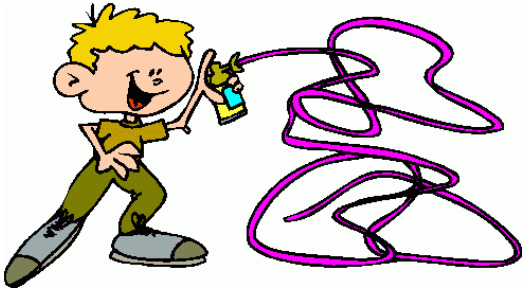
String Matching and Document Processing



Reading:

Kenneth A. Berman and Jerome Paul, *Algorithms: Special Topics*

- Section 2.1 The Naïve Algorithm
- Section 2.2 The Knuth-Morris-Pratt Algorithm
- Section 2.4 The Karp-Rabin Algorithm
- Section 2.5 Tries and Suffix Trees
- Section 2.6 Edit Distance

Definitions

- An **alphabet** is a set **symbols** $A = \{a_1, a_2, \ldots, a_k\}$.
- A **string** $S = S[0:n-1]$ **of length *n*** on A is a sequence of n characters (repetitions allowed) from A, *i.e.*, $S = s_0 s_1, \ldots, s_{n-1}$.
- $S[i]$ denotes the $(i+1)^{st}$ character s_i, $i = 0, \ldots, n-1$.
- $S[i:j]$ denotes the substring of symbols in consecutive positions i through j
- $|S|$ denotes the **length** of S.

String-Matching Problem

- Given a *pattern string* $P = P[0:m-1]$ of length m and *text string* $T = T[0:n-1]$ of length n, where $m \leq n$, the string-matching problem is to determine whether P occurs in T.
- In our string matching algorithms, we assume that we are looking for the first occurrence (if any) of the pattern string in the text string. The algorithms can be readily modified to return all occurrences of the pattern string.

Naïve String-Matching Algorithm

A naive algorithm for finding the first occurrence of P in T is to position P at the start of T and simply shift the pattern P along T, one position at a time, until either a match is found or the string T is exhausted, i.e., position $n - m + 1$ in T is reached without finding a match.

Pseudocode for Naïve Algorithm

```
function NaiveStringMatcher(P[0:m – 1],T[0:n – 1])
Input:      P[0:m – 1] (a pattern string of length m)
            T[0:n – 1] (a text string of length n)
Output:     returns the position in T of the first occurrence of P, or – 1
            if P does not occur in T
            for s ← 0 to n – m do
                        if T[s : s + m – 1]  = P then
                                    return(s)
                        endif
            endfor
            return(– 1)
end NaiveStringMatcher
```

The Knuth-Morris-Pratt Algorithm

- The Knuth-Morris-Pratt (KMP) is based on a strategy of utilizing information from partial matchings of P to not only skip over portions of the text that cannot contain a match but also to avoid checking characters in T that we already know match a prefix of P.

- The KMP algorithm achieves $O(n)$ worst-case complexity by preprocessing the string P to obtain information that can exploit partial matchings.

Strategy for improvement on naïve algorithm

- Pattern string *P* = "**00100201**“ Text string *T* = "**0010010020001002012200**".
- Placing *P* at the beginning of *T*, note that a mismatch occurs in (index) position 5:

  P = "**00100201**“ Text string *T* = "**0010010020001002012200**".
- Looking at *P*[0:4], the first position in *T* where a match could occur is at *i* = 3,

  “00100”
- Note that next position where a match can occur was obtained by subtracting the length of the largest prefix of *P*[0:3] that was also a suffix of *P*[1:4] from the position where the mismatch occurred.
- We don’t need to check the first 2 characters in *P* since they already match the first two characters of *T* at this new position for *P*.

- Given a text string $T[0:n-1]$ and pattern string $P[0:m-1]$, suppose we have detected a mismatch at position i in T, where $T[i] \neq P[j]$, but we know that the previous j characters of T match with $P[0: j-1]$.
- Suppose d_j is the length of the longest prefix of $P[0: j-2]$ that also occurs as a suffix of $P[1: j-1]$. Then the next position where a match can occur is at position $i - d_j$.
- To see whether we actually have a match starting at position $i - d_j$, we can avoid checking the characters that we already know agree with those in T, that is, the characters in the substring $T[i - d_j : i-1]$.
- Hence, we need only check the characters in the substring $T[i : i + m - d_j - 1]$ with those in the substring $P[d_j : m-1]$ to see if a match occurs.

