ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

20120375	Cao Thị Phương Thảo
20120380	Nguyễn Phúc Thuần
20120439	Nguyễn Tạ Huy Hoàng
20120456	Lê Phước Đôn

BÁO CÁO ĐỒ ÁN GIỮA KÌ
CÁC THUẬT TOÁN SO KHỚP CHUỖI

Môn học: Nhập môn Phân tích độ phức tạp thuật toán

GVHD: Lê Phúc Lữ

Thành phố Hồ Chí Minh – 2022

Sơ lược bài toán

Bài toán so khớp chuỗi là một bài toán kinh điển trong việc xử lí dữ liệu, đặc biệt là các dữ liệu dạng xâu. Cho 2 xâu A và B. Hãy tìm các vị trí trên xâu/hoặc đếm số lần xâu pattern khớp với xâu con trong data/text. Ví dụ:

Pattern = “dolor”

Text = “Lorem ipsum dolor sit amet, consectetur adipiscing elit.”

Thì số lần match là 1, vị trí match là 12

Bài toán so khớp chuỗi mang ứng dụng rất lớn trong thực tế, như tra cứu từ điển, so khớp chuỗi AND trong y học, kiểm tra chính tả trong câu, sửa lỗi sai.

Dưới đây là 03 cách tiếp cận đến bài toán so khớp chuỗi (hay string matching) này:

I. Thuật toán Naive/Bruteforce

1. Ý tưởng thuật toán:

Brute Force là một thuật toán vét cạn, thuật toán này sẽ chạy tất cả các trường hợp có thể có để giải quyết một vấn đề nào đó (Bao gồm cả trường hợp đúng và các trường hợp sai hay còn gọi là trường hợp dư thừa).

Bằng cách dịch chuyển biến đếm j qua lần lượt các kí tự của file văn bản sau đó lấy m kí tự liên tiếp trong file văn bản (bắt đầu từ vị trí j) tạo thành một chuỗi phụ R rồi ta lấy R so sánh với S (S là chuỗi kí tự cần tìm) nếu giống nhau thì xuất ra kết quả. Thực hiện quá trình cho đến khi $j > n - m + 1$

2. Đặc điểm

- Thuật toán thực hiện tuần tự theo 1 chiều
- Không cần qua giai đoạn tiền xử lí
- Độ phức tạp cao O(mn)

3. Mô tả thuật toán

Input: Xâu mẫu (pattern) x = (x_0,x_1,…,x_m-1) độ dài m, Xâu văn bản (data) y = (y_0, y_1,…, y_n-1) độ dài n

Output: tất cả các vị trí của x trong y

Giải thuật:

- Lần lượt xét từng vị trí i trong xâu ký tự gốc từ (i = 0 đến n-m).
- So sánh <y_i,y_i+1,...y_i+m-1> với x<0…m-1> bằng cách xét từng cặp ký tự một.
- Trả về vị trí y nếu 2 xâu trên trùng khớp, ngược lại đi đến vị trí i kế tiếp nếu có xuất hiện 1 sự sai khác.

4. Ưu điểm và hạn chế

Ưu điểm:

- Các tiếp cận vét cạn là một cách đảm bảo đề tìm giải phép chính xác bằng cách liệt kê các giải pháp để giải quyết vấn đề.
- Phù hợp để giải quyết các vấn đề nhỏ và đơn giản hơn
- Thuật toán đơn giản

Hạn chế: tốc độ chậm (thực tế với sức mạnh của máy tính hiện tại thì dữ liệu và pattern rất lớn mới làm cho thuật toán này chạy đến vài giây hoặc hơn).

II. Thuật toán KMP

1. Ý tưởng thuật toán

Với bài toán trên, khi sử dụng cách giải đơn thuần (naive solution) thì ta phải duyệt lần lượt các ký tự trên T và so sánh các ký tự với P. Nếu ở vị trí i, P không xuất hiện trên T, thì ta tiếp tục duyệt lần lượt các ký tự của P trên T bắt đầu tại tại vị trí i + 1.

