Text Searching and Processing Notes

Vishnu

Saturday 11th May, 2019

1 Alphabet and String descriptions

1.1 Foundations

An **alphabet** ($\sum$) is a finite non-empty set. Members of an alphabet are called **letters**. A sequence of letters is called a **string**.

The set of all possible strings from an alphabet $\sum$ is represented with $\sum^*$: this includes the **empty string** (a sequence of length 0), which is denoted with ε. Lastly, the length of a string s is denoted as $|s|$, which means strings can be indexed with s[i] *(zero-indexed)*. **e.g.**

$$\sum^* \text{ for } \sum = 0, 1 : \varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots, 01100011, 1001010, \ldots$$

Strings of length 2 from $\left(\sum = a, b, c\right)$: $aa, ab, ac, ba, bb, bc, ca, cb, cc$

1.1.1 String Operators/Definitions

x, y, u and v are strings for all of the below.

Identity x=y, which denotes: $|x| = |y|$ and $x[i] = y[i]\ \forall i \in [0, |x|)$
  e.g decaf=decaf, han $\neq$ hands

Concatenation xy denotes the letters of x followed by the letters of y
  e.g concat(play,ground) = playground

Factor/Substring x is a factor of y if y can be written as uxv for some u,v
  e.g port is a factor of sports: s + port + s

Proper Factor x is a factor of y and $x \neq y$ (i.e u and v aren't both ε)
  e.g hall is a factor of hall, all is a proper factor

(Proper) Superstring If x is a (proper) factor of y, y is a (proper) superstring of x
  e.g hikers is a superstring of hikers and a proper superstring of ike

Prefix x is a prefix of y if y can be written as xv for some v
  e.g boy is a prefix of boyfriend

Suffix x is a suffix of y if y can be written as ux for some u
  e.g ring is a suffix of earring

Occurrence if x is a factor of y, x *occurs* in y. We can mark these occurrences by the start position (first letter of x in y) or end position (last letter of x in y).
  e.g ab occurs in abacbab at (0,6) for start positions and (1,7) for end positions

1.2 Powers and Primitives

Powers For a string x, x^0 is ε and x^k is x^{k-1}: essentially, x^n is x repeated n times, so is undefined for negative numbers.
Similar to real numbers, if $x^a = y^b$ and $a > 0, b > 0$ then x and y are both powers of another string z.
e.g ab^3 = ababab, $\text{aa}^2 = \text{aaaa}^1$ (both are powers of a)

Primitive A primitive string is one that isn't a power of any other string (i.e if x=a^b, then a =x and b=1). A test for this is if the string x only appears twice (as a prefix and suffix) in x^2

Root and Exponent A non-empty string x can be written as $(root)^{exponent}$, where the root is a primitive string.
e.g abc = abc^1 , abababab = ab^4

1.3 Conjugate Strings

Two non-empty strings x and y are conjugate if they can be written if x = uv and y = vu for some strings v,u (Implying that x and y are the same length). v or u can be ε, but not both.

$$\text{x and y are conjugate} \iff \text{root(x) and root(y) are conjugate}$$
$$\text{x and y are conjugate} \iff \text{there is a z such that xz=zy}$$

Proof of the first:

$\implies$ Given: x = uv = a^n, y = vu = b^m.
This implies a has the same prefix as u and suffix as v

$\impliedby$ Given root(x) = ab , root(y) = ba $\to$ x = $(\text{ab})^n$, y=$(\text{ba})^n$.
We can write these as x = $\text{a(ba)}^{n-1}\text{b}$, $(\text{ba})^{n-1}\text{ba}$.
With u = a and v = $(\text{ba})^{n-1}\text{b}$, x=uv and y = vu.

