6.851: Advanced Data Structures Spring 2010

Lecture 23 — May 4, 2010

Prof. Erik Demaine *Scribe: Martí Bolívar, heavily edited by Sarah Eisenstat*

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 showing the equivalence of 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 introduce compact data structures for suffix arrays and suffix trees. Recall the problem that we are trying to solve. Given a text T over the alphabet Σ, we wish to preprocess T to create a data structure. We then want to be able to use this data structure to search for a pattern P, also over Σ.

A suffix array is an array containing all of the suffixes of T in lexicographic order. In the interests of space, each entry in the suffix array stores an index in T, the start of the suffix in question. To find a pattern P in the suffix array, we perform binary search on all suffixes, which gives us all of the positions of P in T.

2 Survey

In this section, we give a brief survey of results for compact suffix arrays. Recall that a compact data structure uses $O(OPT)$ bits, where OPT is the information-theoretic optimum. For a suffix array, we need $|T| \lg |\Sigma|$ bits just to store the text T.

Grossi and Vitter 2000 [3] Suffix array in

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

bits, with query time

$$O\left(\frac{|P|}{\log^{\varepsilon}_{|\Sigma|} |T|} + |\text{output}| \cdot \log^{\varepsilon}_{|\Sigma|} |T|\right)$$

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

Ferragina and Manzini 2000 [1] The space required is

$$5H_k(T)|T| + o(|T|) + O\left(|T|^{\varepsilon} \cdot |\Sigma|^{O(|\Sigma|)}\right)$$

bits, for all fixed values of k. $H_k(T)$ is the k^{th}-order empirical entropy, or the regular entropy conditioned on knowing the previous k characters. More formally:

$$H_k(T) = \sum_{|w|=k} \Pr\{w \text{ occurs}\} \cdot H_0(\text{characters following an occurrence of } w \text{ in } T).$$

Note that because we're calculating this in the empirical case,

$$\Pr\{w \text{ occurs}\} = \frac{\#\text{ of occurrences of } w}{|T|}.$$

For this data structure, query time is

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

Sadakane 2003 [5] 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 [2] This is the only known succinct result. Space in bits is

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

and query time is

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

3 Compressed suffix arrays

For the rest of these notes, we will assume that the alphabet is binary (in other words, that $|\Sigma| = 2$). In this section, we will cover a simplified (and less space-efficient) data structure, which we will adapt in the next section for the compact data structure.

3.1 Top-Down

The data structure uses ideas similar to those in the DC3 algorithm presented in Lecture 7. For this data structure, however, we will group the characters in our string into pairs rather than triples. If we were starting from the original suffix array, the definitions would be as follows:

start The initial text $T_0 = T$, the initial size $n_0 = n$, and the initial suffix array $SA_0 = SA$ (the suffix array of T). Recall that $SA[i]$ is the index in T where the i^{th} suffix begins.

step We define the recursive texts to be:

$$T_{k+1} = \left\langle (T_k[2i], T_k[2i+1]) \text{ for } i = 0, 1, \ldots, \tfrac{n}{2} \right\rangle.$$

The recursive sizes are:

$$n_{k+1} = \tfrac{n_k}{2} = \tfrac{n}{2^k}.$$

And the recursive suffix array is:

$$SA_{k+1} = \tfrac{1}{2} \cdot (\text{extract even entries of } SA_k)$$

where "even suffixes" are defined to be suffixes whose index in T_k are even.

Clearly, it is fairly easy to calculate SA_{k+1} from SA_k, and since SA_0 is known, this means that we can go top-down without much difficulty. However, in order to make this data structure work, we need to go bottom-up.

3.2 Bottom-Up

We need a way to represent SA_k using SA_{k+1}. To do so, we define the following functions:

is-even$_k(i)$ This tells us whether $SA_k[i]$ is an even suffix. More formally:

$$\textit{is-even}_k(i) = \begin{cases} 1 & \text{if } SA_k[i] \text{ is even} \\ 0 & \text{otherwise} \end{cases}$$

even-succ$_k(i)$ The "even successor" of i, defined as i if $SA_k[i]$ is even, and otherwise equal to the j such that $SA_k[i] = SA_k[j] - 1$.

even-rank$_k(i)$ The "even rank" of i, or the number of even suffixes preceding the i^{th} suffix. Note that if $\textit{is-even}_k(\cdot)$ is stored as a bit vector, then this is equivalent to $rank_1(i)$ in that vector.