Tuy nhiên, việc làm như trên có thể mất thời gian và không hiệu quả vì khi P không xuất hiện tại vị trí i thì ta có thể kết luật nó cũng không thể xuất hiện ở vị trí i + 1 hoặc các vị trí tiếp theo.

Ví dụ:

T = “abcabcd”

P = “abcd”

P không thể xuất hiện trong T tại ví trí 0, nếu theo cách giải đơn thuần, ta tiếp tục xét vị trí 1, 2 điều này là không cần thiết vì P cũng không thể xuất hiện tại vị trí 1, 2 ($P[0] \neq T[1]$, $P[0] \neq T[2]$).

Thuật toán KMP cải tiến thuật toán vét cạn bằng việc tiền xử lý chuỗi mẫu P để lưu lại thông tin về chuỗi tiền tố hậu tố dài nhất và sử dụng thông tin đó để đẩy nhanh quá trình tìm kiếm P xuất hiện trong T.

2 Đặc điểm

- Cần có tiền xử lí
- Phù hợp với các chuỗi có nhiều tiền tố tương tự nhau
- Hiệu quả hơn với pattern dài hơn.
- Độ phức tạp O(m + n)

3. Mô tả thuật toán

Lúc đầu, ta tìm chuỗi tiền tố hậu tố dài nhất (longest prefix suffix – LPS) tại vị trí i sao cho:

- Tiền tố độ dài k của P và hậu tố độ dài k của chuỗi $[P_0P_1..P_i]$.
- Giá trị của k phải lớn nhất có thể.

Các giá trị trên được lưu trong mảng LSP[] trong đó LPS[i] là độ dài chuỗi tiền tố hậu tố lớn nhất của chuỗi $[P_0P_1..P_i]$.

Ví dụ:

P = “AAAA” ⇒ LPS[] = {0, 1, 2, 3}

P = “ABCDE” ⇒ LPS[] = {0, 0, 0, 0, 0}

P = “AABAACAABAA” ⇒ LPS[] = {0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5}

Ta có: LPS[0] = 0 vì với chuỗi 1 ký tự, chuỗi tiền tố hậu tố lớn nhất nhỏ hơn chính nó là chuỗi rỗng.

Với các phần tử LPS[i] (i > 0), ta xử lý như sau:

Trước hết ta đặt một biến tạm k = LPS[i-1]. Đó là độ dài của hậu tố đúng lớn nhất đang khớp ngay trước đó.

Chừng nào P[k] và P[i] chưa khớp nhau và k còn mang giá trị dương, ta rút ngắn hậu tố đúng cần đối chứng lại. Công thức rút ngắn sẽ là: k = LPS[k-1].

Do lệnh lặp ở trên nên sẽ xảy ra 2 khả năng:

- Nếu P[k] đã khớp với P[i], tức là ta đã tìm được hậu tố đúng thỏa mãn ở vị trí i. Ta tăng giá trị tmp thêm 1 đơn vị, và gán giá trị mới được xử lý này vào LPS[i].
- Nếu không, tức là k = 0, và không có hậu tố đúng thỏa mãn cho vị trí i. Như vậy rõ ràng LPS[i] = 0.

Mã giả:

```
lps[0] ← 0
k ← 0
i ← 1
while ( i < m )
     if (P[i] == P[k])
          k++
          lps[i] = k
          i++
     else
          if (k == 0)
               lps[i]=0
               i++
          else
               k = lps[k-1]
```

Tiếp theo, ta sẽ duyệt đoạn text T để tìm các vị trí P xuất hiện.

Ta đặt biến i = 0 và j = 0 rồi bắt đầu duyệt từ đầu đoạn T.

Khi P[j] và T[i] vẫn chưa khớp nhau và j còn mang giá trị dương, j = LPS[j-1].

Nghĩa là, ta đẩy xâu P sang phải cho tới khi nào tất cả các ký tự trước của P và T là trùng nhau, j là độ dài của hậu tố dài nhất của T{i} mà trùng khớp với tiền tố độ dài tương ứng của P.

