**HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG**



**BÁO CÁO**

**CHUYÊN ĐỀ CÔNG NGHỆ PHẦN MỀM**

**Giảng viên: Nguyễn Duy Phương**

**Nhóm môn học: 01**

**Sinh viên: Nguyễn Đình Thắng**

**Mã sinh viên B16DCCN319**

MỤC LỤC

[I. Các thuật toán đối sánh mẫu 3](#_Toc45391597)

[II. Phân loại các thuật toán đối sánh mẫu 4](#_Toc45391598)

[III. Trình bày các thuật toán tìm kiếm mẫu từ trái qua phải 5](#_Toc45391599)

[1. Brute Force algorithm 5](#_Toc45391600)

[2. Search with an automaton 8](#_Toc45391601)

[3. Karp-Rabin algorithm 12](#_Toc45391602)

[4. Shift Or algorithm 15](#_Toc45391603)

[5. Morris - Pratt algorithm 18](#_Toc45391604)

[6. Knuth-Morris-Pratt algorithm: 21](#_Toc45391605)

[7. Apostolico-Crochemore algorithm 24](#_Toc45391606)

[8. Not so naive algorithm 27](#_Toc45391607)

[IV. Trình bày các thuật toán tìm kiếm mẫu từ phải qua trái 31](#_Toc45391608)

[1. Boyer Moore algorithm 31](#_Toc45391609)

[2. Zhu Kataoka algorithm 35](#_Toc45391610)

[3. Berry-Ravindran algorithm 38](#_Toc45391611)

[4. Turbo-BM algorithm 40](#_Toc45391612)

[5. Tuned boyer-moore algorithm 44](#_Toc45391613)

[6. Apostolico-giancarlo algorithm 47](#_Toc45391614)

[7. Quick Search algorithm 51](#_Toc45391615)

[V. Trình bày các thuật toán tìm kiếm mẫu từ vị trí cụ thể 53](#_Toc45391616)

[1. Colussi algorithm 53](#_Toc45391617)

[2. Skip Search Algorithm 58](#_Toc45391618)

[3. Alpha Skip Search Algorithm 60](#_Toc45391619)

[VI. Trình bày các thuật toán tìm kiếm mẫu từ vị trí bất kỳ 63](#_Toc45391620)

[1. Horspool algorithm 63](#_Toc45391621)

[2. Smith algorithm 66](#_Toc45391622)

[3. Raita algorithm 69](#_Toc45391623)

# I. Các thuật toán đối sánh mẫu

Có 35 thuật toán đối sánh mẫu:

1. Thuật toán Knuth-Morris-Partt

2. Thuật toán Karp-Rabin

3. Thuật toán Shift or

4. Thuật toán Morris-Partt

5. Thuật toán Automat hữu hạn

6. Thuật toán Simon

7. Thuật toán Colussi

8. Thuật toán Galil-Giancarlo

9. Apostolico-Crochemore algorithm

10. Not So Naive algorithm

11. Turbo BM algorithm

12. Apostolico-Giancarlo algorithm

13. Reverse Colussi algorithm

14. Boyer-Moore algorithm

15. Horspool algorithm

16. Quick Search algorithm

17. Tuned Boyer-Moore algorithm

18. Zhu-Takaoka algorithm

19. Berry-Ravindran algorithm

20. Smith algorithm

21. Raita algorithm

22. Reverse Factor algorithm

23. Turbo Reverse Factor algorithm

24. Forward Dawg Matching algorithm

25. Backward Nondeterministic Dawg 2.38. Matching algorithm

26. Backward Oracle Matching algorithm

27 Galil-Seiferas algorithm

28. Two Way algorithm

29. String Matching on Ordered

30. Alphabets algorithm

31. Optimal Mismatch algorithm

32. Maximal Shift algorithm

33. Skip Search algorithm

34. KMP Skip Search algorithm

35. Alpha Skip Search algorithm

# II. Phân loại các thuật toán đối sánh mẫu

Phân loại các thuật toán:

**Nhóm 1: các thuật toán duyệt từ trái sang phải:**

1. Brute force algorithm

2. Search with an automaton

3. Karp-Rabin algorithm

4. Shift Or algorithm

5. Morris-Pratt algorithm

6. Knuth-Morris-Pratt algorithm

7. Apostolico-Crochemore algorithm

8. Not so naive algorithm

**Nhóm 2: các thuật toán duyệt từ phải sang trái:**

1. Boyer-Moore algorithm

2. Turbo-BM algorithm

3. Tuned boyer-moore algorithm

4. Apostolico-giancarlo algorithm

5. Zhu-Takaoka algorithm

6. Berry-Ravindran algorithm

7. Quick Search algorithm

**Nhóm 3: các thuật toán duyệt tại vị trí cụ thể:**

1. Colussi algorithm

2. Skip search algorithm

3. Alpha skip search

**Nhóm 4: các thuật toán duyệt tại vị trí bất kỳ:**

1. Horspool algorithm

2. Smith algorithm

3. Raita algorithm

# III. Trình bày các thuật toán tìm kiếm mẫu từ trái qua phải

## 1. Brute Force algorithm

*1.1. Đặc điểm*

- Không có pha chuẩn bị

- Bộ nhớ cần dùng cố định

- Luôn luôn dịch 1 bước sang phải

- Việc so sánh có thể phải dùng trong các trường hợp

- Độ phức tạp pha thực thi là O(m x n)

*1.2. Trình bày thuật toán*

Thuật toán Brute Force bao gồm kiểm tra, tất cả các vị trí trong đoạn văn bản giữa 0 và n-m, không cần quan tâm liệu mẫu này có tồn tại ở vị trí đó hay không. Sau đó, sau mỗi lần kiểm tra mẫu sẽ dịch sang phải một vị trí.

*1.3. Kiểm thử thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Quá trình tìm kiếm:

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4

G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4 5 6 7 8

G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1 2

G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1 2

G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1 2

G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1

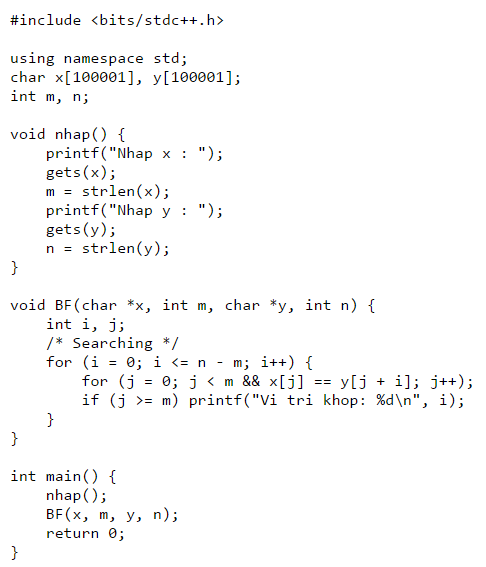
G C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

*1.4. Chương trình chạy*



## 2. Search with an automaton

*2.1. Đặc điểm*

- Yêu cầu xây dựng automaton đơn định (DFA).

- Pha xử lý có độ phức tạp tính toán là O(n∂).

