Cấu trúc dữ liệu và giải thuật

ĐỐI SÁNH CHUỖI

Giảng viên:
Văn Chí Nam – Nguyễn Thị Hồng Nhung – Đặng Nguyễn Đức Tiến

Nội dung trình bày

- Giới thiệu
- Thuật toán Brute-Force
- Thuật toán Morris-Pratt
- Cải tiến với Knuth-Morris-Pratt
- Thuật toán Rabin-Karp

Giới thiệu

- Đối sánh chuỗi
  - Từ khóa: String matching, String searching, Pattern searching, Text Searching
  - Một trong những thuật toán quan trọng và có ứng dụng rộng rãi.

Giới thiệu

- Ứng dụng của đối sánh chuỗi:
  - Máy tìm kiếm
  - Trình soạn thảo văn bản
  - Trình duyệt web
  - Sinh học phân tử (Tìm mẫu trong dãy DNA).
  - ..

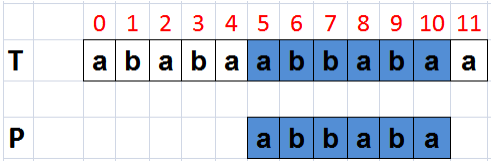
Giới thiệu

- Mục tiêu:
  - Kiểm tra sự tồn tại của một chuỗi ký tự (mẫu, pattern) trong một chuỗi ký tự có kích thước lớn hơn nhiều (văn bản, text).
  - Nếu tồn tại, trả về một (hoặc nhiều) vị trí xuất hiện.
- Quy ước:
  - Mẫu cần tìm: P (chiều dài *m*).
  - Văn bản: T (chiều dài *n*).
  - P và T có cùng tập hữu hạn ký tự ∑. (∑ = {0, 1}; ∑={A,..,Z},…)
  - m ≤ n

Giới thiệu

- Đối sánh chuỗi:
  - Bằng cách lần lượt dịch chuyển (cửa sổ) P trên T.
  - P tồn tại trên T tại vị trí bắt đầu là i ($0 \le i \le n - m$) nếu
    - T[i + j] = P[j] với mọi $0 \le j \le m$ - 1.
- Ví dụ:
  - P = **abbaba**
  - T = **ababaabbabaa**

  => i = 5



Giới thiệu

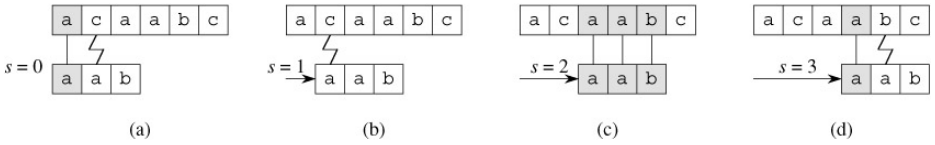
- Các thuật toán tiêu biểu:
  - Brute Force
  - Morris-Pratt
  - Knuth-Morris-Pratt
  - Rabin-Karp
  - Boyer-Moore
  - …

Thuật toán Brute-Force

Ý tưởng

- Lần lượt kiểm tra điều kiện P[0…*m*-1] = T[*i*…*i*+*m*-1] tại mọi vị trí có thể của *i*.

- Ví dụ
  - Tìm kiếm P = **aab** trong T = **acaabc**



Cài đặt

```
bruteForceMatcher(T, P)
    n ← length[T]
    m ← length[P]
    for i ← 0 to n - m
        if P[0..m-1] = T[i…i+m-1]
            return i
```

Đánh giá

- Trường hợp tốt nhất – không tìm thấy: O(n).
- Trường hợp xấu nhất – không tìm thấy: O(n*m).
- Trường hợp trung bình: O(n+m).

Đặc điểm chính

- Không cần thao tác tiền xử lý trên P.
- Luôn luôn dịch chuyển mẫu (cửa sổ) sang phải một vị trí.
- Thao tác so sánh có thể thực hiện theo bất kỳ chiều nào.
- Trường hợp xấu nhất: O((n-m+1)*m).

