Data Structures & Algorithms

Các thuật toán tìm kiếm chuỗi
(String Searching Algorithms)

Nguyễn Tri Tuấn
Khoa CNTT – ĐH.KHTN.Tp.HCM
Email: nttuan@fit.hcmus.edu.vn

String Searching

- Giới thiệu
- Các bước xử lý
- Các cách tiếp cận
- Brute-Force
- Knuth-Morris-Pratt

Brute-Force

- Ý tưởng:

  Đối với vị trí thứ i của văn bản T (i=0…n-m), ta so sánh các ký tự của P tương ứng từ trái sang phải:

  P[0] với T[i], P[1] với T[i+1],…, P[m-1] với T[i+m-1]

  P[j] với T[i+j] (j = 0..m-1)

- Ví dụ:
  - T = “TWO RED ROADS CROSSING”
  - n = length(T) = 22
  - P = “ROADS”
  - m = length(P) = 5

Brute-Force (tt)

- Ví dụ:
  - Bước 1: TWO RED ROADS CROSSING
    ROADS
  - Bước 2: TWO RED ROADS CROSSING
    ROADS
    …
  - Bước 5: TWO RED ROADS CROSSING
    ROADS
    …
  - Bước 9: TWO RED ROADS CROSSING
    ROADS

Brute-Force (tt)

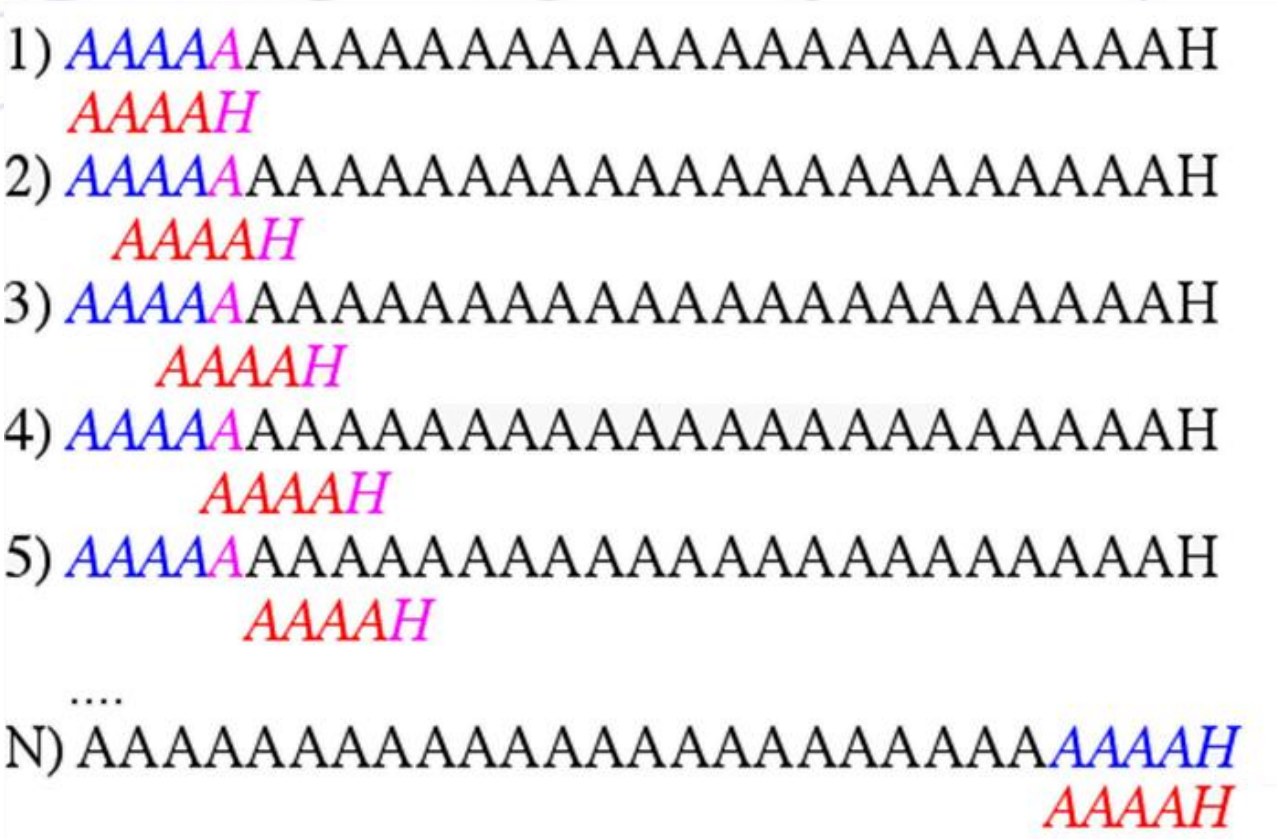
- Cài đặt bằng C:
  - `int stringSearchBF (char *P, char *T);`
  - Kết quả:
    - -1 : nếu P không nằm trong T, hoặc
    - >=0: vị trí của P trong T

