

Some Basic Terms of “Stringology”

A **string** $S = S[0 : n] = \langle S[0], \dots, S[n - 1] \rangle$ is a sequence of characters from some **alphabet** Σ .

$\Sigma = [1..K]$ if not otherwise mentioned (integer alphabet).

$S[0 : i]$ is a **prefix** of length i

$S[n - i : n]$ is a **suffix** of length i

Consider a **text string** $T[0 : n]$ and a **pattern** $P[0 : m]$

An **occurrence** of P in T is a **substring** $T[j : j + m] = P$

Task: find all occurrences of P in T

Example: $P = \text{an}$ occurs in $T = \text{banana}$ at positions 1,3.

Brute Force Algorithm

for $j := 0$ **to** $n - m$ **do**

if $T[j : j + m] = P$ **then report occurrence at** j

Operator = (A, B)

// just pass pointers

 // **convention:** $A[|A|] = 0$ special character serves as sentinel

foreach $i \in [0 : m)$ **do**

if $A[i] \neq A[j]$ **then return** *false*

return *true*

Worst case: $\mathcal{O}(nm)$ time. Example: $T = a^n, P = a^m$

Best case: $\mathcal{O}(n)$ time. Example: $T = a^n, P = b^m$

Average case (random pattern): $\mathcal{O}(n)$

Overview

□ Plain text searching

– An $\mathcal{O}(nm)$ brute force algorithm

– Finite Automata: time $\mathcal{O}(n + m^2K)$

– Morris-Pratt: time $\mathcal{O}(n + m)$

□ String Sorting (\rightsquigarrow fast search for strings rather than substrings)

□ Fulltext indexing by sorting all suffixes \rightsquigarrow

$\mathcal{O}(m \log n) \cdots \mathcal{O}(m)$ search time

Pattern Matching Using Finite Automata A

General idea: preprocess pattern to speed up matching.

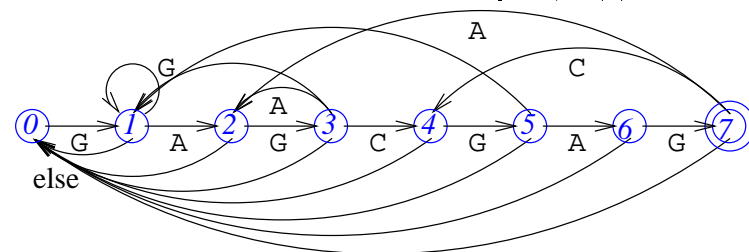
States: $[0 : m]$ Automaton in state $i \Leftrightarrow P[0 : i]$ has been matched.

Accepting States: $\{m\}$. If A is in the accepting state before character $i + m$ is input then i is an occurrence of the pattern.

Transition Function:

$$\delta(i, c) = \max \{j \in [0 : m] : P[0 : j] = P[i - j : i] \circ \langle c \rangle\}$$

(match the longest **prefix** of P that is a **suffix** of $P[0 : i] \circ \langle c \rangle$)



Analysis

Space: $\Theta(Km)$ for a big lookup table (the main problem)

Preprocessing time: (naive) $\mathcal{O}(Km^3)$ (the m -dependence can be improved)

Running the automaton: $\mathcal{O}(n)$

Perhaps good for small K (cache!) and VERY long texts.

Can be generalized to regular expression patterns
(exponential space in m , linear time in n)

More Space Efficient Precomputation

Idea: emulate the automaton using more space efficient tables.

match the longest prefix of P that is a suffix of $P[0 : i] \circ \langle c \rangle$

\leadsto

match the longest prefix of P that is a suffix of $P[0 : i]$

this prefix $P[0 : F[i]]$ is independent of T

Disadvantage: we may have to compare c several times

The Morris-Pratt Algorithm

$\forall i < m : F[i] = \max \{j < i : P[0 : j] = P[i - j : i]\}$

$i := 0$ // next character in P

$j := 0$ // next character in T

invariant $P[0 : i] = T[j - i : j]$

invariant occurrences starting before $T[j - i]$ are already reported

while $j + (m - i) \leq n$ **do** // hope for another occurrence

while $i < m \wedge P[i] = T[j]$ **do** $i++ ; j++$ // scan matching chars

if $i = m$ **then** report occurrence $P = T[j - m : j]$

if $i = 0$ **then** $j++$ // shift by one

else $i := F[i]$ // failure transition

Example: $P = \text{ABCABCACAB}$, $T = \text{BABCABCABCAABCABCACABC}$

Correctness 1

$\forall i < m : F[i] = \max \{j < i : P[0 : j] = P[i - j : i]\}$

$i := 0$

$j := 0$

invariant A: $P[0 : i] = T[j - i : j]$

invariant B: occurrences starting before $T[j - i]$ are already reported

while $j + (m - i) \leq n$ **do**

while $i < m \wedge P[i] = T[j]$ **do** $i++ ; j++$

 // A is maintained.

 // $j - i$ does not change. Hence, B is not affected.

if $i = m$ **then** report occurrence $P = T[j - m : j]$

if $i = 0$ **then** $j++$

else $i := F[i]$

Computing F

Idea: bootstrap. \approx run MP on $T = P[1 : m]$

or, implicitly, run MP on $T = P[1 : k]$ for each $k \leq m$

$i := 0$ // next character in $P[0 : m]$ (prefix)

$j := 1$ // next character in $P[1 : m]$ (suffix)

invariant A: $P[0 : i] = P[j - i : j]$

// Observation: $\forall j$ the longest match will be found first

$F[j] := i$

invariant B: $\forall i' < j : F[i'] = \max \{j' < i' : P[0 : j'] = P[i' - j' : i']\}$

invariant C: $i < j$

while $j < m$ **do**

while $j < m \wedge P[i] = T[j]$ **do** $i++ ; j++ ; F[j] := i$

if $i = 0$ **then** $j++ ; F[j] := i$

else $i := F[i]$ // legal due to Invariant C

More Exact String Matching

Knuth-Morris-Pratt: stronger failure function

$$F'[i] = \max \{-1 \leq j < i : P[0 : j] = P[i - j : i] \wedge P[j] \neq P[i]\}$$

Boyer-Moore: Scan pattern **right to left**.

Sublinear best case, e.g., $T = a^n, P = b^m$.

Perhaps best algorithm in practice.

(Worst case efficient variant difficult to analyze.)

Fingerprinting: Consider a hash function $h : \Sigma^m \rightarrow \mathbb{N}$ such that

$h(T[i - m : i])$ can be computed in time $\mathcal{O}(1)$ from

$h(T[i - m : i])$ and $T[i]$, e.g.

$$h(S) = \sum_{0 \leq j < m} a_j S[j] \bmod p \text{ for some prime } p \gg m\Sigma .$$

Possible occurrence when $h(T[i - m : i]) = h(P)$.

Simple, generalizable to 2D, . . .