Kevin Gao

Notes on String Algorithms

With Applications in Computational Biology

TERRA-INCOGNITA.DEV

Contents

*These notes are primarily based on the book **Algorithms on Strings, Trees, and Sequences** by **Dan Gusfield** [1].*

Thanks Professor Gusfield for his great book.

*The part on indexing and compression is based on the book **Genome-Scale Algorithm Design** [2] and Ben Langmead's lectures.*

Part I

Comparison-Based Exact Matching

1

Naive Exact Matching

(Gusfield Section 1.1)

Terminology Confusion. Before starting the discussion of string matching algorithms, we should note the difference between a ***substring*** and a ***subsequence***. Given a string S, characters in a substring of S must occur contiguously in S; whereas characters in a subsequence may be interspersed gaps (or indels, as we call them in biology) and/or characters not in the original string.

Exact String Mathching Problem

Given a text string T and pattern P, the goal of the exact string matching problem is to find all occurrences of P in T.

The "Naïve" Algorithm

```
NAIVE-MATCH(P, T)
 1  matches = [ ]
 2  for i = 1 to |T| − |P| + 1
 3       match = TRUE
 4       for j = 1 to |P|
 5            if T[i + j] ≠ P[j]
 6                 match = FALSE
 7                 break
 8            if match
 9                 matches.APPEND(i)
10       return matches
```

Figure 1.1: Naive exact matching with $P = abxyabxa$ and $T = xabxyabxyabxz$.

T : $xabxyabxyabxz$
P : $abxyabxa$
$abxyabxa$
$abxyabxa$
$abxyabxa$
$abxyabxa$
$abxyabxa$

The naïve exact matching algorithm aligns the left end of P with the left end of T and compares the characters of P and T left to right until a mismatch is found, or until it reaches the end of P, in which case we report the position of P. Then, P is shifted to the right by one place. We repeat this procedure until the right end of P passes the right end of T.

Runtime of the Naive Algorithm

Let $n = |P|$ and $m = |T|$. The worst-case comparisons made by the naive algorithm is $\Theta(nm)$. We have the lower bound $\Omega(nm)$ when P and T contains the same repeated characters (e.g. $P = aaa$, $T = aaaaaaaaa$), in which case the algorithm makes $n(m - n + 1) \in \Omega(nm)$ comparisons.

2

Z Algorithm

(Gusfield Chapter 1)

Speeding Up the Naive Algorithm

The naive exact matching algorithm is stupid. It always shifts P by only one even if it knows for sure that the next shift will not yield a match. This gives us some ideas on how to improve the algorithm. If we can shift P by more than one character, but never shift so far as to miss the next occurrence of P in T, we can improve the runtime of the naive algorithm.

Doing this, however, requires us to have some prior knowledge of the pattern P or the text T.

Foundamental Preprocessing

A fundemental preprocessing is a generalized way to process the pattern P to gain knowledge of the pattern, independent of any particular algorithm.

Definition 2.1. *Given a string S and a position $i > 1$, let $Z_i(S)$ be the length of the longest substring of S that starts at i and matches a prefix of S.*

In other words, $Z_i(S)$ is the length of the longest prefix of $S[i \dots |S|]$ that matches a prefix of S.

Definition 2.2 (Z-box)**.** *For any position $i > 1$ where Z_i is greater than 0, the **Z-box** at i is the interval starting at i and ending at $i + Z_i - 1$.*

Definition 2.3. *For every $i > 1$, r_i is the right endpoint of the Z-box that begins at or befor position i (i.e. the closest Z-box to the left). More formally, r_i is the largest value of $j + Z_j - 1$ over all $1 < j \leq i$ such that $Z_j > 0$.*

Figure 2.1: Relations between i, l_i, r_i and the Z-box at l_i.

The linear-time computation of the Z_i values for all $i \in \{1, \ldots, |S|\}$ from the string S is called the ***fundamental preprocessing*** task.

The Z Algorithm

The ***Z algorithm*** is an algorithm for computing the fundamental preprocessing. The Z algorithm is similar to a dynamic programming algorithm in the sense that it uses memoized information to speed up the computation and reduce the number of comparison needed. Namely, assume there exists $j < i$ such that $j + Z[j] - 1 > i$, then we can use $Z[i - j + 1]$ to infer $Z[i]$.