Brute-Force (tt)

```
int stringSearchBF(char *P, char *T)
{
  for (int i=0; i<=strlen(T)-strlen(P); i++)
  {
      int j = 0;
      while (j < strlen(P))
            if (T[i+j]==P[j])  j++;
            else  break;
      if(j==strlen(P)) return i;  // tìm thấy
  }
  return -1 ; // không tìm thấy
}
```

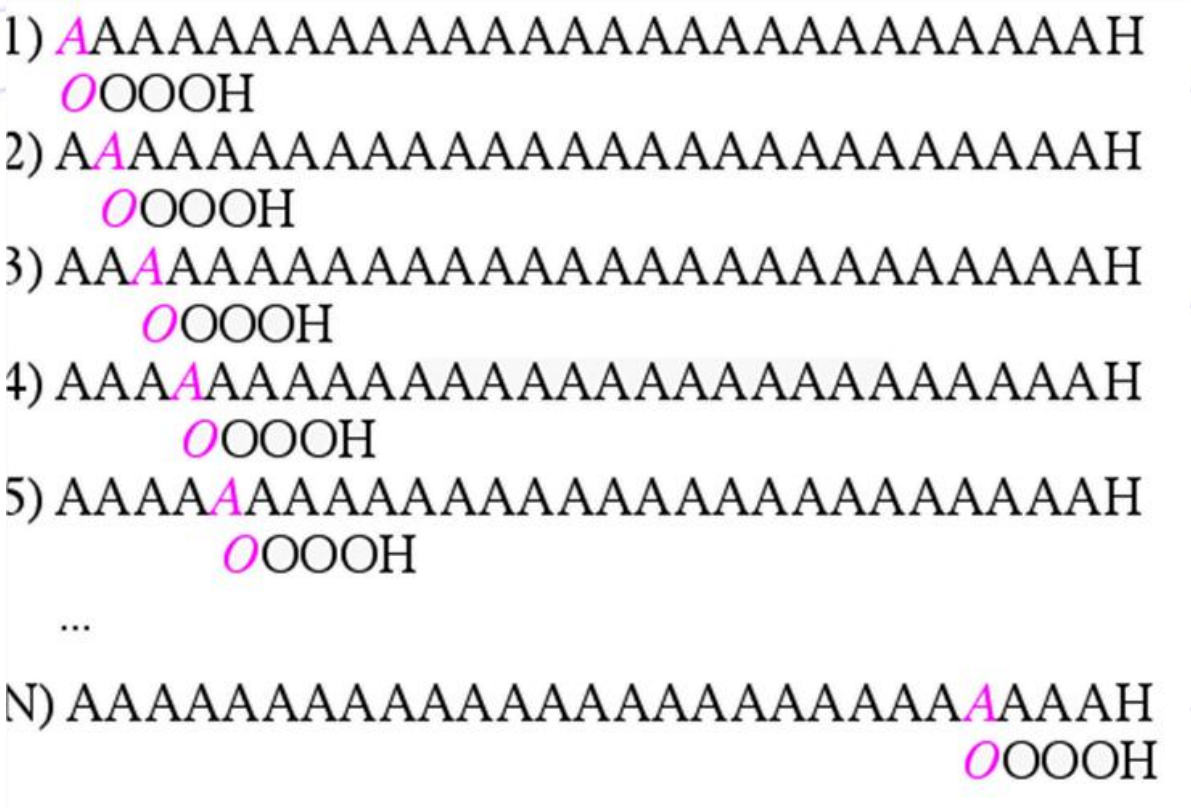
Brute-Force (tt)

- Đánh giá:
  - Trường hợp xấu nhất O(m*n) – *tự chứng minh*



Brute-Force (tt)

- Đánh giá:
  - Trường hợp tốt nhất O(n) – *tự chứng minh*



  - Trường hợp trung bình O(n+m) – *tự chứng minh*

String Searching

- Giới thiệu
- Các bước xử lý
- Các cách tiếp cận
- Brute-Force
- Knuth-Morris-Pratt

Đặt vấn đề

- Trong thuật toán Brute-Force: khi xảy ra không so khớp tại một ký tự, ta đã xóa bỏ tất cả thông tin có được bởi các phép so sánh trước đó và bắt đầu lại việc so sánh từ ký tự đầu tiên của mẫu P ***[i=i+1; j=0]***
- Ví dụ:
  - T = 101010100111; P = 10100
  - 1010**1**0100111

    10100 (không so khớp tại ***i=0, j=4***)
  - 1**0**1010100111

    10100 (Brute-Force: bắt đầu lại từ ***i=1; j=0***)

Dịch chuyển *i* và *j* ra sao cho có lợi ?

