Supplemental Notes on String Algorithms

James Aspnes

April 17, 2004

Note: These notes will make more sense if you read Chapter 32 of [CLRS01] first. Section 1 describes *suffix trees* and *suffix arrays*, and some of their applications for substring problems. Section 2 describes the Burrows-Wheeler transform, which is closely related to suffix arrays, and which is the basis of some of the best general-purpose text-compression algorithms currently known.

There is a vast literature on string matching algorithms, which have become very popular recently because of applications to DNA sequencing. A very thorough survey of string matching algorithms and their biological applications is [Gus97].

1 Suffix trees and suffix arrays

Suffix trees and *suffix arrays* are data structures for representing texts that allow substring queries like "where does this pattern appear in the text" or "how many times does this pattern occur in the text" to be answered quickly. Both work by storing all suffixes of a text, where a *suffix* is a substring that runs to the end of the text. Of course, storing actual copies of all suffixes of an n-character text would take $O(n^2)$ space, so instead each suffix is represented by a pointer to its first character.

In a *suffix array*, the suffixes are sorted in dictionary order. For example, a suffix array of the string `abracadabra` is depicted in Figure 1.

A suffix tree is similar, but instead using an array, we use some sort of tree data structure to hold the sorted list. A common choice given an alphabet of some fixed size k is a *trie*, in which each node at depth d represents a string of length d, and its up to k children represent all $(d+1)$-character extensions of the string. The advantage of using a suffix trie is that searching for a string of length m takes $O(m)$ time, since we can just walk down the trie at the rate of one node per character in m. A further optimization is to replace

```
10  a
 7  abra
 0  abracadabra
 3  acadabra
 5  adabra
 8  bra
 1  bracadabra
 4  cadabra
 6  dabra
 9  ra
 2  racadabra
```

Figure 1: Suffix array for `abracadabra`. The actual contents of the array are the indices in the left-hand column; the right hand column shows the corresponding suffixes.

any long chain of single-child nodes with a "compressed" edge labeled with the concatenation all the characters in the chain. Such compressed suffix tries can not only be searched in linear time but can also be constructed in linear time with a sufficiently clever algorithm. See [Gus97, Chapter 6]. Of course, we could also use a simple balanced binary tree, which might be preferable if the alphabet is large.

The disadvantage of suffix trees over suffix arrays is that they tend to require more space to store all the internal pointers in the tree. If we are indexing a huge text (or collection of texts), this extra space may be too expensive.

1.1 Searching a suffix array

Suppose we have a suffix array corresponding to an n-character text and we want to find all occurrences in the text of an m-character pattern. Since the suffixes are ordered, we can do binary search for the first and last occurrences of the pattern (if any) using $O(\log n)$ comparisons. Unfortunately, each comparison may take as much as $O(m)$ time, since we may have to check all m characters of the pattern. So the total cost will be $O(m \log n)$ in the worst case.

By storing additional information about the amount of overlap between adjacent suffixes, it is possible to avoid having to re-examine every character in the pattern for every comparison, reducing the search cost to $O(m+\log n)$.

This information can also be used to solve quickly such problems as finding the longest duplicate substrings, or most frequently occurring strings. See [Gus97, §7.14.4] for details.

1.2 Building a suffix array

A straightforward approach to building a suffix array is to run any decent comparison-based sorting algorithm on the set of suffixes (represented by pointers into the text). This will take $O(n \log n)$ comparisons, but in the worst case each comparison will take $O(n)$ time, for a total of $O(n^2 \log n)$ time. The original suffix array paper by Manber and Myers [MM93] gives an $O(n \log n)$ algorithm, somewhat resembling radix sort, for building suffix arrays "in place" with only a small amount of additional space. They also note that for random text, simple radix sorting works well, since most suffixes become distinguishable after about $\log_k n$ characters (where k is the size of the alphabet).

The fastest approach is to build a suffix tree in $O(n)$ time and extract the suffix array by traversing the tree. The only complication is that we need the extra space to build the tree, although we get it back when we throw the tree away.

2 The Burrows-Wheeler transform

Some of the best currently-known algorithms for text data compression are based on a technique, known as the *Burrows-Wheeler transform* [BW94], that is closely related to suffix arrays. The idea of the Burrows-Wheeler Transform is to construct an array whose rows are all cyclic shifts of the input string in dictionary order, and return the last column of the array. The last column will tend to have long runs of identical characters, since whenever some substring (like `the`) appears repeatedly in the input, shifts that put the first character (`t`) in the last column will put the rest of the substring (`he`) in the first columns, and the resulting rows will tend to be sorted together. The relative regularity of the last column means that it will compress well with even very simple compression algorithms.