Preprocessing the pattern string *P* by computing the array *Next*[0:*m* – 1]

Let *Next*[*j*] is length of longest prefix of *P*[0 : *j* – 2] that agrees with a suffix of *P*[1 : *j* – 1], *j* = 2, …, *m*.

Set *Next*[0] = *Next*[1] = 0.

PSN. Computer the array *Next*[0:5] for *P* = "987987"

Pseudocode for algorithm to compute array *Next*[0:*m* – 1]

```
function CreateKMPNext(P[0:m – 1], Next[0:m – 1])
Input:    P[0:m – 1] (a pattern string of length m)
Output:   Next[0:m – 1] (Next[i] is length of longest prefix of P [0:i – 2] that is
                                                  a suffix of P[1:i – 1], i = 0, … , m – 1)
   Next[0] ← Next[1] ← 0
   i ← 2
   j ← 0
  while i < m  do
       if P[j]  = P[i – 1]  then
            Next[i] ← j  + 1
            i ← i + 1
            j ← j + 1
         else
            if j > 0 then
                 j ← Next[j – 1]
            else
                Next[i] ← 0
                 i ← i + 1
            endif
       endif
  endwhile
end CreateKMPNext
```

Psuedocode for *KMPStringMatcher*

```
function KMPStringMatcher(P[0:m – 1],T[0:n – 1])
Input: P[0:m – 1] (a pattern string of length m)
                  T[0:n – 1] (a text string of length n)
Output:           returns the position in T of the first occurrence of P, or – 1
                  if P does not occur in T
   i ← 0   //i runs through text string T
   j ← 0   //j runs through pattern string P in manner dictated by key fact
   CreateNext(P[0:m – 1], Next[0:m – 1])
   while i < n  do
      if P[j]  = T[i]  then
         if j = m – 1 then  //match found at position i – m + 1
            return(i – m + 1)
         endif
         i ← i + 1   //continue scan of T
         j ← j + 1   //continue scan of P
      else     //P[j]  ≠ T[i]
         j ← Next[j]  //continue looking for a match of P which now
                                          could begin at position i – Next[j]  in T
         if j = 0 then
            if T[i] ≠ P[0] then   //no match at position i
               i ← i + 1
            endif
         endif
      endif
   endwhile
```

Karp-Rabin String Matcher

We assume without loss of generality that our strings are chosen from the k-ary alphabet $A = \{0,1,\ldots,k-1\}$.

Each character of A can be thought of as a digit in radix-k notation, and each string $S \in A^*$ can be identified with the base k representation of an integer.

For example, when $k = 10$, the string of numeric characters “98323777” can be identified with the integer 98323777.

Karp-Rabin cont'd

Given a pattern string $P[0:m-1]$, we can compute the corresponding integer $\bar{P}$ using *m* multiplications and *m* additions by employing Horner's Rule.

$$\bar{P} = P[m-1] + k(P[m-2] + k(P[m-3] + \cdots + kP[0])\ldots))$$

Karp-Rabin cont'd

- Given a text string $T[0: n - 1]$ and an integer s, we denote the substring $T[s, s + m - 1]$ by T_s.
- We denote the integer corresponding to T_s by $\overline{T_s}$.
- A string matching algorithm is obtained by using Horner's Rule to successively compute $T_0, T_1, T_2, \ldots$ where the computation continues until $\overline{P} = \overline{T_s}$ for some s (a match), or until we reach the end of the text T.
- Unfortunately, this is no better than the naïve string matching algorithm.
- However, the following key fact is the basis of a linear time algorithm.

Key Fact

Given the integers $\bar{T}_{s-1}$ and k^m, we can compute the integer $\bar{T}_s$ in constant time.

Formula yielding Key Fact

We achieve the result of key fact using:

$$\bar{T}_s = k\left(\bar{T}_{s-1} - k^{m-1}\overline{T}[s]\right) + \overline{T}[s+m],$$

$$s = 1, \ldots, n-m.$$

PSN. Illustrate this recursion for

$$k = 10, m = 7, T = \text{" } \ldots 49372458 \ldots \text{"}$$

$\bar{T}_{s-1} = 4937245$, and $\bar{T}_s = 9372458$

Analysis

Thus, each application of computing the next integer takes constant time. Hence, the $n - m + 2$ integers $\bar{T}_s, s = 1, \ldots, n - m$ can be computed in total time $O(n)$.