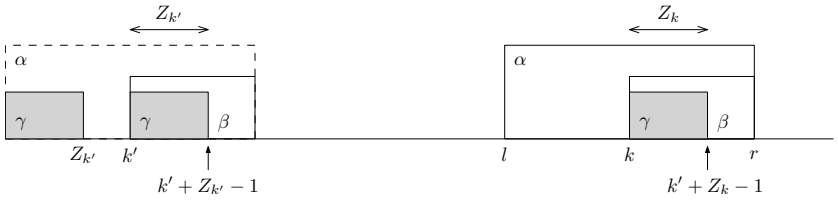


Figure 2.2: Case 2a: The longest string starting at k' that matches a prefix of S is shorter than $|\beta| = r - k + 1$. In this case, $Z_k = Z_{k'}$.

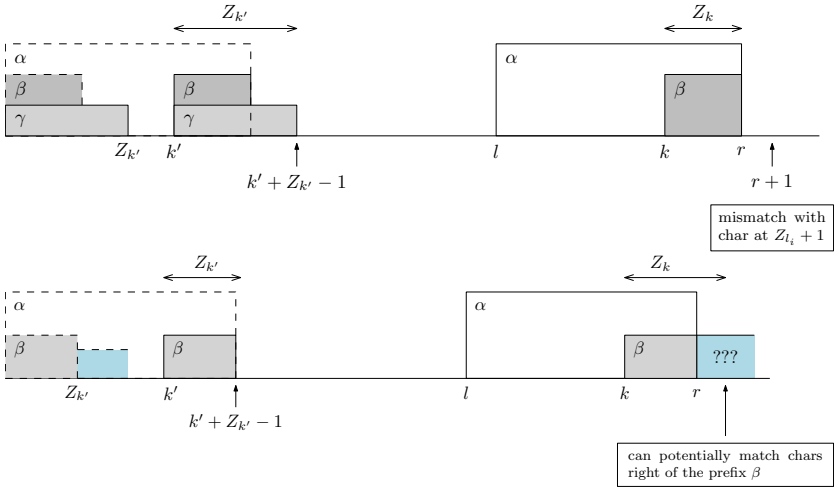


Figure 2.3: Case 2b: The longest string starting at k' that matches a prefix of S is at least $|\beta| = r - k + 1$. In this case, we continue to compare characters right of r with characters right of the $Z_{k'}$th character until mismatch.

Given Z_i for all $1 < i \leq k - 1$ and the current l and r, we can compute Z_k using the following procedure:

1. Case 1: If $k > r$, k is not in an existing Z-box. Compute Z_k by explicitly comparing each character starting at k with prefix of S.
2. Case 2: If $k \leq r$, k is in an existing Z-box, denoted α, that matches a prefix of S. Hence, character $S[k]$ also appears in the prefix at position $k' = k - l + 1$.

We also consider the substring $S[k \dots r]$, denoted β. By the same reasoning, β matches the substring $S[k' \dots Z_l]$.

(a) $Z_{k'} < |\beta|$. This implies that the longest string starting at k' that matches a prefix of S, γ, is shorter $|\beta|$. Then, $Z_k = Z_{k'}$, and l, r remain unchanged. Note that $|\gamma|$ can be 0.

(b) $Z_{k'} \geq |\beta|$. This means $S[k \dots r]$ is at least a prefix of S. However, Z_k might be larger than $|\beta|$. We cannot determine this solely based on the existing information as characters beyond the Z_lth character are not included in α. So, we compare the characters starting at position $r+1$ of S to the characters starting at position $|\beta|+1$ of S, until a mismatch occurs. Say the mismatch occur at position q. Then, $Z_k = q - k$, $r = q - 1$, and $l = k$. (If $Z_{k'} > |\beta|$, a mismatch occurs immediate after the Z_lth character because otherwise, the $(r+1)$th character matches the Z_lth character, and we would have a longer α and a larger Z_l)

COMPUTE-Z(S)

```
1  n = |S|
2  l, r, k = 1
3  Z = empty array of length n
4  for k = 2 to n
5      if k > r                                          // Case 1
6          // compute Z[k] from scratch
7          l, r = k
8          while r ≤ n and S[r − l + 1] == S[r]
9              r = r + 1
10         Z[k] = r − l
11         r = r − 1
12     else                                              // Case 2
13         k' = k − l + 1
14         if Z[k'] < r − k + 1                          // Case 2a
15             Z[k] = Z[k']
16         else                                          // Case 2b
17             q = r + 1
18             while q ≤ n and S[q − l + 1] == S[q]
19                 q = q + 1
20             Z[k] = q − k
21             l, r = k, q − 1
22 return Z
```

For a step-by-step animation of the Z algorithm, see
https://personal.utdallas.edu/~besp/demo/John2010/z-algorithm.htm

Running Time of the Z Algorithm

Theorem 2.4. *All the Z_i values can be computed by the Z algorithm with at most $2|S| \in O(|S|)$ comparisons.*

Proof. The time is proportional to the number of iterations, $|S|$, plus the number of character comparisons.