- Quá trình tìm kiếm có độ phức tạp là O(n). Trường hợp DFA đươc xây dựng bằng cây cân bằng thì độ phức tạp là O(n\*log(∂)).

*2.2. Trình bày thuật toán*

Trong thuật toán này, quá trình tìm kiếm được đưa về một quá trình biến đổi trạng thái automat. Hệ thống automat trong thuật toán DFA sẽ được xây dựng dựa trên xâu mẫu. Mỗi trạng thái (nút) của automat sẽ đại diện cho số ký tự đang khớp của mẫu với văn bản. Các ký tự của văn bản sẽ làm thay đổi các trạng thái. Và khi đạt được trạng cuối cùng có nghĩa là đã tìm được một vị trí xuất hiện ở mẫu.

Thuật toán này có phần giống thuật toán Knuth-Morris-Pratt trong việc nhảy về trạng thái trước khi gặp một ký tự không khớp, nhưng thuật toán DFA có sự đánh giá chính xác hơn vì việc xác định vị trí nhảy về dựa trên ký tự không khớp của văn bản (trong khi thuật toán KMP lùi về chỉ dựa trên vị trí không khớp).

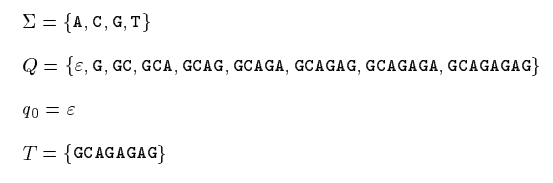
Việc xây dựng hệ automat khá đơn giản khi được cài đặt trên ma trận kề. Khi đó thuật toán có thời gian xử lý là O(n) và thời gian và bộ nhớ để tạo ra hệ automat là O(m\*d) (tùy cách cài đặt). Nhưng ta nhận thấy rằng trong DFA chỉ có nhiều nhất m cung thuật và m cung nghịch, vì vậy việc lưu trữ các cung không cần thiết phải lưu trên ma trận kề mà có thể dùng cấu trúc danh sách kề Forward Star để lưu trữ. Như vậy thời gian chuẩn bị và lượng bộ nhớ chỉ là O(m). Tuy nhiên thời gian tìm kiếm có thể tăng lên một chút so với cách lưu ma trận kề.

*2.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y =“GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lý xây đựng DFA:



Pha tìm kiếm:

Current state is: 0

**G** C A T C G C A G A G A G T A T A C A G T A C G

1

G **C** A T C G C A G A G A G T A T A C A G T A C G

2

G C **A** T C G C A G A G A G T A T A C A G T A C G

3

G C A **T** C G C A G A G A G T A T A C A G T A C G

0

G C A T **C** G C A G A G A G T A T A C A G T A C G

0

G C A T C **G** C A G A G A G T A T A C A G T A C G

1

G C A T C G **C** A G A G A G T A T A C A G T A C G

2

G C A T C G C **A** G A G A G T A T A C A G T A C G

3

G C A T C G C A **G** A G A G T A T A C A G T A C G

4

G C A T C G C A G **A** G A G T A T A C A G T A C G

5

G C A T C G C A G A **G** A G T A T A C A G T A C G

6

G C A T C G C A G A G **A** G T A T A C A G T A C G

7

G C A T C G C A G A G A **G** T A T A C A G T A C G

8

G C A T C G C A G A G A G **T** A T A C A G T A C G

0

G C A T C G C A G A G A G T **A** T A C A G T A C G

0

G C A T C G C A G A G A G T A **T** A C A G T A C G

0

G C A T C G C A G A G A G T A T **A** C A G T A C G

0

G C A T C G C A G A G A G T A T A **C** A G T A C G

0

G C A T C G C A G A G A G T A T A C **A** G T A C G

0

G C A T C G C A G A G A G T A T A C A **G** T A C G

1

G C A T C G C A G A G A G T A T A C A G **T** A C G

0

G C A T C G C A G A G A G T A T A C A G T **A** C G

0

G C A T C G C A G A G A G T A T A C A G T A **C** G

0

G C A T C G C A G A G A G T A T A C A G T A C **G**

1

*2.4. Chương trình chạy*

|  |
| --- |
| **package** traisangphai.Automaton;  **public class** Demo\_Automaton {   **public static void** main(String[] args) {  **char**[] pat = **"ABCABAB"**.toCharArray();  **char**[] txt = **"ABABDABACDABABCABAB"**.toCharArray();  *searchAtm*(pat, txt);  }   **static int** getNextState(**char**[] pat, **int** M, **int** state, **int** x) {  **if** (state < M && x == pat[state])  **return** state + 1;  **int** ns, i;  **for** (ns = state; ns > 0; ns--) {  **if** (pat[ns - 1] == x) {  **for** (i = 0; i < ns - 1; i++)  **if** (pat[i] != pat[state - ns + 1 + i])  **break**;  **if** (i == ns - 1)  **return** ns;  }  }  **return** 0;  }   **static void** compute(**char**[] pat, **int** M, **int**[][] TF) {  **int** state, x;  **for** (state = 0; state <= M; ++state)  **for** (x = 0; x < 256; ++x)  TF[state][x] = *getNextState*(pat, M, state, x);  }   **static void** searchAtm(**char**[] pat, **char**[] txt) {  **int** M = pat.**length**;  **int** N = txt.**length**;   **int**[][] TF = **new int**[M + 1][256];   *compute*(pat, M, TF);   **int** i, state = 0;  **for** (i = 0; i < N; i++) {  state = TF[state][txt[i]];  **if** (state == M)  System.***out***.println(**"Pattern found "** + **"at index "** + (i - M + 1));  }  } } |

## 3. Karp-Rabin algorithm

*3.1. Đặc điểm*

- Thực chất là sử dùng hàm băm để biến các ký tự thành số để tăng tốc độ so sánh

- Pha chuẩn bị có độ phức tạp thuật toán và không gian lưu trữ O(m).

- Chương trình chạy có độ phức tạp O(m+n).

*3.2. Trình bày thuật toán*

Hàm băm cung cấp phương thức đơn giản để tránh những con số phức tạp trong việc so sánh những kí tự trong hầu hết các trường hợp thực tế. Thay cho việc kiểm tra từng vị trí trong văn bản nếu như có mẫu xuất hiện, nó chỉ phải kiểm tra những đoạn “gần giống” xâu mẫu. Để kiểm tra sự giống nhau giữa 2 từ sử dụng hàm băm. Giúp cho việc đối chiếu xâu, hàm băm hash:

- Có khả năng tính toán được.

- Đánh giá xâu mức cao.

- Hash(y[j+1…j+m]) được tính toán dễ hơn dựa trên hash(y[j…j+m-1]) và hash(y[j+m]):

hash(y[j+1 .. j+m])= rehash(y[j], y[j+m], hash(y[j .. j+m-1]).

Với từ w có độ dài m có hash(w) là:

hash(w[0 .. m-1])=(w[0]\*2^(m-1)+ w[1]\*2^(m-2)+···+ w[m-1]\*2^0) mod q

Với q là một số lớn.

