ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA TOÁN - CƠ - TIN HỌC

Báo cáo cuối kì
Dự đoán bệnh tim sử dụng mô hình học máy

Thành viên nhóm

Nguyễn Thùy Dương - 22000077

Lê Thị Hạnh - 22000090

Đặng Thùy Dung - 22000074

Giảng viên

TS. Nguyen Hong Minh

Hà Nội, Tháng 5 năm 2024

Mục lục

Chương 1

Giới thiệu

1.1 Đặt vấn đề

Bài toán khớp xâu (string matching) là một trong những bài toán cổ điển và được nghiên cứu rộng rãi nhất trong lĩnh vực khoa học máy tính. Mục tiêu của bài toán là tìm kiếm một hoặc nhiều mẫu (pattern) trong một chuỗi văn bản lớn (text). Bài toán này có ứng dụng thực tiễn trong nhiều lĩnh vực như: phân tích trình tự DNA, xử lý ngôn ngữ tự nhiên, nén dữ liệu, truy xuất thông tin, và xây dựng trình biên dịch.

Trải qua nhiều thập kỷ phát triển, nhiều thuật toán khớp xâu đã được đề xuất nhằm tối ưu thời gian xử lý và giảm thiểu tài nguyên tính toán. Tuy nhiên, vẫn tồn tại một khoảng cách giữa lý thuyết và thực tiễn: trong khi các nhà nghiên cứu chú trọng vào độ phức tạp lý thuyết, thì người phát triển lại quan tâm đến hiệu năng thực tế và khả năng triển khai dễ dàng. Do đó, một thuật toán khớp xâu được xem là thực tiễn khi nó vừa đảm bảo hiệu suất tốt trên dữ liệu thực, vừa dễ hiện thực và bảo trì.

Các phương pháp khớp xâu thường được chia thành hai nhóm: khớp xâu chính xác (exact matching) và khớp xâu xấp xỉ (fuzzy matching). Khớp xâu xấp xỉ cho phép một số sai lệch ký tự giữa mẫu và văn bản, thường được đo bằng khoảng cách Levenshtein. Tuy nhiên, trong khuôn khổ báo cáo này, chúng tôi chỉ tập trung vào các thuật toán khớp xâu chính xác.

1.2 Mục tiêu

Trình bày và phân tích các thuật toán kinh điển:

- Brute Force
- Rabin Karp
- Boyer Moore

Thông qua việc trình bày đặc điểm và so sánh ưu/nhược điểm của từng thuật toán, báo cáo hướng tới việc cung cấp cái nhìn tổng quan về bài toán, đồng thời đưa ra định hướng áp dụng phù hợp trong thực tiễn.

1.3 Môi trường phát triển và công nghệ sử dụng

Trong quá trình triển khai và kiểm thử các thuật toán khớp xâu, chúng tôi sử dụng ngôn ngữ lập trình **Python 3.11** do đặc điểm cú pháp đơn giản, dễ đọc và thư viện phong phú hỗ trợ thao tác chuỗi, xử lý dữ liệu và trực quan hóa. Các thư viện chính được sử dụng bao gồm:

- `re`: Thư viện chuẩn hỗ trợ biểu thức chính quy, giúp kiểm thử và đối chiếu mẫu.

- `time`: Dùng để đo thời gian thực thi của các thuật toán, phục vụ đánh giá hiệu năng.
- `matplotlib.pyplot`: Hỗ trợ trực quan hóa dữ liệu, vẽ biểu đồ so sánh hiệu năng giữa các thuật toán.

Chương 2

Tổng quan về bài toán khớp xâu

2.1 Tổng quan về bài toán khớp xâu

2.1.1 Định nghĩa bài toán khớp xâu (String Matching)

Bài toán khớp xâu (string matching) là một trong những bài toán cơ bản và quan trọng trong khoa học máy tính, nhằm tìm kiếm tất cả các vị trí xuất hiện của một chuỗi con (gọi là mẫu hay *pattern*, ký hiệu là P) trong một chuỗi lớn hơn (gọi là văn bản hay *text*, ký hiệu là T).

Cụ thể, cho văn bản T có độ dài n và mẫu P có độ dài m (thường $m \leq n$), mục tiêu là xác định tất cả các chỉ số i trong T sao cho $T[i..i+m-1] = P$, tức là mẫu P xuất hiện tại vị trí bắt đầu từ i trong văn bản T.

Bài toán có thể được mở rộng với các biến thể:

- **Khớp chính xác:** Tìm kiếm chuỗi con khớp hoàn toàn với mẫu.
- **Khớp gần đúng:** Cho phép một số sai lệch (chèn, xóa, thay thế ký tự) giữa mẫu và văn bản. Ứng dụng trong tìm kiếm DNA hoặc sửa lỗi chính tả.
- **Khớp với ký tự đại diện (wildcard):** Mẫu có thể chứa các ký tự đặc biệt đại diện cho bất kỳ ký tự nào.
- **Khớp đa mẫu:** Tìm kiếm nhiều mẫu trong cùng một văn bản.

Bài toán khớp xâu không chỉ là một vấn đề lý thuyết mà còn có ý nghĩa thực tiễn lớn, là nền tảng cho nhiều ứng dụng trong các lĩnh vực như xử lý văn bản, phân tích dữ liệu, và trí tuệ nhân tạo.

2.1.2 Các ứng dụng thực tế

Bài toán khớp xâu có nhiều ứng dụng quan trọng trong thực tế:

a. Tìm kiếm văn bản

Khớp xâu là cốt lõi của các công cụ tìm kiếm văn bản như "Ctrl+F", Microsoft Word, Google Docs. Trong các hệ thống tìm kiếm thông tin (information retrieval), khớp xâu giúp xác định các tài liệu chứa từ khóa cụ thể. Các công cụ như Elasticsearch hoặc Apache Lucene sử dụng kỹ thuật khớp xâu để xử lý truy vấn.

b. Xử lý dữ liệu sinh học

Trong lĩnh vực sinh học và tin sinh học (bioinformatics), khớp xâu được dùng để:

- Tìm kiếm các mẫu chuỗi đặc trưng (motif) liên quan đến gen hoặc vùng điều hòa.
- So sánh chuỗi DNA giữa các loài để nghiên cứu tiến hóa hoặc phát hiện đột biến.

