performance?

- Algorithm runs for m iterations.
- Each computation of occ_B is constant-time, as is each lookup in C.
- Hence, each loop iteration takes constant time.
- Conclude that entire search runs in time $O(m)$ to determine the result range $A[i..j]$.
- Search to determine range bounds i and j uses nothing except C and occ_B – not even the text, the BWT, or the suffix array! This is enough to count # of hits.
- Once the range $[i..j]$ is known, we need A itself to enumerate the results, which can be done in time proportional to their number.

Conclude that *suffix arrays support $O(m)$-time exact pattern matching given only $\Theta(n)$ preprocessing.*