Proof of the second:

$\implies$ Given: x = uv, y = vu. By appending u: xu = uvu, uy = uvu $\to$ z = u

$\impliedby$ Given: xz = zy. Therefore we know that either x and z have a common prefix, or x is a prefix for z $\to$ x = ab, z = $x^n a$ (n=0 means common prefix, else x is prefix).

$$\begin{aligned}
xz &= zy \quad \textit{(Expanding z)} \\
xx^n a &= x^n ay \quad \textit{(Removing } x^n \textit{ from prefix of both sides)} \\
xa &= ay \quad \textit{(Expanding x)} \\
aba &= ay \quad \textit{(Removing a from prefix of both sides)} \\
ba &= y
\end{aligned}$$

1.4 Periods and Borders

Period p such that x[i] = x[i+p] $\forall i \in [0, x-p)$. i.e the length of a substring that repeats itself within x (note that it need not finish repeating). By definition, $|x|$ is a period of x for any string x.

per(x) The smallest period of a string.

Border a proper factor of x that is a prefix and a suffix (these can overlap)

border(x) The longest border of a string.

* per(x) + $|border(x)| = |x|$

e.g. x = abbacabba → Periods = [6,10]; Borders:[ε, a, abba]
e.g. x = aabaabaa → Periods = [3,6,7,8]; Borders:[ε, a, aabaa]

Using the above descriptions, we can say that any border for a string s is either:

- border(s)
- A border of border(s)

We can also describe a period p of a string s as:

- s is a prefix of y^k, where y is a string of length p
- s = yw = wz, where y and z are strings of length p and w is the border

2 Sequential String Matching

Checking if a string x is present in string y. This can be done naively in O($|x||y|$), by checking x against each position of y. If there is a mismatch, we increment the checking position of x by 1 and check again.

2.1 MP Algorithm

Since we know that that the prefix of s repeats on per(s), we can instead shift by the per of the portion of the string that matched. Since we know that the shifted part matches, we don't need to check it again, so we can then just start checking from the border of the string. In the algorithm, we generate the border for each substring of x and use that to shift/check.

Border Algorithm (where border[i] represents the border for substring x[0:i]):

A Border[0] = -1 *(This is where the string doesn't match at all, so we start checking from the beginning.)*

B for i in range (0, $|x|$):

  1 j = border[i] *(We know that any border is either border(s) or a border for border(s), so we can use previous values to speed up the search)*

  2 while j≥ 0 and x[i]≠x[j]:

    j = border[j]

  3 border[i+1]=j+1

In the algorithm, we 'shift' x to the border[$|s|$], were s was the length of the match between x and y. This effectively moves the start of x by per(s), and the next iteration checks from after the matching section.

MP(x,y):

A i = 0, j = 0 *(i tracks the position in x, j tracks the position in y)*

B while j < $|y|$:

  1 while $i = |x|$ or $x[i] \neq x[j]$: *($i = |x|$ to move per(x) after finding a full match, $x[i] \neq x[j]$ for a mismatch)*

    j = border[j]

  2 border[i+1]=j+1

e.g. x = abacabacab, y = ababacadab, u = matching substring
j=0, i=0 → u=a
j=1, i=1 → u=ab
j=2, i=2 → u=aba
j=3, i=3 → mismatch: i = border[3] = 1

j=4, i=2 $\rightarrow$ u = ab
$\vdots$
j=8, i=6 $\rightarrow$ u = abacaba
j=9, i=7 $\rightarrow$ mismatch: i = border[7] = 3
j=10, i=4 $\rightarrow$ mismatch: i = border[4] = 1
$\vdots$

2.2 KMP Algorithm

The KMP Algorithm uses strict borders and interrupted periods, but is otherwise the same.

Interrupted Period A period of the string where the repeat doesn't finish ($|s| - |strict\_border(s)|$)

Strict Border For a string x and substring u, w is a strict border for u if:

1. w is a border for u
2. wt is a prefix for x, ut is not (t is a single character)

Effectively, a strict border is one where the border doesn't continue past the end of the string. If we consider u=x, all borders are strict borders.

