STRING MATCHING

Nguyễn Thùy Dương, Đặng Thùy Dung, Lê Thị Hạnh

Ha Noi University of Science

Ngày 10 tháng 5 năm 2025

Đặt vấn đề

- Bài toán khớp xâu (string matching) là một trong những bài toán cổ điển trong khoa học máy tính.
- Ứng dụng trong: phân tích DNA, xử lý ngôn ngữ tự nhiên, nén dữ liệu, truy xuất thông tin, biên dịch,...
- Mục tiêu: tìm kiếm mẫu (pattern) trong chuỗi văn bản lớn (text).
- Nhiều thuật toán đã được phát triển để tối ưu tốc độ và tài nguyên tính toán.
- Khoảng cách giữa lý thuyết và thực tiễn: cần thuật toán vừa hiệu quả vừa dễ triển khai.
- Hai nhóm chính:
  - Khớp xâu chính xác (Exact Matching)
  - Khớp xâu xấp xỉ (Fuzzy Matching) – không xét trong báo cáo này

Mục tiêu và Công nghệ sử dụng

Mục tiêu:

- Trình bày và phân tích các thuật toán kinh điển trong khớp xâu:
  - Brute Force
  - Rabin-Karp
  - Boyer-Moore
- So sánh ưu/nhược điểm từng thuật toán, đưa ra cái nhìn tổng quan và định hướng áp dụng thực tiễn.

Môi trường phát triển:

- **Ngôn ngữ:** Python 3.11 – cú pháp đơn giản, thư viện hỗ trợ phong phú.
- **Thư viện chính:**
  - `re` – hỗ trợ biểu thức chính quy để kiểm thử mẫu.
  - `time` – đo thời gian thực thi, đánh giá hiệu năng.
  - `matplotlib.pyplot` – trực quan hóa, vẽ biểu đồ so sánh thuật toán.

Thuật toán Brute Force

Brute Force (Naive String Matching) là phương pháp đơn giản nhất để tìm chuỗi mẫu trong văn bản. Không dùng kiến thức bổ sung, chỉ so sánh lần lượt từng vị trí trong văn bản.

Nguyên lý hoạt động:

- Với văn bản `text` độ dài n, mẫu `pattern` độ dài m.
- Duyệt từng vị trí i từ 0 đến $n - m$:
  - So sánh từng ký tự: `text[i + j] == pattern[j]` với $0 \leq j < m$
  - Nếu tất cả ký tự khớp, trả về vị trí i là nơi chuỗi mẫu xuất hiện.
- Lặp cho đến hết văn bản.

Đặc điểm: Dễ cài đặt, nhưng hiệu suất kém trong trường hợp xấu ($O(nm)$).

Brute Force: Mã giả và Ví dụ minh họa

Mã giả:

- `n` $\leftarrow$ độ dài `text`, `m` $\leftarrow$ độ dài `pattern`
- **for** i từ 0 đến $n - m$:
  - $j \leftarrow 0$
  - **while** $j < m$ **and** `text[i + j] == pattern[j]`:
    - $j \leftarrow j + 1$
  - **if** $j == m$: báo vị trí khớp i

Ví dụ: `text = "abracadabra"`, `pattern = "bra"`
Các vị trí khớp: **1**, **8**

- Vị trí 1: `"bra"` $=$ `"bra"`
- Vị trí 8: `"bra"` $=$ `"bra"`

Brute Force: Độ phức tạp

Thời gian thực hiện:

- Tổng số lần thử: $n - m + 1$
- Mỗi lần so sánh tối đa m ký tự Tối đa $m(n - m + 1) = O(nm)$ phép so sánh

Độ phức tạp thời gian:

- Trường hợp tốt nhất: $O(n)$ (phát hiện sai khác ngay ký tự đầu)
- Trường hợp trung bình xấu nhất: $O(nm)$

Độ phức tạp không gian:

- Sử dụng biến đếm đơn giản $O(1)$

Kết luận:

- **Thời gian:** tốt $O(n)$, xấu $O(nm)$
- **Không gian:** $O(1)$

Brute Force: Ưu/Nhược điểm và Kết luận

Ưu điểm:

- Đơn giản, dễ hiểu và dễ cài đặt.
- Không cần tiền xử lý hay cấu trúc dữ liệu đặc biệt.
- Phù hợp cho văn bản nhỏ hoặc số lần tìm kiếm ít.

Nhược điểm:

- Hiệu suất thấp với văn bản dài hoặc tìm nhiều mẫu.
- Không tận dụng được đặc trưng mẫu như các thuật toán nâng cao.

Kết luận:

- Thích hợp cho bài toán đơn giản, yêu cầu tối thiểu.
- Không phù hợp với quy mô lớn – cần xem xét các thuật toán như Rabin-Karp, KMP, Boyer-Moore.

