CSE 584A Class 8

Jeremy Buhler

February 15, 2016

We're going to look at one of several linear-time algorithms for constructing the suffix array of a string. This approach is due to Kärkkäinen and Sanders (2003).

1 Important Preliminaries

- For a string S of length n, σ_i denotes ith suffix $S[i..n]$.
- Let A be a list of (not necessarily consecutive) integers $\leq n$, without repetition.
- In time $\Theta(|A|)$ and space $\Theta(n)$, can compute a (partial) mapping R from $1..n$ to A.

- $R[i]$ records position of element i in A.
- If A is a permutation of $1..n$, then R is inverse of this permutation (i.e. $R[A[i]] = i$) and is defined for all of $1..n$.
- If A is viewed as an ordering of objects (say, suffixes!) with labels in interval $1..n$, then $R[i]$ gives *rank* of object i in this ordering. (Ranks run from 1 to $|A|$).
- If we want to know relative order of objects i, j in A, we need only compare $R[i]$ and $R[j]$.
- **Defn**: when A is a suffix array for a string S, the array of ranks R for elements of A is sometimes called the *inverse suffix array* for S.
- $R[i]$ is rank of suffix σ_i in A, i.e. number of suffixes in A lexicograhically less than σ_i

We will use ranks extensively in what follows.

2 Difference Covers

We now introduce a key mathematical idea that will be used in upcoming algorithms.

- Let $Z_h = \{1, \ldots h-1\}$ be the set of positive integers $< h$.

- **Defn**: an h-difference cover is a subset $D \subseteq \{0\} \bigcup Z_h$ such that, for every $i \in Z_h$, there exist $j, k \in D$ with $i = k - j \pmod{h}$.
- In particular $D = \{0, 2\}$ is a 3-difference cover.
- One can find difference covers of size $\Theta(\sqrt{h})$ for arbitrarily large h (Colbourn and Ling 2000), but we are going to focus on the $h = 3$ case.
- **Lemma**: if D is an h-difference cover, then for any $i, j < h$, there exists a value $\delta(i, j) < h$ such that both $(i + \delta(i, j)) \bmod h$ and $(j + \delta(i, j)) \bmod h$ are in D.
- (Put another way, we can add the same amount to i and j and get two values in D.)
- Can easily verify for $h = 3$:

i	j	δ
0	1	2
0	2	0
1	2	1

3 Difference Cover Strategy for Sorting

What good are difference covers for creating a suffix array?

- Given string S and an h-difference cover D, build suffix array A from S as follows:
  1. Extract from S the subset of suffixes $S_D = \{\sigma_i \mid i \bmod h \in D\}$
  2. Sort *only* the suffixes in S_D, creating sorted array A_D.
  3. Compute R_D, the array of ranks for the suffixes in A_D.
  4. Use R_D to efficiently construct A from S.
- Step 1 is trivially $\Theta(n)$.
- We're going to punt on Step 2 until later.
- Step 3 is $\Theta(|A_D|) = \Theta(n)$ by our previous observation about building rank array from a sorted array.
- We're going to talk about Step 4 now.

How to build A from S?

1. Collect all suffixes *not* in S_D into "leftover" set S_L.
2. Sort suffixes in S_L to produce partial suffix array A_L.
3. Merge sorted suffix arrays A_L and A_D to create final A.

Let's first see how to sort S_L efficiently given R_D.

- Suppose that there exists $k < h$ such that, for *all* $i \in Z_h - D$, $(i + k) \bmod h \in D$.

- Not all difference covers have such a k, but some do.
- In particular, $h = 3$, $D = \{0, 2\}$ allows $k = 1$.
- Given such a k, proceed as follows.
- For each $\sigma_i \in S_L$, set

$$\text{key}(\sigma_i) = S[i..i + k - 1] \cdot R_D[i + k].$$

- (The rank $R_D[i + k]$ is always defined because $i + k \in D$, and so $\sigma_{i+k} \in S_D$.)
- Observe that $\sigma_i < \sigma_j$ iff $\text{key}(\sigma_i) < \text{key}(\sigma_j)$.
- (Either suffixes differ in first k chars, or their order is determined by the order of the rest of the strings, which is the order of the later suffixes from S_D.)
- Hence, if we sort suffixes of S_L by the given keys, we will lexicographically order them, producing A_L.
- We can use a radix sort, taking time $\Theta(nk) = \Theta(nh)$.

Next, let's see how to merge A_L with A_D efficiently.

- To merge two sorted lists, one can use a linear-time merge operation: repeatedly choose least element (in either list) not used so far.
- To implement this algorithm, we must be able to efficiently compare a suffix from A_L to a suffix from A_D to find which one is lexicographically least.
- Given $\sigma_i \in A_L$ and $\sigma_j \in A_D$, proceed as follows.
- Compare first $\delta(i, j)$ characters of σ_i, σ_j explicitly.
- If they are not the same, their order is determined.
- Otherwise, their order is given by relative ordering of suffixes $\sigma_{i+\delta(i,j)}$ and $\sigma_{j+\delta(i,j)}$, both of which are in S_D.
- In this case, it suffices to compare $R_D[i + \delta(i, j)]$ and $R_D[j + \delta(i, j)]$.
- Conclude that each comparison takes time $\Theta(h)$.
- Hence, entire merge takes time $\Theta(nh)$.

What is our cost so far, exclusive of computing the sorted subarray A_D?

- Assume $h = O(1)$; $h = 3$ is a good choice as noted above.
- As we said earlier, steps 1 and 3 are $\Theta(n)$.
- We can get A_L from S_L, and A from A_L and A_D, each in time $\Theta(nh) = \Theta(n)$.
- So far, so linear!