Strict Border Algorithm (where KMP_next[i] represents the strict border for substring x[0:i]):

A kmp_next[0] = -1, k = 0 *(This is where the string doesn't match at all, so we start checking from the beginning.)*

B for i in range (0, $|x|$):

  1 if x[i] = x[k]:

    i. kmp_next[i] = kmp_next[k]

  2 else:

    i. kmp_next[i] = k
    ii. do
    $k = kmp\_next[k]$
    while $k \geq 0$ and $x[i] \neq x[k]$

  3 k = k+1

C kmp_next[m] = k

k marks the end of the border as a prefix, **i** as a suffix: if they're the same it's not a strict border so we can re-use values, if they're different that means we've found a strict border, but have to re-calculate the distance between k and i.

2.3 String Matching Automaton

To find the string x in y, we make an automaton that accepts the sequence x, then give in the letters of y till a match is found. For space efficiency, we try to find the smallest automaton possible. Forward transitions imply a match, and backward transitions occur on a mismatch: we can make these smarter using periods and borders.

Each state of the automaton represents u, a prefix of x, and the transitions each represent a letter t. Forward arcs go from u to ut if ut is a prefix of x, and to the initial state (ε) otherwise. Backward arcs go to vt, where vt is the longest suffix of ut that is a prefix of x (effectively a strict border of u).

3 Dictionary Matching

Rather than matching a single string, this method matches multiple strings (a dictionary) from a known language. The output is the full set of occurrences of every string in the dictionary in the given string.

Trie ($\tau(X)$) A tree where each node represents a prefix of a string from X (a list of strings): it essentially matches to every string in X

Dictionary Matching Automaton (D(X)) An automaton where each state is a prefix of a string in X, terminal states represent the strings in X, and arcs are in the form: (source, transition, destination)
The destination for a transition T from a source U is the longest suffix of UT that matches prefix of a string in X. It's effectively a multiple string matching automaton.

3.1 DMA With Failure Function

In a basic DMA, a lot of links are repeated, so we can reuse them using relations between the nodes. If we get to a node U that doesn't have a forward transition for T, we can go to fail[U] and apply T there, rather than directly moving to h(UT). fail[U] is pre-computed for every U with **TargetByFail**(fail[u], a), and is done in a breadth-first manner (a is the last character of U). The generation algorithm is:

TargetByFail(U,T):

A while $u \neq Nil$ and $transition(U, T) = None$ *(U is a non-empty string which doesn't have a transition for T)*

U = fail[U]

B if $U = Nil$

return Initial State *(The algorithm is unable to find a match, return to the empty string)*

else

return transition(U,T) *(We found a match, so apply T and move on)*

Nil is a constant which means 'go to initial state', and we set fail[ε]= Nil.
This method also easily allows us to identify terminal states: if fail[X] is a terminal state, then X is a terminal state.

Rather than separately checking if there is a valid transition, the algorithm calls TargetByFail at each step since if there is a valid transition it will just apply it.

To further optimise transitions:

1. If fail[x] doesn't have the required transition, we can set fail[x] = fail[fail[x]]
2. If every transition from x and fail[x] all lead to the same place, we can set fail[x] = fail[fail[x]

4 Searching through a Set of Strings

The inverse of dictionary searching, this takes a lexicographically (alphabetically) sorted set of strings (L) and either:

- Searches for a string X, and if not present finds where it would be placed
- Finds the strings that have a string X as a prefix

The methods use LCP(a,b), which is the longest common prefix between strings a and b.

4.1 Binary Search

An simple method that uses constant space and is $O(|x| * log|L|)$ is binary search.
Binary Search(L, X)

A d=-1, f=$|L|$

B while $d + 1 < f$:

1 $i = int((d + f)/2)$, $l = lcp(x, L[i])$

2 if $l = |x|$ and $l = |L[i]|$:

return i *(The LCP of both strings = length of both strings, so they're equal)*

else if $l = |L[i]|$ OR ($l \neq |x|$ and $L[i][l] < x[l]$):

d = i *(L[i] matches but is shorter than X OR L[i] partially matches, and the next character is smaller in L[i] than X → X is lexicographically greater than L[i])*

else

f = i

C return (d,f) *(X isn't in the list of strings, but if it was would be between d and f (f=d+1))*

4.2 Binary Search Tree

To make the above easier for multiple searches, and for use in the next algorithm, we can make a binary search tree of indices. Each node is of the form(d,f).