Thuật toán Karp-Rabin

- **Giới thiệu**: Được đề xuất bởi Richard M. Karp và Michael O. Rabin (1987). Sử dụng hàm băm để tìm chuỗi con hiệu quả.
- **Nguyên lý hoạt động**:
  - Tính giá trị băm cho *chuỗi mẫu* và *chuỗi con* có cùng độ dài trong văn bản.
  - So sánh giá trị băm: Nếu trùng, kiểm tra trực tiếp từng ký tự để xác nhận khớp hoàn toàn.
  - Ý tưởng cốt lõi: Ánh xạ chuỗi sang giá trị số qua hàm băm, giảm số phép so sánh.
- **Ưu điểm**: Hiệu quả, nhanh chóng nhờ so sánh giá trị băm thay vì ký tự.

Karp-Rabin: Hàm băm

- **Hàm băm**: Hàm tuyến tính $h(k) = k \mod q$, trong đó:
  - k: Số nguyên đại diện chuỗi (hệ cơ số d, thường $d = 256$ cho ASCII).
  - q: Số nguyên tố lớn để giảm xác suất va chạm.
- **Rolling hash**: Cập nhật giá trị băm:

$$h_i = \left(d \cdot \left(h_{i-1} - \text{text}[i-1] \cdot d^{m-1}\right) + \text{text}[i+m-1]\right) \mod q$$

- **Tham số**:
  - h_i: Giá trị băm tại vị trí i.
  - $\text{text}[i-1]$: Ký tự đầu chuỗi con trước.
  - $\text{text}[i+m-1]$: Ký tự mới thêm vào.
- **Lợi ích**: Giảm độ phức tạp từ $O(m)$ xuống $O(1)$ cho mỗi chuỗi con.

Ví dụ Hàm băm Karp-Rabin

- **Ví dụ**: Văn bản "abracadabra", chuỗi mẫu độ dài 3, $d = 256$, $q = 101$.
- **Tính băm cho "abr"** (ASCII: a=97, b=98, r=114):

$$h(\text{"}abr\text{"}) = [(((97 \cdot 256) \% 101 + 98) \% 101) \cdot 256 + 114] \% 101 = 4$$

- **Tính băm cho "bra" từ băm trước**:

$$h(\text{"}bra\text{"}) = \left(\left(4 + 101 - 97 \cdot (256 \% 101)^2\right) \cdot 256 + 97\right) \% 101 = 30$$

- **Kiểm tra trực tiếp cho "bra"**:

$$h'(\text{"}bra\text{"}) = [(((98 \cdot 256) \% 101 + 114) \% 101) \cdot 256 + 97] \% 101 = 30$$

Karp-Rabin: Độ phức tạp và Xác suất va chạm

- **Độ phức tạp**:
  - Trung bình: $O(n+m)$ (n: độ dài văn bản, m: độ dài mẫu).
  - Tệ nhất: $O(mn)$ nếu nhiều so khớp hoặc va chạm hàm băm.
  - Bộ nhớ: Chỉ cần vài biến cho giá trị băm và chỉ số, không dùng cấu trúc dữ liệu phức tạp.
- **Xác suất va chạm**:
  - Xác suất: $\mathbb{P}\{\text{collision}\} = \mathcal{O}\left(\frac{1}{q}\right)$, q: số nguyên tố lớn.
  - Thực nghiệm: Chỉ 3 va chạm trong 10^7 lần băm với bảng chữ cái lớn.
  - Theo Karp-Rabin:
    $$\text{Prob(collision)} \leq \frac{\pi(m(n-m+1))}{\pi(M)},$$
    với $\pi(z)$: số nguyên tố $\leq z$, $M = O(mn^2)$, $m(n-m+1) \geq 29$.
  - Thực tế:
    $$\text{Prob(collision)} = \frac{1}{q} - O\left(\frac{1}{c^m}\right) < \frac{1}{q},$$
    cho hàm băm đồng đều, $c = 2$.
- **Ưu điểm**: Hiệu quả thời gian, tiết kiệm bộ nhớ, đáng tin cậy cho văn bản lớn khi chọn q lớn.

Karp-Rabin: So sánh kỳ vọng và Kết quả thực nghiệm

- **Số lần so sánh kỳ vọng**:

$$\frac{C_n}{n} = \mathcal{H} + \frac{m}{c^m}\left(1 - \frac{1}{q}\right)\frac{1}{q} + O\left(\frac{1}{c^m}\right)$$

($\mathcal{H}$: chi phí tính toán hàm băm).