Sau đó rehash(a,b,h)= ((h-a\*2^(m-1))\*2+b) mod q.

Pha chuẩn bị của Karp - Rabin có hàm hash(x) có thể tính toán được. Nó được dùng lại không gian nhớ và có độ phức tạp O(m). Trong quá trình thực thi nó so sánh hash(x) với hash([j..j+m-1]) với 0<= j<=n-m. Nếu so sánh đúng, nó phải kiểm tra lại xem các kí tự trong x và y có đúng bằng nhau hay không x=y[j…j+m-1].

*3.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lí: Tính được hash(X) = 17597

Pha tìm kiếm thực hiện như sau:

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[0 .. 7]) = 17819

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[1 .. 8]) = 17533

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[2 .. 9]) = 17979

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[3 .. 10]) = 19389

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[4 .. 11]) = 17339

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4 5 6 7 8

G C A G A G A G

hash(y[5 .. 12]) = 17597

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[6 .. 13]) = 17102

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[7 .. 14]) = 17117

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[8 .. 15]) = 17678

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[9 .. 16]) = 17245

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[10 .. 17]) = 17917

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[11 .. 18]) = 17723

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[12 .. 19]) = 18877

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[13 .. 20]) = 19662

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[14 .. 21]) = 17885

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[15 .. 22]) = 19197

G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

hash(y[16 .. 23]) = 16961

*3.4. Chương trình chạy*

|  |
| --- |
| **package** traisangphai.Rabin\_Karp;  **public class** Demo\_Rabin\_Karp {  **static final int *BASE*** = 3;  **static final int *PRIME*** = 517;  **static long** *valueCurrent*;  **static int** *hashCurrent*;  **static long** *valueOrigin*;  **static int** *hashOrigin*;  **static** String *pattern* = **"ABABCABAB"**;  **static** String *text* = **"ABABDABACDABABCABAB"**;   **public static void** main(String[] args) {   *hashOrigin* = *initHashP*(*pattern*);  *hashCurrent* = *initHashT*(*text*.substring(0, *pattern*.length()));   System.***out***.println(*pattern* + **" : "** + *hashOrigin*+**":"**+*valueOrigin*);  System.***out***.println(*text*.substring(0, *pattern*.length()) + **" : "** + *hashCurrent*+**":"**+*valueCurrent*);    **for** (**int** i = 1; i < *text*.length() - *pattern*.length() + 1; i++) {  *hash*(*text*.charAt(i - 1), *text*.charAt(i + *pattern*.length() - 1));  System.***out***.println(*text*.substring(i, i + *pattern*.length()) + **" : "** + *hashCurrent*);  }  }   **private static int** initHashP(String init) {  **char**[] temp = init.toCharArray();  **int** mul = 1;  **for** (**int** i = temp.**length** - 1; i >= 0; i--) {  *valueOrigin* += temp[i] \* mul;  mul \*= ***BASE***;  }  *hashOrigin* = (**int**) (*valueOrigin* % ***PRIME***);  **return** *hashOrigin*;  }   **private static int** initHashT(String init) {  **char**[] temp = init.toCharArray();  **int** mul = 1;  **for** (**int** i = temp.**length** - 1; i >= 0; i--) {  *valueCurrent* += temp[i] \* mul;  mul \*= ***BASE***;  }  *hashCurrent* = (**int**) (*valueCurrent* % ***PRIME***);  **return** *hashCurrent*;  }   **private static void** hash(**char** old, **char** next) {  *valueCurrent* = (**int**) ((*valueCurrent* - old \* Math.*pow*(***BASE***, *pattern*.length()-1)) \* ***BASE*** + next);  *hashCurrent* = (**int**) (*valueCurrent* % ***PRIME***);  } } |

## 4. Shift Or algorithm

*4.1. Đặc điểm*

- Thuật toán sử dụng kỹ thuật thao tác với bit.

- Thực tế là so sánh từng ký tự nhưng chuyển sang dạng thao tác với bit để tăng tốc độ

- Thuật toán đạt hiệu quả nếu độ dài mẫu không vượt quá kích thước bộ nhớ máy.

- Pha tiền xử lý có độ phức tạp O(m +∂).

- Pha tìm kiếm có độ phức tạp O(n). Nó còn phụ thuộc vào kích thước bảng chữ cái và đồ dài mẫu.

*4.2. Trình bày thuật toán*

Thuật toán Shift Or sử dụng công nghệ bitwise. Ta có R là mảng bit độ dài là m. Vector Rj là giá trị của mảng R sau vị trí y[j] đã xử lý. Nó chứa các thông tin về tất cả các tiền tố x đã phù hợp tại vị trí j hay chưa.

For 0 < i <= m-1:

Vector Rj+1 có thể xác định sau khi có Rj và nếu Rj+1[m-1] = 0. Có nghĩa là ta đã hoàn thành so sánh và tìm được vị trí x xuất hiện trong y. Với mỗi kí tự c ta có Sc là mảng bít độ dài m:

for 0 <= i <m-1, Sc[i] =0 nếu và chỉ nếu x[i] = c.

Mảng Sc xác định vị trí của kí tự c trong mẫu x. mỗi Sc sẽ được chuẩn bị trước. Ta có thể xác định nhanh Rj+1:

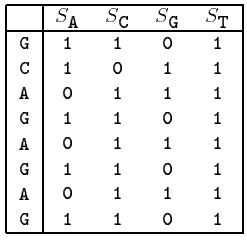
Rj+1=SHIFT(Rj) OR Sy[j+1]

*4.3. Kiểm nghiệm thuật toán*

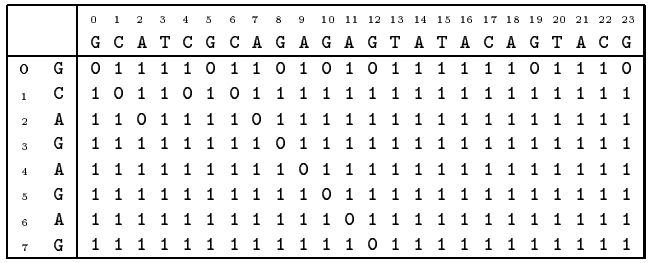
X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lý xác định:



Pha tìm kiếm thực hiện qua bảng sau:



Tại vị trí 12 ta có R12 [7] = 0. Nghĩa là có x xuất hiện trong y tại vị trí 12-8+1 = 5;

*4.4. Chương trình chạy*

|  |
| --- |
| **package** traisangphai.Shift\_Or;  **public class** Demo\_Shift\_Or {  **public static void** main(String[] args) {  **char**[] pat = **"ABCABAB"**.toCharArray();  **char**[] txt = **"ABABDABACDABABCABAB"**.toCharArray();  *searchSO*(pat, txt);  }   **static int** preSo(**char**[] x, **int** m, **long**[] S) {  **long** j, lim;  **int** i;  **for** (i = 0; i < 256; ++i)  S[i] = ~0;  **for** (lim = i = 0, j = 1; i < m; ++i, j <<= 1) {  S[x[i]] &= ~j;  lim |= j;  }  lim = ~(lim >> 1);  **return** (**int**) lim;  }   **static void** searchSO(**char**[] x, **char**[] y) {  **int** m = x.**length**;  **int** n = y.**length**;   **long** lim, state;  **long**[] S = **new long**[256];  **int** j;   lim = *preSo*(x, m, S);   **for** (state = ~0, j = 0; j < n; ++j) {  state = (state << 1) | S[y[j]];  **if** (state < lim)  System.***out***.println(j - m + 1);  }  } } |

## 5. Morris - Pratt algorithm

*5.1. Đặc điểm*

- Thực hiện so sánh từ trái qua phải.