Root: (-1, $|L|$)

Children: (int(d+f/2),f) , (d,int(d+f/2))

Leaves: (x,x+1)

4.3 LCP Search

Effectively the Binary Search with some extensions, this method reduces the overall number of iterations and comparisons by making more use of LCPs. In addition to d, f and i, the algorithm keeps track of l_d and l_f, the LCP between X and L[d]/L[f] respectively.

The algorithm uses the following three cases, all of which assume that $l_d < x < l_f$ (x is somewhere between d and f) and use $l_{if} = lcp(L[i], L[f])$:

1. $l_d \leq l_{if} < l_f \rightarrow$ x has more in common with f than i does with f $\rightarrow$ move d=i
   X is somewhere between i and f, and $l_i = l_{if}$
2. $l_d \leq l_f < l_{if} \rightarrow$ x has less in common with f than i does with f $\rightarrow$ move f=i
   X is somewhere between d and i, and $l_i = l_f$
3. $l_d \leq l_f < l_{if} \rightarrow$ f has the same prefix for x and i $\rightarrow$ check L[i] and x from index l_f

These cases can also be flipped, checking against d rather than f.

LCP Search(L, X)

A d=-1, f=$|L|$, l_d=0, l_f=0
(Since L[d] and l[f] don't exist, lcp with anything is 0 by default)

B while $d + 1 < f$:

1 $i = int((d + f)/2)$, $l_{if} = lcp(L[i], L[f]), l_{id} = lcp(L[i], L[d])$
2 if $l_d \leq l_{if} < l_f$: *(Case 1)*
    d=i, l_d=l_if *(Move d up, $l_d = l_i = l_{if}$)*
  else if $l_d \leq l_f < l_{if}$: *(Case 2)*
    f = i *(Move f down, l_f already equals l_i)*
  else if $l_f \leq l_{id} < ld$: *(Case 1 flipped for d)*
    f=i, l_f=l_id *(Move f down, $l_f = l_i = l_{id}$)*
  else if $l_f \leq ld < l_{id}$: *(Case 2 flipped for d)*
    d = i *(Move d up, l_d already equals l_i)*
  else *(Case 3 for both, so a manual check)*
  i. $l = max(l_d, l_f)$
  ii. $l = l+$ lcp(x[l:],L[i][l:]) *(We know the first l match, so check after that)*
  iii. if $l = |x|$ and $l = |L[i]|$:
      return i
    else if $l = |L[i]|$ OR ($l \neq |x|$ and $L[i][l] < x[l]$):
      d = i, $l_d = l$
    else
      f = i, $l_f = l$

C return (d,f) *(X isn't in the list of strings, but if it was would be between d and f)*

5 Suffix Trees

For a string X, a suffix tree is a Trie for the set of suffixes of X (normally represented as $X_{[0...n]}$). One application of such a tree is string matching: any substring of X has to be a prefix of a string in $X_{[0...n]}$.

Naively constructing a tree gives $O(n^2)$ nodes: the following algorithm brings that to $O(n)$, with the following assumptions/constraints:

- The alphabet's size is constant
- All the nodes that represent a suffix will be leaves
- No suffix is a prefix for another suffix : this can be achieved by adding a new character that is not part of the language (usually $) to the end of the string

Some notation for the rest of the section:

$\boldsymbol{S_u}$ String represented by node u

child(u,c) node connected from u where the connecting edge starts with character c

parent(u) node that connects to u

depth(u) $|S_u|$

start(u) starting position of S_u in X

$\boldsymbol{S_u =}$ T[start(u) : start(u)+depth(u)-1]

edge(parent(u), u) = S_u[depth(parent(u) :]

5.1 Locus

In the context of a suffix tree, a locus is a pair (u,d) where:

$$depth(parent(u)) < d \leq depth(u)$$

Effectively, it represents the node (and string) that would be at depth d along the edge (parent(u), u) in the uncompacted tree. Every factor of X has a locus in the suffix tree.

