Stringology

Sasha Bell, Tiffany Yong

March 17, 2022

This is an augmented transcript of a lecture given by Luc Devroye on the 17th of March 2022 for the Honours Data Structures and Algorithms class (COMP 252, McGill University). The subject was string processing data structures and algorithms.

Introduction

Stringology deals with strings, patterns and text. We think of a string in a broad way — it could represent text, DNA, numerical data or even images. In the first half of the lecture, we present data structures that work with text and strings, and in the second half of the lecture, we introduce some algorithms on these data structures.

Definition 1. An **alphabet** A is a set of possible values that a string can take.

Examples of alphabets include the 26-letter Latin alphabet ($|A| = 26$), and the alphabets used to express binary numbers ($|A| = 2$), or even DNA sequences ($|A| = 4$).

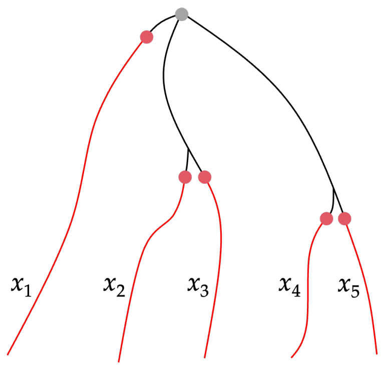

Figure 1: An example of an infinite trie. The red paths are the "spaghetti"-like infinite ends that get trimmed, and the red nodes are the leaves of the resulting trie.

Tries

Definition 2. A **trie** is a k-ary search tree that stores strings $x_1, \ldots, x_n$, where each x_i is a string consisting of symbols from A, and $k = |A|$. The term "trie" comes from the word "retrieval", and was first coined by Fredkin[1].

[1] Fredkin [1960]

We begin with the case where x_i's are all infinite strings. This allows us to form an infinite trie, where a string can be viewed as a path in a k-ary tree if we use the symbols in the string to determine the child that is followed (Figure 1).

Since we are given a finite number of paths, if we do not allow identical paths[2], eventually each path must split. We can **"trim the spaghetti"**, where we remove the unnecessary infinite ends of each branch of the tree, such that each path has a unique leaf. We do this by cutting off the trailing end of each path at the first node unique to that path, visualised by removing the red paths in Figure 1. Now we have our trie (Figure 2), with n leaves and one leaf per string, where the leaf stores the index of the string.

[2] Assume that the x_i's are all distinct. Otherwise, there would still be an infinite path after the trim.

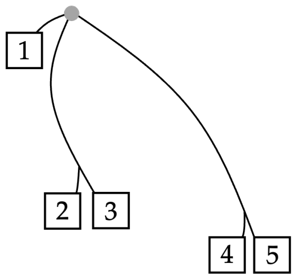

Figure 2: The trie obtained from trimming Figure 1.

In the case where x_i's are finite, it is possible that some x_i is a prefix for some x_j in the list of strings (e.g., x_i = "goo" and x_j = "google"). Clearly we cannot count on having only one leaf per path anymore.

To get around this issue, we mark the nodes where one string terminates, as shown in Figure 3. This indicates that one path represents two strings rather than one. We don't need to mark the leaves, since these will always represent a string. Then, we have that all path endings are either in a leaf or a marked node.

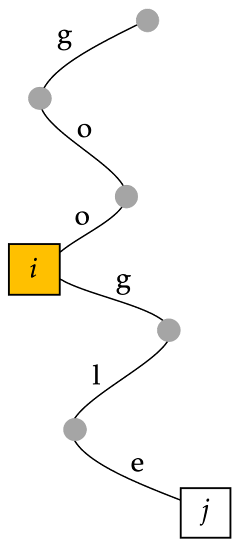

Figure 3: The "goo" and "google" paths, with a yellow marked node representing the word ending "goo".