- Pha tiền xử lí có độ phức tạp O(m).

- Pha thực thi có độ phức tạp O(m+n) phụ thuộc vào kích thước của mảng chữ cái.

- Thực hiện nhiều nhất 2n-1 các so sánh kí tự trong pha thực thi.

- Từ pattern xây dựng 1 mpNext để xác định độ dài lớn nhất hậu tố mà trùng với tiền tố của pattern, từ đó xác định bước nhảy khi bị mismatch

*5.2. Trình bày thuật toán*

Thiết kế của thuật toán Morris Partt là thuật toán Brute Force với các ràng buộc chặt chẽ hơn, sử dụng những thông tin thu thập được trong quá trình quét văn bản.

Cùng nhìn lại thuật toán Brute Force, chúng có thể tăng độ dài bước dịch và đồng thời nhớ một vài phần của văn bản, phần mà nó chiếu với mẫu. Nó sẽ hạn chế số lần so sánh giữa các kí tự trong xâu mẫu và xâu văn bản và tăng tốc độ thực hiện.

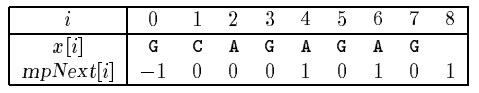
Quan tâm tới bên trái vị trí j trong y khi cửa sổ đang là vị trí của đoạn y[j…j+m-1]. Giả sử rằng vị trí đầu tiên mà chúng không giống nhau là x[i] và y[i+j] với 0<j<m và x[0..i-1] = y[j .. i+j-1] = u và a = x[i] # y[i+j] = b. Khi dịch, nó là cơ sở để chắc chắn rằng tiền tố v của mẫu trùng với một hậu tố của u trong văn bản. Độ dài lớn nhất của v được gọi là biên của u. Mảng mpNext[i] là độ dài lớn nhất của biên của x[0…i-1] với 0<i<=m. Bên cạnh đó sau mỗi bước dịch, ta tiếp tục so sánh kí tự c = x[mpNext[i]] và y[i+j]= b trong trường hợp không tồn tại kí tự nào của x trong y và tránh bị backtrack. Giá trị của mpNext[0] được thiết lập là -1.

*5.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lí chuẩn bị bảng mpNext:



Pha tìm kiếm thực hiện :

G C A **T** C G C A G A G A G T A T A C A G T A C G

1 2 3 4

G C A **G** A G A G

Shift by: 3 (i-mpNext[i]=3-0)

G C A **T** C G C A G A G A G T A T A C A G T A C G

1

**G** C A G A G A G

Shift by: 1 (i-mpNext[i]=0- -1)

G C A T **C** G C A G A G A G T A T A C A G T A C G

1

**G** C A G A G A G

Shift by: 1 (i-mpNext[i]=0- -1)

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4 5 6 7 8

G C A G A G A G

Shift by: 7 (i-mpNext[i]=8-1)

G C A T C G C A G A G A G **T** A T A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (i-mpNext[i]=1-0)

G C A T C G C A G A G A G **T** A T A C A G T A C G

1

**G** C A G A G A G

Shift by: 1 (i-mpNext[i]=0- -1)

G C A T C G C A G A G A G T **A** T A C A G T A C G

1

**G** C A G A G A G

Shift by: 1 (i-mpNext[i]=0- -1)

G C A T C G C A G A G A G T A **T** A C A G T A C G

1

**G** C A G A G A G

Shift by: 1 (i-mpNext[i]=0- -1)

G C A T C G C A G A G A G T A T **A** C A G T A C G

1

**G** C A G A G A G

Shift by: 1 (i-mpNext[i]=0- -1)

*5.4. Chương trình chạy*

|  |
| --- |
| **package** traisangphai.Morris\_Pratt;  **public class** Demo\_Morris\_Pratt {   **public static void** main(String[] args) {  String pattern = **"ABABCABAB"**;  String text = **"ABABDABACDABABCABAB"**;  *searchMP*(pattern, text);  }   **static void** searchMP(String pattern, String text) {  **int**[] mpNext = *preMP*(pattern);  **for** (**int** i : mpNext) System.***out***.print(i + **" "**);  **int** i = 0;  **int** j = 0;  **while** (i < text.length()) {  **if** (text.charAt(i) == pattern.charAt(j)) {  i++;  j++;  }  **if** (j == pattern.length()) {  System.***out***.println(**"index : "** + (i - j));  j = mpNext[j - 1];  } **else if** (i < text.length() && text.charAt(i) != pattern.charAt(j)) {  **if** (j != 0) j = mpNext[j - 1];  **else** {  i++;  }  }  }  }   **static int**[] preMP(String pattern) {  **int**[] mpNext = **new int**[pattern.length()];  **int** i = 0;  **int** len = i;  mpNext[i] = len;  i++;  **char**[] X = pattern.toCharArray();  **while** (i < mpNext.**length**) {  **if** (X[i] == X[len]) {  len++;  mpNext[i] = len;  i++;  } **else** {  **if** (len != 0) {  len = mpNext[len - 1];  } **else** {  mpNext[len] = 0;  i++;  }  }  }  **return** mpNext;  } } |

## 6. Knuth-Morris-Pratt algorithm:

*6.1. Đặc điểm*

- Thực hiện so sánh từ trái qua phải

- Là phiên bản nâng cấp của thuật toán MP

- Tự sử dụng bảng mpNext với pattern để cái tiến bảng mpNext

- Pha tiền xử lí có độ phức tạp thuật toán là O(m)

- Pha thực thi có độ phức tạp O(m+n) phụ thuộc vào kích thước của mảng chữ cái.

- Thực hiện nhiều nhất 2n-1 các so sánh kí tự trong pha thực thi

- Độ trễ biên được tính bằng logɸ(m) với ɸ được tính là:

*6.2. Trình bày thuật toán*

Knuth Morris Partt là một phát triển chặt chẽ hơn của thuật toán Morris partt. Hãy nhìn lại Morris Partt, nó có thể tăng được số bước dịch chuyển.