- Lắp ráp chuỗi (sequence assembly) trong quá trình giải trình tự gen.

c. Công cụ tìm kiếm và xử lý ngôn ngữ tự nhiên

Khớp xâu được sử dụng để xử lý truy vấn người dùng, phân tích cú pháp, nhận diện thực thể (NER), phát hiện đạo văn, và tiền xử lý văn bản (tokenization, loại bỏ stop words).

d. An ninh mạng

Trong an ninh mạng, khớp xâu giúp:

- Phát hiện xâm nhập (IDS).
- Quét virus bằng cách so khớp với chữ ký mã độc.
- Phân tích log hệ thống để phát hiện hành vi bất thường.

e. Các ứng dụng khác

- **Nén dữ liệu:** Thuật toán như LZW sử dụng khớp xâu để phát hiện chuỗi lặp lại.
- **Xử lý hình ảnh:** Nhận diện mẫu thông qua chuỗi đặc trưng.
- **Hệ thống đề xuất:** Tìm kiếm sản phẩm hoặc nội dung dựa trên mô tả văn bản.

2.2 Các yếu tố đánh giá thuật toán

Khi nghiên cứu và so sánh các thuật toán khớp xâu, việc đánh giá hiệu quả và đặc điểm hoạt động của từng thuật toán là rất quan trọng. Các yếu tố đánh giá không chỉ giúp xác định thuật toán nào phù hợp hơn trong từng tình huống cụ thể, mà còn giúp làm sáng tỏ các ưu nhược điểm cốt lõi của mỗi phương pháp. Dưới đây là các khía cạnh cơ bản thường được sử dụng để đánh giá các thuật toán khớp xâu:

- **Độ phức tạp thời gian**
  - Trường hợp tốt nhất
  - Trường hợp trung bình
  - Trường hợp xấu nhất
- **Độ phức tạp không gian**
- **Hiệu năng thực tế**
  - Thời gian trên văn bản ngắn vs văn bản dài
  - Ảnh hưởng kích thước mẫu
  - Đặc điểm nội dung văn bản
- **Tính ổn định và độ tin cậy**
  - Xử lý chuỗi trống, đặc biệt, ký tự không hợp lệ
  - Đảm bảo kết quả chính xác, không bị lỗi
- **Khả năng mở rộng**
  - Xử lý dữ liệu lớn
  - Tìm kiếm nhiều mẫu đồng thời
- **Khả năng cài đặt và bảo trì**
- **Khả năng xử lý các trường hợp đặc biệt**

  - Văn bản không phải ASCII
  - Mẫu trùng lặp nhiều lần
  - Ký tự đặc biệt, khoảng trắng
- **Tính phù hợp với ứng dụng thực tế**

Chương 3

Cơ sở lí thuyết

3.1 Định nghĩa khớp xâu

3.2 Thuật toán Brute Force

Thuật toán Brute Force (hay còn gọi là thuật toán so khớp chuỗi tuyến tính hoặc Naive String Matching) là phương pháp đơn giản và cơ bản nhất để tìm kiếm một chuỗi mẫu trong một văn bản. Thuật toán này không sử dụng bất kỳ kiến thức bổ sung nào về cấu trúc của chuỗi mẫu hay văn bản, mà chỉ đơn thuần kiểm tra lần lượt từng vị trí có thể trong văn bản.

3.2.1 Nguyên lý hoạt động tổng quát

Ý tưởng cốt lõi của thuật toán là duyệt qua toàn bộ văn bản và so sánh chuỗi mẫu tại từng vị trí có thể xuất hiện. Cho văn bản `text` có độ dài n và chuỗi mẫu `pattern` có độ dài m, thuật toán thực hiện các bước sau:

1. Duyệt từng vị trí i từ 0 đến $n - m$ trong văn bản.
2. Tại mỗi vị trí i, so sánh lần lượt các ký tự:

$$\text{text}[i + j] \stackrel{?}{=} \text{pattern}[j] \quad \text{với } 0 \leq j < m$$

3. Nếu tất cả m ký tự khớp nhau, trả về vị trí i là một vị trí xuất hiện của chuỗi mẫu trong văn bản.

Thuật toán sẽ tiếp tục tìm kiếm cho đến hết văn bản.

3.2.2 Mã giả (Pseudocode)

Input: text, pattern

$n \leftarrow$ độ dài của chuỗi văn bản (text)

$m \leftarrow$ độ dài của chuỗi mẫu (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** $j = m$:

– Thông báo tìm thấy mẫu tại vị trí i

3.2.3 Ví dụ minh họa

Giả sử văn bản là `"abracadabra"` và chuỗi mẫu là `"bra"`:

Vị trí 0: `"abr"` $\neq$ `"bra"` (không khớp)

Vị trí 1: `"bra"` = `"bra"` (khớp) $\Rightarrow$ đánh dấu vị trí 1

Vị trí 2: `"rac"` $\neq$ `"bra"`

Vị trí 3: `"aca"` $\neq$ `"bra"`

Vị trí 4: `"cad"` $\neq$ `"bra"`

Vị trí 5: `"ada"` $\neq$ `"bra"`

Vị trí 6: `"dab"` $\neq$ `"bra"`

Vị trí 7: `"abr"` $\neq$ `"bra"`

Vị trí 8: `"bra"` = `"bra"` (khớp) $\Rightarrow$ đánh dấu vị trí 8

Kết quả: Chuỗi mẫu xuất hiện tại các vị trí 1 và 8 trong văn bản.

3.2.4 Độ phức tạp

Thời gian thực hiện:

Với văn bản có độ dài n và chuỗi mẫu có độ dài m, thuật toán Brute Force thử so khớp chuỗi mẫu tại mỗi vị trí từ $i = 0$ đến $i = n - m$ trong văn bản, tổng cộng là $(n - m + 1)$ lần.