Do đó lệnh lặp ở trên nên sẽ xảy ra 2 khả năng:

- Nếu P[j] đã khớp với T[i], tức là ta đã tìm được hậu tố đúng thỏa mãn ở vị trí i. Ta tăng giá trị j thêm 1 đơn vị.
    - Khi j = m, hậu tố độ dài |P| của T{i} đã khớp hoàn toàn với xâu P. Đặt lại $j = LPS[j-1]$ và tiếp tục quá trình tìm kiếm.
- Nếu không, tức là j = 0, và không có tiền tố nào của P khớp với hậu tố tương ứng cùng độ dài của xâu T{i}. Ta bỏ qua và duyệt tiếp tới i tiếp theo.

Mã giả:

```
i ← 0
j ← 0
result = []
```

```
      while ( i < n )
      if ( P[j] == T[i] )
            j++
            i++
            if ( i == m )
                  j = lps[j-1]
                  result.push(i-m)
            else if ( i < n && P[j] != T[i] )
                  if ( j == 0 )
                        i++
                  else
                        j = lps[k-1]
      return result
```

4. Ưu điểm và hạn chế

Ưu điểm:

- Tối ưu hóa cho các chuỗi chứa prefix dài.
- Giảm bớt các xử lí dư thừa.
- Tối ưu những lần so khớp sau bằng việc tái sử dụng bảng LPS.

Hạn chế:

- Tốn chi phí tiền xử lí
- Chưa thực sự hiệu quả nếu số lượng prefix ít

III. Thuật toán Rabin – Karp

1. Ý tưởng thuật toán

Thuật toán Rabin-Karp là một thuật toán được sử dụng để tìm kiếm hoặc so khớp chuỗi trong đoạn văn bản bằng cách sử dụng một hàm băm.

Không giống như thuật toán so khớp chuỗi thông thường, nó sẽ không đi qua mọi ký tự trong giai đoạn đầu mà nó sẽ lọc các ký tự không khớp và sau đó thực hiện so sánh.

Hàm băm là một công cụ để ánh xạ giá trị đầu vào lớn hơn với giá trị đầu ra nhỏ hơn. Giá trị đầu ra này được gọi là giá trị băm.

Hàm băm: $Hash(x[0,1,...,m-1]) = x[0]*2^{m-1} + x[1]*2^{m-2} + ... + x[m-1]*2^0$

Hàm băm tốt:

- Các thao tác cơ bản được thực hiện hiệu quả
- Khi băm hai xâu con khác nhau có cùng độ dài mm, xác suất hai giá trị băm giống nhau là nhỏ.

2. Đặc điểm

- Phụ thuộc vào việc lựa chọn hàm băm
- Tốc độ và các thao tác được cải thiện do chỉ tương tác trên giá trị băm
- Độ phức tạp: Trung bình O(m+n), Tốt nhất O(m+n), xấu nhất O(mn)

3. Mô tả thuật toán

Input:

- T[0…n-1]: là văn bản có n ký tự
- P[0…k-1]: là pattern có k ký tự với $k \leq n$
- Hash_T: là giá trị băm của chuỗi con tuần tự T[s…s+k-1] trong T với độ dịch chuyển là s, trong đó $0 \leq s \leq n-k$.
- Hash_P: là giá trị băm của P.

Output: số lần xuất hiện của P trong T

Khi này thuật toán so sánh lần lượt giá trị hash_T với hash_P với s chạy từ 0 đến n-k , bước tiếp theo của thuật toán sẽ xảy ra với hai trường hợp như sau:

- TH1: hash_T = hash_P, thực hiện phép so khớp chuỗi giữa T[s…s+k-1] và P[0…k-1]
- TH2: hash_T ≠ hash_P, nếu $s \leq k$, tính gán s = s+1 và tính tiếp giá trị băm hash_T.