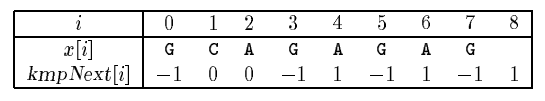
Quan tâm tới bên trái vị trí j trong y khi cửa sổ đang là vị trí của đoạn y[j…j+m-1]. Giả sử rằng vị trí đầu tiên mà chúng không giống nhau là x[i] và y[i+j] với 0<j<m. và x[0..i-1] = y[j .. i+j-1] = u và a = x[i] # y[i+j]=b . Khi dịch, nó là cơ sở để chắc chắn rằng tiền tố v của mẫu trùng với một hậu tố của u trong văn bản. Tuy nhiên, nếu chúng ta muốn tránh những mẫu không khớp ngay lập tức, kí tự ngay sau tiền tố v trong mẫu phải khác a. Độ dài lớn nhất của tiền tố v được gọi là nhãn biên của u (nó là kí tự xuất hiện ở 2 đầu của u theo sau bởi kí tự khác nhau trong x). KmpNext[i] là độ dài dài nhất của biên x[0…i-1] nếu theo sau là kí tự c khác với x[i] và bằng -1 nếu nhãn biên tồn tại với 0 < i <= m. Bên cạnh đó sau mỗi bước dịch, ta tiếp tục so sánh kí tự c = x[mpNext[i]] và y[i+j] = b trong trường hợp không tồn tại kí tự nào của x trong y và tránh bị backtrack. Giá trị của kmpNext[0] được thiết lập bằng -1 .

*6.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lý xác định bảng kmpNext:



Pha tìm kiếm thực hiện như sau :

G C A **T** C G C A G A G A G T A T A C A G T A C G

1 2 3 4

G C A **G** A G A G

Shift by: 4 (i-kmpNext[i]=3- -1)

G C A T **C** G C A G A G A G T A T A C A G T A C G

1

**G** C A G A G A G

Shift by: 1 (i-kmpNext[i]=0- -1)

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4 5 6 7 8

G C A G A G A G

Shift by: 7 (i-kmpNext[i]=8-1)

G C A T C G C A G A G A G **T** A T A C A G T A C G

2

G **C** A G A G A G

Shift by: 1 (i-kmpNext[i]=1-0)

G C A T C G C A G A G A G **T** A T A C A G T A C G

1

**G** C A G A G A G

Shift by: 1 (i-kmpNext[i]=0- -1)

G C A T C G C A G A G A G T **A** T A C A G T A C G

1

**G** C A G A G A G

Shift by: 1 (i-kmpNext[i]=0- -1)

G C A T C G C A G A G A G T A **T** A C A G T A C G

1

**G** C A G A G A G

Shift by: 1 (i-kmpNext[i]=0- -1)

G C A T C G C A G A G A G T A T **A** C A G T A C G

1

**G** C A G A G A G

Shift by: 1 (i-kmpNext[i]=0- -1)

*6.4. Chương trình chạy*

|  |
| --- |
| **package** traisangphai.Knuth\_Morris\_Pratt;  **public class** Demo\_Knuth\_Morris\_Pratt {   **public static void** main(String[] args) {  String pattern = **"ABCABAB"**;  String text = **"GCATCGCAGAGAGTATACAGTACG"**;  *searchKMP*(pattern, text);  }   **static void** searchKMP(String pattern, String text) {  **int**[] kmpNext = *preKMP*(pattern);  **for** (**int** i : kmpNext) System.***out***.print(i + **" "**);  **int** i = 0;  **int** j = 0;  **while** (i < text.length()) {  **if** (text.charAt(i) == pattern.charAt(j)) {  i++;  j++;  }  **if** (j == pattern.length()) {  System.***out***.println(**"index : "** + (i - j));  j = kmpNext[j - 1];  } **else if** (i < text.length() && text.charAt(i) != pattern.charAt(j)) {  **if** (j != 0) j = kmpNext[j - 1];  **else** {  i++;  }  }  }  }   **static int**[] preKMP(String pattern) {  **int**[] kmpNext = **new int**[pattern.length()];  **int** i = 0;  **int** len = i;  kmpNext[i] = len;  i++;  **char**[] X = pattern.toCharArray();  **while** (i < kmpNext.**length**) {  **if** (X[i] == X[len]) {  len++;  kmpNext[i] = len;  **if** (i < X.**length** - 1 && X[i + 1] == X[len]) {  kmpNext[i] = kmpNext[len];  }  i++;  } **else** {  **if** (len != 0) {  len = kmpNext[len - 1];  } **else** {  kmpNext[len] = 0;  i++;  }  }  }  **return** kmpNext;  } } |

## 7. Apostolico-Crochemore algorithm

*7.1. Đặc điểm*

- Sử dụng phép dịch của thuật toán KMP

- Là thuật toán so sánh xâu

- Sử dụng khi pattern có nhiều kí tự đầu giống nhau 🡪 giảm số lượt so sánh

- Pha tiền xử lý có độ phức tạp O(m).

- Pha tìm kiếm có độ phức tạp O(n).

- Trường hợp xấu nhất thuật toán thực hiện 3/2 n lần so sánh kí tự.

*7.2. Trình bày thuật toán*

Thuật toán Apostolico – Crochemor sử dụng bảng kmpNext từ thuật toán KMP. Định nghĩa l là số lượng kí tự giống nhau đầu tiên trong xâu x. Trường hợp x = c^m thì l = 0. Tổng quát x = a^l \* bu với a, b là các kí tự, u là một xâu con. a != b trong mỗi một phép thứ ta sẽ so sánh mẫu lần lượt các vị trí : l, l+1,… , m-2, m-1, 0, 1,.., l-1.

Trong pha tìm kiếm ta quan tâm tới bộ 3 số (i, j, k):

Khi của sổ đặt ởvị trí khớp với y[j… j+m-1]; 0 <=k <= l và x[0..k-1] = y[j.. j+k-1]; l<=i < m và x[l.. i-1] = y[j+l.. i+j-1]; trạng thái khới tạo bạn đâu với bộ (l,0,0). Chúng ta sẽ giải thích làm cách nào để xác định được bộ (i, j, k) tiếp theo.

Có 3 trường hợp tùy thuộc vào các giá trị của i như sau :

- Trường hợp i = l : nếu x[i] = y[i+j] thì bộ kế tiếp (i+1,j,k) nếu x[i] != y[i+j] thì bộ kế tiếp là (l,j+1,max{0,k-1}).

- Trường hớp l < i < m: nếu x[i] = y[i+j] thì bộ kế tiếp là (i+1,j,k) nếu x[i] != y[i+j] thì sẽ có 2 trường hợp xẩy ra phụ thuộc vào giá trị kmpNext[i]:

Nếu kmpNext[i] <= l thì bộ tiếp theo : ( , i+jkmpNext[i], max{0, kmpNext[i]})

Nếu kmpNext[i] > l thì bộ tiếp theo : (kmpNext[i], i+j-kmpNext[i], )

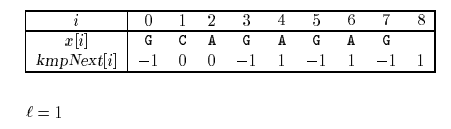
- Trường hợp l = m : nếu k < l và x[k]=y[j+k] thì bộ tiếp theo là (i, j, k+1).