Tại mỗi vị trí, ta cần so sánh tối đa m ký tự giữa chuỗi mẫu và đoạn văn bản hiện tại. Như vậy:

- Tổng số phép so sánh ký tự tối đa là:

$$T(n, m) \leq m(n - m + 1) = \mathcal{O}(nm)$$

- Trong trường hợp xấu nhất, không có ký tự nào trùng giữa chuỗi mẫu và đoạn văn bản đang xét (hoặc chỉ trùng một phần nhỏ trước khi phát hiện sai khác ở ký tự cuối), mỗi lần lặp có thể cần đến m phép so sánh $\rightarrow$ độ phức tạp thời gian là:

$$\mathcal{O}(nm)$$

- Trường hợp tốt nhất: nếu các ký tự đầu tiên của chuỗi mẫu và văn bản luôn khác nhau (ví dụ mẫu là `"z"` và văn bản là toàn bộ `"a"`), thì mỗi lần so khớp chỉ cần 1 phép so sánh để phát hiện sai khác. Do đó độ phức tạp tốt nhất là:

$$\mathcal{O}(n)$$

Không gian bộ nhớ:

Thuật toán chỉ sử dụng một số biến đếm và biến chỉ số để duyệt văn bản và mẫu (thường là các biến i, j, và biến tạm), không sử dụng thêm cấu trúc dữ liệu bổ trợ, nên độ phức tạp không gian là:

$$\mathcal{O}(1)$$

Kết luận:

- **Độ phức tạp thời gian:**
  - Trường hợp tốt nhất: $\mathcal{O}(n)$
  - Trường hợp trung bình: $\mathcal{O}(nm)$
  - Trường hợp xấu nhất: $\mathcal{O}(nm)$
- **Độ phức tạp không gian:** $\mathcal{O}(1)$

3.2.5 Ưu điểm và Nhược điểm

Ưu điểm:

- Rất đơn giản, dễ hiểu và dễ cài đặt.
- Không yêu cầu bước tiền xử lý chuỗi mẫu.
- Không sử dụng bất kỳ cấu trúc dữ liệu đặc biệt nào.
- Phù hợp với các văn bản nhỏ hoặc khi số lần tìm kiếm ít.

Nhược điểm:

- Hiệu suất thấp khi văn bản hoặc mẫu dài, hoặc khi số lượng chuỗi cần tìm lớn.
- Không tận dụng được tính chất trùng lặp của mẫu (như KMP hay Boyer-Moore).

3.2.6 Kết luận

Thuật toán Brute Force là phương pháp cơ bản nhất trong các thuật toán tìm kiếm chuỗi. Dù hiệu suất không cao trong trường hợp tổng quát, nhưng nhờ vào tính đơn giản và không yêu cầu cấu trúc dữ liệu đặc biệt, thuật toán này vẫn được sử dụng trong các tình huống đơn giản, khi hiệu suất không phải là vấn đề chính.

Trong các bài toán thực tế, đặc biệt với văn bản lớn hoặc khi cần xử lý nhiều mẫu tìm kiếm, các thuật toán nâng cao như Karp-Rabin, Knuth-Morris-Pratt (KMP), hay Boyer-Moore thường sẽ là những lựa chọn tốt hơn nhờ hiệu suất cao và khả năng tận dụng thông tin mẫu để giảm thiểu số phép so sánh.

3.3 Thuật toán Karp-Rabin

Thuật toán Rabin Karp là một thuật toán khớp xâu hiệu quả, được phát triển bởi Richard M. Karp và Michael O. Rabin vào năm 1987. Không giống như thuật toán Brute Force, vốn so sánh từng ký tự của mẫu với văn bản, Rabin Karp sử dụng kỹ thuật băm (hashing) để giảm số lần so sánh ký tự. Ý tưởng chính là tính giá trị băm của mẫu và các đoạn con có độ dài bằng mẫu trong văn bản, sau đó so sánh các giá trị băm này. Nếu giá trị băm khớp, thuật toán mới thực hiện so sánh ký tự để xác nhận khớp chính xác, giúp tiết kiệm thời gian trong trường hợp trung bình.

Thuật toán này đặc biệt hữu ích khi cần tìm kiếm nhiều mẫu trong cùng một văn bản hoặc khi văn bản và mẫu có kích thước lớn. Rabin Karp được đánh giá là một thuật toán có hiệu suất trung bình tốt, mặc dù trong trường hợp xấu nhất, nó có thể hoạt động kém hơn các thuật toán như Boyer Moore.

3.3.1 Nguyên lý hoạt động tổng quát

Thuật toán hoạt động dựa trên việc tính toán một giá trị băm cho chuỗi mẫu và cho từng chuỗi con có cùng độ dài trong văn bản. Nếu phát hiện giá trị băm của chuỗi con và chuỗi mẫu trùng nhau, thuật toán sẽ thực hiện kiểm tra trực tiếp ký tự để xác nhận sự trùng khớp hoàn toàn.

Ý tưởng cốt lõi là thay vì so sánh trực tiếp từng ký tự trong chuỗi văn bản với chuỗi mẫu, ta có thể ánh xạ các chuỗi này sang một không gian số thông qua một hàm băm, rồi so sánh các giá trị số này, điều này giúp giảm đáng kể số phép so sánh cần thiết.

3.3.2 Hàm Băm

Ý tưởng chính của thuật toán Rabin-Karp bắt đầu từ việc mã hóa chuỗi ký tự thành một dãy số dựa trên mã ASCII. Giả sử ta có một chuỗi ký tự, mỗi ký tự sẽ được ánh xạ sang một giá trị nguyên trong khoảng từ 0 đến 255. Ví dụ, chuỗi `"abcd"` sẽ được mã hóa thành dãy {97, 98, 99, 100}.

Tiếp theo, ta coi dãy số này như một số trong hệ cơ số *base*. Hệ cơ số *base* phải lớn hơn mã ASCII lớn nhất trong chuỗi để tránh trùng lặp khi chuyển đổi. Sau đó, ta tính giá trị thập phân của dãy số đó giống như cách tính giá trị trong hệ cơ số. Dễ nhận thấy rằng nếu hai chuỗi có cùng giá trị thập phân sau khi mã hóa thì chúng là giống nhau.

Tuy nhiên, vì chuỗi có thể rất dài, giá trị số nguyên thu được có thể vượt quá giới hạn lưu trữ thông thường. Để khắc phục, ta không tính trực tiếp giá trị thập phân, mà thay vào đó, tính phần dư khi chia cho một số nguyên lớn q . Cụ thể, nếu:

- x là giá trị thập phân của chuỗi a,
- y là giá trị thập phân của chuỗi b,

thì ta coi $a = b$ nếu như:

$$x \bmod q = y \bmod q$$

Đây chính là nguyên lý hoạt động của hàm băm (*hash function*) trong thuật toán Rabin-Karp.