Drawback

The drawback with the preceding approach is that the integers $\bar{P}$ and $\bar{T}_s$ may be too large to work with efficiently and the assumption that they can be performed in constant time becomes unreasonable.

Getting around Drawback

- To get around this difficulty, we reduce these integers modulo q for some randomly chosen integer q.
- To avoid multiple-precision arithmetic, q is often chosen to be a random prime number such that kq fits within one computer word.

Getting around Drawback cont'd

$$\overline{P}^{(q)} = \overline{P} \bmod q,$$
$$\overline{T_s}^{(q)} = \overline{T_s} \bmod q.$$

- $\bar{P}^{(q)}$ and $\bar{T}_s^{(q)}$ can be computed in time $O(n)$ in the same way we computed $\bar{P}$ and $\bar{T}_s$, except that all arithmetic operations are performed modulo q.
- If $\bar{P}^{(q)} \neq \bar{T}_s^{(q)}$, then $\bar{P} \neq \bar{T}_s$ and there is no match. However, we need to check for **spurious** matches, i.e., $\bar{P}^{(q)} = \bar{T}_s^{(q)}$ but $\bar{P} \neq \bar{T}_s$.
- For sufficiently large q, the probability of a spurious match can be expected to be small.

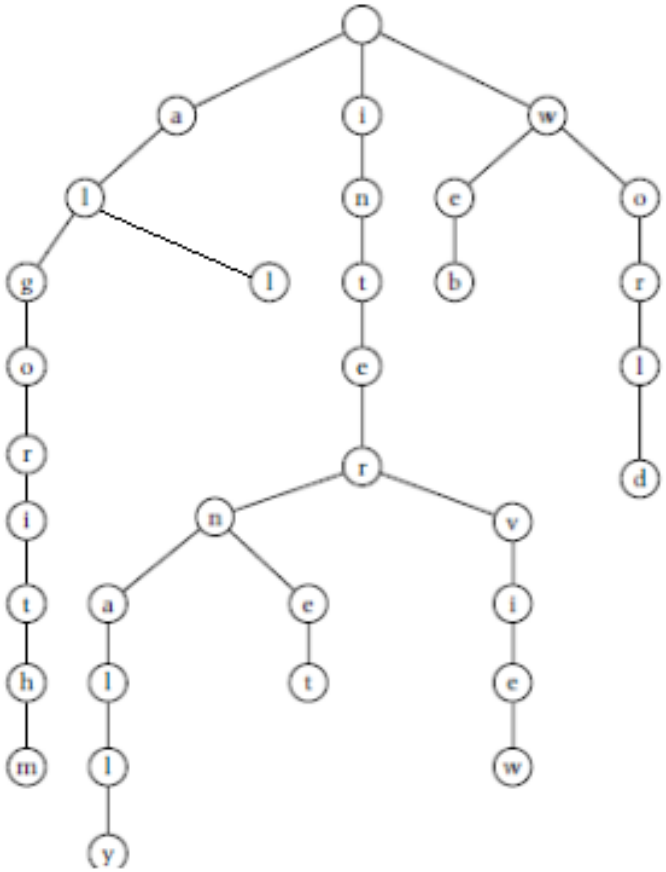
Psuedocode KarpRabinStringMatcher

function *KarpRabinStringMatcher*(*P*[0:*m* – 1],*T*[0:*n* – 1],*k*,*q*)
Input: *P*[0:*m* – 1] (a pattern string of length *m*)
T[0:*n* – 1] (a text string of length *n*)
k (*A* is the *k*-ary alphabet {0, 1, . . . , *k* – 1})
q (a random prime number *q* such that *kq* fits in one computer word)
Output: returns the position in *T* of the first occurrence of *P*, or – 1 if *P* does not occur in *T*

$c \leftarrow k^{m-1}$ **mod** q
$\overline{P}^{(q)} \leftarrow 0$
$\overline{T_s}^{(q)} \leftarrow 0$