Remarks on storage:

- The storage required for finite tries can easily blow up, as it is not bound as a function of n. Even when $n = 2$, we can create two arbitrarily long strings that only split at the end. Then, the space needed is the length of the arbitrarily large string.
- Storing strings the classical way results in each internal node having $k = |A|$ children pointers. However, the further an internal node is from the root, the more likely that most of its children pointers will point to nil. For example, there are many words that start with "t", so most of the children pointers will point to another node. However, there are fewer words that start with "trem", so most children pointers will point to nil. This results in a lot of wasted space due to unnecessary pointers.
- de la Briandais[3] developed a more space-efficient method for storing strings in an alphabet A. Let $|A| = k$. Instead of each internal node having k child pointers, we replace it with one pointer to a linked list of $\leq k$ children (Figure 4). However, this disadvantage of this method is that one needs to search a linked list to find a child. Another similar alternative would be having a node point to a search tree of children.
- There is an even more space-efficient method for storing strings: the PATRICIA tree, which we will explore below.

[3] de la Briandais [1959]

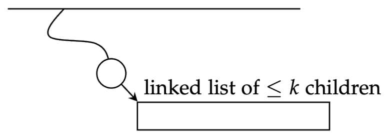

Figure 4: de la Briandais method for storing strings.

PATRICIA Trees

Definition 3. A PATRICIA tree is a trie where subsequent nodes with one child each are compacted into one node, to save space. PATRICIA stands for Practical Algorithm to Retrieve Information Coded in Alphanumeric, and was first described by Morrison[4].

[4] Morrison [1968]

For example, in a binary tree, one can describe a path using a sequence of zeros and ones. If there is a chain of nodes with one

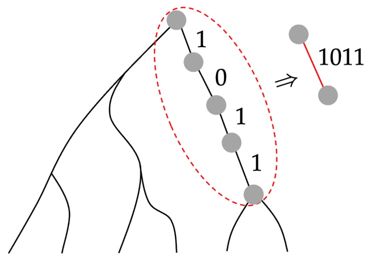

Figure 5: PATRICIA tree node reduction.

child each then we can reduce these nodes to an edge that stores the relevant path information (a finite binary sequence) (Figure 5).

If we have a marked node in a chain of nodes with one child, we cannot compact it into one node to save space. We can only compact chains of unmarked nodes. If a marked node interrupts a chain of nodes with one child, we compact the chain above and below the marked node (Figure 6).

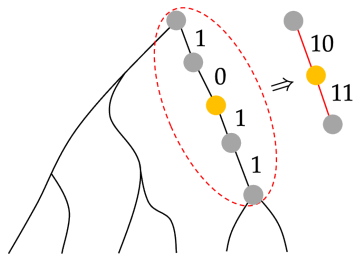

Figure 6: PATRICIA tree marked node reduction.

The storage improvement of a PATRICIA tree is significant, since all the nodes in the tree have at least 2 children, except the parent and child of marked nodes. However, the storage will not be $O(n)$ if we store the relevant path information in the edges with linked lists. A better way to store path information in edges is to store a string at the edge, specifying the string number, array start index and array end index. For example, if we see the string "17, 8, 11" stored in the edge, we go to x_{17} and take the values from position 8 to position 11 to find our values (Figure 7).

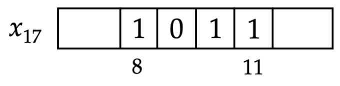

Figure 7: Retrieval of path information from string.

Exercise 4. Show that the number of nodes in a PATRICIA tree $\leq 2n$.

Now we look at data structures designed to prepare text for future search commands. Let T be a text (an array) of size n over some alphabet A. Our aim is to return information about the text, such as how many times or where a certain pattern occurs.

Suffix Data Structures

Definition 5. A **suffix trie** is a trie for all n suffixes of T.

Let $T = 001011$. Then, Figure 8 has the list of suffixes of T. The suffix trie for T would be a binary tree with each leaf and marked node corresponding to a suffix (Figure 9). The number stored in the leaf or marked node is the index of T where the suffix begins.

```
          1
        1 1
      0 1 1
    1 0 1 1
  0 1 0 1 1
0 0 1 0 1 1
```

Figure 8: The suffixes of $T = 001011$.

