6.897: Advanced Data Structures Spring 2005

Lecture 22 — April 28, 2005

Prof. Erik Demaine *Scribe: Nick Harvey*

1 Overview

In the previous lecture, we saw how to represent binary tries that support a variety of efficient query operations, but do not use much space. The data structures that we saw were ***succinct***: the space they require is

$$\text{information theoretic optimal} \; + \; o(\text{information theoretic optimal}).$$

In this lecture, we will present low-space data structures for representing suffix arrays and suffix trees. These data structures are ***compact***: the space they require is

$$O(\text{information theoretic optimal}).$$

The suffix array data structure that we present is due to Grossi and Vitter [1]. It uses a recursive construction that inflates the alphabet size, much like the the suffix array construction that we saw in Lecture 18. Building on this, we will construct a low-space suffix tree by augmenting this suffix array with an additional tree structure. This construction is due to Munro, Raman and Rao [2], and it relies on techniques that we saw in the previous lecture.

2 Compact Suffix Arrays

Let T be a text string over an alphabet Σ. Let P be a pattern that we wish to find in text T. The data structure of [1] achieves the following space and time bounds.

$$\begin{array}{ll} \text{Space:} & (\frac{1}{\varepsilon} + O(1)) \cdot |T| \cdot \log|\Sigma| \text{ bits} \\ \text{Query:} & O\Big(\frac{|P|}{\log_{|\Sigma|}|T|} + \log^{\varepsilon}_{|\Sigma|}|T| \cdot |\text{output}|\Big) \text{ time} \end{array}$$

Here ε is an arbitrary constant that may be used to achive a tradeoff between time and space. These bounds are quite close to optimal, except for the $(1/\varepsilon + O(1))$ factor in the space bound and the $\log^{\varepsilon}_{|\Sigma|}|T|$ factor in the query bound. The structure that we present herein is somewhat simplified. We will assume that $\Sigma = \{0, 1\}$ and we will only achieve a bound of $O(|P| \cdot \log^{1+\varepsilon}|T|)$ for the query time.

2.1 Recursive Construction

Let us say that our given text T has length n. We use a recursive construction. At level $k \geq 0$, we have a text T_k of length n_k over alphabet Σ_k, and a suffix array SA_k.

Base Level: At level $k = 0$, we set T_0 to be our original text T, and hence $n_0 = n$ and $\Sigma_0 = \Sigma$. Let us assume for now that SA_0 is an ordinary suffix array. In other words, suppose that we have ordered the suffixes of T_0 by their string value. Then the i^{th} suffix under this ordering starts as position $SA_0[i]$ in string T_0. Of course, we cannot afford to store such a large array: each entry would require roughly $\log n$ bits, so the total space required would be $n \log n$, which exceeds our goal of $O(n)$. We will explain later how to reduce the space of SA_0.

Higher Levels: Given the structure at level k, we now explain how to build the structure at level $k+1$. First, we increase the alphabet size by setting $\Sigma_{k+1} = \Sigma_k \times \Sigma_k$. The text T_{k+1} consists of consecutive pairs of symbols of T_k.

$$T_{k+1} \;=\; < (T_k[2i], T_k[2i+1]) \quad \text{for } i = 0, 1, \ldots, n_k/2 >$$

The size of the text is clearly halved, so $n_{k+1} = n_k/2 = n/2^{k+1}$. Again, we assume for now that SA_{k+1} is an ordinary suffix array on the text T_{k+1}.

Note that the suffix arrays SA_k and SA_{k+1} are somewhat related. If we delete all odd values from SA_k and divide the remaining values by 2 then the resulting array is precisely SA_{k+1}. Let us illustrate this relationship with an example. Let $T = T_0 = 01100111$ and $T_1 = (01)(10)(01)(11)$. The corresponding suffix arrays are as follows.