5.2 Brute Force Construction

To make a tree, we add in suffixes one at a time, starting from the longest:

1. If the new suffix has a prefix in common with an existing suffix, we find the locus where they diverge, make a new node there, then make a leaf connected to that
2. Otherwise, or if there are no existing nodes, we make a leaf connected to the root

BruteForce($T_{[0...n]}$):

A root = empty_node(), depth(root) = 0

B u=root, d=0

C for i in range(0, n):

1 while $d = depth(u)$ and $child(u, T[i + d])\, exists$: *(Common Prefix)*

i. u = child(u,T[i+d]), d+=1

ii. while $d < depth(u)$ and T[start(u)+d] = T[i+d]: *(Traverses till it finds the difference)*

d+=1

2 if $d < depth(u)$:

u = CreateNode(u,d)

3 CreateLeaf(i,u,d)

4 u=root, d=0

In the brute-force algorithm, to add a suffix S, if it finds a node (say N) with a common prefix it traverses S_N till it finds a difference or reaches the end of the string. If it finds a difference it creates the new node then a leaf, otherwise it creates a leaf node at the end.

However, if it reaches the end of the string that means S=S_N (which is impossible, two suffixes can't be identical) or S is a prefix of S_N (also impossible due to our assumption). It only reaches the end of a string after it breaks out of the loop in 1: if it fully matches a string before that it means a locus so it just loops, and so there are no issues.

5.3 Suffix Links

For a node u, slink(u) is the node n where S_n is the longest proper suffix of S_u . For the root, this is undefined, but set as slink(root) = root.
If we take slink(u) = S_u[1:] for all u, depth(slink(u)) = depth(u)-1. Below is the proof that this exists for all u:

- Leaf Nodes: Every suffix is present, so this is trivially true
- Internal Nodes: If an internal node u exists, there's a division after a character c between two edges. This character (and split) will re-appear in any suffix of S_u that contains c: including $S_u[1:]$, so a node $S_u[1:]$ will be created

ComputeSuffixLink(u):

A d = depth(u), v = slink(parent(u))

B while $depth(v) < d - 1$:

v = child(v, X[start(u)+depth(v)+1]) *The next node in the substring u[1:]*

C if $depth(v) > d - 1$: *If the algorithm overshot and s[1:] doesn't exist yet*

Create the node s[1:], v = new node

D slink(u) = v

The algorithm effectively goes to the suffix link for the parent, then traverses nodes to find the required suffix link. If it hasn't been created yet, it finds the first node that overshoots (i.e u[s:1]+x) then inserts a new node before that.

5.4 McCreight's Algorithm

Effectively the Brute-Force algorithm with a single improvement: rather than searching from the node at each iteration, it instead goes to the suffix link of the leaf's parent (which by definition will be a prefix of the next suffix to be added to the tree) and finds the divergence from there. If the leaf's parent doesn't have a suffix link, it creates one at the same time it creates the leaf.

BruteForce($T_{[0...n]}$):

A root = empty_node(), depth(root) = 0

B u=root, d=0, slink(root) = root

C for i in range(0, n):

1 while $d = depth(u)$ and $child(u, T[i + d])\,exists$: *(Common Prefix)*

i. u = child(u,T[i+d]), d+=1

ii. while $d < depth(u)$ and T[start(u)+d] = T[i+d]: *(Traverses till it finds the difference)*

d+=1

2 if $d < depth(u)$:

u = CreateNode(u,d)

3 CreateLeaf(i,u,d)

4 if slink(u) = null:

slink(u) = ComputeSuffixLink(u)

5 u=slink(u), d=max(d-1,0)

d = max(d-1,0) is the same as d = depth(slink(u)), since the suffix link is either one shorter or the root,

5.5 Generalised Suffix Tree

To create a single suffix tree for multiple strings, just use a different ending character (e.g $\$_1$,$\$_2$) for each. Both the brute-force and McCreight's algorithms can be used.