1010**1**0100111

10**1**00 (bắt đầu lại từ ***i=2; j=2***)

Morris-Pratt

- Hướng giải quyết của Morris-Pratt:
  - Lợi dụng thông tin đã biết về các ký tự đã so sánh
  - Biến j thể hiện số ký tự đã được so khớp giữa mẫu (P) và văn bản (T). Khi gặp vị trí không so khớp, thay vì gán $j = 0$ để quay lại từ đầu chuỗi P, ta sẽ gán cho j một giá trị thích hợp

Morris-Pratt (tt)

```
// Tư tưởng chính của thuật toán Morris-Pratt
// giai đoạn tìm kiếm P trong T

i = j = 0;
while (i+j < n) {
   if (p[j]==t[i+j]) {
        j++;
        // khi đã đi hết độ dài của chuỗi P
        // → đã tìm được P → trả về vị trí tìm được
        if (j==m) return i;
   }
   else {
        // khi không so khớp, dịch chuyển về vị
        // trí NEXT[j]
        i = i + j - NEXT[j];
        if (j > 0) j = NEXT[j];
}
return -1; // không tìm thấy
```

Morris-Pratt (tt)

- Giai đoạn tiền xử lý – tính giá trị bảng NEXT
  - Xây dựng mảng NEXT[0..m-1] (có m phần tử)
  - NEXT[j] chứa giá trị dùng để dịch chuyển con trỏ j khi xảy ra sự không khớp tại vị trí j

Morris-Pratt (tt)

- 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

i=0 AATAAAATA
j=2 AAATA (không so khớp → i=i+j–NEXT[2]=1

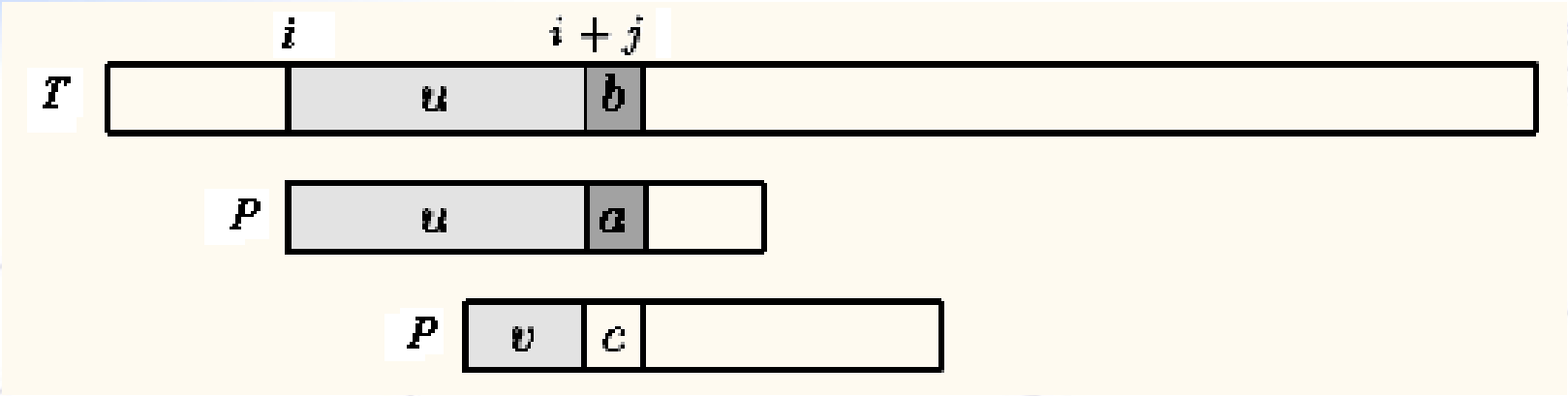
j=NEXT[2]=1)

Morris-Pratt (tt)