SA_0	Suffix of T_0
3	00111
0	01100111
4	0111
2	100111
1	1100111
5	111
6	11
7	1

SA_1	Suffix of T_1
0	(01)(10)(01)(11)
2	(01)(11)
1	(10)(01)(11)
3	(11)

This relationship between SA_k and SA_{k+1} can be exploited to reduce the size of each entry to just a single bit. Let's first consider the "even" entries of SA_k (i.e., the ones that contain even values). Since these entries are also represented in SA_{k+1}, intutively we just "mark" them to indicate that this is the case. How can we determine which entry in SA_k corresponds to which entry in SA_{k+1}. The key observation is that the even entries in SA_k are in the same order as their corresponding entries in SA_{k+1}. So the suffix of T_k denoted by the i^{th} even entry in SA_k is precisely $T_{k+1}[SA_{k+1}[i] :]$. Refering to our example above, the entry 2 in SA_0 is the third even entry. Its corresponding string is the same as the third string in SA_1.

From this discussion, it follows that the query operations we need to support are *rank* on a static n_k-bit string. The previous lecture presented the Jacobson structure which supports this operation in $O(1)$ time while using only $o(n)$ bits of additional space. Using this structure, we define the following operation.

$$\textit{even-rank}_k(i) \;=\; \text{the number of even values in the set } \{SA_k[1], \ldots, SA_k[i]\}.$$

Now let's consider how to handle the odd entries of SA_k. These correspond to strings that are not directly represented in SA_{k+1}. However, if we consider the suffix that starts one character later,

this suffix is in SA_{k+1}. Given this even suffix, it is easy to recover the original odd suffix. So, what we need is a map from each odd entry $SA_k[i]$ to the entry containing the value $SA_k[i]+1$. More formally, we define the following operation.

$$\textit{even-succ}_k(i) = \begin{cases} i & \text{if } SA_k[i] \text{ is even} \\ j & \text{if } SA_k[i] \text{ is odd and } SA_k[j] = SA_k[i]+1 \end{cases}$$

We will discuss the implementation of $\textit{even-succ}_k$ in the following section. Given the two preceding operations, we can compute SA_k recursively as follows.

$$SA_k[i] \;=\; 2 \cdot SA_{k+1}[\textit{even-rank}_k(\textit{even-succ}_k(i))] - \{1 \text{ if } SA_k[i] \text{ is odd}\}$$

When does this recursive construction stop? If $n_k \leq n/\log n$ then we can afford $\log n$ bits on each entry of the suffix array without exceeding our goal of $O(n)$ space. Since $n_k = n/2^k$, we see that we can stop when $2^k = \log n$, i.e., $k = \log\log n$.

Let us now analyze the space required by this scheme. At level k, the bitvector to represent the suffix array takes only n_k bits. The rank-computation structure at level k takes only $o(n_k)$ bits. Summing over all levels, the space required by these two structures is is $\sum_k O(n_k) = O(n)\sum_k 1/2^k \leq O(n)$.

2.2 The Even-Successor Operation

How can we implement $\textit{even-succ}_k$? The high-level idea is to simply write down all the answers in order in a table R_k. A naïve implementation of this approach would use $\log n$ bits for each entry of R_k, and hence $O(n_k \log n)$ bits in total at level k. With this approach, even the base level $k=0$ would require $O(n \log n)$ bits, which exceeds our goal of $O(n)$. In this section, we will reduce the space to $O(n)$ bits per level by storing the table R_k implicitly using more compact structures.

Define $\textit{odd-rank}_k(i) = i - \textit{even-rank}_k(i)$. So then if $SA_k[i]$ is odd, we can compute

$$\textit{even-succ}_k(i) \;=\; R_k[\textit{odd-rank}_k(i)].$$

It turns out that the table R_k has some structure that allows us to store it more compactly. Let us now consider how the entries in the table R_k are ordered. We now characterize the ordering of R_k in several different ways, imagining i as varying from 1 to $n_k/2$.

1. By construction, the entries of R_k are ordered in the same manner as the (odd) entries in SA_k.
2. Since SA_k is a suffix array, its entries are ordered by the lexicographic ordering of the suffixes $T_k[SA_k[i]:]$.
3. Peeling off the first character of each suffix, we see that this order is the same as for the pair $(T_k[SA_k[i]], T_k[SA_k[i]+1:])$.
4. Since $SA_k[i]$ is odd, the definition of $\textit{even-succ}_k$ implies that $SA_k[i]+1 = SA_k[\textit{even-succ}_k(i)]$. Hence the ordering is the same as for the pair $(T_k[SA_k[i]], T_k[SA_k[\textit{even-succ}_k(i)]:])$.