*7.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lý xác định :



Pha tìm kiếm thực hiện như sau :

G C A **T** C G C A G A G A G T A T A C A G T A C G

1 2 3

G C A **G** A G A G

Shift by: 4 (i-kmpNext[i]=3- -1)

G C A T C **G** C A G A G A G T A T A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (i-kmpNext[i]=1-0)

G C A T C G C A G A G A G T A T A C A G T A C G

8 1 2 3 4 5 6 7

G C A G A G A G

Shift by: 7 (i-kmpNext[i]=8-1)

G C A T C G C A G A G A G **T** A T A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (i-kmpNext[i]=1-0)

G C A T C G C A G A G A G T **A** T A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (i-kmpNext[i]=1-0)

G C A T C G C A G A G A G T A **T** A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (i-kmpNext[i]=1-0)

G C A T C G C A G A G A G T A T **A** C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (i-kmpNext[i]=1-0)

G C A T C G C A G A G A G T A T A C A G **T** A C G

1 2 3 4

G C A G **A** G A G

Shift by: 3 (i-kmpNext[i]=4-1)

*7.4. Chương trình chạy*

|  |
| --- |
| **package** traisangphai.Apostolico\_Crochemore;  **public class** Demo\_Apostolico\_Crochemore {  **public static void** main(String[] args) {  **char**[] pat = **"ABCABAB"**.toCharArray();  **char**[] txt = **"ABABDABACDABABCABAB"**.toCharArray();  *AC*(pat, txt);  }   **static void** AC(**char**[] x, **char**[] y) {  **int** m = x.**length**;  **int** n = y.**length**;  **int** i, j, k, ell;  **int**[] kmpNext = **new int**[m+1];  *preKmp*(x, m, kmpNext);  **for**(**int** h : kmpNext) System.***out***.print(h+**" "**);   **for** (ell = 1; x[ell - 1] == x[ell]; ) ell++;  **if** (ell == m) ell = 0;    i = ell;  j = k = 0;  **while** (j <= n - m) {  **while** (i < m && x[i] == y[i + j])  ++i;  **if** (i >= m) {  **while** (k < ell && x[k] == y[j + k])  ++k;  **if** (k >= ell)  System.***out***.println(**"index : "** + j);  }  j += (i - kmpNext[i]);  **if** (i == ell)  k = Math.*max*(0, k - 1);  **else if** (kmpNext[i] <= ell) {  k = Math.*max*(0, kmpNext[i]);  i = ell;  } **else** {  k = ell;  i = kmpNext[i];  }  }  }   **static void** preKmp(**char**[] x, **int** m, **int**[] kmpNext) {  **int** i, j;  i = 0;  j = kmpNext[0] = -1;  **while** (i < m) {  **while** (j > -1 && x[i] != x[j])  j = kmpNext[j];  i++;  j++;  **if** (i < m && x[i] == x[j])  kmpNext[i] = kmpNext[j];  **else** kmpNext[i] = j;  }  } } |

## 8. Not so naive algorithm

*8.1. Đặc điểm*

- Pha chuẩn bị có độ phức tạp thuật toán không thay đổi.

- So sánh 2 ký tự đầu của pattern, từ đó thực hiện so sánh để xác định bước nhảy nhằm tăng tốc độ dịch

- Cộng thêm phức tạp không gian lưu trữ.

- Độ phức tạp pha thực thi là O(mxn).

*8.2. Trình bày thuật toán*

Trong quá trình thực thi của Not So Naïve các kí tự so sánh với các vị trí của mẫu theo thứ tự 1,2,3…m-2, m-1, 0. Với mỗi trường hợp, trên cửa sổ dịch chuyển là các vị trí của xâu văn bản y[j…j+m-1]: nếu x[0] = x[1] và x[1] != y[j+1] hoặc nếu   
x[1] != x[0] và x[1] = y[j+1] thì số bước dịch là 2 so với vị trí trước đó. Và dịch 1 bước với các trường hợp còn lại. Do vậy, pha tiền xử lí có thể xác định được độ phức tạp thuật toán và không gian nhớ.

*8.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lí:

k = 1; l = 2;

Pha tìm kiếm thực hiện như sau :

G C A **T** C G C A G A G A G T A T A C A G T A C G

1 2 3

G C A **G** A G A G

Shift by: 2 (l)

G C A **T** C G C A G A G A G T A T A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (k)

G C A T C **G** C A G A G A G T A T A C A G T A C G

1 2

G C **A** G A G A G

Shift by: 2 (l)

G C A T C G C A G A G A G T A T A C A G T A C G

8 1 2 3 4 5 6 7

G C A G A G A G

Shift by: 2 (l)

G C A T C G C A **G** A G A G T A T A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (k)

G C A T C G C A G **A** G A G T A T A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (k)

G C A T C G C A G A **G** A G T A T A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (k)

G C A T C G C A G A G **A** G T A T A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (k)

G C A T C G C A G A G A **G** T A T A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (k)

G C A T C G C A G A G A G **T** A T A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (k)

G C A T C G C A G A G A G T **A** T A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (k)

G C A T C G C A G A G A G T A **T** A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (k)

G C A T C G C A G A G A G T A T **A** C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (k)

G C A T C G C A G A G A G T A T A C A G **T** A C G

1 2 3 4

G C A G **A** G A G

Shift by: 2 (l)

*8.4. Chương trình chạy*

|  |
| --- |
| **package** traisangphai.Not\_So\_Naive;  **public class** Demo\_Not\_So\_Naive {   **public static void** main(String[] args) {  String pattern = **"ABABCABAB"**;  String text = **"ABABDABACDABABCABAB"**;  *searchNsn*(pattern, text);  }   **static void** searchNsn(String pattern, String text) {  **int** shiftIfEqual = 1;  **int** shiftIfNotEqual = 2;  **if** (pattern.charAt(0) != pattern.charAt(1)) {  shiftIfEqual = 2;  shiftIfNotEqual = 1;  }   **int** i = 0;  **while** (i <= text.length() - pattern.length()) {  **if** (pattern.charAt(1) == text.charAt(i + 1)) {  **if** (text.startsWith(pattern, i)) {  System.***out***.println(**"index : "** + i);  }  i += shiftIfEqual;  } **else** {  i += shiftIfNotEqual;  }  }   } } |

# IV. Trình bày các thuật toán tìm kiếm mẫu từ phải qua trái

## 1. Boyer Moore algorithm

*1.1. Đặc điểm*

- Thuật toán thực thi việc so sánh từ phải sang trái.

- Pha cài đặt có độ phức tạp thuật toán và không gian nhớ là O(m+σ).

- Pha thực thi có độ phức tạp là O(mxn).

- 3n kí tự xâu văn bản được so sánh trong trường hợp xấu nhất khi thực thi với mẫu không tuần hoàn.

- Độ phức tạp O(n/m) khi thực thi tốt nhất.

- Gồm 2 phương pháp tính toán số bước dịch: good suffix và bad character

*1.2. Trình bày thuật toán*

Boyer-Moore quan tâm tới thuật toán đối sánh xâu hiệu quả nhất thường thấy. Các biến thể của nó thường được dùng trong các bộ soạn thảo cho các lệnh như <<search>> và <<subtitute>>. Thuật toán sẽ quét các ký tự của mẫu (pattern) từ phải sang trái bắt đầu ở phần tử cuối cùng.