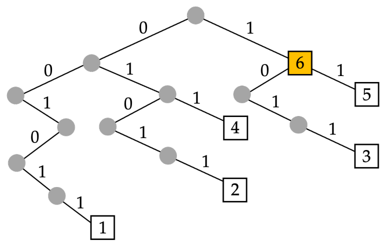

Figure 9: The suffix trie for $T = 001011$.

This allows us to find patterns in the text T. For example, if we wanted to know all the substrings of T that start with "01", we could take the path "01" in the typical way that one navigates a binary tree, and view the resulting subtree. All the integers that are stored in the leaves and the marked nodes of this subtree are indices of T where the pattern "01" is found. Then, you know how many matches there are, and where these matches are in the text T. Not only that, but if you perform an in-order traversal, the substrings will be sorted lexicographically.

This example easily extends to k-ary position trees when an alphabet of size k is used to construct T. However, suffix tries blow up the storage required as well. This leads us to our next data structure, the suffix tree.

Definition 6. A **suffix tree** is the PATRICIA version of the suffix trie. We perform compactification on unbroken chains of unmarked nodes with one child in the suffix trie, exactly as described for PATRICIA trees. The new edge will store the start and end indices of the substring of T corresponding to the edges that have been replaced (Figure 10).

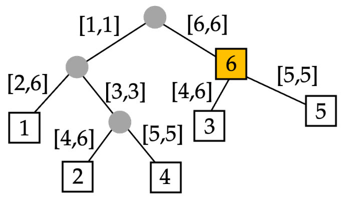

Figure 10: The suffix tree for $T = 001011$.

WE WILL REQUIRE $O(n \log n)$ bits to store the suffix tree. We are asked to show in Exercise 4 that PATRICIA trees have $\leq 2n$ nodes. This property also applies to suffix trees. The storage required per edge is $O(\log n)$, since the maximum number of bits required to store the start and end indices is $2\lceil \log n \rceil$. With $O(n)$ nodes and $O(\log n)$ bits per edge, storage of the suffix tree requires $O(n \log n)$ bits.

This data structure is used for the Oxford English Dictionary, as it is extremely space efficient and easy to search. It is also easy to construct: a more efficient but very complicated algorithm can create a suffix tree in $O(n)$ time, but we won't cover this.

IF WE WANT TO MAKE CHANGES to T after constructing a suffix tree, then adding one element to the front at most affects one path, since this element will be added to a leaf. However, adding an element at the end means it will have to be added to every leaf. The way to build a suffix tree or trie, then, is to insert suffixes from right to left.

Definition 7. A **suffix array** is a sorted array of all suffixes in T.

We can construct a suffix array by looking at all the leaves of a suffix tree in lexicographical order. In the case of the ordering of marked nodes, we assume that the ancestor is before the descendant. For example, when $T = 001011$ as above, the suffix array is $[1, 2, 4, 6, 3, 5]$. We can use binary search with a string to see if the pattern comes before or after in the array.

However, in terms of storage, suffix arrays still require $O(n \log n)$ bit space, since we have n entries in the array, and each entry requires $\leq \lceil \log n \rceil$ bits to store.

Pattern Search / Matching

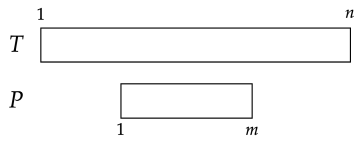

Figure 11: Pattern search for P in T.

Now we turn to an example of algorithms involving strings. The setup is: let the text T and the pattern P be unknown, so we cannot conduct any pre-processing. We want to determine if the pattern P of size m is contained in text T of size n, where $n \geq m$. We may additionally want to determine how many times P repeats in T, or where it is located within T.

