1 APL13: Suffix Arrays: more space reduction

In Section **??**, we saw that when alphabet size is included in the time and space bounds, the suffix tree for a string of length m either requires $\Theta(m|\Sigma|)$ space or the minimum of $O(m \log m)$ and $O(m \log |\Sigma|)$ time. Similarly, searching for a pattern P of length n using a suffix tree can be done in $O(n)$ time only if $\Theta(m|\Sigma|)$ space is used for the tree, or if we assume that up to $|\Sigma|$ character comparisons cost only constant time. Otherwise, the search takes the minimum of $O(n \log m)$ and $O(n \log |\Sigma|)$ comparisons. For these reasons, a suffix tree may require too much space to be practical in *some* applications. Hence a more space efficient approach is desired that still retains most of the advantages of searching with a suffix tree.

In the context of the *substring* problem (see Section **??**) where a fixed string T will be searched many times, the key issues are the time needed for the search, and the space used by the fixed data structure representing T. The space used during the preprocessing of T is of less concern, although it should still be "reasonable".

Manber and Myers [1] proposed a new data structure, called a *suffix array*, that is very space efficient and yet can be used to solve the exact matching problem or the substring problem almost as efficiently as with a suffix tree. Suffix arrays are likely to be an important contribution to certain string problems in computational molecular biology, where the alphabet can be large (we will discuss some of the reasons for large alphabets below). Interestingly, although the more formal notion of a suffix array and the basic algorithms for building and using it were developed in [1], many of the ideas were anticipated in the biological literature by Martinez [2].

After defining suffix arrays we show how to convert a suffix tree to a suffix array in linear time. It is important to be clear on the setting of the problem. String T will be held fixed for a long time, while P will vary. Therefore the goal is to find a space efficient representation for T (a suffix array) that will be held fixed and that facilitates search problems in T. But the amount of space used during the construction of that representation is not so critical. In the exercises we consider a more space efficient way to build the representation itself.

Definition: Given an m character string T, a *suffix array* for T, called *Pos*, is an array of the integers in the range 1 to m, specifying the lexicographic order of the m suffixes of string T.

That is, the suffix starting at position *Pos(1)* of T is the lexically smallest suffix, and in general suffix *Pos(i)* of T is lexically smaller than suffix *Pos(i+1)*.

As usual, we will affix a terminal symbol \$ to the end of S, but now interpret it to be lexically *less* than any other character in the alphabet. This is in contrast to its interpretation in the previous section. As an example of a suffix array, if T is *mississippi*, then the suffix array *Pos* is $11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3$. Figure 1 lists the eleven suffixes in lexicographic order.

Notice that the suffix array holds only integers and hence contains no information about the alphabet used in string T. Therefore, the space required by suffix arrays

```
11:  i
 8:  ippi
 5:  issippi
 2:  ississippi
 1:  mississippi
10:  pi
 9:  ppi
 7:  sippi
 4:  sisippi
 6:  ssippi
 3:  ssissippi
```

Figure 1: The eleven suffixes of *mississippi* listed in lexicographic order. The starting positions of those suffixes define the suffix array *Pos*.

is modest – for a string of length m, the array can be stored in exactly m computer words, assuming a word size of at least $\log m$ bits.

When augmented with an additional $2m$ values (called *Lcp* values and defined later), the suffix array can be used to find all the occurrences in T of a pattern P in $O(n + \log_2 m)$ single-character comparison and bookkeeping operations. Moreover, this bound is independent of the alphabet size. Since for most problems of interest $\log_2 m$ is $O(n)$, the substring problem is solved by using suffix arrays as efficiently as by using suffix trees.

1.1 Suffix tree to suffix array in linear time

We assume that sufficient space is available to build a suffix tree for T (this is done once during a preprocessing phase), but that the suffix tree cannot be kept intact to be used in the (many) subsequent searches for patterns in T. Instead we convert the suffix tree to the more space efficient suffix array. Exercises **??**, **??** and **??** develop a more space efficient (but slower) method for *building* a suffix array.