Thuật toán Morris-Pratt

Đặt vấn đề

- Điểm hạn chế của thuật toán Brute-Force:
  - Không ghi nhớ được thông tin đã trùng khớp (trước) khi xảy ra tình trạng không so khớp.
  - Phải so sánh lại từ đầu (trên P) trong tất cả trường hợp

Đặt vấn đề

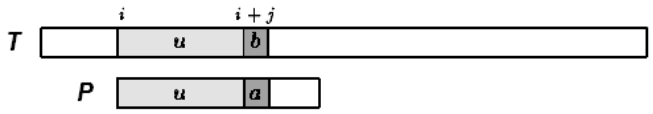
- Ví dụ:
  - T: **10101**011101001;
  - P: **10100**
  - Brute Force: i = 0, j = 4, T[i+j] != P[j] => i = 1, j = 0
    - T : 1**0**101011101001
    - P: **1**0100
  - Cách khác? i = 0, j = 4, T[i+j] != P[j] => i = 2, j = 2
    - T : 10**101**011101001
    - P: **101**00

Đề xuất của thuật toán

- Ghi nhận lại những phần của T đã trùng với P trước đó.
- Cố gắng tăng số bước dịch chuyển P trên T (**thay vì 01** đơn vị).
- Cố gắng **bỏ qua một số bước so sánh** giữa P và T tại vị trí mới (thay vì j=0, gán *j* bằng một số thích hợp).

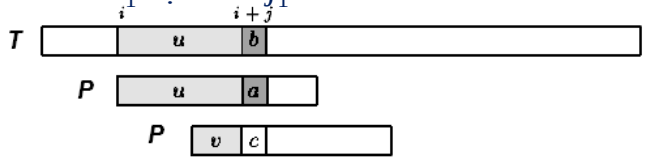
Đề xuất của thuật toán

- Giả sử:
  - i là vị trí bắt đầu sự đối sánh (trên T).
  - j là vị trí đang so sánh (trên P). (Ký tự tương ứng trên T tại vị trí $i+j$).
  - T[i+j] != P[j] => không so khớp



Đề xuất của thuật toán

- Tìm:
  - Vị trí mới i_1 (trên T) và j_1 (trên P) sao cho
    - i+j = i_1+j_1 (ngay tại vị trí đang xem xét)
    - v =T[i_1 … i_1+j_1–1] là đoạn so khớp mới giữa P và T.
- Khi đó:
  - Đoạn dịch chuyển cửa sổ: $j - j_1$.(do $j_1 < j$)
  - Có thể tìm i_1 dựa trên j_1.



Đề xuất của thuật toán

- Vấn đề:
  - Tìm giá trị j_1 dựa trên j.
- Cách thức:
  - Tính sẵn các giá trị của j_1 ứng với mỗi vị trí j (trên P).
- Câu hỏi:
  - Có thể làm được không? Tại sao?

Xây dựng bảng NEXT

- Bảng NEXT:
  - Bảng chứa các giá trị j_1 ứng với các giá trị j.
- Ví dụ:

j	0	1	2	3	4	5	6
j_1	-1	0	1	1	0	3	2

Xây dựng bảng NEXT

- Hoàn toàn dựa trên P.
- Cách thức:
  - NEXT[0] = -1
  - Với mỗi vị trí j > 0, giá trị của NEXT[j] (j_1) là số k **lớn nhất** ($k < j$) sao cho:
    - k ký tự đầu tiên khớp với k ký tự cuối cùng của chuỗi trước vị trí j.
    - Nghĩa là P[0..k-1] = P[j-k ..j-1]

Xây dựng bảng NEXT

- Ví dụ:
  - P = AAATA
  - Bảng NEXT:
    - NEXT[0] = -1

Xây dựng bảng NEXT

- P = AAATA
- j = 1
- NEXT[1] = 0

Xây dựng bảng NEXT

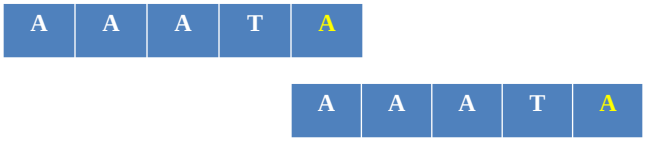
- P = AAATA
- j = 2
- NEXT[2] = 1

Xây dựng bảng NEXT

- P = AAATA
- j = 3
- NEXT[3] = 2

Xây dựng bảng NEXT

- P = AAATA
- j = 4
- NEXT[4] = 0



Xây dựng bảng NEXT

- P = AAATA
- Bảng NEXT
  - NEXT[0] = -1
  - NEXT[1] = 0
  - NEXT[2] = 1
  - NEXT[3] = 2
  - NEXT[4] = 0

	0	1	2	3	4
NEXT	-1	0	1	2	0

Xây dựng bảng NEXT

- Xây dựng bảng NEXT cho P = 10100
- Xây dựng bảng NEXT cho P = ABACAB
- Xây dựng bảng NEXT cho P = GCAGAGAG
- Xây dựng bảng NEXT cho P = AABAABA