We could easily do this with our suffix tree as described above. However, there are shorter algorithms to do this, some which take as little as $O(n)$ time. We will study the KMP algorithm, devised by Knuth, Morris and Pratt[5]. Another way to do this is Rabin-Karp[6], but this requires knowledge of hashing which is not covered this year.

[5] Knuth et al. [1977]

[6] Karp and Rabin [1987]

CONDUCTING THE PATTERN SEARCH consists of 2 steps:

1. Prepare the pattern, by making a magic table M.
2. Perform the KMP algorithm.

STEP 1: Build a magic table $M[1,\dots,m]$

To build the magic table, for each $k \in \{1,...,m\}$, we define

$$M_k = \max\{0 \le j < k : P[1,\dots,j] = P[k-j+1,\dots,k]\}.$$

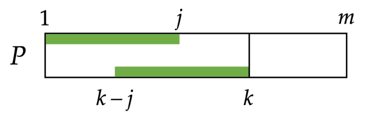

Figure 12: $P[1,\dots,j] = P[k-j,\dots,k]$.

This means that the k-th value in the magic table is the largest value j for which the pattern from 1 to j is the same as the pattern from $k-j$ to k. This is visualised as the two green areas having equal patterns in Figure 12. If there is no j where the patterns match, $M_k = 0$. If $j = k$ then the trivial solution is $M_k = k$, so we restrict $j < k$. We can construct M with the following algorithm:

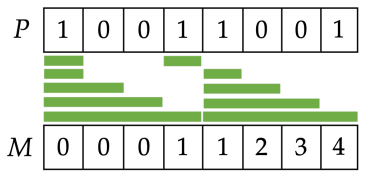

Figure 13: Magic table M for P = 10011001.

BUILD-MAGIC-TABLE(P, M)

```
$j = 0, k = 1, M_1 = 0$
while $k < m$
    if $P[j+1] = P[k+1]$
        $M_{k+1} = j+1$
        $j = j+1, k = k+1,$
    else if $j > 0$
            $j = M_j$
    else    // $j = 0$
        $M_{k+1} = 0$
        $k = k+1$
```

This algorithm works by starting with two pointers $j = 0$ and $k = 1$, where $M_1 = 0$ by definition. We only increment $j = j+1$ if $P[j+1] = P[k+1]$. Thus, if $j > 0$, then we know that $P[1,\dots,j] = P[1,\dots,k]$. Then, to check if $P[1,\dots,j+1] = P[1,\dots,k+1]$, we just need to check if $P[j+1] = P[k+1]$. If this is true, then $j+1$ must be the largest index where this overlap occurs. We set $M_{k+1} = j+1$, $j = j+1$ and $k = k+1$, and keep iterating to see if the shared substring continues.

Otherwise, since $P[j+1] \neq P[k+1]$, denoted by the red shaded region in Figure 14, then $M_{k+1} \neq j+1$. If $j > 0$, then we know that

$P[1,\ldots,j] = P[1,\ldots,k]$ as explained above. This implies that the tail of $P[1,\ldots,j]$ is the tail of $P[1,\ldots,k]$, as denoted by the yellow shaded regions in Figure 14. For any other index i, the largest common substring between $P[1,\ldots,i]$ and the tail $P[j-i,\ldots,j]$ is found when $i = M_j$ based on our definition of M. Thus, we set $j = M_j$, and continue iterating to check if $P[M_j+1] = P[k+1]$. This explains the else if statement.

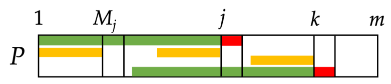

Figure 14: Construction of the magic table M by iterating through M_j's.

However, if $j = 0$ and $P[j+1] \neq P[k+1]$, then we have run through the entire pattern and not found a match. This implies that $M_{k+1} = 0$, and we can continue to build the magic table for when $k = k+1$.