Each iteration that does any character comparison ends as soon as there is a mismatch, so there are at most $|S|$ **mismatches**. In every iteration where there is a match, r moves to the right by an amount at least as large as the number of matches. This implies that there are at most $|S|$ **matches**. Note that once a character results in a match, it is not compared again.

Every character comparison is either a match or mismatch, so the total number of comparisons is at most $2|S|$. □

Correctness of the Z Algorithm

Theorem 2.5. *At the kth iteration of the algorithm,* Compute-Z *correctly calculates Z_k, and l, r are updated correctly.*

Proof. By a case analysis.

Case 1: Trivial since it's just explicit comparison.

Case 2a: We claim that the substring at position k can match a prefix of S only for length $Z_{k'} < |\beta|$. Prove the claim by contradiction (suppose not, we would have a longer prefix matching the substring at k', contradicting the maximality of $Z_{k'}$).

Case 2b: β must be a prefix of S as established above. Characters beyond r are explicitly compared. □

Corollary 2.6. Compute-Z *correctly calculates Z_k for all $k \in \{2, \ldots, |S|\}$.*

Proof. By induction on k. □

Z Algorithm for Exact Matching

The Z algorithm lends itself to a linear-time algorithm for exact matching. Given a pattern P and text T, we construct a new string $P\$T$ where \$ is a ***separator*** (a.k.a. ***delimiter*** or ***sentinel***) such that $\$ \notin \Sigma$. We construct the Z-array for this new string. It takes $O(|P| + |T|)$

time to construct the Z-array. After this, we can simply make a pass through the Z-array and read the result from it. This takes $O(|T|)$ time.

Z-EXACT-MATCHING(P, T)

```
1  query = P + "$" + T
2  Z = COMPUTE-Z(query)
3  for i = |P| + 1 to |P| + |T| + 1
4       if Z[i] == |P|
5            pattern found at index i − |P|
```

3
Boyer-Moore

(Gusfield Chapter 2)

KEY IDEAS of the Boyer-Moore algorithm: right-to-left scan, bad character rule, good suffix rule.

Bad Character Rule

The intuition behind the bad character rule is as follows. Suppose we have the pattern P and text T, and the rightmost character of P is aligned to the character x in T, where $x \neq y$. If we know the position of the rightmost x in P, we can safely shift P by that amount to the right so that the x in P aligns with the x in T. Furthermore, if we know that x is not in P, we can shift P completely past the x in T. This intuition is formalized below, as shown in Gusfield's book.

Definition 3.1. *For each character x in the alphabet, let $R(x)$ be the position of the rightmost occurrence of character x in P. If x does not occur in P, $R(x) = 0$.*

Rule 3.2 (Bad Character Rule)**.** Suppose for a particular alignment of P against T, the rightmost $n - i$ characters of P match their counterparts in T, but the next character to the left $P(i)$ mismatches with its counterpart in T, say at position k in T. The ***bad character rule*** says that P should be shifted right by $\max\{1, i - R(T(k))\}$ places. That is, if the rightmost occurrence of $T(k)$ in P is in position $j < i$, then shift P so that the jth character of P is matched to the kth character of T (or completely shifting P past kth position if the kth character of T does not occurr in P). Otherwise, if $j > i$, shift P by one position.

Extended Bad Character Rule

Note that the bad character rule allows us to shift more than one character only when the mismatched character show up at a position left of the mismatched point. However, it is not as helpful if the character only occurs in P on the right side of the mismatched point. The ***extended bad character rule*** addresses this issue.

Rule 3.3 (Extended Bad Character Rule)**.** When a mismatch occurs at position i of P and the mismatched character in T is x, then shift P to the right so that the closest x to the left position i in P is matched with the x in T.

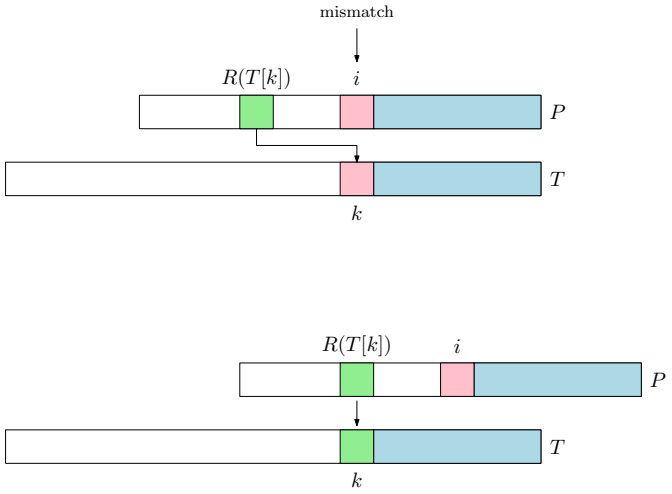


Figure 3.1: The bad character rule says to shift as much as possible so that a mismatch becomes a match.

