Thuật toán KMP (Knuth – Morris - Pratt)

I. Giới thiệu thuật toán KMP

Thuật toán được phát minh bởi James H. Moris và Vaughan Pratt vào năm 1970 song song với Donald Knuth dựa trên lý thuyết automata. Năm 1977, cả ba chính thức xuất bản báo cáo chuyên ngành cho thuật toán này. KMP là thuật toán so khớp chuỗi đầu tiên có độ phức tạp tuyến tính.

II. Mô tả thuật toán

Thuật toán KMP dựa trên tính chất phân nhỏ (degenerating property) của chuỗi kí tự cần tìm S. Theo đó bên trong chuỗi S có thể tồn tại các các chuỗi con nhỏ lặp lại. Dựa vào đó có thể cải thiện được hạn chế của thuật toán Naive – Brute Force khi gặp phải trường hợp nhiều kí tự không khớp nối tiếp sau các kí tự khớp.

Ví dụ: txt[] = “AAAAAAAAAAAAAAAAB"

pat[] = “AAAB”

Ý tưởng của thuật toán: mỗi khi gặp phải các kí tự không khớp sau các kí tự khớp, thay vì dịch tới một vị trí trên chuỗi txt[] như thuật toán Naïve, ta vận dụng những phần đã duyệt qua để tránh lặp lại các bước so chuỗi đã từng thực hiện.

Trước tiên để hiểu được thuật toán, cần nắm được một vài khái niệm thuật toán dùng đến. Cho trước chuỗi kí tự:

Pattern: a b c d a b c

- Prefix: các chuỗi con được lấy từ vị trí đầu của pattern. Các chuỗi con có thể có với chuỗi pattern trên: “” (chuỗi rỗng), a, ab, abc…
- Suffix: tương tự như prefix nhưng được bắt đầu từ kí tự cuối cùng của pattern. Ví dụ: “”, c, bc, abc, dabc…
- Từ đó ta phát triển thêm 2 khái niệm: proper prefix và proper suffix. Chúng thực chất chỉ là prefix/suffix nhưng không được lấy hết cả chuỗi pattern. (abcdabc có thể là prefix/suffix nhưng không là proper prefix/suffix)

Thực tế thấy rằng có thể có các proper prefix trùng với proper suffix. Trong ví dụ trên, "abc" vừa là proper prefix, vừa là proper suffix. Trong thuật toán KMP, ta sẽ đi tìm chuỗi LPS (chuỗi dài nhất vừa là proper prefix, vừa là proper suffix).

Trước tiên khởi tạo một mảng lps[] cùng kích thước với pattern. Mỗi phần tử lps[i] của mảng là chiều dài của LPS được tạo bởi i kí tự đầu tiên trong pattern. Cụ thể với ví dụ pattern: a b a b a a b a

- i = 0, substr = "a", LPS = "" lps[0] = 0
- i = 1, substr = "ab" LPS = "" lps[1] = 0
- i = 2, substr = "aba" LPS = "a" lps[2] = 1
- i = 3, substr = "abab" LPS = "ab" lps[3] = 2
- i = 4, substr = "ababa" LPS = "aba" lps[4] = 3
- i = 5, substr = "ababaa" LPS = "a" lps[5] = 1
- i = 6, substr = "ababaab" LPS = "ab" lps[6] = 2
- i = 7, substr = "ababaaba" LPS = "aba" lps[7] = 3

Tương tự với các trường hợp sau:

- pattern: "AAAA" lps[] = [0, 1, 2, 3]
- pattern “ABCDE” lps[] = [0, 0, 0, 0, 0]
- pattern “AAACAAAAAC” lps[] = [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

…

**Vai trò của mảng lps[] trong thuật toán KMP:*

Như đã đề cập phía trên, những trùng khớp một phần có thể dẫn đến việc so sánh lặp đi lặp lại. Bằng cách sử dụng mảng lps[] ta có thể bỏ qua việc lặp đi lặp lại đó.