Vai Trò của Hàm Băm trong Thuật Toán Rabin-Karp

Hàm băm là thành phần cốt lõi trong thuật toán **Rabin-Karp**, với vai trò chuyển một chuỗi ký tự thành một số nguyên duy nhất gọi là *giá trị băm* (hash value). Mục tiêu chính của việc sử dụng hàm băm là:

- **Tối ưu hóa tốc độ:** Giá trị băm của chuỗi mẫu độ dài m được tính trong thời gian $\mathcal{O}(m)$. Sau đó, khi tìm kiếm trong chuỗi văn bản dài n, giá trị băm của các đoạn con độ dài m được cập nhật trong thời gian $\mathcal{O}(1)$ nhờ kỹ thuật *cửa sổ trượt* (sliding window), giúp giảm độ phức tạp toàn bộ xuống $\mathcal{O}(n)$.
- **Giảm thiểu đụng độ (collision):** Mặc dù có khả năng xảy ra trường hợp hai chuỗi khác nhau có cùng giá trị băm (gọi là *xung đột băm*), việc lựa chọn cơ số b và số nguyên tố q một cách hợp lý (ví dụ: chọn q là số nguyên tố lớn) sẽ giảm xác suất xảy ra xung đột tới mức chấp nhận được.

Khi hai đoạn văn bản có cùng giá trị băm với chuỗi mẫu, thuật toán sẽ kiểm tra trực tiếp từng ký tự trong đoạn con và chuỗi mẫu để xác nhận sự trùng khớp. Do đó, hàm băm giúp sàng lọc nhanh chóng các vị trí nghi ngờ, trong khi việc so khớp chính xác vẫn được đảm bảo sau bước kiểm tra cuối cùng, duy trì cả hiệu suất và độ chính xác.

Công Thức Hàm Băm

Giả sử chuỗi $s = s_0 s_1 \dots s_{m-1}$, với mỗi ký tự s_k được ánh xạ thành một số nguyên (ví dụ: giá trị ASCII). Khi đó, hàm băm được định nghĩa như sau:

$$h(s) = \left(\sum_{k=0}^{m-1} s_k \cdot b^{m-1-k} \right) \bmod q$$

Trong đó:

- b: Cơ số (base), thường chọn là 256 (tương ứng với 256 ký tự ASCII) hoặc một số nguyên tố như 31 hay 101.

- q: Số nguyên tố lớn, dùng để giảm đụng độ và giới hạn phạm vi giá trị băm.

Giải thích chi tiết:

- Ký tự đầu tiên s_0 có trọng số lớn nhất b^{m-1}, ký tự cuối s_{m-1} có trọng số $b^0 = 1$. Cách gán trọng số này giúp phân biệt các chuỗi có cùng tập ký tự nhưng khác thứ tự (ví dụ: "ABC" $\neq$ "CBA").
- Phép modulo q giới hạn giá trị băm trong đoạn $[0, q-1]$ và tăng tính ngẫu nhiên của phân phối.

Ví dụ:

Xét chuỗi $P =$ `"ABC"` với mã ASCII: $A = 65$, $B = 66$, $C = 67$. Chọn $b = 256$, $q = 101$:

$$\begin{aligned} h(\text{"}ABC\text{"}) &= (65 \cdot 256^2 + 66 \cdot 256^1 + 67 \cdot 256^0) \bmod 101 \\ &= (65 \cdot 65536 + 66 \cdot 256 + 67) \bmod 101 \\ &= (4259840 + 16896 + 67) \bmod 101 \\ &= 4276803 \bmod 101 = 12 \end{aligned}$$

Vậy giá trị băm của chuỗi "ABC" là 12.

3.3.3 Kỹ Thuật Rolling Hash

Kỹ thuật **Rolling Hash** (hay còn gọi là băm trượt) là một phương pháp tối ưu để tính toán giá trị băm của các cửa sổ con liên tiếp trong một chuỗi văn bản. Thay vì tính toán lại băm của mỗi cửa sổ từ đầu, Rolling Hash cho phép cập nhật giá trị băm của cửa sổ hiện tại từ giá trị băm của cửa sổ trước đó, nhờ đó giảm độ phức tạp tính toán. Đây là kĩ thuật rất quan trọng trong thuật toán Rabin - Karp.

Giả sử ta có một văn bản T có độ dài n, và ta đang làm việc với một cửa sổ con có độ dài m. Kỹ thuật Rolling Hash sẽ cho phép tính toán giá trị băm cho một cửa sổ và sau đó cập nhật giá trị băm khi cửa sổ này trượt sang vị trí tiếp theo.

Khi trượt cửa sổ từ vị trí $i-1$ sang i, giá trị băm h_i của cửa sổ mới có thể được tính bằng công thức sau:

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

Trong đó :

- T: Chuỗi văn bản.
- $T[i..i+m-1]$: Cửa sổ con có độ dài m bắt đầu tại vị trí i.
- h_{i-1}: Giá trị băm của cửa sổ con trước đó $T[i-1..i+m-2]$.
- h_i: Giá trị băm của cửa sổ con hiện tại $T[i..i+m-1]$.
- d: Cơ số dùng trong phép băm (thường là 256, đại diện cho số lượng ký tự trong bảng mã ASCII).
- q: Số nguyên tố lớn, dùng để tính toán modulo để tránh tràn số.

Giải Thích Các Thành Phần:

- **Loại bỏ ký tự cũ**: Ký tự $\text{text}[i-1]$ trong cửa sổ cũ có giá trị $\text{text}[i-1] \cdot d^{m-1}$, đóng góp vào giá trị băm của cửa sổ trước đó. Khi trượt cửa sổ sang phải, ta cần loại bỏ phần đóng góp này. Phần này sẽ bị trừ đi trong công thức.
- **Dịch trái phần còn lại**: Sau khi loại bỏ ký tự cũ, ta cần nhân toàn bộ phần còn lại của giá trị băm (tức là h_{i-1}) với cơ số d để "dịch" giá trị băm sang trái một vị trí.

- **Thêm ký tự mới**: Sau khi dịch phần còn lại, ta cộng thêm ký tự mới text$[i+m-1]$ vào giá trị băm của cửa sổ mới. Đây là ký tự mới xuất hiện ở cuối cửa sổ khi trượt sang phải.
- **Modulo** q: Sau khi tính toán xong, ta lấy kết quả modulo với số nguyên tố q để đảm bảo rằng giá trị băm không vượt quá giới hạn của số nguyên, tránh tràn số. Điều này cũng giúp bảo vệ khỏi hiện tượng va chạm (collision) trong phép băm.

Lưu ý: Nếu kết quả tính toán trước khi lấy modulo là số âm, ta cần cộng thêm q vào để đảm bảo rằng kết quả sau modulo là một số dương.

