6.851: Advanced Data Structures Spring 2007

Lecture 22 — 7 May

Prof. Erik Demaine *Scribe: Martí Bolívar*

1 Overview

In the last lecture we introduced the concept of implicit, succinct, and compact data structures, and gave examples for succinct binary tries, as well as proving a bijection between binary tries, rooted ordered trees, and balanced parenthesis expressions. Succinct data structures were introduced which solve the *rank* and *select* problems.

In this lecture we expand slightly on our previous discussion of succinct binary tries, and introduce compact data structures for suffix arrays and suffix trees.

2 More on Binary Tries

Note that a leaf in a balanced-parenthesis representation of a binary trie is the string ()). As shown in a paper by Munro, Raman, and Rao [1], the following queries on leaves can be implemented in constant time using a succinct data structure.

leaf-rank(n) The number of leaves to the left of node n; denoted $rank_{())}(n)$.

leaf-select(i) The i^{th} leaf; denoted $select_{())}(i)$.

leaf-count(n) Number of leaves in the subtree of node n; equal to $rank_{())}(\text{matching) of parent}) - rank_{())}(n)$.

leftmost leaf in subtree of n Equal to $select_{())}(rank_{())}(n))$.

rightmost leaf in subree of n Similar to above.

3 Minisurvey

Following is a small survey of results on compact suffix arrays. Recall that a compact data structure uses $O(n)$ bits, where n is the information-theoretic optimum.

Grossi and Vitter 2000 [2] Suffix array in

$$(\frac{1}{\varepsilon} + O(1))|T| \lg |\Sigma|$$

bits, with query time

$$O(\frac{|P|}{\log_{|\Sigma|}^{\varepsilon} |T|} + |\text{output}| \log_{|\Sigma|}^{\varepsilon} |T|)$$

We will follow this paper fairly closely in our discussion today.

Ferragina and Manzini 2000 [3] This technique is known as the FM index. Space is

$$5H_k(T)|T| + O(\frac{|T|}{\lg |T|}(|\Sigma| + \lg\lg |T|) + |T|^{\varepsilon}|\Sigma|2^{|\Sigma| \lg |\Sigma|})$$

bits, for all k, where $H_k(T)$ is the k^{th}-order empirical entropy, or the regular entropy conditioned on knowing the previous k characters. Query time is

$$O(|P| + |\text{output}| \lg^{\varepsilon} |T|).$$

The analysis of the FM index is tricky; in particular, the paper does not claim the above bounds.

Sadakane 2003 [4] Space in bits is

$$\frac{1}{\varepsilon}H_0(T)|T| + O(|T| \lg\lg |\Sigma| + |\Sigma| \lg |\Sigma|),$$

and query time is

$$O(|P| \lg |T| + |\text{output}| \lg^{\varepsilon} |T|).$$

Note that this bound is more like a suffix array, due to the multiplicative log factor.

Grossi, Gupta, Vitter 2003 [5] This is the only known succinct result. Space in bits is

$$H_k(T)|T| + O(|T| \lg |\Sigma| \frac{\lg\lg |T|}{\lg |T|}),$$

and query time is

$$O(|P| \lg |\Sigma| + \lg^{o(1)} |T|).$$

4 Compressed suffix arrays

As a warm-up problem, we'll consider compressed suffix arrays. In the next section, we will build upon the discussion here in a compact suffix array construction. Henceforth, we shall assume that $|\Sigma| = 2$, i.e. a binary alphabet. The words "even" and "odd" refer to the index of a character in T (recall that suffix array elements contain indices into T denoting the beginning of the suffix).

4.1 Intuition

We will follow the three-way divide and conquer suffix array construction discussed in lecture 9, but modify it to be a two-way division. The recursion is as follows.

start The initial text, T_0, is T; the initial size, n_0, is n, and the initial suffix array, SA_0, is SA, the suffix array of T. We'll define $SA[i]$ as the index in T where the i^{th} suffix begins.

step $T_{k+1} =< (T_k[2i], T_k[2i+1]) >$, for $i = 0, 1, ..., n/2$; $n_{k+1} = n_k/2$ (and hence after k recursions, we'll have a text of length $n/2^k$ on an alphabet of size 2^{k+1}); $SA_{k+1} = (1/2)\cdot$extract even entries of old array SA_k.