5. Finally, this ordering is the same as for the pair $(T_k[SA_k[i]], \textit{even-succ}_k(i))$. This holds for the same reason as (2.): the quantity $T_k[SA_k[i]:]$ is ordered in the same manner as the entries of SA_k.

This sequence of equivalent orderings is rather abstruse, but the punch line is quite simple. The first few entries of the table R_k all correspond to suffixes that start with the first symbol in Σ_k. These entries are all in increasing order. The next few entries of R_k all correspond to suffixes that start with the second symbol in Σ_k, and these entries are also in increasing order. This pattern continues throughout R_k, and leads to our efficient represent of R_k.

Imagine representing R_k as several distinct subtables $R_{k,\sigma}$, where each table contains information about suffixes that start with the same symbol $\sigma \in \Sigma_k$. As we have just observed, the elements of each table $R_{k,\sigma}$ are in increasing order. Therefore we can store each table efficiently using ***unary differential encoding***. The idea is to store a bitstring B_σ of the following form

$$B_\sigma \;=\; \underbrace{00\ldots0}_{R_{k,\sigma}[1]} \quad 1 \quad \underbrace{00\ldots0}_{R_{k,\sigma}[2]-R_{k,\sigma}[1]} \quad 1 \quad \underbrace{00\ldots0}_{R_{k,\sigma}[3]-R_{k,\sigma}[2]} \quad 1 \quad \ldots$$

To query this bitstring efficiently, we also store *rank* and select structures introduced in the previous lecture. To find the value of $R_{k,\sigma}[i]$, we use *select* to find the i^{th} 1 in the bitstring, then use *rank* to count the number of 0s preceding it. How much space does this bitstring take? We can determine the length of the bitstring by counting the number of 0s and 1s separately. The total number of 0s in the bitstring is exactly the largest value in $R_{k,\sigma}$. Each entry of $R_{k,\sigma}$ is a value of $\textit{even-succ}_k$, which is upper bounded by n_k. The number of 1s is simply the number of entries in $R_{k,\sigma}$. Summing over all values of σ, the total length of the bitstrings is therefore

$$(\text{total \# of 0s}) + (\text{total \# of 1s}) \;=\; n_k + |\Sigma_k| \cdot n_k \;=\; (1 + 2^k)n_k \;\leq\; 2n.$$

We showed in the previous lecture that the *rank* and *select* structures asymptotically take less space than the bitstring. Thus the total amount of space for the tables $R_{k,\sigma}$ is $O(n)$.

Remaining details:

- Given a value i, how can we decide in which table $R_{k,\sigma}$ to look? The appropriate table is given by $\sigma = T_k[SA_k[i]]$. So to find this correct value of σ, we must store an additional table Q_k where $Q_k[i] = T_k[SA_k[i]]$. The size of this table is only $|\Sigma_k| \cdot n_k = n$.
- Suppose that $SA_k[j]$ is the largest entry in SA_k and furthermore it is odd. In this case, $\textit{even-succ}_k(j)$ is undefined. However, we can easily handle this case by storing the entry $SA_k[j]$ explicitly. This does not affect the claimed bounds on the space.
- To map between the indices of R_k and the indices of $R_{k,\sigma}$, we need to know, for each σ, the index in R_k of the first entry stored in $R_{k,\sigma}$. We store this value in a location $r_{k,\sigma}$.

Summary: Let us now summarize the implementation of the *even-succ*$_k$ operation.

even-succ$_k(i)$**:**

Set $i' = \textit{odd-rank}_k(i)$.

Set $\sigma = Q_k[i']$.

Set $x = \textit{one-select}(B_\sigma, i' - r_\sigma)$.

Set $y = \textit{zero-rank}(B_\sigma, x)$.

Return y.

The *even-succ*$_k$ operation, and hence the computation of SA_k, takes only $O(|\Sigma_k|) = O(2^k)$ time. Since the recursion proceeds for $\log\log n$ levels, the total query time required is $\sum_{k=0}^{\log\log n} O(2^k) = O(\log n)$.

