CISC 320 Introduction to Algorithms
Fall 2003

Lecture 13
String matching

Naïve string-matching algorithm

Naïve-string-matcher(T, P)
1. n = length(T)
2. m = length(P)
3. for i from 0 to n - m
4. for j from 1 to m
5. if P[j] = T[i+j]
6. then break
7. if j = m then
8. print “Match at” i

Running time = O(nm)

Terminologies
- Σ: the alphabet
- Σ*: the set of all finite-length strings formed using characters from Σ.
- xy: concatenation of two strings x and y.
- Prefix: a string w is a prefix of a string x if x=wy for some string y ∈ Σ*.
- Suffix: a string w is a suffix of a string x if x=wy for some string y ∈ Σ*.

Finnite automata
A finite automaton M is a 5-tuple (Q, q₀, A, δ, f), where
- Q is a finite set of states
- q₀ ∈ Q is the start state
- A ⊆ Q is a distinguished set of accepting states
- Σ is a finite input alphabet
- δ is a function from Q × Σ into Q, called transition function of M.

Pattern P = “nano”
Test T = “bananano

Running time is O(nm), excluding the time required to build the transition function δ.
The Knuth-Morris-Pratt algorithm

1. Skip outer iteration \( i = 3 \)
2. Skip first inner iteration testing "n" so "N" at outer iteration \( i = 4 \)

In general, if there is a partial match of \( j \) chars starting at \( l \), then we know what is in position \( T[l]...T[l+j-1] \). So we can save by

- Skip outer iterations (for which no match possible)
- Skip inner iterations (when no need to test know matches).

When a mismatch occurs, we want to slide \( P \) forward, but maintain the longest overlap of a prefix of \( P \) with a suffix of the part of the text that has matched the pattern so far.

KMP algorithm achieves linear time performance by capitalizing on the observation above, via building a simplified finite automaton: each node has only two links, success and fail.

Failure links: \( fail[k] \) is defined as the largest \( r < k \) such that \( x = P[1]...P[r-1] \) matches \( y = P[k-r+1]...P[k-1] \).

- \( kmSetup(P, m, fail) \)
  1. \( fail[1] = 1 \)
  2. for \( k = 2 \) to \( m \)
  3. \( s = fail[k-1] \)
  4. while \( (s >= 1) \)
  5. \( \) if \( P[s] = P[k-1] \)
    6. \( s = s + 1 \)
    7. break
  8. \( s = fail[s] \)
  9. \( fail[k] = s \)
  10. \( \) Running time is \( O(m) \).

KMP Scan \( (P, T, m, fail) \)
\( match = -1; j = 1; k = 1; \)

1. \( \) while \( \) endText \( \) ( \( T, j) \) = false \( ) \)
  2. if \( k = m \)
    3. \( match = j - m; \) //Match found
    4. break
  5. if \( (x = 0) \)
    6. \( j++; \) //Start pattern over
  7. else if \( (T[j] = P[k]) \)
    8. \( j++; \)
    9. \( k++; \)
  10. else \( k = fail[k]; \)
  11. return \( match \)
  12. \( \) Running time is \( O(n) \).

The Knuth-Morris-Pratt algorithm

Boyer-Moore algorithm

The numbers of positions we can "jump" forward when there is a mismatch depends on the last character being read, say \( T[l] \), more precisely, depends on \( T[l] \)'s occurrence in pattern \( P \).
computeJump(p, charJump)

m ← length(p)

1. while i < m:
2. charJump[i+1] = m
3. if p[i+1 ... m] is not present in P
   then we can jump forward m places.
4. charJump[i+1] = charJump[i+1] + m

Running time is O(m^2 + n)

Suppose we have a pattern P and text T, and we want to find all occurrences of P in T. The algorithm to do this is as follows:

1. Compute the jump array for the pattern P
2. Use the jump array to perform a linear search for P in T
3. If P is found, then the starting position is the value of j + 1 in the jump array
4. Repeat steps 2 and 3 until all occurrences of P are found

Running time is linear in worst case, and sub-linear on average.

Summary
- Naive (or brute-force) algorithm: $O(nm)$
- Finite-automaton algorithm: $O(nm)$
- KMP algorithm: $O(nm)$
  - Relatively easier to implement
  - Do not require random access to the text
- BM algorithm: $O(n+m)$ worst, sublinear average
  - Fewer character comparisons
  - The algorithm of choice in practice for string matching.