CSE 584A Class 13

Jeremy Buhler

February 28, 2018

1 Extension to Bidirectional BWT

The BWT is sufficient for simple pattern matching, but can it do more?

- Let A be the suffix array for T.
- Say that a suffix array interval $[i, j]$ is *P-maximal* (or *maximal for string P*) w/r to T if every suffix in $A[i..j]$ starts with P, and suffixes $A[i-1]$ and $A[j+1]$ do not.
- For each substring P of T, there is a unique P-maximal interval w/r to T.
- The usual backward matching algorithm for P starts with the ϵ-maximal interval $[1, n]$ and successively computes $P[k..m]$-maximal intervals for $m \geq k \geq 1$.

We'll now consider a bidirectional analog of maximal intervals.

- Let T^R be the reverse of a text T.
- Let B and $\tilde{B}$ be the BWTs for T and T^R, respectively.
- The pair $(B, \tilde{B})$, with each string augmented to permit $O(1)$ rprev calculations, is called the *bidirectional BWT*, or *2BWT*, of T.
- **Defn**: For any string P, the pair $([i, j]; [i', j'])$ is the *maximal bi-interval* for P w/r to T if $[i, j]$ is P-maximal w/r to T, and $[i', j']$ is P^R-maximal w/r to T^R.
- Note that $[i', j']$ is also the interval in the suffix array of T^R corresponding to all *prefixes* of T that end with the string P.
- Intuitively, a maximal bi-interval is just like a maximal interval, except we simultaneously track the substrings in T that start with a given string *and* those that end with it.

We will show that, just as the BWT lets us extend maximal intervals one character at a time, the 2BWT lets us extend maximal bi-intervals.

- **Claim**: given the maximal bi-interval for P w/r to T and the 2BWT for T, we can in constant time compute the maximal bi-interval for cP, for any $c \in \Sigma$.
- **Pf**: Let $([i, j]; [i', j'])$ be the maximal bi-interval for P.

- The backwards matching algorithm lets us compute the maximal interval $[i^*, j^*]$ for cP in constant time starting from $[i, j]$, using B.
- But how do we figure out the maximal interval for $(cP)^R = P^R c$ in T^R?
- This interval is a *subinterval* of $[i', j']$, since we've extended P^R to the right.
- To find the bounds of the subinterval, we can ask two questions:
  - How many suffixes in $[i', j']$ start with $P^R d$, for $d < c$?
  - How many suffixes in $[i', j']$ start with $P^R d$, for $d \leq c$?
- If x suffixes start with chars $< c$, and y suffixes start with chars $\leq c$, then the desired interval is $[i' + x, i' + y - 1]$.
- We can translate the above two questions into queries about $[i, j]$:
  - How many suffixes in $[i, j]$ are preceded by chars $< c$?
  - How many suffixes in $[i, j]$ are preceded by chars $\leq c$?
- Each of these quantities can be computed in $\Theta(|\Sigma|)$ time using occ queries on B.
- If we store progressive sums of counts over the chars of Σ, rather than individual counts per char, we can reduce this cost to $O(1)$.
- Conclude that, in constant time, we can compute the desired bi-interval $([i^*, j^*]; [i' + x, i' + y - 1])$. QED

The above claim has an interesting corollary...

- **Cor**: given the maximal bi-interval for P w/r to T and the 2BWT for T, we can in constant time compute the maximal bi-interval for Pc, for any $c \in \Sigma$.
- Procedure is exactly as above, except that we do the "easy" backward step w/r to T^R and the fancier forward step w/r to T.
- Hence, the 2BWT lets us refine a set of pattern matches to P by extending P *either* forward or backward, all in constant time!
- This improves on the regular BWT, which only lets us go backwards.

Note: if we use a wavelet tree to compute occ for B and $\tilde{B}$, then the cost per extension is $\Theta(\log |\Sigma|)$.

2 Relation of the 2BWT to Suffix Trees

- **Defn**: a P-maximal interval $[i, j]$ is *right-diverse* if not all suffixes in $A[i..j]$ share a common prefix strictly longer than $|P|$.
- **Defn**: a P-maximal interval $[i, j]$ is *left-diverse* if not all suffixes in $A[i..j]$ have the same previous character, i.e. $B[i..j]$ contains at least two distinct characters.

- **Claim**: a P-maximal interval w/r to T is right-diverse iff P labels an internal node of T's suffix tree.
- (Indeed, right-diversity means that there are two suffixes Pc and Pd, for $c \neq d$.)

Question: can we efficiently enumerate just the *right-diverse* maximal intervals for substrings of T? If so, we would implicitly enumerate all the internal nodes of T's suffix tree!