2.3 Reducing Space

The structure that we have described thus far uses $O(n)$ space per level and has $O(\log\log n)$ levels. We now use a simple idea to reduce the space even further: don't store every level. The consequence of this idea is that we will have to perform a more expensive search operation as we recurse down to lower-level structures.

Set $\ell = \log\log n$, choose $\varepsilon \in (0, 1]$, and suppose that we only store structures at levels $0, \varepsilon\ell, 2\varepsilon\ell, \ldots, \ell$. Our recursive structure will therefore need to relate the suffix arrays $SA_{k\varepsilon\ell}$ and $SA_{(k+1)\varepsilon\ell}$. The construction is largely the same as before. We describe here the main differences. Previously the entries of SA_k that were stored in SA_{k+1} were the even entries. Now the term "even entries" should be taken to mean "entries that appear in $SA_{(k+1)\varepsilon\ell}$". Whereas previously 1/2 the entries of SA_k were even, now only $1/2^{\varepsilon\ell} = \log^\varepsilon n$ entries are even. Instead of defining the *even-succ*$_k$ operation, we now define the operation

$$\textit{succ}_{k\varepsilon\ell}(i) = \begin{cases} i & \text{if } SA_k[i] \text{ is "even"} \\ j & \text{if } SA_k[i] \text{ is "not even" and } SA_k[j] = SA_k[i] + 1 \end{cases}$$

To compute $SA_{k\varepsilon\ell}$ is now to repeatedly call $\textit{succ}_{k\varepsilon\ell}$ until we find an even entry. More precisely, we use the following procedure.

Compute-$SA_{k\varepsilon\ell}(i)$**:**

Set $x = 0$.

While i is not even

  Set $i = \textit{succ}_{k\varepsilon\ell}(i)$.

  Set $x = x + 1$.

End While

Set y to be the one-rank of i in the bitvector representing $SA_{k\varepsilon\ell}$.

Set $z = \textit{Compute-}SA_{(k+1)\varepsilon\ell}(y)$.

Return $(\log^\varepsilon n) \cdot z - x$.

The query time is therefore increased by the time to execute the above while-loop. Since the loop executes at most $\log^{\varepsilon} n$ times, the total query time is $O(\log^{1+\varepsilon} n)$. Since the structure now only stores $1/\varepsilon$ levels, the space required is $O(n/\varepsilon)$ bits.

3 Compact Suffix Trees

In this section, we explain how any compact suffix array can be augmented with a linear number of bits to obtain a suffix tree.

A suffix tree containing all n suffixes of a string is stored as a binary trie on $2n+1$ nodes. Binary tries are typically represented as binary trees where the edges of the tree are augmented with "skip" values. These skip values indicate how many non-branching nodes of the tree have been coalesced into a single edge. Equivalently, the skip values indicate how many characters of the text are shared by all strings in the subtree beneath this edge.

Using our construction from the previous lecture, a binary tree on $2n+1$ nodes can be represented using only $4n+o(n)$ bits. Unfortunately, representing the skip values might take much more space. We will now show how the skip values can be computed without storing them explicitly.

As we descend the tree, we can keep track of our ***letter-depth*** (i.e., the sum of the skip values from the root). Consider an edge of the tree from node x to node y. As we traverse this edge, we must infer its skip value. As explained above, this is just the longest-common prefix length of all nodes below y minus the letter depth at node x. Fortunately, we don't need to examine all nodes in y's subtree to compute the longest-common prefix, it suffices to compare the *first* string and the *last* string in the subtree. In the last lecture, we showed that the left-most and right-most leaves of any subtree can be found in $O(1)$ time. Given the index of these leaves, we look them up in the suffix array to find their corresponding locations in the text. Once their locations are found, we compute their longest-common prefix by simply comparing the characters one-by-one.

Analyzing the query time of this scheme is straightforward. Only $O(1)$ suffix array queries are made for each edge traversed in the suffix tree. When traversing an edge of skip value v, only $O(v)$ work is required to perform the longest-common prefix comparison. Thus the time to query a pattern P is

$$O(|P| \cdot (\text{suffix array query time}) + |\text{output}|).$$

3.1 Reducing Space

In this section, we briefly sketch an approach to reduce the space used by the suffix tree to $O(n/\sqrt{\log n})$. As before, we assume that we have an underlying structure implementing a suffix array.