Figure 2 shows the Burrows-Wheeler transform applied to the string `abracadabra$`. Here `$` serves as the end-of-string marker.

```
abracadabra$      abracadabra$
bracadabra$a      abra$abracad
racadabra$ab      acadabra$abr
acadabra$abr      adabra$abrac
cadabra$abra      a$abracadabr
adabra$abrac      bracadabra$a
dabra$abraca -->  bra$abracada
abra$abracad      cadabra$abra
bra$abracada      dabra$abraca
ra$abracadab      racadabra$ab
a$abracadabr      ra$abracadab
$abracadabra      $abracadabra
```

Figure 2: Cyclic shifts of `abracadabra$`, before and after sorting. The Burrows-Wheeler transform is `$drcraaaabba`, the last column of the sorted array.

```
$    a       $a    ab       $ab    abr
d    a       da    ab       dab    abr
r    a       ra    ac       rac    aca
c    a       ca    ad       cad    ada
r    a       ra    a$       ra$    a$a
a    b       ab    br       abr    bra
a -> b       ab -> br       abr -> bra
a    c       ac    ca       aca    cad
a    d       ad    da       ada    dab
b    r       br    ra       bra    rac
b    r       br    ra       bra    ra$
a    $       a$    $a       a$a    $ab
```

Figure 3: Inverting the Burrows-Wheeler transform the slow way. At each step, the rightmost column is prepended to the k columns reconstructed so far so far, and the resulting rows are sorted to get the first $k + 1$ columns. Only the first three steps are shown.

2.1 Inverting the Burrows-Wheeler transform

The surprising feature of the Burrows-Wheeler transform is that the original string can be recovered from this last column, provided its end is specially marked.

We'll describe two ways to do this; the first is less efficient, but more easily grasped, and involves rebuilding the array one column at a time, starting at the left. Observe that the leftmost column is just all the characters in the string in sorted order; we can recover it by sorting the rightmost column, which we have to start off with. If we paste the rightmost and leftmost columns together, we have the list of all 2-character substrings of the original text; sorting this list gives the first *two* columns of the array. (Remember that each copy of the string wraps around from the right to the left.) We can then paste the rightmost column at the beginning of these two columns, sort the result, and get the first three columns. Repeating this process eventually reconstructs the entire array, from which we can read off the original string from any row. The initial stages of this process for `abracadabra$` are shown in Figure 3.

Rebuilding the entire array takes $O(n^2)$ time and $O(n^2)$ space. In their paper, Burrows and Wheeler [BW94] showed that one can in fact reconstruct the original string from just the first and last columns in the array in $O(n)$ time.

Here's the idea: Suppose that all the characters were distinct. Then after reconstructing the first column we would know all pairs of adjacent characters. So we could just start with the last character `$` and regenerate the string by appending at each step the unique successor to the last character so far. If all characters were distinct, we would never get confused about which character comes next.

The problem is what to do with pairs with duplicate first characters, like `ab` and `ac` in Figure 3. We can imagine that each `a` in the last column is labeled in some unique way, so that we can talk about the first `a` or the third `a`, but how do we know which `a` is the one that comes before `b` or `d`?

The trick is to look closely at how the original sort works. Look at Figure 2. If we look at all rows that start with `a`, the order the are sorted in is determined by the suffix after `a`. These suffixes also appear as the prefixes of the rows that *end* with `a`, since the rows that end with `a` are just the rows that start with `a` rotated one position. It follows that *all instances of the same letter occur in the same order in the first and last columns.* So if we use a *stable sort* to construct the first column, we will correctly match up instances of letters. This method is shown in action in Figure 4.

```
$1      a1
d1      a2
r1      a3
c1      a4
r2      a5
a1      b1
a2 --> b2
a3      c1
a4      d1
b1      r1
b2      r2
a5      $1
```

Figure 4: Inverting the Burrows-Wheeler transform the fast way. Each letter is annotated uniquely with a count of how many identical letters equal or precede it. Sorting recovers the first column, and combining the last and first columns gives a list of unique pairs of adjacent annotated characters. Now start with `$1` and construct the full sequence `$1 a1 b1 r1 a3 c1 a4 d1 a2 b2 r2 a5 $1`. The original string is obtained by removing the end-of-string markers and annotations: `abracadabra`.

Because we are only sorting single characters, we can perform the sort in linear time using counting sort. Extracting the original string also takes linear time if implemented reasonably.

2.2 Suffix arrays and the Burrows-Wheeler transform

A useful property of the Burrows-Wheeler transform is that each row of the sorted array is essentially the same as the corresponding row in the suffix array, except for the rotated string prefix after the $ marker. This means, among other things, that we can compute the Burrows-Wheeler transform in linear time using suffix trees. Ferragina and Manzini [FM00, FM01] have further exploited this correspondence (and some very clever additional ideas) to design compressed suffix arrays that compress and index a text at the same time, so that pattern searches can be done directly on the compressed text in time close to that needed for suffix array searches.

References

[BW94] Michael Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, DEC Systems Research Center, May 1994.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, 2001.

[FM00] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science*, 2000.

[FM01] Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, 2001.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[MM93] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.