Trong trường hợp mis-match (hoặc là trường hợp đã tìm được 1 đoạn khớp với mẫu), nó sẽ dùng 2 hàm được tính toán trước để dịch cửa sổ sang bên phải. Hai hàm dịch chuyển này được gọi là good-suffix shift (còn được biết với cái tên phép dịch chuyển khớp) và bad-character shift (còn được biết với cái tên phép dịch chuyển xuất hiện).

Đối với mẫu x[0..m-1] ta dùng 1 biến chỉ số i chạy từ cuối về đầu, đối với chuỗi y[0..n-1] ta dùng 1 biến j để chốt ở phía đầu.

G/s rằng trong quá trình so sánh ta gặp 1 mis-match tai vị trí x[i]=a của mẫu và y[i+j]=b trong khi đang thử khớp tại vị trí j.

Khi đó, x[i+1..m-1]=y[j+i+1..j+m-1]=u và x[i]≠y[i+j] . Bây giờ ta đi xét xem đối với từng trường hợp, 2 hàm trên sẽ thực hiện việc dịch chuyển như thế nào:

Phép dịch chuyển good-suffix shift sẽ dịch cửa sổ sang bên phải cho đến khi gặp 1 ký tự khác với x[i] trong trường hợp đoạn u lại xuất hiện trong x.

Nếu đoạn u không xuất hiện lại trong x, mà chỉ có 1 phần cuối(suffix) của u khớp với phần đầu (prefix) của x, thì ta sẽ dịch 1 đoạn sao cho phần suffix dài nhất vcủa y[j+i+1..j+m-1] khớp với prefix của x.

Phép dịch chuyển bad-character shift sẽ khớp kí tự y[i+j] với 1 ký tự (bên phải nhất) trong đoạn x[0..m-2].

Nếu y[i+j] không xuất hiện trong x, ta thấy ngay rằng không có xuất hiện nào của x trong y mà lại chứa chấp y[i+j], do đó ta có thể đặt cửa sổ ngay sau y[i+j], tức là y[j+i+1] .

Thuật toán Boyer-Moore sẽ chọn đoạn dịch chuyển dài nhất trong 2 hàm dịch chuyển good-suffix shift và bad-character shift. Hai hàm này được định nghĩa như sau:

Hàm good-suffix shift được lưu trong bảng bmGs có kích thước m+1.

Ta định nghĩa 2 điều kiện sau:

1. Cs(i, s): với mỗi k mà i < k < m, s ≥ k hoặc x[k-s]=x[k] và

2. Co(i, s): nếu s <i thì x[i-s] ≠ x[i]

Khi đó, với 0≤ i <m: bmGs[i+1]=min{s>0 : Cs(i, s) and Co(i, s) hold} và chúng ta định nghĩa bmGs[0] là độ dài chu kỳ của x. Việc tính toán bảng bmGs sử dụng 1 bảng suff định nghĩa như sau: với 1 ≤ i < m, suff[i]=max{k : x[i-k+1 ..i]=x[m-k .. m-1]}

Hàm bad-character shift được lưu trong bảng bmBc có kích thước σ. Cho c trong Σ: bmBc[c] = min{i : 1≤ i <m-1 và x[m-1-i]=c} nếu c xuất hiện trong x, m ngược lại.

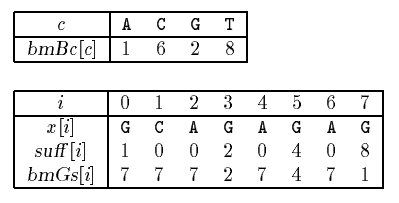
Bảng bmGs và bmBc được tính toán trong thời gian O(m+σ) trước khi thực hiện tìm kiếm và cần 1 không gian phụ là O(m+σ). Giai đoạn tìm kiếm có độ phức tạp thời gian bậc hai nhưng lại chỉ có 3n phép so sánh khi tìm kiếm 1 chuỗi không có chu kì. Đối với việc tìm kiếm trong 1 khối lượng lớn các chữ cái thuật toán thực hiện với tốc độ nhanh “khủng khiếp”. Khi tìm kiếm chuỗi a^(m-1)b trong a^nchuỗi thuật toán chỉsử dụng O(n/m) phép so sánh, đây được coi là “cảnh giới” cho bất cứmột thuật toán tìm kiếm chuỗi nào mà mẫu đã được xử lý trước.

*1.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lí:



Pha thực thi:

G C A T C G C **A** G A G A G T A T A C A G T A C G

1

G C A G A G A **G**

Shift by: 1 (bmGs[7]=bmBc[A]-8+8)

G C A T C G **C** A G A G A G T A T A C A G T A C G

3 2 1

G C A G A **G** A G

Shift by: 4 (bmGs[5]=bmBc[C]-8+6

G C A T C G C A G A G A G T A T A C A G T A C G

8 7 6 5 4 3 2 1

G C A G A G A G

Shift by: 7 (bmGs[0])

G C A T C G C A G A G A G T A T A **C** A G T A C G

3 2 1

G C A G A **G** A G

Shift by: 4 (bmGs[5]=bmBc[C]-8+6)

G C A T C G C A G A G A G T A T A C A G T A **C** G

2 1

G C A G A G **A** G

Shift by: 7 (bmGs[6])

*1.4. Chương trình chạy*

|  |
| --- |
| **package** phaisangtrai.Boyer\_Moore;  **public class** Demo\_Boyer\_Moore {   **public static void** main(String[] args) {  **char**[] x = **"ABCABAB"**.toCharArray();  **char**[] y = **"ABABDABACDABABCABAB"**.toCharArray();  *searchBM*(x, y);  }   **static void** preBmBc(**char**[] x, **int** m, **int** bmBc[]) {  **int** i;   **for** (i = 0; i < 256; ++i)  bmBc[i] = m;  **for** (i = 0; i < m - 1; ++i)  bmBc[x[i]] = m - i - 1;  }   **static void** preBmGs(**char**[] x, **int** m, **int** bmGs[]) {  **int** i, j;  **int**[] suff = **new int**[m];  **int** f = 0, g, h;  suff[m - 1] = m;  g = m - 1;  **for** (h = m - 2; h >= 0; --h) {  **if** (h > g && suff[h + m - 1 - f] < h - g)  suff[h] = suff[h + m - 1 - f];  **else** {  **if** (h < g)  g = h;  f = h;  **while** (g >= 0 && x[g] == x[g + m - 1 - f])  --g;  suff[h] = f - g;  }  }   **for** (**int** temp : suff) System.***out***.print(temp + **" "**);  **for** (i = 0; i < m; ++i)  bmGs[i] = m;  j = 0;  **for** (i = m - 1; i >= 0; --i)  **if** (suff[i] == i + 1)  **for** (; j < m - 1 - i; ++j)  **if** (bmGs[j] == m)  bmGs[j] = m - 1 - i;  **for** (i = 0; i <= m - 2; ++i)  bmGs[m - 1 - suff[i]] = m - 1 - i;  System.***out***.println();  **for** (**int** temp : bmGs) System.***out***.print(temp + **" "**);  }    **static void** searchBM(**char**[] x, **char**[] y) {  **int** m = x.**length**;  **int** n = y.**length**;  **int** i, j;  **int**[] bmGs = **new int**[m];  **int**[] bmBc = **new int**[256];  *preBmGs*(x, m, bmGs);  *preBmBc*(x, m, bmBc);  j = 0;  **while** (j <= n - m) {  **for** (i = m - 1; i >= 0 && x[i] == y[i + j]; --i) ;  **if** (i < 0) {  System.***out***.println(**"index : "** + j);  j += bmGs[0];  } **else** j += Math.*max*(bmGs[i], bmBc[y[i + j]] - m + 1 + i);  }  } } |

## 2. Zhu Kataoka algorithm

*2.1. Đặc điểm*

- Là một biến thể của Boyer Moore.