First we build the suffix array as in Section 2 for the original text T. Next, we imagine building an ordinary suffix tree $\mathcal{T}$ for text T. Set $b = \frac{1}{2}\sqrt{\log n}$. Next, imagine walking through the suffix array in order and selecting every b^{th} string. We build a suffix tree $\mathcal{T}'$ on this smaller set of strings as in the previous section. The space required for this smaller suffix tree is clearly $O(n/b)$ bits.

The two trees $\mathcal{T}$ and $\mathcal{T}'$ differ in that $\mathcal{T}'$ is missing certain contiguous ranges of b leaves, and hence missing certain subtrees as well. Let x be a leaf that is present in $\mathcal{T}$ but missing in $\mathcal{T}'$. What

happens when we search for the string $P = x$ in $\mathcal{T}'$? The search will terminate prematurely, ending at the first ancestor y of x that is in $\mathcal{T}'$. Our goal at this point is to find the "nearest" leaf to node x. For simplicity, assume that the left child of y is the next node on the path from y to x. Then the nearest leaf to x is either y (if y is a leaf), or the left-most leaf in y's right subtree. As observed above, the suffix tree $\mathcal{T}'$ can find this left-most leaf in $O(1)$ time.

Suppose that this nearest leaf is the i^{th} leaf in $\mathcal{T}'$. Then we know that the only suffixes that can match our pattern P are found in our suffix array between the $(i-1)^{\text{th}}$ leaf and the i^{th} leaf. By construction, this range consists of at most b suffixes. It remains to find which of these b suffixes matches the pattern. To accomplish this, we will use a lookup-table scheme.

Along with our suffix tree $\mathcal{T}'$, we will store a lookup-table as follows. For every b bitstrings of length b and every pattern of length b, we store a bitvector of length b indicating which bitstrings matched the pattern. (Note: our actual pattern P's length may greatly exceed b, but this lookup-table is still useful.) The total size of this table is

$$\begin{aligned}
&(\#\text{ of entries}) \cdot (\text{size of entries}) \\
&\quad = \left(2^{(\#\text{ of bitstrings})\cdot(\text{length of bitstrings})} \cdot 2^{(\text{length of pattern})}\right) \cdot (\text{size of entries}) \\
&\quad = 2^{b^2+b} \cdot b \;=\; 2^{b^2+b+\log b} \;\leq\; 2^{\frac{1}{4}\log n + \sqrt{\log n} + \log\log n} \;\leq\; 2^{\frac{1}{2}\log n} \;=\; \sqrt{n}.
\end{aligned}$$

Recall that after our search in $\mathcal{T}'$, we have a range of b suffixes that may match our pattern. For each of those b suffixes, we extract their first b symbols from the text. Since $b \leq \log n$, this can be done using word-operations in $O(b)$ time. We combine these b bitstrings with the first b symbols of our pattern and examine the corresponding position in the lookup-table. This gives us a bitvector telling us which subset of the bitstrings match our pattern. We then repeat this process on that subset of bitstrings, extracting their next b symbols from the text and comparing them with the next b symbols from our bitstring. The process repeats until we reach the end of our pattern or no bitstrings are left.

Let us now analyze the space and query time of this suffix tree. Assume that a query returns a single match. As observed above, the total space required is $O(n/b + \sqrt{n}) = o(n)$. The time required for the search in $\mathcal{T}'$ is still $O(|P| \cdot (\text{suffix array query time}))$. The subsequent lookup process requires at most $O(|P|/b)$ steps, each requiring $O(b)$ time, which is dominated by the search time in $\mathcal{T}'$.

References

[1] R. Grossi, J. Vitter, *Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract)*, ACM Symposium on Theory of Computation, 397–406, 2000.

[2] J. I. Munro, V. Raman, S. S. Rao, *Space Efficient Suffix Trees*, Journal of Algorithms, 39(2):205-222, 2001.