- Ví dụ:

```
i=1          AATAAAATA
j=1            AAATA   (không so khớp → i=i+j–NEXT[1]=2
                                          j=NEXT[1]=0)

i=2          AATAAAATA
j=0            AAATA   (không so khớp → i=i+j–NEXT[0]=3
                                          j=0)

i=3          AATAAAATA
j=0              AAATA                 (tiếp tục…)

i=3          AATAAAATA
j=3              AAATA
                          (không so khớp → i=i+j–NEXT[3]=4
                                          j=NEXT[3]=2)

i=4          AATAAAATA
j=2                AAATA               (tiếp tục…)
i=4          AATAAAATA
j=4                AAATA
→ so khớp !
```

Morris-Pratt (tt)



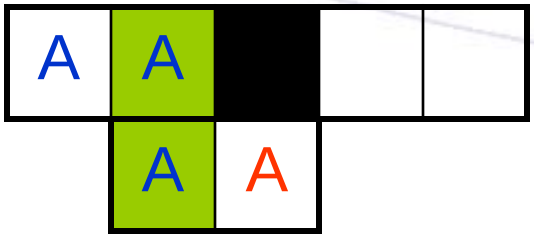
- Xét trạng thái không so khớp tại vị trí thứ *j* trên mẫu P
  - Phần đã so khớp (u), từ P[0] đến P[0..*j-1*]
  - Nếu tồn tại *v* = đoạn so khớp giữa phần đầu P với phần cuối của P[0..*j-1*]
    → *v* là đoạn dịch chuyển của *j*

Morris-Pratt (tt)

- Cách xây dựng bảng NEXT:
  - NEXT[0] = -1
  - Với j>0, giá trị của NEXT[j] là số k lớn nhất nhỏ hơn j sao cho k ký tự đầu tiên của mẫu P khớp với k ký tự cuối của P[0..j-1]
  - Ví dụ: P = AAATA

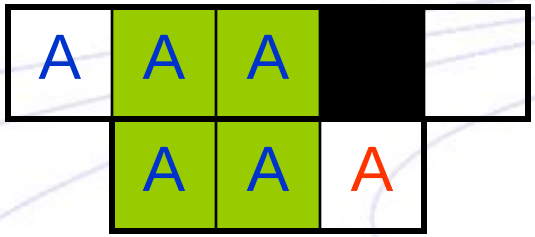
NEXT[1] = 0
(j=1)

NEXT[2] = 1
(j=2)

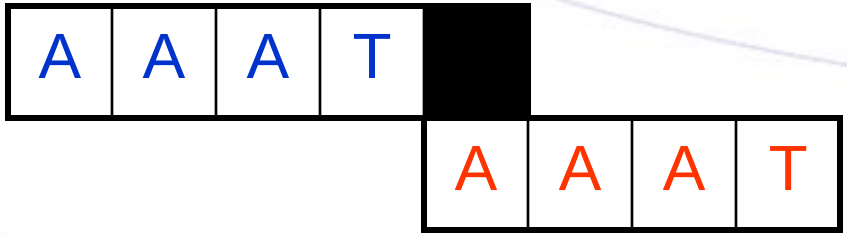


Morris-Pratt (tt)

NEXT[3] = 2



NEXT[4] = 0



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

Morris-Pratt (tt)

```
// Hàm tính giá trị bảng NEXT (Morris-Pratt)
void initNEXT_MP(char *p, int NEXT[]) {
        int i, j;
        int m = strlen(p);
        i = 0;
        j = NEXT[0] = -1;
        while (i < m-1) {
                if (j == -1 || p[i] == p[j]) {
                        i++; j++;
                        NEXT[i] = j;
                }
                else j = NEXT[j];
        }
}
```

Morris-Pratt (tt)

- Cài đặt bằng C:

```
// Thuật toán đối sánh Morris-Pratt
// Kết quả:
// -1    : nếu P không nằm trong T, hoặc
// >=0   : vị trí của P trong T
int stringSearchMP (char *P,char *T) {
  int n = strlen(T);
  int m = strlen(P);
  int *NEXT = new int[m];

  // Tiền xử lý – Tính giá trị bảng NEXT
  initNEXT_MP(p, NEXT);

  // Tìm P trong T
  … (xem slide #23)
}
```

Morris-Pratt (tt)

- Ví dụ:
  - 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

Morris-Pratt (tt)