- Naively, we could use the backward matching algorithm on T^R to walk the entire suffix tree τ of T one character at a time.
- But this would take $\Theta(n^2)$ steps! Can we do better?
- We need to think about the *suffix links* in τ.
- For any suffix tree node x with label P, there exists a chain of $|P|$ suffix links that connects x to the root.
- Now imagine that we *invert* all the suffix links in τ.
- A node with label P has inverse links to up to $|\Sigma|$ nodes with different labels cP. (It could have fewer inverse links if not all $|\Sigma|$ possible nodes exist in τ.)
- The inverse links themselves form a tree τ', since each node has only one outgoing suffix link.
- Note that, unlike τ, τ' might have some nodes with just one child (i.e. nodes that are the target of just one suffix link.)
- If T ends with a unique character \$, then all the leaves, starting with the longest, are connected in a chain of suffix links ending at the root, and no leaf links to any other non-leaf. Let's omit (the inverse of) this chain from τ'.
- Then τ' links all the *internal* nodes of τ.
- If we can figure out a way to traverse τ', we will reach all such nodes!

The 2BWT is key to efficiently traversing τ'.

- **Lemma**: given the P-maximal bi-interval $I = ([i, j]; [i', j'])$ w/r to T, we can, in $O(|\Sigma|)$ time, determine whether the forward interval $[i, j]$ is right-diverse.
- **Pf**: If $[i', j']$ is the maximal interval for P^R, use $\text{occ}_{\tilde{B}}$ to check whether $\tilde{B}[i'..j']$ contains at least two distinct characters. QED
- We now give a recursive algorithm to traverse τ' using the 2BWT for T.
- Traversal will report only the bi-intervals corresponding to internal nodes of τ, along with their depth (i.e. lengths of their labels) in τ.

```
enumTree(I, d)
  if I is right-diverse
    for c ∈ Σ do                                  ▷ possible inverse links in τ′
      extend I backwards by char c to obtain maximal bi-interval I′
      enumTree(I′, d + 1)
    report (I, d) as a suffix tree node of depth d
```

- To traverse the whole tree τ', call enumTree($([1, n]; [1, n])$, 0).
- Right-diversity check ensures that we only report and continue traversal from internal nodes of suffix tree!
- Enumerating the set of child bi-intervals I' and checking their right-diversity costs $\Theta(|\Sigma|)$ time per tree node.
- Conclude that algorithm runs in time $\Theta(|\Sigma|n)$.

What have we achieved?

- Observe that enumTree reports the nodes of τ', and hence the *internal* nodes of τ, in order from highest to lowest depth.
- If we first process the leaves of τ (i.e. the individual suffixes of T), enumTree is actually a *postorder* enumeration of τ's nodes – it always enumerates a node after its children (which are at strictly greater depth)!
- Conversely, if we report each right-diverse interval as soon as it is found, before the extension loop, we do a *preorder* enumeration of τ.
- Neither of these enumerations is really a "traversal" of τ – they might bounce all over the suffix tree.
- But as we've seen, enumerating internal nodes can still be helpful...
- And they can be done using only about twice the space needed for pattern-matching with the BWT. This is still quite a bit less space than is needed to store even the suffix array with adjacent LCP values.

A few further notes on more space and time reduction...

- If we use wavelet trees for occ, we can reduce the cost of enumeration to $\Theta(n \log |\Sigma|)$, making it suitable even for large alphabets.
- (Not completely trivial – need to show that there are only $O(n)$ recursive calls made on non-right-diverse intervals, independent of $|\Sigma|$. In fact, can prove that there are at most n such calls!)
- It is actually possible (and practical) to traverse τ' using only the forward BWT B, instead of the 2BWT, plus an extra stack.
- See Mäkinen's book, Chapter 9 for hairy details.

3 Application: Repeat Finding

Let's return to the maximal repeats problem.

- We gave an $O(|\Sigma|^4 n)$ algorithm to enumerate, in compact form, all maximal instances of repeats of length $\geq k$ in a string T.
- This algorithm required construction of the suffix tree for T.
- Can we get away without the tree, or even the suffix array?
- We know from before that every repeat with at least one maximal instance labels an internal node of this tree, so it suffices to check all such nodes.
- If node x with label P has children $y_1 \dots y_k$ in τ', then each child y_j has a distinct label $c_j P$.
- If $I_1 \dots I_k$ are the maximal intervals w/r T corresponding to nodes $y_1 \dots y_k$, then every pair of suffixes in $I_a \times I_b$, $1 \leq a < b \leq k$ (minus their first characters) represents a *left-maximal* instance of repeat P.
- But we want instances that are *both* left- and right-maximal.
- In $O(|\Sigma|)$ time, we can use $\tilde{B}$ to divide P-maximal interval I into at most $|\Sigma|$ contiguous subintervals I^c, corresponding to suffixes starting with Pc for $c \in \Sigma$.
- For each I_j, $1 \leq j \leq k$, let's divide it into its subintervals I_j^c.
- Enumerate only pairs of suffixes from $I_a^c \times I_b^d$ for $a \neq b$, $c \neq d$.
- Because we spend only $\Theta(|\Sigma|)$ time to subdivide each node's interval, and we need only represent *non-empty* subintervals, this method is actually faster than the original: $\Theta(|\Sigma|n)$ time, plus the cost to enumerate the maximal instances.