- **Bảng kết quả ($\mathcal{H} = 1$):**

m	c				
	2	4	10	30	90
2	1.5 1.4996	1.125 1.12454	1.02 1.0199	1.00222 1.002179	1.00025 1.000244
4	1.25 1.2500	1.01563 1.01563	1.0040 1.00361	1.00000 1.000040	1.00000 1.0000
7	1.05469 1.05422	1.00043 1.000404	1.0000 1.000017	1.0000 1.0000	1.0000 1.0000
10	1.00977 1.00980	1.0001 1.000050	1.0000 1.0000	1.0000 1.0000	1.0000 1.0000
15	1.00046 1.000454	1.0000 1.000	1.0000 1.0000	1.0000 1.0000	1.0000 1.0000

- **Nhận xét**: Kết quả lý thuyết và thực nghiệm tương đồng. $\mathcal{H}$ có thể lớn hơn với chuỗi dài, nhưng tối ưu nhờ phần cứng hỗ trợ chia dư.

Karp-Rabin: Tối ưu hóa và Kết luận

- **Tối ưu hóa:** Thuật toán Karp-Rabin có thể tối ưu hóa bằng cách tận dụng đặc tính tràn số nguyên để thay thế phép chia dư modulo, giúp giảm thời gian tính toán. Cụ thể, thay vì tính modulo, ta sử dụng công thức:

$$h_i = h_{i-1} \cdot d - \text{text}[j-m] \cdot d^m + \text{text}[i+m],$$

  và việc tràn số sẽ tự động giới hạn giá trị giống như modulo.
- **Kết luận:**
  - Thuật toán hiệu quả với văn bản lớn và nhiều mẫu.
  - Hàm băm tuyến tính giúp tính toán nhanh.
  - Va chạm có thể giảm thiểu với tham số thích hợp.
  - Tối ưu hóa phần cứng nhờ tràn số.

Thuật toán Boyer-Moore

Giới thiệu: Thuật toán Boyer-Moore (BM), công bố năm 1977, là một trong những thuật toán tìm kiếm chuỗi hiệu quả nhất. Các biến thể: Simplified Boyer-Moore (SBM) và Boyer-Moore-Horspool (BMH).

Cách hoạt động:

- Dịch chuyển mẫu từ phải sang trái trong văn bản.
- Sử dụng hai chiến lược dịch chuyển (heuristics):
  - **Match heuristic:** Dịch chuyển dựa trên ký tự đã khớp.
  - **Occurrence heuristic:** Dịch chuyển dựa trên ký tự không khớp.
- Các bước:
  1. Căn chỉnh mẫu với ký tự đầu tiên của văn bản và so sánh từ cuối mẫu.
  2. Nếu tất cả khớp, báo cáo vị trí. Nếu không, tính khoảng cách dịch chuyển bằng hai heuristics và chọn giá trị lớn nhất.
  3. Dịch chuyển mẫu và lặp lại.

Bảng Dịch Chuyển trong Thuật toán Boyer-Moore

Thuật toán sử dụng hai bảng dịch chuyển:

Bảng dd (match heuristic):

$$dd[j] = \min\{s + m - j \mid s \geq 1 \text{ và } ((s \geq i \text{ hoặc } pattern[i - s] = pattern[i]) \text{ cho } j < i \leq m)\}$$

Bảng d (occurrence heuristic):

$$d[x] = \min\{s \mid s = m \text{ hoặc } (0 \leq s < m \text{ và } pattern[m - s] = x)\}$$

Phiên bản cải tiến $\widehat{dd}$:

bổ sung điều kiện $pattern[j - s] \neq pattern[j]$.

Ví dụ (mẫu "abracadabra"):

- $\widehat{dd}[j] = [17, 16, 15, 14, 13, 12, 11, 13, 12, 4, 1]$
- $d['a'] = 0$, $d['b'] = 2$, $d['c'] = 6$, $d['d'] = 4$, $d['r'] = 1$, các ký tự khác: 11.

Boyer-Moore: Mã giả

```
PROCEDURE bmsearch(text, n, pattern, m)
    initd(pattern, m, d)  Tạo bảng d
    initdd(pattern, m, dd)  Tạo bảng dd
    k ← m
    skip ← dd[1] + 1
    WHILE k   n DO
        j ← m
        WHILE j > 0 AND text[k] = pattern[j] DO
            j ← j - 1
            k ← k - 1
        ENDWHILE
        IF j = 0 THEN
            Report match at position k + 1
            k ← k + skip
        ELSE
            k ← k + MAX(d[text[k]], dd[j])
        ENDIF
    ENDWHILE
ENDPROCEDURE
```

Ghi chú: Khi không khớp, nó dùng bảng *d* (khoảng cách ký tự) và *dd* (khớp một phần) để nhảy qua nhiều ký tự, giúp tìm nhanh hơn.