Now, we obviously can't actually start where we say we do, since that would require knowing SA_0, which is exactly what we're trying to build. So conceptually, we'll walk *up* the recursion tree from the bottom, which consists of the trivial case in which the entire text T's suffix array is constructed as if T were written as a single letter.

4.2 Crawling Up the Recursion Tree

Since we're recursing "backwards", we need a way to represent SA_k using SA_{k+1}. We'll use the following queries to accomplish this efficiently:

$evensucc_k(i)$ The "even successor" of i, defined as i if $SA_k[i]$ is even, and j if $SA_k[i] = SA_{k+1}[j] - 1$. One of these will be the case, since SA_{k+1} is SA_k with every pair of letters joined together as a single letter on an alphabet of twice the original size.

$evenrank_k(i)$ The "even rank" of i, or the number of even values in $SA_k[: i]$ (using slice notation as in Python [6]). This equals the number of even suffixes preceding the i^{th} suffix.

Additionally, we'll let $SA_k[i] = 2SA_{k+1}[evenrank_k(evensucc_k(i))]$, minus 1 if $SA_k[i]$ is odd.

So constant time per operation on the above queries reduces a query on SA_k to a query on SA_{k+1} in constant time. Hence, a query on SA_0, the array we're trying to build, will take $O(l)$ time if we recurse l times. Only $l = \lg \lg n$ recursions will be necessary, for since we halve the size of the text on each recursion, this will reduce the text to size $n_l = n/\lg n$. We then use a normal suffix array, which will use $O(n_l \lg n_l) = O(n)$ bits of space, and thus be compressed.

4.3 Construction

We will now construct a data structure that answers even-successor and even-rank queries in constant time. We'll begin with an auxiliary query, is-even:

$$iseven_k(i) = \begin{cases} 1 & \text{if} \quad SA_k[i] \quad \text{even,} \\ 0 & \text{else} \end{cases}$$

We can imagine storing this as an n_k-bit vector. Since $n_{k+1} = n_k/2$, the total number of bits is geometric in $n_0 = n$, so we'd need $O(n)$ bits to do it this way, which won't work. But for now, let's pretend that this is how we implement this.

Then we can implement $evenrank_k(i)$ with a rank structure on our imaginary bit vector in $o(n)$ space.

Doing $evensucc_k(i)$ is trivial in the case that $SA_k[i]$ is even; there are $n_k/2$ such values. Intuitively, we could just store the values of j for odd values of $SA_k[i]$. Note that we can't actually write them down, because that would require $n_k \lg n_k$ bits.

Whatever data structure we use, let's order the values of j by i; that is, if we're pretending to store the values of j in an array called $odds$, we'd like $evensucc_k(i) = odds[oddrank_k(i)]$, where $oddrank_k(k) = i - evenrank_k(i)$.

Why is it useful to order the j's by i? Well, that's just ordering by suffix in the suffix array, which is ordering by the odd suffix $T_k[SA_k[i] :]$, or ordering by $(T_k[SA_k[i]], T_k[SA_k[i]+1 :]) = T_k[SA_k[evensucc_k(i)] :]$. This in turn is equivalent to ordering by $(T_k[SA_k[i]], evensucc_k(i))$.

Now recall that we're trying to create a data structure for answering $evensucc_k(i)$ queries. So ordering the j's by i is equivalent to sorting by i and the values of j! That is to say, the values of j are mostly in sorted order. So we will be storing pairs of letters in lexical order.

This means we need a clever way of storing a sorted array of $n_k/2$ values v_i, each of which is $2^k + \lg n_k$ bits long. The 2^k follows since at level k, each letter j will require 2^k bits; similarly, each value i requires $\lg n_k$ bits.

The desired clever trick is to store the leading $\lg n_k$ bits of each v_i in unary differential encoding:

$$0^{\text{lead}(v_1)}10^{\text{lead}(v_2)-\text{lead}(v_1)}1...$$

Where $\text{lead}(v_i)$ is the value of the leading $\lg n_k$ bits of v_i as an unsigned integer. That is to say, we write down the difference between the value of v_i and v_{i-1}'s leading bits in unary, then write a 1, and repeat.

There will then be $n_k/2$ ones and at most $2^{\lg n_k} = n_k$ zeros, and hence at most $(3/2)n_k$ bits total used for this encoding. Again by the geometric nature of successive values of n_k, this will require $O(n)$ bits total, so the overall data structure is still compressed.