A suffix array for T can be obtained from the suffix tree $\mathcal{T}$ for T by performing a "lexical" depth-first traversal of $\mathcal{T}$. Once the suffix array is built, the suffix tree is discarded.

Definition Define an edge (v, u) to be *lexically less* than an edge (v, w) if and only if the first character on the (v, u) edge is lexically less than the first character on (v, w). (In this application, the end of string character $ is lexically less than any other character.)

Since no two edges out of v have labels beginning with the same character, there is a strict lexical ordering of the edges out of v. This ordering implies that the path

from the root of $\mathcal{T}$ following the lexically smallest edge out of each encountered node leads to a leaf of $\mathcal{T}$ representing the lexically smallest suffix of T. More generally, a depth-first traversal of $\mathcal{T}$ which traverses the edges out of each node v in their lexical order will encounter the leaves of $\mathcal{T}$ in the lexical order of the suffixes they represent. Suffix array *Pos* is therefore just the ordered list of suffix numbers encountered at the leaves of $\mathcal{T}$ during the lexical depth- first-search. The suffix tree for T is constructed in linear time, and the traversal also takes only linear time, so we have the following

Theorem 1.1 *The suffix array Pos for a string T of length m can be constructed in $O(m)$ time.*

For example, the suffix tree for $T = tartar$ is shown in Figure 2. The lexical depth-first traversal visits the nodes in the order 5,2,6,3,4,1, defining the values of array *Pos*.

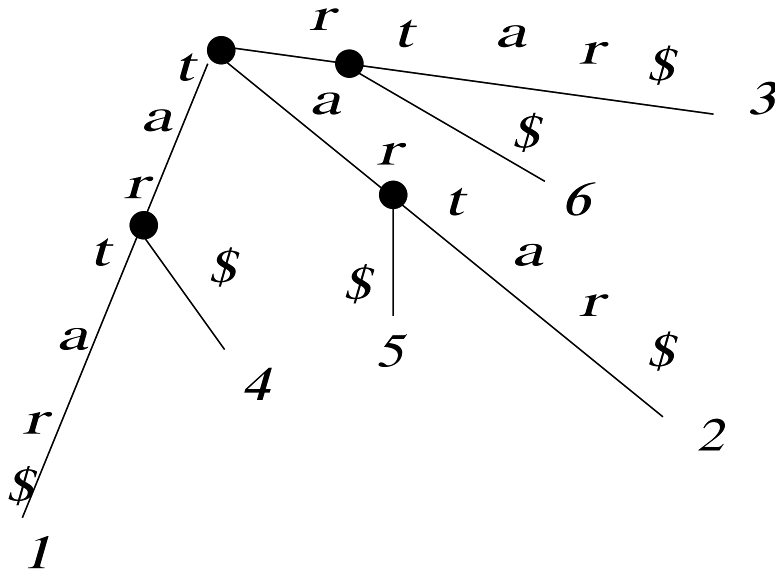

Figure 2: The lexical depth-first traversal of the suffix tree visits the leaves in order 5,2,6,3,4,1.

As an implementation detail, if the branches out of each node of the tree are organized in a *sorted* linked list (as discussed in Section **??**, page **??**), then the overhead to do a lexical depth-first search is the same as for any depth-first search. Every time the search must choose an edge out of a node v to traverse, it simply picks the next edge on v's linked list.