3.3.4 Độ phức tạp và xác suất va chạm

Thuật toán yêu cầu thời gian tỷ lệ với $n+m$ trong hầu hết các trường hợp, trong đó n là độ dài của văn bản và m là độ dài của chuỗi mẫu. Điều đáng chú ý là hiệu suất này đạt được mà không cần sử dụng thêm bộ nhớ phụ đáng kể nào. Thuật toán chỉ cần một số biến để lưu trữ giá trị băm và các chỉ số hiện tại, không cần đến các cấu trúc dữ liệu phức tạp như bảng phụ hay mảng hỗ trợ. Chính vì vậy, Karp-Rabin là một giải pháp vừa hiệu quả về thời gian vừa tiết kiệm về không gian bộ nhớ, rất phù hợp trong các ứng dụng tìm kiếm trên văn bản lớn. Chú ý rằng thuật toán Rabin Karp sẽ tìm vị trí trong văn bản mà có cùng giá trị băm với mẫu, bởi vậy để có thể so khớp ta cần tìm trong văn bản những giá trị băm bằng với giá trị băm của mẫu. Thuật toán này là một thuật toán xác suất, tức là trong quá trình hoạt động, nó không luôn đảm bảo cho kết quả chính xác tuyệt đối ngay lập tức, mà có thể xảy ra sai lệch do hiện tượng va chạm hàm băm (hash collision), tức là hai chuỗi khác nhau lại có cùng một giá trị băm. Tuy nhiên, việc chọn một giá trị lớn cho q (một số nguyên tố dùng trong phép toán modulo) sẽ giúp giảm mạnh xác suất xảy ra va chạm này. Cụ thể, xác suất để xảy ra một va chạm ngẫu nhiên giữa hai chuỗi khác nhau được ước lượng là $\mathcal{O}(1/q)$, nghĩa là tỉ lệ nghịch với độ lớn của q.Nói cách khác, càng chọn q lớn bao nhiêu thì xác suất va chạm càng nhỏ bấy nhiêu, và khi đó, thuật toán có thể hoạt động gần như chắc chắn chính xác với hiệu suất cao. Điều này làm cho thuật toán Karp-Rabin, dù có bản chất xác suất, vẫn rất hiệu quả và đáng tin cậy trong thực tế, đặc biệt là khi xử lý các văn bản dài hoặc nhiều chuỗi mẫu khác nhau.

Xác suất va chạm giữa hai chuỗi khác nhau có thể ước lượng theo công thức:

$$\mathbb{P}\{\text{collision}\} = \mathcal{O}\left(\frac{1}{q}\right),$$

với q là số nguyên tố lớn. Khi q đủ lớn, xác suất va chạm sẽ rất nhỏ.Trên lí thuyết, thuật toán này có thể cần mn bước trong trường hợp tệ nhất Nếu phải kiểm tra quá nhiều phép so khớp hoặc xảy ra đụng độ quá nhiều thì hiệu suất của thuật toán có thể bị ảnh hưởng. Tuy nhiên, trong các kết quả thực nghiệm, chúng tôi chỉ ghi nhận 3 va chạm xảy ra trong tổng số 10^7 lần tính toán hàm băm, khi làm việc với bảng chữ cái có kích thước lớn. Điều này cho thấy rằng với việc lựa chọn tham số phù hợp (như giá trị q lớn và bảng chữ cái rộng), xác suất xảy ra va chạm là rất thấp, và do đó thuật toán hoạt động hiệu quả trong thực tế.

Karp và Rabin chỉ ra rằng:

$$\text{Prob(collision)} \leq \frac{\pi(m(n-m+1))}{\pi(M)},$$

trong đó $\pi(z)$ là số lượng số nguyên tố nhỏ hơn hoặc bằng z, với điều kiện $m(n-m+1) \geq 29$, $c=2$, và q là một số nguyên tố ngẫu nhiên không vượt quá M. Họ sử dụng $M = O(mn^2)$ để đạt được xác suất va chạm ở mức $O(1/n)$. Tuy nhiên, trong thực tế, giới hạn M phụ thuộc vào độ rộng từ (word size) được sử dụng trong các phép toán số học, chứ không phụ thuộc trực tiếp vào m hay n. Dựa trên mô hình tính toán thực tế hơn, chúng ta có kết quả sau:

$$\text{Prob(collision)} = \frac{1}{q} - O\left(\frac{1}{c^m}\right) < \frac{1}{q},$$

cho một hàm băm đồng đều (uniform signature function).

Số lượng so sánh kỳ vọng giữa chuỗi văn bản và mẫu (text-pattern numerical comparisons) mà thuật toán Karp–Rabin thực hiện khi tìm kiếm mẫu có độ dài m trong văn bản độ dài n là:

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

trong đó $\mathcal{H}$ là chi phí tính toán hàm băm (signature function), tính theo số lần so sánh.

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

Bảng 1: Kết quả lý thuyết (hàng trên) và thực nghiệm (hàng dưới) của thuật toán Karp–Rabin.

Kết quả thực nghiệm được so sánh với kết quả lý thuyết trong Bảng 1 với giả định $\mathcal{H} = 1$. Các giá trị này phù hợp tốt với nhiều bộ chữ cái khác nhau. Trong thực tế, giá trị $\mathcal{H}$ có thể lớn hơn do các phép nhân và chia dư (modulo) trong quá trình tính toán hàm băm. Tuy nhiên, điều này chỉ ảnh hưởng rõ rệt với các chuỗi dài. Để khắc phục, ta có thể tận dụng phép toán chia dư ngầm định do phần cứng hỗ trợ, tức là sử dụng kích thước từ tối đa được hỗ trợ để tính toán hiệu quả hơn.

3.3.5 Tối ưu hóa bằng phần cứng

Trong một số hệ thống, có thể tận dụng đặc tính tràn số nguyên để thay thế phép chia dư modulo trong quá trình tính giá trị băm. Thay vì tính toán modulo q ở mỗi bước, hệ thống cho phép:

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

và bỏ qua phép modulo. Khi sử dụng số nguyên giới hạn (ví dụ 32-bit hoặc 64-bit), việc tràn số sẽ tự động giới hạn phạm vi giá trị, giống như một modulo ngầm. Điều này làm giảm số lượng phép toán chia vốn tốn kém về thời gian, qua đó tăng hiệu năng thuật toán.