Given these definitions, we can write:

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

If the predicate $\textit{is-even}_k(i)$ is true, this is equivalent to saying

$$SA_k[i] = 2 \cdot SA_{k+1}[\textit{even-rank}_k(i)],$$

which basically means that we're looking up the correct value in the array SA_{k+1}. If $\textit{is-even}_k(i)$ is false, then this is equivalent to performing the same action on i's "even successor" — which is the index into SA_k of the suffix starting one position after $SA_k[i]$ — and then subtracting 1 to get the correct starting position in the text T_k.

If we can perform the above operations in constant time, then we can reduce a query on SA_k to a query on SA_{k+1} in constant time. Hence, a query on SA_0 will take $O(\ell)$ time if our maximum recursion depth is ℓ. If we set $\ell = \lg \lg n$, we will reduce the size of the text to $n_\ell = n / \lg n$. We can then use a normal suffix array, which will use $O(n_\ell \lg n_\ell) = O(n)$ bits of space, and thus be compressed.

3.3 Construction

We can store $\textit{is-even}_k(i)$ as a bit vector of size n_k. Because n_k decreases by a factor of two with each level, this takes a total of $O(n)$ space. Then we can implement $\textit{even-rank}_k(i)$ with the rank from last lecture on our bit vector, requiring $o(n_k)$ space per level, for a total of $O(n)$ space.

Doing $\textit{even-succ}_k(i)$ is trivial in the case that $SA_k[i]$ is even. If we had as much space as we wanted, we could just store the answer for odd values of $SA_k[i]$. Unfortunately, we can't actually write them down, because that would require $n_k/2$ items of $\lg n_k$ bits each.

Whatever data structure we use, let's order the values of j by i; that is, if the answers are stored in array called *answers*, then we would have $\textit{even-succ}_k(i) = \textit{answers}[i - \textit{even-rank}_k(i)]$, because $i - \textit{even-rank}_k(i)$ is the index of i among odd suffixes. This ordering is equivalent to ordering by the suffix in the suffix array, or $T_k[SA_k[i] :]$. This, in turn, is equivalent to ordering by $(T_k[SA_k[i]], T_k[SA_k[i] + 1 :]) = (T_k[SA_k[i]], T_k[SA_k[\textit{even-succ}_k(i)] :])$. Further, this is equivalent to ordering by $(T_k[SA_k[i]], \textit{even-succ}_k(i))$.

So to store $\textit{even-succ}_k(i)$, we actually store items of the form $(T_k[SA_k[i]], \textit{even-succ}_k(i))$. Each such item requires $(2^k + \lg n_k)$ bits, because the characters in T_k are of length 2^k. The data in these arrays is in sorted order. So we can store the leading $\lg n_k$ bits of each value v_i using unary differential encoding:

$$0^{\text{lead}(v_1)}10^{\text{lead}(v_2)-\text{lead}(v_1)}10^{\text{lead}(v_3)-\text{lead}(v_2)}1\ldots$$

Where $\text{lead}(v_i)$ is the value of the leading $\lg n_k$ bits of v_i. 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 if we maintain rank and select data structures, we can efficiently compute $\text{lead}(v_i) = \text{select}_1(i) - i$.

The remaining 2^k bits can be stored in an array. The space required for such an array on level k is:

$$2^k \cdot \frac{n_k}{2} = 2^k \frac{n}{2^{k+1}} = \frac{n}{2}.$$

Unfortunately, when we sum up this space requirement for all $\ell = \lg\lg n$ levels, we need a total of $O(n \lg\lg n)$ bits. So this is not compact.