Preprocessing for Bad Character Rule

The bad character rule is quite straightforward to implement. In the preprocessing for the bad character rule, we find, for each position i in the pattern P and for each characther $x \in \Sigma$, the position of the closest occurrence of x in P to the left of i. This uses an n matrix. However, the time complexity for building this lookup table and the space required for it could be massive, depending on the length of the pattern and the size of the alphabet. It is also not hard to see that this approach can be quite space inefficient and we can end up store duplicate information over and over.

Alternatively, we can scan P from right to left, and for each character $x \in \Sigma$, we keep a list of positions where x occurs in P (think hash table with chaining but just using another array instead of linked list). Since we scan P from right to left, the indices will be stored in

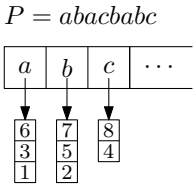


Figure 3.2: Table for storing information obtained from the bad character rule preprocessing.

decreasing order for each character. It takes $O(n)$ space. During the actual execution of the Boyer-Moore algorithm, we can query this table whenever we need to find the rightmost occurrence of character x left of index i. We can use *binary search* for this.

If the alphabet Σ is sparse, we can condense it by simply creating a mapping f from $\Sigma \to \mathbb{N}$. The two approaches can be implemented as follows, respectively.

```
CREATE-BAD-CHAR-TABLE(P, f)
1  table = [ ]
2  row = [0, ..., 0]
3  for i = 1 to |P|
4      c = P[i]
5      table.APPEND(row)
6      row[f(c)] = i + 1
7  return table
```

```
CREATE-BAD-CHAR-TABLE(P, f)
1  table = [ ]
2  for i = |P| downto 1
3      c = P[i]
4      table[f(c)].APPEND(i)
5  return table
```

Good Suffix Rule

Rule 3.4 (Good Suffix Rule)**.** Given an alignment of P against T, suppose a substring t of T matches a **suffix** of P, but a mismatch occurs at the next comparison to the left. Then, find, if exsits, the right-most copy t' of t in P such that t' is not a suffix of P. Shift P to the right so that t' in P is matched with t in T.

Essentially, the ***good suffix rule*** says we can shift as much as possible such that an *existing match does not become a mismatch*.

This rule is the weaker version of the good suffix rule used by Boyer and Moore's original publication. We now present a stronger version of the good suffix rule that allows us to prove properties of the Boyer-Moore algorithm more easily.

Strong Good Suffix Rule

Rule 3.5 (Strong Good Suffix Rule)**.** Given an alignment of P against T, suppose a substring t of T matches a **suffix** of P, but a mismatch occurs at the next comparison to the left. Then, find, if exsits, the right-most copy t' of t in P such that t' is not a suffix of P. Additionally, **we requires that the character left of t' in P differs from the character to the left of t in P.** Shift P to the right so that t' in P is matched with t in T.

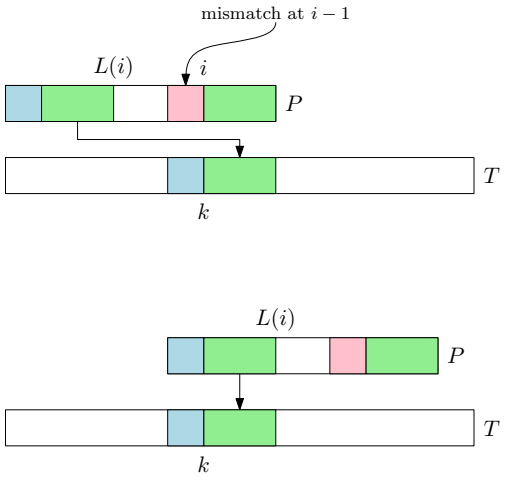


Figure 3.3: The good suffix rule says we can shift as much as possible so long as a match does not become a mismatch. The additional requirement in the strong good suffix rule ensures that we don't encounter the same mismatch at $P[i-1]$.