3.3.6 Kết luận

Thuật toán Karp-Rabin thể hiện tính hiệu quả và tính thực tiễn cao trong các bài toán tìm kiếm chuỗi, đặc biệt với các văn bản lớn và khi cần xử lý nhiều mẫu cùng lúc. Sử dụng hàm băm tuyến tính giúp đại diện chuỗi một cách nhỏ gọn và có thể tính toán lặp nhanh. Mặc dù có thể xảy ra va chạm, việc lựa chọn tham số thích hợp cho phép giảm thiểu vấn đề này tới mức không đáng kể. Đồng thời, thuật toán có thể được tối ưu hóa tốt trên phần cứng nhờ kỹ thuật tràn số. Với những ưu điểm đó, Karp-Rabin là một trong những giải pháp kinh điển và nền tảng trong lĩnh vực xử lý chuỗi và tìm kiếm văn bản.

3.4 Boyer Moore

3.4.1 Giới thiệu

Thuật toán Boyer-Moore (BM), được công bố vào năm 1977 bởi Robert Boyer và J Strother Moore, là một trong những thuật toán tìm kiếm chuỗi hiệu quả nhất, đặc biệt trong các ứng dụng thực tế. Phần này trình bày chi tiết thuật toán Boyer-Moore, cùng với hai biến thể: Simplified Boyer-Moore (SBM) và Boyer-Moore-Horspool (BMH).

3.4.2 Thuật toán Boyer-Moore

Thuật toán Boyer-Moore tìm kiếm chuỗi bằng cách so sánh từ **phải sang trái** trong mẫu, giúp giảm số lần so sánh ký tự. Nó sử dụng hai chiến lược dịch chuyển (heuristics):

- **Match heuristic**: Dựa trên các ký tự đã khớp để dịch chuyển mẫu.
- **Occurrence heuristic**: Dựa trên ký tự gây ra không khớp để căn chỉnh mẫu.

3.4.3 Cách hoạt động

1. Đặt mẫu căn chỉnh với ký tự đầu tiên của văn bản và so sánh từ ký tự cuối cùng của mẫu.
2. Nếu tất cả ký tự khớp, báo cáo vị trí khớp. Nếu có không khớp, tính khoảng cách dịch chuyển bằng hai heuristics và chọn giá trị lớn nhất.
3. Dịch chuyển mẫu sang phải và lặp lại.

3.4.4 Bảng dịch chuyển

Thuật toán sử dụng hai bảng:

- **Bảng dd (match heuristic)**:

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

- **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: Với 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.

3.4.5 Mã nguồn

```
bmsearch(text, n, pat, m) /* Tim pat[1..m] trong text[1..n] */
char text[], pat[];
int n, m;
{
    int k, j, skip;
    int dd[MAX_PATTERN_SIZE], d[MAX_ALPHABET_SIZE];
    initd(pat, m, d); /* Tien xu ly mau de tao bang d */
    initdd(pat, m, dd); /* Tien xu ly mau de tao bang dd-hat */
    k = m; skip = dd[1] + 1;
    while (k <= n) /* Tim kiem */
```

```
    {
        j = m;
        while (j > 0 && text[k] == pat[j])
        {
            j--; k--;
        }
        if (j == 0)
        {
            Report_match_at_position(k + 1);
            k += skip;
        }
        else
            k += max(d[text[k]], dd[j]);
    }
}

initd(pat, m, d) /* Tien xu ly mau: tao bang d */
char pat[];
int m, d[];
{
    int k;
    for (k = 0; k <= MAX_ALPHABET_SIZE; k++)
        d[k] = m;
    for (k = 1; k <= m; k++)
        d[pat[k]] = m - k;
}

initdd(pat, m, dd) /* Tien xu ly mau: tao bang dd-hat */
char pat[];
int m, dd[];
{
    int j, k, t, t1, q, q1;
    int f[MAX_PATTERN_SIZE + 1];
    for (k = 1; k <= m; k++)
        dd[k] = 2 * m - k;
    for (j = m, t = m + 1; j > 0; j--, t--)
    {
        f[j] = t;
        while (t <= m && pat[j] != pat[t])
        {
            dd[t] = min(dd[t], m - j);
            t = f[t];
        }
    }
    q = t; t = m + 1 - q; q1 = 1;
    for (j = 1, t1 = 0; j <= t; t1++, j++)
        f[j] = t1;
    while (t1 >= 1 && pat[j] != pat[t1])
        t1 = f[t1];
    while (q < m)
    {
        for (k = q1; k <= q; k++)
            dd[k] = min(dd[k], m + q - k);
        q1 = q + 1; q = q + t - f[t]; t = f[t];
    }
}
```

3.4.6 Phân tích lý thuyết

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

$$\frac{\bar{C}_n}{n} \geq \frac{1}{m} + \frac{m(m+1)}{2m^2c} + O\left(c^{-2}\right)$$

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

3.4.7 Cải tiến

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

3.5 Thuật toán Simplified Boyer-Moore (SBM)

SBM chỉ sử dụng occurrence heuristic, giảm không gian từ $O(m + c)$ xuống $O(c)$ và đơn giản hóa tiền xử lý.

3.5.1 Phân tích

- **Trường hợp xấu nhất**: $O(mn)$.
- **Trường hợp trung bình**:

$$\frac{\bar{C}_n}{n} \geq \frac{1}{c-1} + \frac{1}{c^4} + O\left(c^{-6}\right) \quad (\text{với } n, m \text{ lớn})$$

$$\frac{\bar{C}_n}{n} \geq \frac{1}{m} + \frac{m^2+1}{2cm^2} + O\left(c^{-2}\right) \quad (\text{với } n, c \text{ lớn})$$

3.6 Thuật toán Boyer-Moore-Horspool (BMH)

BMH, do 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.

3.6.1 Bảng d

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

Ví dụ: Với mẫu `abracadabra`:

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

3.6.2 Mã nguồn

```
bmhsearch(text, n, pat, m) /* Tim pat[1..m] trong text[1..n] */
char text[], pat[];
int n, m;
{
    int d[MAX_ALPHABET_SIZE], i, j, k;
    for (j = 0; j < MAX_ALPHABET_SIZE; j++)
        d[j] = m;
```

```
    for (j = 1; j < m; j++)
        d[pat[j]] = m - j;
    pat[0] = CHARACTER_NOT_IN_THE_TEXT;
    text[0] = CHARACTER_NOT_IN_THE_PATTERN;
    i = m;
    while (i <= n)
    {
        k = i;
        for (j = m; text[k] == pat[j]; j--)
            k--;
        if (j == 0)
            Report_match_at_position(k + 1);
        i += d[text[i]];
    }
}
```

3.6.3 Phân tích lý thuyết

- **Khoảng cách dịch chuyển trung bình**:

$$\bar{S} = c\left(1 - \left(1 - \frac{1}{c}\right)^m\right)$$

- **Số lần so sánh trung bình**:

$$\frac{\bar{C}_n}{n} \geq \frac{1}{c-1} + O\left(\left(1 - \frac{1}{c}\right)^m\right) \quad (\text{với } n, m \text{ lớn})$$

$$\frac{\bar{C}_n}{n} = \frac{1}{m} + \frac{m+1}{2mc} + O\left(c^{-2}\right) \quad (\text{với } n, c \text{ lớn})$$

3.7 Kết luận

BMH là thuật toán hiệu quả nhất trong thực tế, đặc biệt với mẫu dài và bảng chữ cái lớn. SBM đơn giản hơn nhưng kém hiệu quả trong trường hợp xấu. BM gốc mạnh mẽ nhưng phức tạp hơn về tiền xử lý.