Xây dựng bảng NEXT

- P = 10100

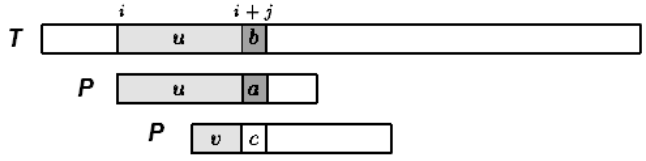
	0	1	2	3	4
NEXT	-1	0	0	1	2

- P = ABACAB

	0	1	2	3	4	5
NEXT	-1	0	0	1	0	1

Sử dụng NEXT trong thuật toán

- Mục tiêu :
  - Xác định vị trí mới i_1 (trên T) và j_1 (trên P) sao cho
    - i+j = i_1+j_1 (vị trí đang xem xét)
    - v =T[i_1 … i_1+j_1–1] là đoạn so khớp mới giữa P và T.
- Đã có j_1 = NEXT[j]
- Vậy, $i_1 = i + j$ – NEXT[j]



Sử dụng NEXT trong thuật toán

- Ví dụ:
  - T = AATAAAATA
  - P = AAATA

	0	1	2	3	4
NEXT	-1	0	1	2	0

  - *i* = 0 AATAAAATA
  - *j* = 0 AAATA

  - *i* = 0 AATAAAATA
  - *j* = 1 AAATA

Sử dụng NEXT trong thuật toán

- Ví dụ:
  - T = AATAAAATA
  - P = AAATA

	0	1	2	3	4
NEXT	-1	0	1	2	0

  - *i* = 0 AATAAAATA
  - *j* = 2 AAAATA

  - *i* = 1 AATAAAATA (i = 0 + 2 – 1)
  - *j* = 1 AAATA (j = 1)

Sử dụng NEXT trong thuật toán

- Ví dụ:
  - T = AATAAAATA
  - P = AAATA

	0	1	2	3	4
NEXT	**-1**	**0**	**1**	**2**	**0**

  - *i* = 2 AATAAAATA (i = 1 + 1 – 0)
  - *j* = 0 AAATA (j = 0)

  - *i* = 3 AATAAAATA (i = 2 + 0 – (-1))
  - *j* = 0 AAATA (j = 0)

Sử dụng NEXT trong thuật toán

- Ví dụ:
  - T = AATAAAATA
  - P = AAATA

	0	1	2	3	4
NEXT	**-1**	**0**	**1**	**2**	**0**

  - *i* = 3 AATAAAATA
  - *j* = 3 AAATA

  - *i* = 4 AATAAAATA (i = 3 + 3 – 2)
  - *j* = 2 AAATA (j = 2)

Sử dụng NEXT trong thuật toán

- Ví dụ:
  - T = AATAAAATA
  - P = AAATA

	0	1	2	3	4
NEXT	-1	0	1	2	0

  - *i* = 4 AATA**AAATA**
  - *j* = 4 **AAATA**

(Hoàn toàn so khớp, vị trí xuất hiện của P trong T tại *i*=4)

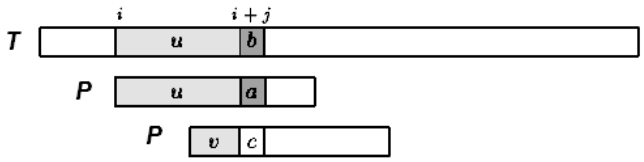
Độ phức tạp

- Tính NEXT: O(m)
- Tìm kiếm: O(n)
- Tổng: O(n+m)

Thuật toán Knuth-Morris-Pratt

Ý tưởng

- Thuật toán Knuth-Morris-Pratt cải tiến Morris-Pratt bằng cách
  - bổ sung thêm điều kiện a ≠ c (*vì nếu a và c như nhau thì sẽ không khớp ngay sau khi dịch chuyển*).