5.6 Suffix Tree Applications

5.6.1 String Matching

As mentioned before, string matching on the string becomes trivial. If x is the string to be found, simply find how many leaf nodes are descendants of x: that gives the number of occurrences of x in the string.

5.6.2 Number of Distinct Substrings

This method requires an uncompacted tree: the number of nodes is the number of distinct substrings. In a compacted tree, the number of nodes + the number of loci, since loci are effectively uncompacted tree nodes.

5.6.3 Longest non-unique substring

Since we know that all leaves are unique, and the closer to the start of the tree the shorter the string, the longest repeating substring is the deepest internal node. Any internal node implies that the string is a prefix for two (or more) substrings, so we know that it's non-unique.

5.6.4 Common Substrings

In a generalised suffix tree, each string has a unique ending character (e.g $\$_1$,$\$_2$). If an internal node is an ancestor of leaves with different ending characters, it's a common substring to the string corresponding to each character. Similary, the longest common substring is the deepest internal node with leaves with each ending character.

5.6.5 Matching Suffix Arrays

If we've pre-processed $T_{[0...N]}$, we can efficiently match $S_{[0...N]}$ against it. i.e if each suffix of S is present in T. The standard would be to just try to match each string, but since we know that $S_{n+1} = S_n[1:]$, we can use suffix links.

The algorithm follows S_n as far as possible: when it hits a mismatch at depth j (i.e S[n:j]), take the suffix link of node at j (i.e S[n+1:j]). If there is no mismatch, then all S_m where $m > n$ are present.

5.6.6 Longest Common Prefix Between Suffixes

For two suffixes of S, namely S_i and S_j, the longest common prefix is the deepest internal node that leads to both leaves. For more than two suffixes, it's the deepest internal node that lead to all the leaves.

6 Table of Prefixes

A **Prefix Table** for a string X is a table t such that:

$$t[i] = |lcp(X, X[i:])|$$

i.e the length of the lcp between X and suffix X_i. Naively, this can be constructed in $O(n^2)$, but by using previously computed values this can be $O(n)$.

6.1 Proofs for Algorithm

For $i > 1$:

$$g = max(j + t[j], 0 < j < i)$$
$$f = j\textit{(This is the j from g)}$$

Using the above values, X[f:g-1] is a prefix of X of length t[j] (g-1-f = t[j], and we know t[j] is the length of a match). Therefore, X[f:g-1] is a border of X[0:g-1].

Using the above, if $\boldsymbol{i < g}$ we can say that:

$$t[i] = \begin{cases} pref[i-f] & , if pref[i-f] < g-i \\ g-i & , if pref[i-f] > g-i \\ g-i+|lcp(x[g-i:|x|-1], x[g:|x|-1]| & , otherwise \end{cases}$$

As a border, x[0:t[f]]=x[f:f+t[f]-1]. Since f+t[f]=g and $i < g$, i $\in$ x[0:t[f]].

1. If t[i-f] < g-i, then the entire match occurs within x[0:t[f]]. The same match will occur for t[i] since it's the same string, so we just copy the value.
2. g-i is the distance between i and the mismatch. The same match occurs for t[i], so we know that the new string only matches till g as well, so the length of the match is g-i.
3. The match occurs to the end of the substring in the original, but we don't know if it ends or continues to match, so we check the remainder.

6.2 Prefix Table Algorithm

Prefixes(X):

A $pref[0] = |X|, g = 0, f = undefined$

B for i in range(1, n):

  1 if $i < g$ and $pref[i - f] \neq g - i$:

    pref[i] = min(pref[i-f],g-i) *(Case 1 and 2)*

  2 else:

    i. g = max(g,i), f=i *(if $i < g$ Case 3, otherwise regular comparison)*

    ii. while $g < |x|$ and $x[g] = x[g + f]$

      g++

    iii. pref[i] = g-f

This algorithm runs in $O(|x|)$: ii can run at most $|x|$ times across the entire run since g only increases, and B only runs $|x|$ times, so the total is $O(2|x| - 1)$.

This algorithm can also be used for string matching: append the pattern to be found P to the string S, then find pref[i]=len(P).