3.8

3.9

Chương 4

Triển khai thuật toán

4.1 Naive

4.1.1 Các bước thực hiện

- Gọi T là văn bản đầu vào có độ dài n.
- Gọi P là chuỗi mẫu cần tìm có độ dài m.
- Với mỗi vị trí i từ 0 đến $n - m$, kiểm tra xem đoạn $T[i..i + m - 1]$ có trùng với $P[0..m - 1]$ không.
- Nếu có, thì in hoặc lưu lại vị trí i là vị trí khớp.

4.1.2 Mã giả

Algorithm 1 Thuật toán Naive khớp xâu

```
for i ← 0 to n − m do
    j ← 0
    while j < m and T[i + j] = P[j] do
        j ← j + 1
    end while
    if j = m then
        In ra vị trí i
    end if
end for=0
```

4.2 Thuật toán Rabin-Karp

4.2.1 Các bước thực hiện

1. Khởi tạo các biến cần thiết:

- n là độ dài của văn bản (text).
- m là độ dài của mẫu (pattern).
- h là giá trị băm của mẫu được tính từ $d^{m-1} \mod q$, trong đó d là kích thước bảng chữ cái và q là số nguyên tố.
- p và t lần lượt là giá trị băm của mẫu và cửa sổ hiện tại trong văn bản.

2. Tính giá trị băm ban đầu cho mẫu và cửa sổ đầu tiên trong văn bản:

- Dùng vòng lặp từ 0 đến $m-1$ để tính giá trị băm của mẫu p và cửa sổ đầu tiên trong văn bản t.

3. Duyệt qua các cửa sổ trong văn bản:

- Duyệt từ 0 đến $n-m$ để kiểm tra mỗi cửa sổ con của văn bản.
- Nếu giá trị băm của cửa sổ và mẫu trùng nhau, kiểm tra từng ký tự trong cửa sổ và mẫu để xác nhận sự khớp thực sự.

4. Cập nhật giá trị băm của cửa sổ cho mỗi bước di chuyển:

- Sử dụng công thức trượt băm để cập nhật giá trị băm cho cửa sổ tiếp theo trong văn bản. Công thức là:

$$t = (d \cdot (t - \text{ASCII}(\text{text}[s]) \cdot h) + \text{ASCII}(\text{text}[s+m])) \mod q$$

- Nếu giá trị băm t nhỏ hơn 0 sau khi tính, cộng thêm q vào để giữ giá trị băm không âm.

5. Kết quả:

- Nếu có sự khớp giữa mẫu và cửa sổ con trong văn bản, in ra vị trí tìm thấy. Nếu không có, tiếp tục kiểm tra các cửa sổ tiếp theo.

4.2.2 Mã giả

Algorithm 2 Rabin-Karp String Matching

Require: text, pattern, alphabet size d, prime q

```
n ← length(text)
m ← length(pattern)
h ← (d^{m-1}) mod q
p ← 0, t ← 0
for i = 0 to m − 1 do
    p ← (d · p + ASCII(pattern[i])) mod q
    t ← (d · t + ASCII(text[i])) mod q
end for
for s = 0 to n − m do
    if p = t and text[s : s + m] = pattern then
        print "Found at", s
    end if
    if s < n − m then
        t ← (d · (t − ASCII(text[s]) · h) + ASCII(text[s + m])) mod q
        if t < 0 then
            t ← t + q
        end if
    end if
end for=0
```

4.3 Thuật toán Boyer Moore

4.3.1 Các Bước Thực Hiện

1. **Khởi tạo các biến cần thiết:**
    - n: Độ dài của văn bản (text).
    - m: Độ dài của mẫu (pattern).
    - Bảng Bad Character ($last$): Một mảng lưu vị trí xuất hiện cuối cùng của mỗi ký tự trong mẫu. Nếu ký tự không có trong mẫu, giá trị là -1.

- Bảng Good Suffix ($goodSuffix$): Một mảng lưu thông tin về các hậu tố khớp để xác định khoảng nhảy tối ưu.

2. **Tạo bảng Bad Character:**
   - Duyệt qua mẫu từ trái sang phải.
   - Với mỗi ký tự $pattern[i]$, cập nhật $last[\text{ASCII}(pattern[i])] = i$.
   - Các ký tự không có trong mẫu được gán giá trị -1.
3. **Tạo bảng Good Suffix:**
   - Khởi tạo mảng $suffix$ để lưu độ dài của hậu tố dài nhất bắt đầu từ vị trí i khớp với một tiền tố của mẫu.
   - Duyệt mẫu từ phải sang trái để xây dựng mảng $suffix$:
     - Nếu hậu tố tại vị trí i khớp với tiền tố, lưu độ dài khớp vào $suffix[i]$.
     - Sử dụng các biến phụ (g, f) để theo dõi hậu tố và tiền tố khớp.
   - Dựa trên $suffix$, xây dựng mảng $goodSuffix$:
     - Với mỗi vị trí i, $goodSuffix[i]$ lưu khoảng nhảy tối ưu khi không khớp tại i.
     - Xử lý hai trường hợp:
       - *Trường hợp 1*: Nếu hậu tố khớp, nhảy để căn chỉnh với tiền tố tương ứng.
       - *Trường hợp 2*: Nếu không có tiền tố khớp, nhảy dựa trên hậu tố dài nhất có thể.
4. **Duyệt qua văn bản:**
   - Bắt đầu từ vị trí $s = 0$ (cửa sổ đầu tiên).
   - So sánh mẫu với cửa sổ từ phải sang trái (từ $i = m - 1$ đến 0):
     - Nếu $text[s + i] = pattern[i]$, tiếp tục so sánh ký tự trước.
     - Nếu không khớp tại $text[s + i]$, tính khoảng nhảy:
       - *Bad Character*: Lấy ký tự không khớp $c = text[s + i]$. Nhảy dựa trên $last[c]$:
         - Nếu $last[c] \geq 0$, nhảy để căn chỉnh c với vị trí $last[c]$ trong mẫu.
         - Nếu $last[c] = -1$, nhảy toàn bộ mẫu qua vị trí không khớp.
         - Công thức: badCharShift $= \max(1, i - last[c])$.
       - *Good Suffix*: Nhảy dựa trên hậu tố đã khớp, sử dụng $goodSuffix[i + 1]$.
       - Chọn khoảng nhảy lớn nhất: $s = s + \max(\text{badCharShift}, \text{goodSuffixShift})$.
5. **Kiểm tra khớp và in kết quả:**
   - Nếu toàn bộ mẫu khớp (tất cả ký tự từ $i = m - 1$ đến 0), in vị trí s.
   - Tiếp tục nhảy đến cửa sổ tiếp theo.
6. **Kết thúc:**
   - Khi $s + m > n$, dừng thuật toán.