4 Compact suffix arrays

To reduce the space requirements of the data structure, we want to store fewer levels of recursion. We choose to store $(1 + 1/\varepsilon)$ levels of recursion, one for the following values of k:

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

In other words, instead of pairing two letters together with each recursive step, we are now clustering $2^{\varepsilon\ell}$ letters in a single step. We now need to be able to jump $\varepsilon\ell$ levels at once.

4.1 Level jumping

In order to find a formula for $SA_{k\varepsilon\ell}$ in terms of $SA_{(k+1)\varepsilon\ell}$, we first redefine the word "even." Now a suffix $SA_{k\varepsilon\ell}[i]$ is "even" if its index in $T_{k\varepsilon\ell}$ is divisible by $2^{\varepsilon\ell}$. This changes the definition of $\textit{even-rank}_{k\varepsilon\ell}(i)$ in the obvious way. However, we will not change the definition of $\textit{even-succ}_{k\varepsilon\ell}(i)$: it should still return the value j such that $SA_{k\varepsilon\ell}[i] = SA_{k\varepsilon\ell}[j] - 1$. It should do this for all "even" values of $SA_{k\varepsilon\ell}[i]$.

With these modified definitions, we can compute $SA_{k\varepsilon\ell}[i]$ as follows:

- Calculate the even successor repeatedly until index j is at the next level down — in other words, so that $SA_{k\varepsilon\ell}[j]$ is divisible by $2^{\varepsilon\ell}$.
- Recursively compute $SA_{(k+1)\varepsilon\ell}[\textit{even-rank}_{k\varepsilon\ell}(j)]$.
- Multiply by $2^{\varepsilon\ell}$ (the number of letters we clustered together) and then subtract the number of calls to successor in the first step.

This process works for much the same reason that it does in the compressed suffix array. We first calculate the j such that $SA_{k\varepsilon\ell}[j]$ is divisible by $2^{\varepsilon\ell}$ and $SA_{k\varepsilon\ell}[j]-m = SA_{k\varepsilon\ell}[i]$, where $0 \le m < 2^{\varepsilon\ell}$. The recursive computation gives us the index in $T_{(k+1)\varepsilon\ell}$ of the suffix corresponding to $SA_{k\varepsilon\ell}[j]$. We can compute the true value of $SA_{k\varepsilon\ell}[j]$ if we multiply the result of the recursive computation by $2^{\varepsilon\ell}$. We then subtract the value m to get $SA_{k\varepsilon\ell}[i]$.

4.2 Analysis

We may have to look up the even successor of an index $2^{\varepsilon\ell}$ times before getting the value we can recur on. Therefore, the total search time is $O(2^{\varepsilon\ell} \lg\lg n) = O(\lg^{\varepsilon} n \lg\lg n) = O(\lg^{\varepsilon'} n)$ for any $\varepsilon' > \varepsilon$.

We use the same unary differential encoding for successor as in the compressed construction, for a total of $2n_{k\varepsilon\ell} + n + o(n)$ bits per level in total. We also must store the $\textit{is-even}_{k\varepsilon\ell}(\cdot)$ vectors and the rank data structure, for a total of $n_{k\varepsilon\ell} + o(n)$ per level. There are $1 + 1/\varepsilon$ levels in total. Hence, the total space is something like $(6 + 1/\varepsilon)n + o(n)$ bits, which is compact.

There are some optimizations we can perform to improve the space. We don't have to store the data for $\textit{even-succ}_0(\cdot)$, because it's the top level, which means that the total space required storing even successor information is:

$$O\left(\frac{n}{2^{\varepsilon\ell}}\right) = O\left(\frac{n}{\lg^{\varepsilon} n}\right) = o(n).$$

If we store the $\textit{is-even}_{k\varepsilon\ell}(\cdot)$ vectors as succinct dictionaries (because they are, after all, fairly sparse), then the space required is:

$$\lg \binom{n_{k\varepsilon\ell}}{n_{(k+1)\varepsilon\ell}} \approx n_{(k+1)\varepsilon\ell} \lg \frac{n_{k\varepsilon\ell}}{n_{(k+1)\varepsilon\ell}} = n_{(k+1)\varepsilon\ell} \lg 2^{\varepsilon\ell} = \frac{n_{k\varepsilon\ell} \cdot \varepsilon\ell}{2^{\varepsilon\ell}} = \frac{n_{k\varepsilon\ell} \cdot \varepsilon \lg\lg n}{\lg^{\varepsilon} n} = o(n_{k\varepsilon\ell})$$

Hence, the total space is $o(n)$. This gives us a total space of $(1 + 1/\varepsilon)n + o(n)$ bits.