Boyer-Moore: Mã giả

```
PROCEDURE initd(pattern, m, d)
    FOR each char in alphabet DO
        d[char] ← m
    ENDFOR
    FOR k ← 1 TO m DO
        d[pattern[k]] ← m - k
    ENDFOR
ENDPROCEDURE

PROCEDURE initdd(pattern, m, dd)
    FOR k ← 1 TO m DO
        dd[k] ← 2 * m - k
    ENDFOR
    FOR j ← m DOWNTO 1 DO
        Compute f[j] and update dd[j]  Xử lý khớp một phần
    ENDFOR
     Cập nhật dd cho các đoạn khớp, chi tiết lược bớt
ENDPROCEDURE
```

Ghi chú: Hàm *initd* tạo bảng *d* để biết cách nhảy khi ký tự không khớp. Hàm *initdd* tạo bảng *dd* để tối ưu khi mẫu khớp một phần, giúp nhảy xa hơn.

Boyer-Moore: Độ phức tạp và cải tiến

Phân tích lý thuyết:

- **Trường hợp xấu nhất**: $O(n + r \cdot m)$, có thể đạt $O(mn)$ nếu r lớn (số lần khớp).
- **Trường hợp trung bình**: Với c (bảng chữ cái lớn), $m < n$:

$$\frac{n}{m} + \frac{m(m+1)}{2m^2c + O(c^{-2})}$$

- **Không gian**: $O(m + c)$.
- **Thời gian tiền xử lý**: $O(m)$.

Cải tiến:

- Galil (1979): Ghi nhớ chồng lấn, giảm độ phức tạp xấu nhất xuống $O(n + m)$.
- Apostolico-Giancarlo (1986): Giảm số lần so sánh xuống $2n - m + 1$.

Ghi chú:

Thuật toán hiệu quả với bảng chữ cái lớn, cải tiến giúp giảm so sánh và tối ưu trường hợp xấu.

Thuật toán Boyer-Moore đơn giản hóa (SBM)

Giới thiệu:

- Thuật toán Boyer-Moore đơn giản hóa (SBM) chỉ sử dụng **occurrence heuristic** .
- Lý do chính: mẫu tìm kiếm trong thực tế thường không có tính chu kỳ.
- Không gian phụ giảm từ $O(m+c)$ xuống $O(c)$ (với c là kích thước bảng chữ cái, thường cố định).
- Không sử dụng cải tiến của Galil vì cần $O(m)$ bộ nhớ để tính độ dài các ký tự chồng lấp.

Độ phức tạp:

- Trường hợp xấu nhất: $O(mn)$, nhưng trung bình nhanh hơn đáng kể.
- Với $m \ll n$: $\frac{C_n}{n} \geq \frac{1}{c-1} + \frac{1}{c^4} + O(c^{-6})$
- Với $c \ll n$: $\frac{C_n}{n} \geq \frac{1}{m} + \frac{m^2+1}{2cm^2} + O(c^{-2})$

Thuật toán Boyer-Moore-Horspool

Giới thiệu

- Horspool đề xuất năm 1980, cải tiến SBM bằng cách sử dụng ký tự văn bản tương ứng với ký tự cuối của mẫu để tra bảng d.
- BM gốc dùng hai bảng (d và dd) để nhảy qua ký tự khi so sánh mẫu với văn bản. BMH đơn giản hơn: chỉ dùng một bảng (gọi là bảng xuất hiện). Bảng này dựa trên ký tự cuối cùng của mẫu.

Bảng d (Heuristic xuất hiện):

Định nghĩa: $d[x] = \min\{s \mid s = m \text{ hoặc } (1 \leq s < m \text{ và pattern}[m - s] = x)\}$

Ví dụ: Với mẫu `abracadabra`:

$d[\text{'a'}] = 3$, $d[\text{'b'}] = 2$, $d[\text{'c'}] = 6$, $d[\text{'d'}] = 4$, $d[\text{'r'}] = 1$

Thuật toán Boyer-Moore-Horspool

```
PROCEDURE bmhsearch(text, n, pattern, m)
    FOR each char in alphabet DO
        d[char] ← m
    ENDFOR
    FOR j ← 1 TO m-1 DO
        d[pattern[j]] ← m - j
    ENDFOR
    pattern[0] ← NOT_IN_TEXT
    text[0] ← NOT_IN_PATTERN
    i ← m
    WHILE i   n DO
        k ← i
        j ← m
        WHILE j > 0 AND text[k] = pattern[j] DO
            j ← j - 1
            k ← k - 1
        ENDWHILE
        IF j = 0 THEN
            Report match at k + 1
        ENDIF
        i ← i + d[text[i]]
    ENDWHILE
ENDPROCEDURE
```