Note the additional requirement stated in the second-to-last sentence. The addition of this requirement ensures that we don't get the same mismatch. A copy of t is not worth looking at if it results in the same mismatch at the next character to the left. We call this rule the ***strong good suffix rule***. It should be obvious that it is safe to shift using the strong good suffix rule. That is, we won't miss any matches if we shift P using the strong good suffix rule. We formalize this idea of correctness in the following theorem.

Theorem 3.6. *The strong good suffix rule does not shift P past an occurrence in T.*

Proof. By contradiction. See Gusfield Theorem 2.2.1. □

Preprocessing for Good Suffix Rule

Definition 3.7. *For each i, let $L(i)$ be the **largest position** less than n such that string $P[i \ldots n]$ matches a suffix of $P[1 \ldots L(i)]$. If no such position exists, $L(i) = 0$.*

For each i, let $L'(i)$ be the largest position less than n such that string $P[i \ldots n]$ matches a suffix of $P[1 \ldots L'(i)]$ and such that the character preceding that suffix is not equal to $P[i-1]$. If no such position exists, $L'(i) = 0$.

By definition, $L'(i)$ gives the right endpoint of the rightmost copy of $P[i \ldots n]$ that is not a suffix of P, and whose preceding character does not equal to $P[i-1]$.

Definition 3.8. *For string P, $N_j(P)$ is the length of the **longest suffix** of the substring $P[1 \ldots j]$ that is also a **suffix** of the whole string P.*

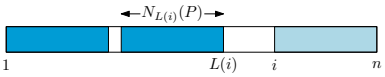


Figure 3.4: $L(i)$ gives the right-most copy of t that is not a suffix of the whole string.

Theorem 3.9. *$L(i)$ is the largest index j less than n such that $N_j(P) \geq |P[i \ldots n]|$.*

Proof. By definition of L, $L(i)$ is the largest index less than n such that $P[i \ldots n]$ matches a suffix of $P[1 \ldots L(i)]$.

Let $j = L(i)$ in the definition of N_j. Then, $N_j(P)$ is the length of the longest suffix of the substring $P[1 \ldots L(i)]$ that is also a suffix of the whole string. A suffix of the whole string must also be a suffix of $P[i \ldots n]$. Then, clearly, $N_j(P) \geq |P[i \ldots n]|$ because the substring $P[L(i) - N_j(P) + 1 \ldots L(i)]$ can possibly match more characters left of $P[i]$. □

Corollary 3.10. *$L'(i)$ is the largest index j less than n such that $N_i(P) = |P[i \ldots n]| = n - i + 1$.*

Proof. Immediate from Theorem 3.9. The definition requires that the characters left of $P[i]$ must not be contained in the suffix ending at $L'(i)$. □

One crucial observation is that N_j is, in fact, the exact reverse of Z_i when we talked about the Z-algorithm. Hence, $N_j(P)$ can be computed in linear time for all j in linear time by calling COMPUTE-Z on P^R (the reversal of P).

COMPUTE-L'(P)
1 $N =$ COMPUTE-Z(P^R)
2 **for** $i = 1$ **to** $|P|$
3 $\quad L'[i] = 0$
4 **for** $j = 1$ **to** $|P| - 1$
5 $\quad i = n - N[j] + 1$
6 $\quad L'[i] = j$
7 **return** L'

Definition 3.11. *For string P, $l'(i)$ is the length of the **longest suffix** of $P[i \ldots n]$ that is also a **prefix** of the whole string P. If none exists, $l'(i) = 0$.*

Figure 3.5: $l'(i)$ gives the index of the end of the longest prefix that matches a suffix of $P[i \ldots n]$.

Theorem 3.12. *$l'(i)$ is the largest index $j \leq |P[i \ldots n]| = n - i + 1$ such that $N_j(P) = j$.*

Proof. By definition of l', the prefix $P[1 \ldots l'(i)]$ is the longest prefix that matches a suffix of $P[i \ldots n]$. So $l'(i) \leq |P[i \ldots n]|$. A suffix of $P[i \ldots n]$ is also a suffix of P, so $P[1 \ldots l'(i)]$ also matches a suffix of P.