Ý tưởng

- Thay đổi cách tính bảng NEXT:
  - Nếu `p[i]` ≠ `p[j]` thì **`NEXT[i] = j`**
  - Ngược lại **`NEXT[i] = NEXT[j]`**
- Thao tác tìm kiếm vẫn không thay đổi

Xây dựng bảng NEXT

- P = 10100

	0	1	2	3	4
MP	-1	0	0	1	2
KMP	-1	0	-1	0	2

- P = ABACAB

	0	1	2	3	4	5
MP	-1	0	0	1	0	1
KMP	-1	0	-1	1	-1	0

Thuật toán Rabin-Karp

Giới thiệu

- Là thuật toán tìm kiếm chuỗi được đề xuất bởi Michael O. Rabin và Richard M. Karp vào 1987.
- Sử dụng phép băm(hashing).


Rabin


Karp

Ý tưởng thuật toán

- T = "AADCABADCA"
- P = "BACD"

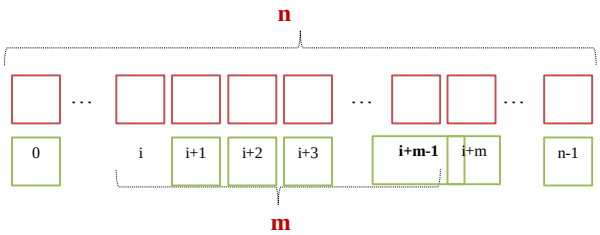
Ý tưởng thuật toán

❖ Giải pháp: Rolling Hash

❖ Tận dụng được mã hash của lần tính trước.
❖ Lần tính kế tiếp không phụ thuộc vào độ dài chuỗi con.

Ý tưởng thuật toán

- Hàm băm của Rabin–Karp
  - h(x) = x mod q (q là số nguyên tố lớn)
  - $x_i = a[i]d^{m-1} + a[i+1]d^{m-2} + ... + a[i+m-1]$
    - d là cơ số



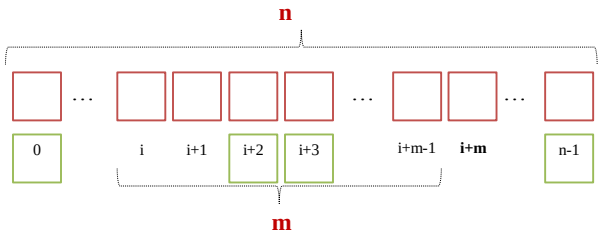
Ý tưởng thuật toán

- Ví dụ: T = "abracadabra" P = "arb" d = 10
  m = length(P) = 3
  $x\_P = [a].10^2 + [r].10^1 + [b].10^0$
  $x_0 = [a].10^2 + [b].10^1 + [r].10^0$
  $x_1 = [b].10^2 + [r].10^1 + [a].10^0$
  …
  $x_8 = [b].10^2 + [r].10^1 + [a].10^0$

- Dùng cơ số nhằm giảm khả năng đụng độ.
- Hạn chế:
  - Giá trị hàm hash tăng rất nhanh.
  - Chi phí tính toán lớn.

Ý tưởng thuật toán

- Rolling Hash
  - $x_i = a[i]d^{m-1} + a[i+1]d^{m-2} + ... + a[i+m-1]$



  - $x_{i+1} = (x_i - a[i]d^{m-1})d + a[i+m]$

Cài đặt

```
function RabinKarp(string s[1..n],
  string sub[1..m])
    hsub := hash(sub[1..m]);
     hs := hash(s[1..m])
    for i from 1 to n-m+1
        if hs = hsub
            if s[i..i+m-1] = sub
                return i
        hs := hash(s[i+1..i+m])
    return not found
```

Cài đặt

```
RABIN-KARP-MATCHER( T, P, d, q )
   n ← length[ T ]
   m ← length[ P ]
   h ← d^(m-1) mod q
   p ← 0
   t0 ← 0
   for i ← 1 to m                          ► Preprocessing
       do  p ← ( d*p + P[ i ] ) mod q
           t0 ← ( d*t0 + T[ i ] ) mod q
   for s ← 0 to n – m                      ► Matching
       do if p = ts
           then if P[ 1..m ] = T[ s+1 .. s+m ]
                then print "Pattern occurs with shift" s
       if s < n – m then
     ts+1 ← ( d * ( ts – T[ s + 1 ] * h )
                + T[ s + m + 1 ] ) mod q
```


Nhận xét về thuật toán

- Gần như tuyến tính
- Độ phức tạp:
  - Tốt nhất: O(m + n)
  - Xấu nhất: O(m.n)
  - Trung bình: O(m + n)
- Được sử dụng trong tìm kiếm đa mẫu (multiple pattern search)

Hỏi và Đáp