Open problem: Is it possible to achieve $o(\lg^{\varepsilon} n)$ in linear space?

5 Suffix trees [4]

5.1 Construction

In Lecture 22, we saw how to store a binary trie with $2n + 1$ nodes on $4n + o(n)$ bits. We can use this to store the structure of the compressed suffix tree. Unfortunately, we don't have enough

space to store the edge lengths or the letter depth, which would allow us to traverse the tree with no further effort.

To search for a pattern P in the tree, we must calculate the letter depth as we go along. Say that we know the letter depth of the current node x. To descend to its child y, we need to compute the difference in letter depths, or the length in letters of the edge between them.

The letter depth of y is equivalent to the length of the substring shared by the leftmost descendant of y and the rightmost descendant of y. Let ℓ be the leftmost descendant, and let r be the rightmost descendant. If we know the index in the suffix array of both ℓ and r, then we can use the suffix array to find their indices in the text. Because ℓ and r are both descendants of x, we know that they both match for *letter-depth*(x) characters. So we can skip the first *letter-depth*(x) characters of both, and start comparing the characters of ℓ, r, and P. If P differs from ℓ and r before they differ from each other, we know that there are no suffixes matching P in the tree, and we can stop the whole process. Otherwise, ℓ and r will differ from each other at some point, which will give us the letter depth of y. Note that the total number of letter comparisons we perform is $O(|P|)$, for the entire process of searching the tree.

5.2 Necessary binary trie operations

To find ℓ, r, and their indices into the suffix array, note that in the balanced parentheses representation of the trie, each leaf is the string "())".

leaf-rank(here**)** The number of leaves to the left of the node which is at the given position in the string of balanced parentheses. Can be computed by getting $rank_{())}((n)$

leaf-select(i**)** The position in the balanced parentheses string of the i^{th} leaf. Can be computed by calculating $select_{())}(i)$.

leaf-count(here**)** The number of leaves in the subtree of the node at the given position in the string of balanced parens. Can be computed using the formula:

$$rank_{())}(\text{matching) of parent}) - rank_{())}(\text{here}).$$

leftmost-leaf(here**)** The position in the string of the leftmost leaf of the node at the given position. Given by the formula:

$$\textit{leaf-select}(\textit{leaf-rank}(\text{here}) + 1).$$

rightmost-leaf(here**)** The position in the string of the rightmost leaf of the node at the given position. Given by the formula:

$$\textit{leaf-select}(\textit{leaf-rank}(\text{matching) of parent} - 1)).$$

Hence, we can use a rank and select data structure on the string of balanced parentheses to find the first and last leaves in the part of the string representing the subtree rooted at y. We can then calculate the rank of those leaves to determine the index into the suffix array. Hence, we can perform the search described in the previous paragraph at a cost of $O(1)$ operations per node.

5.3 Analysis

The total time required to search for the pattern P is

$$O(|P| + |\text{output}|) \cdot O(\text{cost of suffix array lookup}).$$

5.4 Improvement

It is also possible to improve this, creating a succinct suffix tree given a suffix array. In the above algorithm, storing the suffix tree takes too much space to achieve succinctness. Instead, we store the above compact suffix tree on every b^{th} entry in the suffix array, which gives us an extra storage cost of $O(n/b)$.

First, modify the tree to return something reasonable if P doesn't match any of the items in the tree, such as returning the predecessor of P in the set of leaves. If we consider the suffixes to be divided into blocks of size b, then when we query on P, the suffix tree will give us an interval of block dividers such that any suffix matching P must lie in a block touching one of those dividers. This gives us a range of blocks in which to look for the true answer.

The rest of the algorithm was not covered in lecture, but is explained in [4] and in the handwritten lecture notes.

References

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

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

[3] 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.

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

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