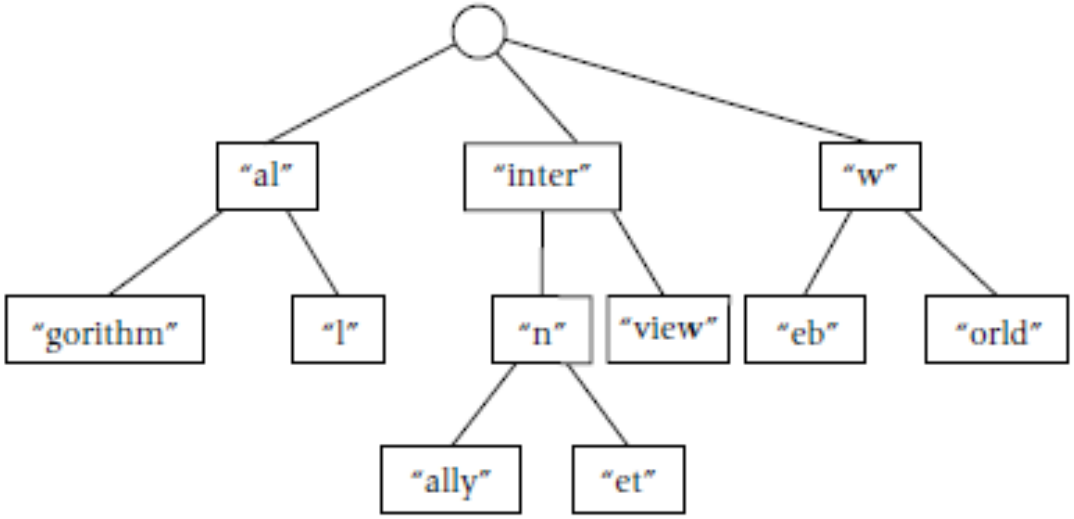
for $i \leftarrow 1$ **to** m **do** //apply Horner's Rule to compute $\overline{P}^{(q)}$ and $\overline{T_s}^{(q)}$
$\overline{P}^{(q)} \leftarrow (k\overline{P}^{(q)} + P[i])$ **mod** q
$\overline{T_0}^{(q)} \leftarrow (k\overline{T_0}^{(q)} + H[i])$ **mod** q
endfor
for $s \leftarrow 0$ **to** $n - m$ **do**
if $s > 0$ **then**
$\overline{T_s}^{(q)} \leftarrow (k(\overline{T_{s-1}}^{(q)} - T[s]c) + T[s + m])$ **mod** q
endif
if $\overline{T_s}^{(q)} = \overline{P}^{(q)}$ **then**
if $T_s = P$ **then** //match is not spurious
return $(s + 1)$
endif
endif
endfor
return(0)
end *KarpRabinStringMatcher*

Tries



Standard trie for the collection of strings *C* = {“internet”, “interview”, “internally”, “algorithm”, “all”, “web”, “world”}

Compressed Tries

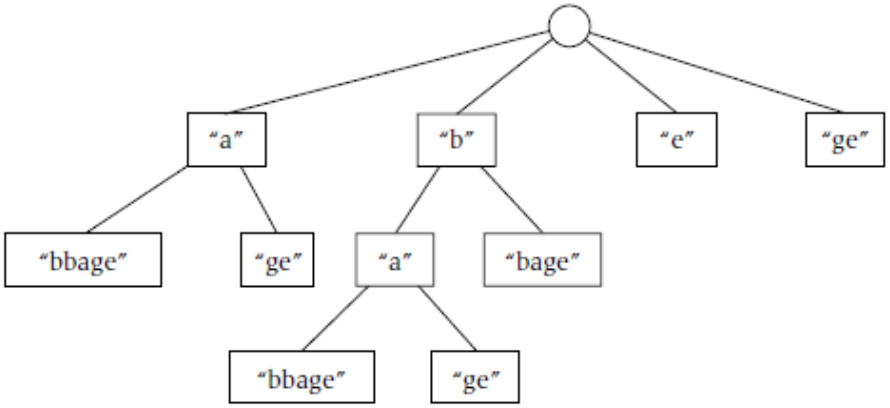


Compressed Trie for the same string set *C* = {“internet”, “interview”, “internally”, “algorithm”,“all”, “web”, “world”}