Mã giả:

Algorithm 3: Rabin-Karp string search algorithm

```
Data: P: The pattern to look for
      T: The text to look in
      hashT, hashP: Hash structures for strings T, P
Result: Returns the number of occurrences of P in T
answer ← 0;
n ← length(T);
k ← length(P);
for i ← 0 to n − k do
    textHash ← hashT.getHash(i, i + k − 1);
    patternHash ← hashP.getHash(0, k − 1);
    if textHash = patternHash then
        valid ← true;
        for j ← 0 to k − 1 do
            if T[i + j] ≠ P[j] then
                valid ← false;
                break;
            end
        end
        if valid = true then
            answer ← answer + 1;
        end
    end
end
return answer;
```

4. Ưu điểm và hạn chế

Ưu điểm:

- Giảm thiểu số lượng phép gán và phép so sánh.
- Cải thiện thời gian thực hiện bài toán.
- Tối ưu những lần so khớp sau bằng việc tái sử dụng bảng băm.

Hạn chế:

- Phụ thuộc vào độ tốt của hàm băm

IV. Cài đặt thử nghiệm

1. Môi trường

Ngôn ngữ: C++

2. Cài đặt

2.1. Đánh giá thuật toán:

Đánh giá hiệu suất dựa trên việc đếm số phép gán và phép so sánh được thực hiện trong các thuật toán.

Đánh giá dựa trên các bộ test có dạng (LP, LT), với LP là độ dài pattern, LT là độ dài văn bản mẫu.

2.2. Các bộ test (LP, LT) đã áp dụng:

Văn bản trong các bộ test có text ngắn là prefix trong bộ test có text dài.

Văn bản trong các bộ test có pattern ngắn là prefix trong bộ test có pattern dài.

Việc test chưa xét đến việc tối ưu do tái sử dụng bảng băm (đối với Rabin Karp) hay bảng LPS (đối với KMP).

2.3. Cài đặt các thuật toán với kiểm thử:

2.3.1. Bruteforce:

(Xem trong file Brute-force.cpp)

2.3.2. KMP

(Xem trong file KMP.cpp)

Bao gồm cài đặt tiền xử lí LPS và thuật toán KMP sử dụng LPS.

Việc đếm số phép gán và so sánh tính trên cả quá trình tiền xử lí LPS và quá trình so khớp.

2.3.3. Rabin – Karp

(Xem trong file Rabin-Karp.cpp)

Bao gồm cài đặt tiền xử lí hash và so khớp sử dụng hash function.

Việc đếm số phép gán và so sánh tính trên cả quá trình tiền xử lí LPS và quá trình so khớp.

2.4. Thực nghiệm

Tiến hành thực hiện nhanh thuật toán bằng cách chạy file run.bat

Data (pattern và text) có thể tùy chỉnh trong các file thuộc thư mục data.

3. Kết quả thử nghiệm

Kết quả thực nghiệm của từng thuật toán được ghi vào các file sau trong thư mục result:

- Bruteforce.csv
- KMP.csv
- RabinKarp.csv

Có thể xem nhanh cả 3 kết quả bằng cách chạy file result.ipynb

V. Nhận xét

- Toàn bộ thực nghiệm và cài đặt có thể được xem tại: Repository này.
- Qua thực nghiệm có thể thấy:
    - Nếu không thông qua tối ưu bảng băm hay LPS, Rabin Karp hay không quá hiệu quả hơn quá nhiều so với Bruteforce. Tuy nhiên khi đã tối ưu, sự chênh lệch giữa các thuật toán là đáng kể.
    - Thuật toán KMP ổn định trong hầu hết trường hợp, trong khi Rabin không quá vượt trội so với Bruteforce.

VI. Phân công công việc

Họ tên	Công việc	Mức độ hoàn thành
Cao Thị Phương Thảo	Cài đặt thuật toán KMP	10/10
Nguyễn Phúc Thuần	Thực nghiệm, báo cáo	10/10
Nguyễn Tạ Huy Hoàng	Cài đặt thuật toán Bruteforce	10/10
Lê Phước Đôn	Cài đặt thuật toán Rabin Karp	10/10