1.2 How to search for a pattern using a suffix array

The suffix array for string T allows a very simple algorithm to find all occurrences of any pattern P in T. The key is that if P occurs in T then all the locations of those

occurrences will be grouped consecutively in *Pos*. For example, P = *issi* occurs in *mississippi* starting at locations 2 and 5, which are indeed adjacent in *Pos* (see Figure 1). So to search for occurrences of P in T simply do binary search over the suffix array. In more detail, suppose that P is lexically less than the suffix in the middle position of *Pos* (i.e., suffix $Pos(\lceil m/2 \rceil)$). In that case, the first place in *Pos* that contains a position where P occurs in T must be in the first half of *Pos*. Similarly, if P is lexically greater than suffix $Pos(\lceil m/2 \rceil)$, then the places where P occurs in T must be in the second half of *Pos*. Using binary search, one can therefore find the smallest index i in *Pos* (if any) such that P exactly matches the first n characters of suffix $Pos(i)$. Similarly, one can find the largest index i' with that property. Then pattern P occurs in T starting at every location given by $Pos(i)$ through $Pos(i')$.

The lexical comparison of P to any suffix takes time proportional to the length of the common prefix of those two strings. That prefix has length at most n, hence

Theorem 1.2 *By using binary search on array Pos, all the occurrences of P in T can be found in $O(n \log m)$ time.*

Of course, the true behavior of the algorithm depends on how many long prefixes of P occur in T. If very few long prefixes of P occur in T then it will rarely happen that a specific lexical comparison actually takes $\Theta(n)$ time and generally the $O(n \log m)$ bound is quite pessimistic. In "random" strings (even on large alphabets) this method should run in $O(n + \log m)$ expected time. In cases where many long prefixes of P do occur in T, then the method can be improved with the following two tricks.

1.3 A simple accelerant

As the binary search proceeds, let L and R denote the left and right boundaries of the "current search interval". At the start, L equals 1 and R equals m. Then in each iteration of the binary search, a query is made at location $M = \lceil (R + L)/2 \rceil$ of *Pos*. The search algorithm keeps track of the longest prefixes of $Pos(L)$ and $\mathrm{Pos}(R)$ that match a prefix of P. Let l and r denote those two prefix lengths respectively, and let $mlr = \min(l, r)$.

The value mlr can be used to accelerate the lexical comparison of P and suffix $Pos(M)$. Since array *Pos* gives the lexical ordering of the suffixes of T, if i is any index between L and R, the first mlr characters of suffix $Pos(i)$ must be the same as the first mlr characters of suffix $Pos(L)$, and hence of P. Therefore, the lexical comparison of P and suffix $Pos(M)$ can begin from position $mlr + 1$ of the two strings, rather than starting from the first position.

Maintaining mlr during the binary search adds little additional overhead to the algorithm but avoids many redundant comparisons. At the start of the search, when $L = 1$ and $R = m$, explicitly compare P to suffix $Pos(1)$ and suffix $Pos(m)$ to find l and r and mlr. However, the *worst* case time for this revised method is still $O(n \log m)$. Myers and Manber report that the use of mlr alone allows the search to

run as fast in practice as the $O(n + \log m)$ worst case method that we first advertised. Still, if only because of its elegance, we present the full method that guarantees that better worst case bound.

1.4 A super-accelerant

Call an examination of a character in P *redundant* if that character has been examined before. The goal of the acceleration is to reduce the number of redundant character examinations to at most one per iteration of the binary search – hence $O(\log m)$ in all. The desired time bound, $O(n + \log m)$, follows immediately. The use of *mlr* alone does not achieve this goal. Since *mlr* is the *minimum* of l and r, whenever $l \neq r$, all characters in P from $mlr + 1$ to the maximum of l and r will have already been examined. So any comparisons of those characters will be redundant. What is needed is a way to begin comparisons at the *maximum* of l and r.

Definition *Lcp(i,j)* is the length of the longest common prefix of the suffixes specified in positions i and j of *Pos*. That is, *Lcp(i,j)* is the length of the longest prefix common to suffix *Pos*(i) and suffix *Pos*(j). The term *Lcp* stands for *longest common prefix*.

For example, when $T =$ *mississippi*, suffix *Pos*(3) is *issippi*, suffix *Pos*(4) is *ississippi*, and so *Lcp(3,4)* is four (see Figure 1).