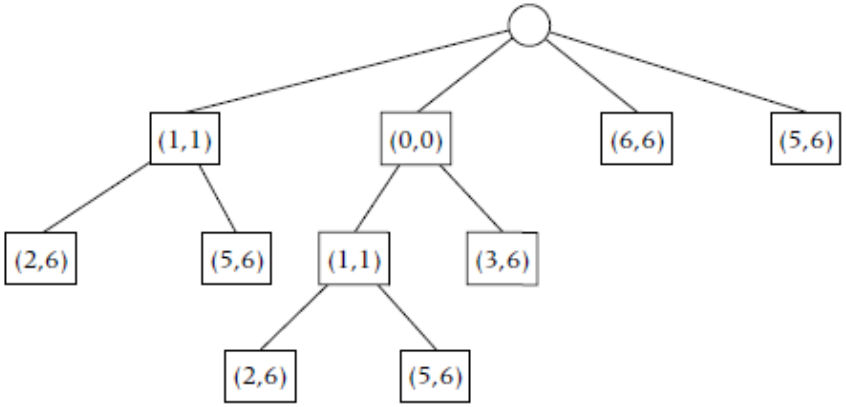
Suffix Tree

A **suffix tree** (also called a **suffix trie**) with respect to a given text string *T* is a compressed trie for the string collection *C* consisting of all suffixes of *T*.

Example Suffix Tree



Suffix tree for string *T* = "babbage"



More compact representation of the node labels

Edit Distance

- Edit distance measures how far two strings are apart.
- Can be used for a spell checker.
- Given a lexicon of words find the word in the lexicon whose edit distance with the word being spellchecked is the smallest.

Edit Operations and *k*-approximation

A pattern string $P = P_1 \ldots P_p$ is k-approximation of a text string $T_1 \ldots T_t$ if T can be converted to *P* using at most *k* operations involving one of

(i) changing a character of T (substitution),
(ii) adding a character to T (insertion),
(ii) removing a character of T (deletion).

For example, suppose P is the string “algorithm”

(i) elgorithm algorithm (substitution of e with a)
(ii) algorthm algorithm (insertion of letter i)
(iii) lagorithm algorithm (deletion of letter l)

Edit Distance

We define the edit distance, $D(P, T)$, between P and T to be the minimum number of operations of substitution, deletion and insertion needed to convert T to P.

For example, the string “algorithm” and “logarithm” have edit distance 3

logarithm → alogarithm → algarithm → algorithm

Computing the Edit Distance

- Let $D[i,j]$ denote the edit distance between the substring $P[0{:}i]$ consisting of the first $i + 1$ characters of the pattern string P and $T[0{:}j - 1]$ consisting of the first $j + 1$ characters of the text string T.
- If $p[i] = t[j]$, then, $D[i,j] = D[i - 1, j - 1]$.
- Otherwise, consider an optimal intermixed sequence involving the three operations substitution, insertion and deletion that converts $T[0{:}j]$ into $P[0{:}i]$. The number of such operations is the edit distance.
- Note that in transforming T to P inserting a character into T is equivalent to deleting a character from P. For convenience we will perform the equivalent operation of deleting characters from P rather than adding characters to T.
- We can assume without loss of generality that the sequence of operations involving the first i characters of P and the first j characters of T are operated on first.

Dynamic Programming Solution

To obtain a recurrence relation for $D[i, j]$, we examine the last operation.

- If the last operation is substitution of $t[j]$ with $p[i]$ in T, then $D[i, j] = D[i - 1, j - 1] + 1$.
- If the last operation is the deletion of $p[i]$ from P, then $D[i, j] = D[i - 1, j] + 1$.
- If the last operation is deletion of $t[j]$ from T, then $D[i, j] = D[i, j - 1] + 1$
- The edit distance is realized by computing the minimum of these three possibilities.

Recurrence Relation for Edit Distance

Observing that the edit distance between a string of size *i* and the null string is *i*, we obtain the following recurrence relation for the edit distance:

$$D[i,j] = \begin{cases} D[i-1,j-1], & p[i] = t[j], \\ \min\{D[i-1,j-1]+1, D[i-1,j]+1, D[i,j-1]+1\}, & otherwise. \end{cases}$$

init. cond. $D[0,i] = D[i,0] = i.$

An Integer walks up to a String and asks for its number.

String replies, "Sorry, you're not my type.“