Let $j = l'(i)$ in the definition of N_j. Then, by definition of N_j, $P[l'(i) - N_{l'(i)}(P) + 1 \ldots l'(i)]$ is the longest suffix of $P[1 \ldots l'(i)]$ that matches a

suffix of P. The longest suffix of $P[1 \dots l'(i)]$ is $P[1 \dots l'(i)]$ itself, and indeed $P[1 \dots l'(i)]$ matches a suffix of P. Hence, $l'(i) = N_{l'(i)}(P)$. The maximality follows from the definition of $l'(i)$. □

COMPUTE-$\ell'(P)$

```
1  N = COMPUTE-Z(P^R)
2  // initialize l' array
3  for i = 1 to |P|
4      l'[i] = 0
5  // set entires of l' based on Theorem 3.12
6  for j = 1 to |P|
7      if N[j] == j
8          l'[|P| - j + 1] = j
9  // fill in l' for i's left of the longest suffix that matches a prefix
10 for i = |P| - 1 downto 1
11     if l'[i] == 0
12         l'[i] = l'[i + 1]
13 return l'
```

It may not be clear at first what the last for-loop on line 10-12 does.

It copies the values of l' from the right to fill in the blanks. Consider the example shown in Figure 3.6. $P[1 \dots l'(i')]$ is the longest prefix that matches a suffix of P. For any $i < i'$, $P[i \dots n]$ will not match a prefix of P. However, for those values of $i < i'$, $P[i' \dots n]$ is **still a proper suffix** of $P[i \dots n]$, so they "inherit" the l' values from $l'(i')$ – the left most position such that the entirety of $P[i' \dots n]$ matches a prefix of P.

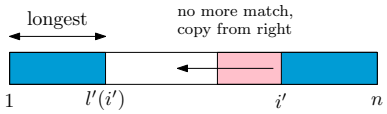


Figure 3.6: "Smear" to the left. This handles the cases when $P[i \dots n]$ itself does not match a prefix of P, but some **proper suffix** $P[i' \dots n]$ does.

Putting Everything Together

The Shifting Rules

We summarize the shifting rules here. Remember that we are scanning from right to left.

1. Bad character rule: Mismatch at i with $T[k] \neq P[i]$; Shift P to the right by $i - R(T[k])$.
2. Good suffix rule: Mismatch at i with $T[k] \neq P[i]$ but the suffix $P[i+1 \dots n]$ matches some substring of T
   (a) If $L'(i) > 0$, then shift P to the right by $n - L'(i)$
   (b) If $L'(i) = 0$, then shift P to the right by $n - l'(i)$

3. No mismatch and P is entirely matched with some substring of T, apply good suffix rule and shift P to the right by $n - l'(2)$.

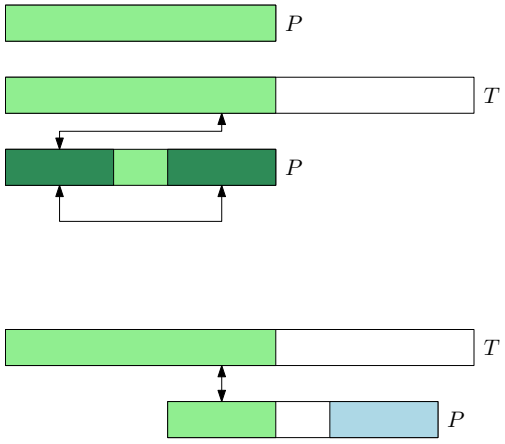


Figure 3.7: The special good suffix rule when P matches with T. We shift P to the right so that a prefix of P remains matched with T, if exists.

Implementation

For simplicity sake, we use I, the identity function when creating the bad character table. Again, in practice, if we are dealing with a sparse alphabet, we may want to change it to something else.

```
BOYER-MOORE(P, T)
 1  L' = COMPUTE-L'(P)
 2  l' = COMPUTE-ℓ'(P)
 3  R = CREATE-BAD-CHARACTER-TABLE(P, I)
 4  k = |P|
 5  while k ≤ |T|
 6      i = n, h = k
 7      while i > 0 and P[i] == P[h]
 8          i = i − 1
 9          h = h − 1
10      if i = 0
11          P found at position h in T
12          k = k + |P| − l'[2]
13      else
14          bc = i − R[T[h]]
15          gs = |P| − L'[i]
16          if gs == 0
17              gs = |P| − l'[i]
18          k = k + max{bc, gs}
```

4
Knuth-Morris-Pratt Algorithm

Part II

Suffix Trees

5
Suffix Trees

Suffix Tries

Let us recall the definition of a suffix.

Definition 5.1 (Suffix)**.** *For any string S, $S[i \dots j]$ is the **substring** starting at position i and ending at position j; $S[1 \dots i]$ is the prefix of S ending at i; and $S[j \dots |S|]$ is the **suffix** of S starting at position j. A proper substring, prefix, or suffix is a substring, prefix, or suffix that is neither the entire string S nor the empty string.*

Then, we define a trie and a suffix trie as follows.