4.3.2 Mã Giả Boyer-Moore

Algorithm 3 Boyer-Moore String Matching with Bad Character and Good Suffix

Require: text, pattern

```
0: n ← length(text)
0: m ← length(pattern)
0: Initialize last[256] to -1 {Bad Character table}
0: Initialize goodSuffix[m + 1] {Good Suffix table}
0: Initialize suffix[m + 1] {Auxiliary array for Good Suffix}
0: procedure PREPROCESSBADCHARACTER(pattern, m, last)
0:     for i = 0 m − 1 do
0:         last[ASCII(pattern[i])] ← i {Store last position of each character}
0:     end for
0: end procedure
0: procedure PREPROCESSGOODSUFFIX(pattern, m, goodSuffix, suffix)
0:     Initialize suffix[0 to m] to 0
0:     suffix[m] ← m
0:     g ← m − 1
0:     for i = m − 2 −1 step −1 do
0:         if i > g  & suffix[i + m - 1 - f] < i - g then          suffix[i] ← suffix[i + m − 1 − f]
0:
0:             if i < g then
0:                 g ← i
0:             end if
0:             f ← i
0:             while g ≥ 0  & pattern[g] = pattern[g + m - 1 - f] do          g ← g − 1
0:         end while
0:         suffix[i] ← f − g
0:
0:
0:         for i = 0 m do
0:             goodSuffix[i] ← m {Initialize Good Suffix shifts}
0:         end for
0:         j ← 0
0:         for i = m − 1 −1 step −1 do
0:             if suffix[i] = i + 1 then
0:                 while j < m − 1 − i do
0:                     if goodSuffix[j] = m then
0:                         goodSuffix[j] ← m − 1 − i
0:                     end if
0:                     j ← j + 1
0:                 end while
0:             end if
0:         end for
0:         for i = 0 m − 1 do
0:             goodSuffix[m − suffix[i]] ← max(goodSuffix[m − suffix[i]], m − i)
0:         end for
0:
0:         PREPROCESSBADCHARACTER(pattern, m, last)
0:         PREPROCESSGOODSUFFIX(pattern, m, goodSuffix, suffix)
0:         s ← 0 {Starting position in text}
0:         while s ≤ n − m do
0:             i ← m − 1 {Compare from right to left}
0:             while i ≥ 0  & pattern[i] = text[s + i] do          i ← i − 1
0:             end while
0:             if i < 0 then
0:                 print "Found at", s
0:                 s ← s + goodSuffix[0] {Shift using Good Suffix}
0:             else
0:                 c ← ASCII(text[s + i])
0:                 badCharShift ← max(1, i − last[c]) {Bad Character shift}
0:                 goodSuffixShift ← goodSuffix[i + 1] {Good Suffix shift}
0:                 s ← s + max(badCharShift, goodSuffixShift) {Choose maximum shift}
0:             end if
0:         end while
```

algorithm algpseudocode

4.3.3 MÃ GIẢ

Algorithm 4 Boyer–Moore with Bad Character Good Suffix

```
0: procedure PREPROCESSBADCHARACTER(pattern, m)
0:     for i ← 0 to m − 1 do
0:         last[ASCII(pattern[i])] ← i
0:     end for
0: end procedure
0: procedure PREPROCESSGOODSUFFIX(pattern, m)
0:     initialize suffix[0..m − 1], goodSuffix[0..m]
0:     suffix[m − 1] ← m
0:     g ← m − 1, f ← 0
0:     for i ← m − 2 downto 0 do
0:         if i > g and suffix[i + m − 1 − f] < i − g then
0:             suffix[i] ← suffix[i + m − 1 − f]
0:         else
0:             g ← min(g, i),  f ← i
0:             while g ≥ 0 and pattern[g] = pattern[g + m − 1 − f] do
0:                 g ← g − 1
0:             end while
0:             suffix[i] ← f − g
0:         end if
0:     end for
0:     for i ← 0 to m do goodSuffix[i] ← m end for
0:     j ← 0
0:     for i ← m − 1 downto 0 do
0:         if suffix[i] = i + 1 then
0:             while j < m − 1 − i do
0:                 if goodSuffix[j] = m then
0:                     goodSuffix[j] ← m − 1 − i
0:                 end if
0:                 j ← j + 1
0:             end while
0:         end if
0:     end for
0:     for i ← 0 to m − 2 do
0:         goodSuffix[m − 1 − suffix[i]] ← m − 1 − i
0:     end for
0: end procedure
0: PREPROCESSBADCHARACTER(pattern, m)
0: PREPROCESSGOODSUFFIX(pattern, m)
0: s ← 0
0: while s ≤ n − m do
0:     i ← m − 1
0:     while i ≥ 0 and pattern[i] = text[s+i] do
0:         i ← i − 1
0:     end while
0:     if i < 0 then
0:         print "Found at ", s
0:         s ← s + goodSuffix[0]
0:     else
0:         c ← ASCII(text[s + i])
0:         bc ← max(1, i − last[c])
0:         gs ← goodSuffix[i + 1]
0:         s ← s + max(bc, gs)
0:     end if
0: end while=0
```

Tài liệu tham khảo

[1] R. A.Baeza-Yates, "Algorithm for string searching," *ACM SIGIR Forum*, pp. 1–25, 1989, https://dl.acm.org/doi/pdf/10.1145/74697.74700.