Tương tự với thuật toán Naïve, dùng hai biến i, j để duyệt qua chuỗi txt[] và pat[]. Ở mỗi bước, so sánh txt[i] và pat[j]:

- Nếu trùng khớp, tăng biến i và j lên 1. Nếu đã so hết chuỗi pattern, tức đã tìm thấy được vị trí xuất hiện của pat[] trong txt[], xuất vị trí và đặt j = lps[j – 1].
- Nếu không khớp sau một chuỗi khớp, đặt j = lps[j – 1], tiếp tục so sánh.

- Nếu j = 0 và không trùng, nghĩa là chưa có bất kì kí tự nào trùng khớp, dời chuỗi pat qua một kí tự bằng cách tăng biến i.

Ý nghĩa của việc đặt j = lps[j – 1]: Gọi k = lps[j – 1]. Ta vừa so sánh (j + 1) kí tự đầu tiên của pat[] (j bắt đầu từ 0) và không trùng ở kí tự pat[j] (hoặc tìm thấy chuỗi) nên dời về kí tự pat[j – 1]. Ứng với pat[j – 1], ta đã ngầm so sánh k kí tự đầu tiên (do k kí tự đầu tiên trùng với k kí tự cuối của chuỗi con) do đó có thể bỏ qua k kí tự này và bắt đầu so sánh txt[i] với pat[k].

III. Cách cài đặt thuật toán

Mã giả cho giải thuật KMP:

```
LPS = calLPS(Pattern) // calculate LPS array function
i = 0
j = 0
n = text length
m = pattern length
while i < n do
    if pattern[j] == text[i] then  // if the characters are a match
        i++
        j++
    if j == m then  // j reaches end of pattern
        print(i - m) // index of the match
            j = LPS[j - 1]

    else if i < n && pattern[j] != text[i] then //no match
            if j > 0
            j = LPS[j - 1]
          else
            i++
```

**Cài đặt hàm tính mảng lps[]*

- Khởi tạo mảng lps với phần tử đầu tiên là 0. (Phần tử đầu tiên luôn là 0 vì với chuỗi 1 kí tự LPS luôn là chuỗi rỗng)
- Biến i = 1 để duyệt từ kí tự thứ 2 đến hết chuỗi pat[].
- Biến m lưu kích thước pat[].
- Biến prevLPS lưu độ dài LPS với chuỗi con của kí tự ngay trước đó.

- Lặp từ 1 đến m – 1, nếu kí tự pat[prevLPS] == pat[i] thì lps[i] = prevLPS. Nghĩa là nếu kí tự thứ i của chuỗi pat[] (cũng là kí tự cuối của chuỗi con) trùng với kí tự liền sau LPS của pat[i – 1] trước đó, ta nối dài chuỗi LPS trước đó.
- Nếu không trùng, ta có 2 trường hợp sau:
    - Trong trường hợp pat[i – 1] cũng có LPS là chuỗi rỗng, prevLPS == 0, thì LPS của pat[i] cũng là chuỗi rỗng, lps[i] = 0.
    - Nếu không, giảm độ dài của LPS xuống bằng với độ dài của LPS trước cả đó và tiếp tục so sánh.

IV. Phân tích độ phức tạp

Gọi k = i – j, n là độ dài của chuỗi txt[], m là độ dài chuỗi pat[].

Trong một vòng lặp while có 3 trường hợp có thể xảy ra:

- So sánh khớp, i và j cùng tăng => k giữ nguyên.
- So sánh không khớp, j = 0, i tăng => k tăng 1 giá trị.
- So sánh không khớp, j > 0, i không thay đổi, j đổi sang lps[j – 1] => k tăng ít nhất 1 giá trị.

Do đó trong mỗi vòng lặp, đều có ít nhất i hoặc k tăng, vậy số vòng lặp tối đa có thể có là 2n.

Với hàm tính mảng lps[], duyệt từ kí tự i = 1 đến m nên độ phức tạp tuyến tính O(m).

Từ đó tổng độ phức tạp của thuật toán KMP về mặt thời gian là O(n + m).

Nguồn tham khảo:

- https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm
- https://www.scaler.com/topics/data-structures/kmp-algorithm/
- https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/