Definition 5.2 (Suffix Trie)**.** *A **trie** is the smallest tree such that each edge is labeled with a character from the alphabet Σ, each node has at most one outgoing edge labeled with c for each $c \in \Sigma$, and each node has a key that is the concatenation of the edge labels along the path from the root to that node. A **suffix trie** is a trie where each root-to-leaf path represents a suffix.*

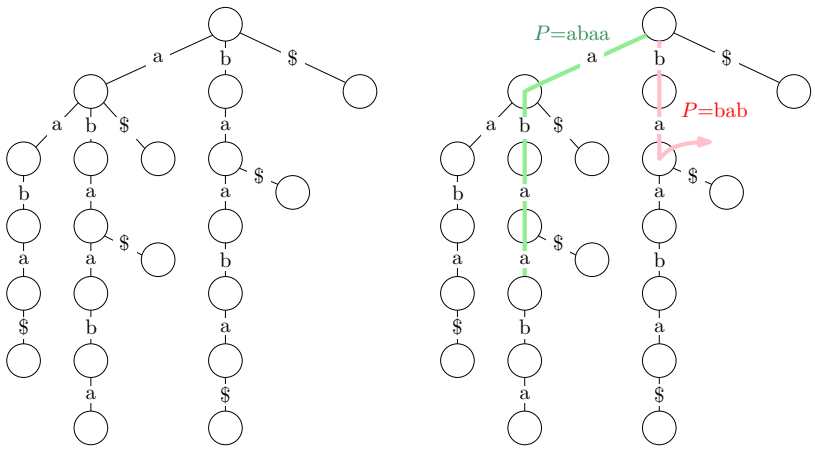


Figure 5.1: Suffix trie for $T = abaaba$. On the right: the search path for $P = abaa$ and $P = bab$. When searching for a pattern that is not in T, we "fall off" the trie.

In a regular tree (e.g. binary search tree), the key is stored at each node. In a trie, the keys are *implicitly* represented by the edge labels along the path. Figure 5.1 shows a suffix trie constructed for $T = abaaba$.

It is important to add the terminator character \$ at the end of the string. If we remove the terminator \$, it is not hard to see the result trie may no longer be a valid suffix trie. We assume that \$ is *lexicographically smaller than all characters* in Σ.

Search in Suffix Trie

SEARCH FOR PATTERN: It is easy to search for a pattern P given a suffix trie. We can **start from the root and follow the edges labeled with the characters** in P until we either finish reading the pattern and find a match, or "fall off" the trie, in which case we can return that a match is not found.

SEARCH-TRIE(P, T)

```
1  cur = T.root
2  for c in P
3      if c ∉ cur.edges
4          return FALSE
5      else cur = cur.edges[c]
6  return cur ≠ NULL
```

Assume that at each node, we maintain a hash table for each outgoing edges. Then, the algorithms runs in expected time $\Theta(|P|)$.

SEARCH FOR SUFFIX: Similarly, if we want to see if a given pattern P is a suffix of T, we can run the same algorithm and checks if the node at the end of the path has an outgoing edge labeled \$.

SEARCH FOR NUMBER OF OCCURRENCES: If we are interested in the number a pattern P occurs as a substring in T, we can run SEARCH-TRIE. Once we arrive at the end of our search path, we run a **depth-first search** from the node at the end of the search path and count the number of leaf nodes reachable from that node. Since a trie is a tree, DFS runs in $O(|V|)$ time. In this case, it takes $O(|P| + |T|)$ time to find the number of occurrences of a given pattern.

SEARCH FOR LONGEST REPEATED SUBSTRING: Find the deepest (internal) node with more than one children.

Space Complexity of Suffix Trie

Construct a Suffix Trie

Suffix Tree

From Suffix Trie to Suffix Tree

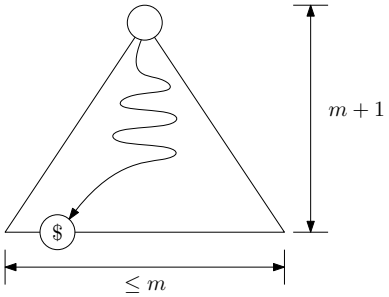


Figure 5.2: Max width and height of a suffix trie. The path from the root to the deepest leaf represents the longest suffix (the whole string plus the terminator).