To speed up the search, the algorithm uses *Lcp(L,M)* and *Lcp(M,R)* for each triple (L, M, R) that arises during the execution of the binary search. For now, we assume that these values can be obtained in constant time when needed, and show how they help the search. Later we will show how to compute the particular *Lcp* values needed by the binary search during the preprocessing of T.

How to use *Lcp* values

Simplest case:

In any iteration of the binary search, if $l = r$, then compare P to suffix *Pos(M)* starting from position $mlr + 1 = l + 1 = r + 1$, as before.

General case:

When $l \neq r$, let us assume without loss of generality that $l > r$. Then there are three subcases:

- If *Lcp(L,M)* $> l$, then the common prefix of suffix *Pos(L)* and suffix *Pos(M)* is longer than the common prefix of P and *Pos(L)*. Therefore P agrees with suffix *Pos(M)* up through character l. In other words, characters $l+1$ of suffix *Pos(L)* and suffix *Pos(M)* are identical and lexically *less* than character $l + 1$ of P (the last fact follows since P is lexically greater than suffix *Pos(L)*). Hence all (if any) starting locations of P in T must occur to the right of position M in *Pos*. So in any iteration of the binary search where this case occurs, *no* examinations

of P are needed; L just gets changed to M, and l and r remain unchanged. (See Figure 3).

- If *Lcp(L,M)* $< l$, then the common prefix of suffix *Pos(L)* and *Pos(M)* is smaller than the common prefix of suffix *Pos(L)* and P. Therefore P agrees with suffix *Pos(M)* up through character *Lcp(L,M)*. The *Lcp(L,M)+1* characters of P and suffix *Pos(L)* are identical and lexically less than character *Lcp(L,M)+1* of suffix *Pos(M)*. Hence all (if any) starting locations of P in T must occur to the left of position M in *Pos*. So in any iteration of the binary search where this case occurs, *no* examinations of P are needed; r is changed to *Lcp(L,M)*, l remains unchanged, and R is changed to M.
- If *Lcp(L,M)* $= l$, then P agrees with suffix *Pos(M)* up to character l. The algorithm then lexically compares P to suffix *Pos(M)* starting from position $l + 1$. In the usual manner, the outcome of that lexical comparison determines which of L or R change, along with the corresponding change of l or r.

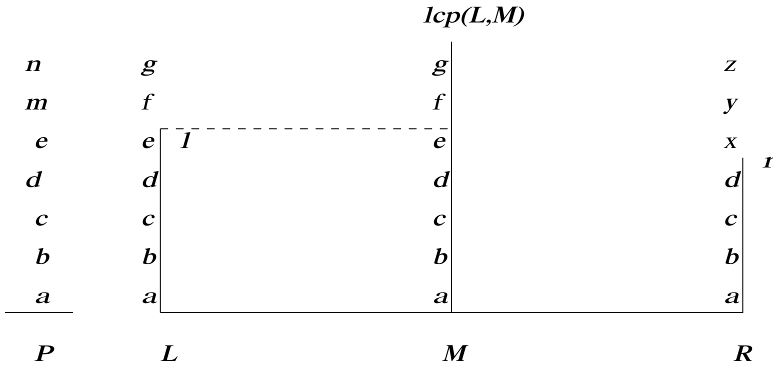

Figure 3: Subcase 1 of the super-accelerant. Pattern P is *abcdemn* shown vertically running upwards from the first character. The suffixes *Pos(L), Pos(M), Pos(R)* are also shown vertically. In this case, *Lcp(L,M)* > 0, and $l > r$. Any starting location of P in T must occur in *Pos* to the right of M, since P agrees with suffix *Pos(M)* only up to character l.