4 Subsorting Suffixes in the Difference Cover

We will describe an approach to transform the sorting problem for S_D, which consists of *non-contiguous* suffixes, into an equivalent suffix array construction problem.

- **Idea**: construct a string T such that the order of T's suffixes matches the order of S_D's elements.
- In what follows, let $m = |S_D|$.
- **First step**: bin suffixes of S_D according to lexicographic order of their first h characters.
- Suffixes have bin numbers between 1 and m.
- Suffixes with same h-prefix have same bin number
- Let B be the array of bin numbers for S_D after this h-sort.

Now to construct T itself...

- For each $x \in D$, construct a list I_x that enumerates, in order, all $i \leq n$ such that $i \equiv x \pmod{h}$.
- Form a string T_x from each I_x using the mapping

$$T_x[j] = B[\sigma_{I_x[j]}].$$

- (T_x is a string over the alphabet $\Sigma = \{1, 2, \ldots n\}$.)
- Finally, let T be the concatenation of T_x for all $x \in D$ in order.
- We can certainly construct the I_x's in total time $\Theta(m)$.
- The mappings from I_x to T_x take total time $\Theta(m)$, assuming we can look up the bin of a suffix in B in constant time. (Can be done in $\Theta(n)$ space, or $\Theta(m)$ if we are a bit clever - left as exercise.)

So what's so great about T?

- Let I_D be the concatenation of I_x for all $x \in D$ in order.
- Define pos(i) to be the position of the number "i" in I_D.
- Let τ_i be the ith suffix of string T.
- **Theorem**: For any $i, j \in D$,

$$\sigma_i < \sigma_j \text{ iff } \tau_{\text{pos}(i)} < \tau_{\text{pos}(j)}.$$

- (In other words, if we order the suffixes of T, we can infer the order of any two suffixes in S_D.)

- **Pf**: Observe that binning S_D by its first h chars induces a mapping ϕ from h-mers to integers $\leq m$ (bin #'s).
- For any two h-mers x and y,

$$x < y \text{ iff } \phi(x) < \phi(y).$$

- Consider an operation $\Phi(\sigma)$ on strings that maps each successive (non-overlapping) h-mer to its image under ϕ.
- For any two strings σ and π, surely

$$\sigma < \pi \text{ iff } \Phi(\sigma) < \Phi(\pi).$$

- Now, consider the lists I_x as defined above. Note that the indices listed in I_x are exactly h apart.
- Hence, T_x is simply $\Phi(\sigma_x)$, the h-mer-wise mapping of the string starting at position x of S.
- (As a special case, we take $I_0 = I_h$, since strings are 1-based.)
- Note that for every $i \in I_x$, $\Phi(\sigma_i)$ is a suffix of T_x.
- Conclude that, for each σ_i in S_D, $\Phi(\sigma_i)$ occurs in T.
- In particular, $\Phi(\sigma_i)$ is a prefix of $\tau_{\text{pos}(i)}$.
- Finally,

$$\begin{aligned} \sigma_i < \sigma_j \quad &\text{iff} \quad \Phi(\sigma_i) < \Phi(\sigma_j) \\ &\text{iff} \quad \tau_{\text{pos}(i)} < \tau_{\text{pos}(j)}. \end{aligned}$$

  where the last step follows because σ_i and σ_j are of different lengths and so will be correctly ordered by the time we reach the end of the shorter one, regardless of what comes afterwards in T. QED

5 Exploiting T to Create a Suffix Array

We've established that we can create T from S_D in time $\Theta(mh)$, and that the order of T's suffixes tells us something about the order of S_D's suffixes.

- How do we exploit T to sort S_D?
- First, construct a suffix array A_T for T.
- Then, compute the rank array R_T corresponding to A_T.
- Now let R_D be array of ranks of each suffix in S_D.
- Observe that for each $i \in S_D$,

$$R_D[i] = R_T[\text{pos}(i)].$$

  (Follows immediately from our theorem.)

- Finally, we can create sorted array A_D of suffixes in S_D by inverting rank array R_D.

What does this cost?

- Let's punt for a second on construction cost of A_T.
- Computing R_T from A_T is $\Theta(m)$.
- Computing R_D from R_T is $\Theta(m)$, provided we've got the mapping "pos" in constant time.
- Computing A_D from R_D is $\Theta(m)$ as well.

6 Putting It All Together: Recursive Strategy

KSBuild(S)
  extract S_D from S
  compute T from S_D
  $A_T \leftarrow$ KSBuild(T)
  $R_T \leftarrow$ rank array for A_T
  compute R_D, A_D from R_T

  $S_L \leftarrow S - S_D$
  sort S_L into array A_L using order induced by R_D
  merge A_L with A_D to create final array A
  return A

- (Above code skips the base case – directly sort strings of up to some constant size.)
- Every line *except* the recursive call to KSBuild is either $\Theta(m)$ or $\Theta(n)$.
- Let $C(n)$ be cost of calling KSBuild on string of length n.
- Then we have
$$C(n) = C(m) + \Theta(n)$$
(since $m < n$).
- If we set $h = 3$ and $D = \{0, 2\}$ (the so-called "DC3 algorithm"), then $m = 2n/3$.
- Hence, we are left with recurrence
$$C(n) \leq C(2n/3) + dn$$
for some constant d.
- Solution to this recurrence is given by
$$C(n) = \sum_{i=0}^{\log_{3/2} n} d \cdot \left(\frac{2}{3}\right)^i n$$
which is $< 3dn$.
- Conclude that recursive algo is in fact $\Theta(n)$!
- For larger h, m is an even smaller fraction of n.