Note that this also gets you random access – the leading bits of v_i have value equal to $\text{rank}_0(\text{select}_1(i))$.

The remaining 2^k bits can be stored in the obvious way, in an array, as that will use $2^k \frac{n_k}{2} = 2^k \frac{n/2^k}{2} = \frac{n}{2}$ bits, for total of $n/2 + 3n_k/2 + o(n_k)$ bits. This completes the construction of a compressed suffix array.

5 Compact suffix arrays

The problem with our compressed suffix array construction is that its $\lg \lg n$ levels require $n \lg \lg n$ space. As in the last lecture on binary tries, we would prefer to reduce the number of recursion levels to a constant. This would give us linear space for a time tradeoff.

To accomplish this, we will store only $1/\varepsilon + 1$ levels of recursion, namely those values of k equal to

$$0, \varepsilon l, 2\varepsilon l, ..., l = \lg \lg n.$$

In essence, we are clustering $2^{\varepsilon l}$ letters in a single stroke. We now need to be able to *jump* εl levels at once. We are not able to do this in constant time.

5.1 Level jumping

In order to present a method of representing $SA_{k\varepsilon l}$ with $SA_{(k+1)\varepsilon l}$, the concept of "even" and "successor" need to be generalized from the compressed construction.

In particular, the previous notion of an index i being "even" in the text T will henceforth mean "even for εl recursions". This buys us a generalized notion of $evenrank_k(i)$ as well, in the obvious way. We'll define successor similarly, with $evensucc_k(i) = j$, where $SA_{k\varepsilon l}[i] = SA_{k\varepsilon l}[j] - 1$.

Using these definitions, computing $SA_{k\varepsilon l}[i]$ is as follows:

- Follow the successor pointer repeatedly until index j is at the next level down, namely $(k+1)\varepsilon l$.
- Recurse: $SA_{(k+1)\varepsilon l}[evenrank_{k+1}(j)]$
- Multiply by $2^{\varepsilon l}$, as this, modulo rounding errors, is how many letters we clustered per recursion level. We then correct the round-off error by subtracting the number of calls to successor in the first step.

The runtime is then clearly linear in the number of times we call successor in the first step. This equals $2^{\varepsilon l}$, because the successor walking is done in text space, not suffix space. That is, each recursion level in effect halves the number of letters in T εl times.

5.2 Analysis

From the arguments outlined in the previous subsection, search time is $2^{\varepsilon l} \lg\lg n = \lg^{\varepsilon} n \lg\lg n = O(\lg^{\varepsilon'} n)$. (We have introduced a new variable ε', which can be made arbitrarily small by appropriate choice of ε).

Space is $O(n)$: we use the same unary differential encoding for successor as in the compressed construction. This is a linear number of bits per level, but we have a (large) multiplicative constant factor due to level jumping. Nevertheless, with a constant number of levels, space is linear overall.

This gets us a compact suffix array.

Open problem: is it possible to achieve constant query time in linear space?

6 Suffix trees

Suffix arrays are somewhat troublesome due to the log factor paid to search. Suffix trees eliminate this problem. An algorithm for creating a compact suffix tree given a compact suffix array was also given in [1]. This converts a suffix array which uses m bits into a suffix tree which uses $o(m)$ bits.

6.1 Implementation

This is how it's done.

6.2 Analysis

6.3 Improvement

It is also possible to improve this, creating a succinct suffix tree given a suffix array. This algorithm is complicated; the reader is referred to [1] for the details.

References

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

[2] R. Grossi and J. S. Vitter, *Compressed suffix arrays and suffix trees with applications to text indexing and string matching*, Thirty-Second Annual ACM Symposium on Theory of Computing, vol. STOC,pp. 397-, 2000.

[3] P. Ferragina and G. Manzini, *Indexing Compressed Text*, Journal of the ACM, Vol. 52 (2005), 552-581.

[4] K. Sadakane, *New text indexing functionalities of the compressed suffix arrays.* Journal of Algorithms, 48(2): 294-313 (2003).

[5] R. Grossi, A. Gupta, J. S. Vitter, *High-order entropy-compressed text indexes*, SODA 2003: 841-850.

[6] G. van Rossum, *Python Tutorial*, http://docs.python.org/tut/node5.html#SECTION005140000000000000000