THE TIME COMPLEXITY of the BUILD-MAGIC-TABLE algorithm is $O(m)$. Note first that both k and $k-j$ are monotonically increasing. Each time we go through the while loop, either k increases by one, or $k-j$ increases (in the $j = M_j$ step). Both can happen at most m times.

STEP 2: KMP-Algorithm

Now we can move on to the KMP-Algorithm, which uses the table we just constructed. At the beginning of the algorithm, we compare the $P[1]$ to $T[1]$, so P and T can be thought of as "lined up" at their first index (Figure 15). After this comparison we essentially "slide" P along T until we find a match, repeating this process until the largest match has been found or T has been exhausted (Figure 16).

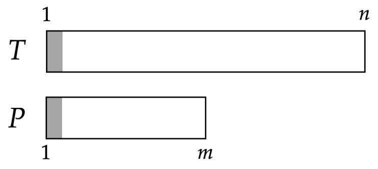

Figure 15: Alignment of T and P at the first index.

KMP-ALGORITHM(T, P, M)

```
1   $i = j = 1$     // initialization
2   while $i \leq n$
3       if $T[i] = P[j]$
4           $i = i+1$, $j = j+1$
5       else if $j > 1$
6           $j = M_{j-1} + 1$
7       else    // $j = 1$
8           $i = i+1$
9       if $j = m+1$
10          return $i - m$ and halt
11  return “no match"
```

This algorithm works by starting with two pointers, $i = j = 1$. We only increment $j = j+1$ if $T[i] = P[j]$. Thus, if $j > 1$, we know that $T[1,\ldots,i-1] = P[1,\ldots,j-1]$. Then, to check if $T[1,\ldots,i] = P[1,\ldots,j]$, we just need to check if $T[i] = P[j]$. If this is true, then we can continue iterating through the loop to check for more matches.

Otherwise, since $T[i] \neq P[j]$ denoted by the red shaded region in Figure 16, then the P no longer matches T at the start point s_1.

If $j > 1$, then we know that $T[1,\ldots,i-1] = P[1,\ldots,j-1]$ as explained above. This implies that the tail of $T[1,\ldots,i-1]$ is the tail of $P[1,\ldots,j-1]$, as denoted by the yellow shared regions in Figure 16. For any other index k, the largest common substring between $P[1,\ldots,k]$ and the tail $T[i-k,\ldots,i]$ is found when $k = M_{j-1}$ based on our definition of M constructed before. Thus, we set $k = M_{j-1}$, effectively sliding the pattern P until M_{j-1} is lined up with $i-1$, at a new start point s_2.

However, if $j = 1$ and $T[i] \neq P[j]$, then we have run through the entire pattern and not found a match between $T[i-j,\ldots,i-1]$ and $P[1,\ldots,j]$. Thus, we set $i = i+1$ and continue iterating to try and find a match.

We know we have found a match when $j = m+1$, since this means that $T[i-m,\ldots,i-1] = P[1,\ldots,m]$, which implies that there is a complete match. Otherwise, if we run through the entire pattern where $i \leq n$ and there is still no match, we return "no match".

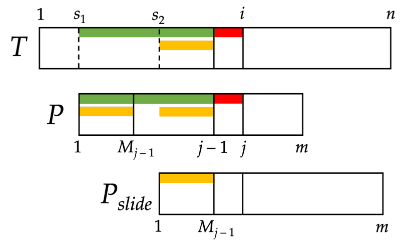

Figure 16: Sliding P along T to find a pattern match.

The time complexity of the KMP-Algorithm is $O(n)$. Note that both i and $i-j$ are monotonically increasing. In each iteration of the while loop, either i increases by 1 (n iterations), or $i-j$ increases by at least one ($\leq n$ iterations).

Exercise 8. Modify the code of the KMP-algorithm to return all matches.

References

R. de la Briandais. File searching using variable length keys. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), page 295–298. Association for Computing Machinery, 1959. ISBN 9781450378659.

E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

D.E. Knuth, J.H. Morris Jr., and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

D.R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.