6

Constructing Suffix Trees

Part III

Dynamic Programming

7
Edit Distance

Part IV

Compression and Indexing

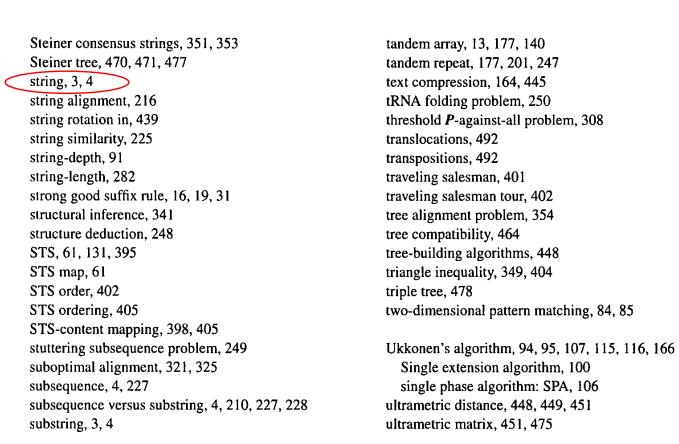
8

Entropy and Information

9
k-mer Index and Suffix Array

Idea Behind Indexing

Suppose you have a textbook and you want to look for a certain term, say, the word "*string*". It would be really time-consuming, and in some cases, impossible to look for the term by going through the book page by page. That's why most books usually (and hopefully) have an index at the end, which list the pages where each term occurs.



534 Index

Steiner consensus strings, 351, 353
Steiner tree, 470, 471, 477
string, 3, 4
string alignment, 216
string rotation in, 439
string similarity, 225
string-depth, 91
string-length, 282
strong good suffix rule, 16, 19, 31
structural inference, 341
structure deduction, 248
STS, 61, 131, 395
STS map, 61
STS order, 402
STS ordering, 405
STS-content mapping, 398, 405
stuttering subsequence problem, 249
suboptimal alignment, 321, 325
subsequence, 4, 227
subsequence versus substring, 4, 210, 227, 228
substring, 3, 4
tandem array, 13, 177, 140
tandem repeat, 177, 201, 247
text compression, 164, 445
tRNA folding problem, 250
threshold *P*-against-all problem, 308
translocations, 492
transpositions, 492
traveling salesman, 401
traveling salesman tour, 402
tree alignment problem, 354
tree compatibility, 464
tree-building algorithms, 448
triangle inequality, 349, 404
triple tree, 478
two-dimensional pattern matching, 84, 85

Ukkonen's algorithm, 94, 95, 107, 115, 116, 166
  Single extension algorithm, 100
  single phase algorithm: SPA, 106
ultrametric distance, 448, 449, 451
ultrametric matrix, 451, 475

Figure 9.1: The index of Gusfield's textbook, with the term "string" circled.

Index can help us speed up query speed and also give us a compact representation of the data.

Two big ideas behind indexing are ***grouping*** and ***ordering***. The first index that we will look at, *k*-mer index, uses grouping.

k-mer Index

Given a text T, we call all its substrings of length k, the ***k-mers*** of T. For example, consider the string `CGTGCGTGCTT`, it has the following 5-mers: `CGTGC`, `GCGTG`, `GTGCC`, `GTGCT`, `TGCCT`, and `TGCTT`. The substrings *can have overlaps*.

k-mer index groups the indices of T by the k-mer that starts at the index. So, for the string `CGTGCGTGCTT`, we have the k-mer index for $k = 5$:

k-mer	indices
`CGTGC`	0, 4
`GCGTG`	3
`GTGCC`	1
`GTGCT`	5
`TGCCT`	2
`TGCTT`	6

It is easy to construct a k-mer index. We can scan the string from left to right and record the position of each k-mer. It takes $|T| - k + 1$ time.

Querying k-mer Index

To query a k-mer index efficiently, we should first *sort the index lexicographically* by the k-mers. It takes $k(|T| - k)\log(|T| - k) \in O(|T|^2 \log |T|)$ steps. To compare two k-mers, we need k steps (unlike comparing two numbers or characters, which takes constant time), and sorting the list of $|T| - k$ k-mers takes $\Theta((|T| - k)\log(|T| - k))$. There is a bit of tradeoff here. If we choose a small k, we spend less time comparing two substrings during sorting, but we can possibly end with many k-mers in the list; in we choose a large k, it takes more time to compare two substrings, but we will have fewer k-mers to sort.

Once we sort the k-mer index, we can use *binary search* for patterns P such that $|P| \leq k$. It takes $O(|P| \log |T|)$ to query a sorted k-mer index. However, if $|P| > k$, k-mer index can be inefficient since we need to *manually extend the match* once the first k characters of P matches a substring in T.

10

Burrows-Wheeler Transform

11

FM Index

12

Wavelet Tree

Bibliography

[1] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi: 10.1017/CBO9780511574931.

[2] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015. doi: 10.1017/CBO9781139940023.

Index