Theorem 1.3 *Using the Lcp values, the search algorithm does at most* $O(n + \log m)$ *comparisons, and runs in that time.*

Proof First, by simple case analysis it is easy to verify that neither l nor r ever decreases during the binary search. Also, every iteration of the binary search either terminates the search, or examines no characters of P, or ends after the first mismatch occurs in that iteration.

In the two cases ($l = r$ or *Lcp(L,M)* $= l > r$) where the algorithm examines a character during the iteration, the comparisons start with character *max(l,r)* of P.

Suppose there are k characters of P examined in that iteration. Then there are $k-1$ matches during the iteration, and at the end of the iteration *max(l,r)* increases by $k-1$ (either l or r is changed to that value). Hence at the start of any iteration, character *max(l,r)* of P may have already been examined, but the next character in P has not been. That means at most one redundant comparison per iteration is done. So no more than $\log_2 m$ redundant comparisons are done overall. There are at most n non-redundant comparisons of characters of P, giving a total bound of $n + \log m$ comparisons. All the other work in the algorithm can clearly be done in time proportional to these comparisons. □

1.5 How to obtain the *Lcp* values

The original method of Manber and Myers has been improved and notes on it will be distributed.

1.6 Where do large alphabet problems arise?

A large part of the motivation for suffix arrays comes from problems that arise in using suffix trees when the underlying alphabet is large. So it is natural to ask where large alphabets occur.

First, there are natural languages, such as Chinese, with large "alphabets" (using some computer representation of the Chinese pictograms.) But most large alphabets of interest to us arise because the string contains *numbers*, each of which is treated as a character. One simple example is a string that comes from a picture where each character in the string gives the color or grey level of a pixel.

String and substring matching problems where the alphabet contains numbers, and where P and T are large, also arise in computational problems in molecular biology. One example is the *map matching* problem. A *restriction enzyme map* for a single enzyme specifies the locations in a DNA string where copies of a certain substring (a restriction enzyme recognition site) occurs. Each such site may be separated from the next one by many thousands of bases. Hence, the restriction enzyme map for that single enzyme is represented as a string consisting of a sequence of integers specifying the distances between successive enzyme sites. Considered as a string, each integer is a character of a (huge) underlying alphabet. More generally, a map may display the sites of many different patterns of interest (whether or not they are restriction enzyme sites), so the string (map) consists of characters from a finite alphabet (representing the known patterns of interest) alternating with integers giving the distances between such sites. The alphabet is huge because the range of integers is huge, and since distances are often known with high precision, rounding is not used. Moreover, the variety of known patterns of interest is itself large (see [3] for a library of several hundred known, significant patterns).

It often happens that a DNA substring is obtained and studied without knowing

where that DNA is located in the genome or whether that substring has been previously researched. If both the new and the previously studied DNA are fully sequenced and put in a database, then the issue of previous work or locations would be solved by exact string matching. But most DNA substrings that are studied are not fully sequenced – maps are easier and cheaper to obtain than sequences. So the following matching problem on *maps* arises and translates to an matching problem on *strings* with large alphabets:

> Given an established (restriction enzyme) map for a large DNA string, and a map from a smaller string, determine if the smaller string is a substring of the larger one.

Since each map is represented as an alternating string of characters and integers, the underlying alphabet is huge. This provides one motivation for using suffix arrays for matching or substring searching in place of suffix trees. Of course, the problems become more difficult in the presence of errors, when the integers in the strings may not be exact, or when sites are missing or spuriously added. That problem, called *map alignment*, is discussed in Section **??**.

References

[1] U. Manber and G. Myers. Suffix arrays: a new method for on-line search. *SIAM J. on Computing*, pages 935–948, 1993.

[2] H. Martinez. An efficient method for finding repeats in molecular sequences. *Nucl. Acids Res.*, 11:4626–4634, 1983.

[3] E. Trifonov and V. Brendel. *Gnomic: A dictionary of genetic codes*. VCH Press, Deerfield, Florida, 1986.