P = 10100	0	1	2	3	4
NEXT	-1	0	0	1	2

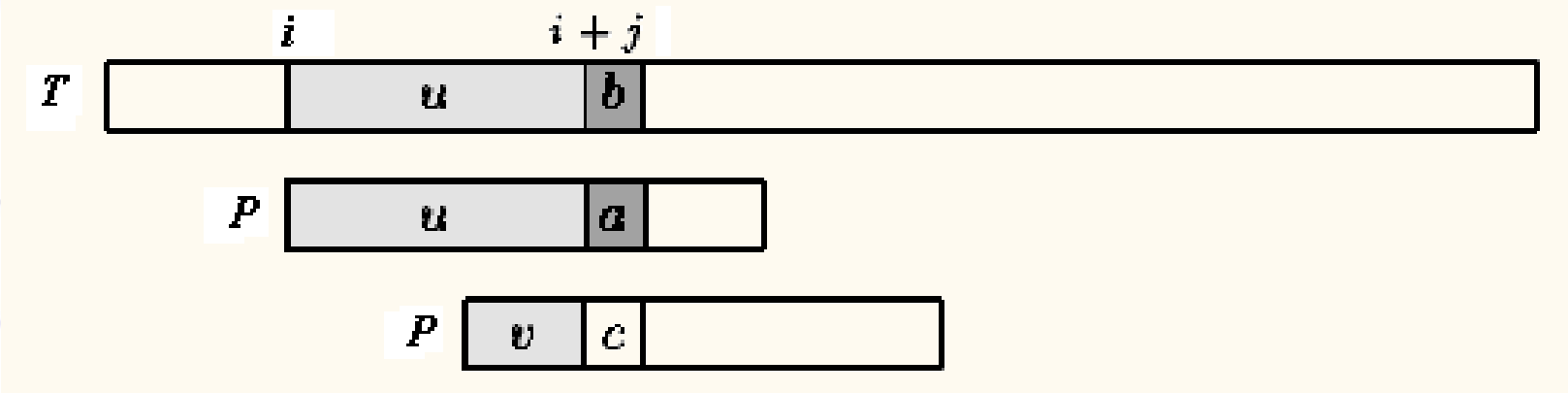
P = ABACAB	0	1	2	3	4	5
NEXT	-1	0	0	1	0	1

P = GCAGAGAG	0	1	2	3	4	5	6	7
NEXT	-1	0	0	0	1	0	1	0

Morris-Pratt (tt)

- Độ phức tạp:
  - Giai đoạn tiền xử lý: O(m) (tính NEXT)
  - Giai đoạn tìm kiếm: O(n)
  - Tổng: O(n+m)
  - Số phép so sánh lớn nhất của một ký tự <= m

Knuth-Morris-Pratt



- Cải tiến của KMP so với Morris-Pratt
  - Khi xây dựng bảng NEXT, Knuth bổ sung kiểm tra điều kiện $c \neq a$ để tránh trường hợp "mis-match" ngay vị trí đầu tiên sau khi dịch chuyển j
  - *Giải thích: nếu a == c, tức là c ≠ b (vì a ≠ b) → sẽ "mis-match" ngay lần so sánh đầu tiên sau khi dịch chuyển j*

Knuth-Morris-Pratt (tt)

- Tóm lại: thuật toán KMP
  - Giai đoạn tiền xử lý có một cải tiến nhỏ:
    - Tính độ dịch chuyển tốt hơn → tránh so sánh cùng một ký tự trong T hai lần
  - Giai đoạn tìm kiếm: hoàn toàn giống thuật toán Morris-Pratt

Knuth-Morris-Pratt (tt)

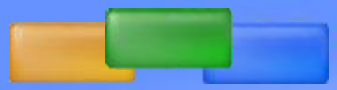
```
// Hàm tạo lập bảng NEXT (KMP)
void initNEXT_KMP(char *p, int NEXT[]) {
        int i, j;
        int m = strlen(p);
        i = 0;
        j = NEXT[0] = -1;
        while (i < m-1) {
             if (j == -1 || p[i] == p[j]) {
                  i++; j++;
                  if (p[i] != p[j]) NEXT[i] = j;
                  else NEXT[i] = NEXT[j];
             }
             else j = NEXT[j];
        }
}
```

Knuth-Morris-Pratt (tt)

- Độ phức tạp:
  - Giai đoạn tiền xử lý: O(m) (tính NEXT)
  - Giai đoạn tìm kiếm: O(n)
  - Tổng: O(n+m)
  - Số phép so sánh lớn nhất của một ký tự <= $\log_a m$

với $a = \frac{1+\sqrt{5}}{2}$

Thank you
for your attention



Q & A