- Sử dụng 2 kí tự liên tiếp nhau để tính toán bước dịch bad charater.

- Pha cài đặt có độ phức tạp thuật toán và không gian nhớ là O(m+σ^2).

- Pha thực thi có độ phức tạp là O(m x n).

- Khác với BM thì bad character trong thuật toán này dùng 2 ký tự

*2.2. Trình bày thuật toán*

Zhu và Takaoka thiết kế thuật toán mà chúng thực thi dựa trên bad charater. Trong quá trình tìm kiếm, việc so sánh được thực hiện từ phải qua trái, và khi cửa sổ đang ở vị trí y[j…j+m-1] và xuất hiện sự khác nhau giữa x[m- k] và y[j+m-k] trong khi x[m-k+1 .. m-1]=y[j+m-k+1 .. j+m-1]. Bước dịch good suffix cũng được sử dụng để tính toán bước dịch.

Pha tiền xử lí của thuật toán bao gồm việc tính toán mỗi cặp kí tự(a,b) với a,b là mút bên phải của đoạn x[0…m-2]   
  
Với a,b thuộc , ztBc[a, b]=k và k có các giá trị:

k<m-2 và x[m-k .. m-k+1]=ab và ab không xuất hiện trong đoạn x[m-k+2..m-2]

Hoặc

k=m-1 và x[0]=b và ab không xuất hiện trong đoạn x[0 .. m-2]

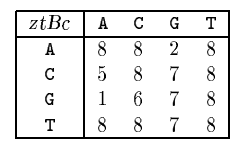
Hoặc

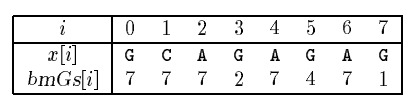
k=m and x[0] bvà ab không xuất hiện trong đoạn x[0 .. m-2]

*2.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lí:



Pha tìm kiếm thực hiện như sau :

G C A T C G C **A** G A G A G T A T A C A G T A C G

1

G C A G A G A **G**

Shift by: 5 (ztBc[C][A])

G C A T C G C A G A G A G T A T A C A G T A C G

8 7 6 5 4 3 2 1

G C A G A G A G

Shift by: 7 (bmGs[0])

G C A T C G C A G A G A G T A T A **C** A G T A C G

3 2 1

G C A G A **G** A G

Shift by: 7 (bmGs[6])

*2.4. Chương trình chạy*

|  |
| --- |
| **package** phaisangtrai.Zhu\_Takaoka;  **public class** Demo\_Zhu\_Takaoka {  **public static void** main(String[] args) {  **char**[] x = **"AAA"**.toCharArray();  **char**[] y = **"AAAAAAA"**.toCharArray();  *searchZT*(x, y);  }  **static void** preZtBc(**char**[] x, **int** m, **int**[][] ztBc) {  **int** i, j;   **for** (i = 0; i < 256; ++i)  **for** (j = 0; j < 256; ++j)  ztBc[i][j] = m;  **for** (i = 0; i < 256; ++i)  ztBc[i][x[0]] = m - 1;  **for** (i = 1; i < m - 1; ++i)  ztBc[x[i - 1]][x[i]] = m - 1 - i;  }   **static void** preBmGs(**char**[] x, **int** m, **int** bmGs[]) {  **int** i, j;  **int**[] suff = **new int**[256];  **int** f = 0, g, h;  suff[m - 1] = m;  g = m - 1;  **for** (h = m - 2; h >= 0; --h) {  **if** (h > g && suff[h + m - 1 - f] < h - g)  suff[h] = suff[h + m - 1 - f];  **else** {  **if** (h < g)  g = h;  f = h;  **while** (g >= 0 && x[g] == x[g + m - 1 - f])  --g;  suff[h] = f - g;  }  }  **for** (i = 0; i < m; ++i)  bmGs[i] = m;  j = 0;  **for** (i = m - 1; i >= 0; --i)  **if** (suff[i] == i + 1)  **for** (; j < m - 1 - i; ++j)  **if** (bmGs[j] == m)  bmGs[j] = m - 1 - i;  **for** (i = 0; i <= m - 2; ++i)  bmGs[m - 1 - suff[i]] = m - 1 - i;  }    **static void** searchZT(**char**[] x, **char**[] y) {  **int** m = x.**length**;  **int** n = y.**length**;  **int** i, j;  **int**[] bmGs = **new int**[256];  **int**[][] ztBc = **new int**[256][256];  *preZtBc*(x, m, ztBc);  *preBmGs*(x, m, bmGs);  j = 0;  **while** (j <= n - m) {  i = m - 1;  **while** (i >= 0 && i < m && x[i] == y[i + j])  --i;  **if** (i < 0) {  System.***out***.println(**"index : "** + j);  j += bmGs[0];  } **else** j += Math.*max*(bmGs[i],  ztBc[y[j + m - 2]][y[j + m - 1]]);  }  } } |

## 3. Berry-Ravindran algorithm

*3.1. Đặc điểm*

- Đây là thuật toán kết hợp giữa thuật toán Quick search và Zhu and Takaoka.

- Pha chuẩn bị có độ phức tạp O(m +∂^2).

- Pha tìm kiếm có độ phức tạp O(mn).

- Quick search 2 ký tự

*3.2. Trình bày thuật toán*

Berry và Ravindran đã thiết kế ra thuật toán thực hiện bước dịch dựa vào tư tưởng bad-character. Nhưng ở đây lấy 2 kí tự liên tiếp ngoài cùng bên phải của sổ để xác định bước dịch.

Quá trình chuẩn bị của thuật toán bao gồm việc xác định với mỗi cặp (a,b) vị trị ngoài cũng bên phải gần nhất bắt đầu xuất hiện ab trong x.

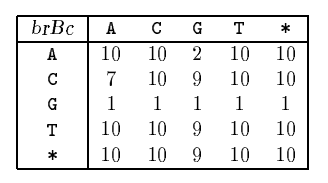
Sau mỗi lần thử mẫu khi cửa sổ đang ở vị trí tương ứng y[j .. j+m-1] bước dịch tiếp theo sẽ là brBc[y[j+m],y[j+m+1]]

*3.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lý xác định:



Pha tìm kiếm thực hiện như sau:

G C A **T** C G C A G A G A G T A T A C A G T A C G

1 2 3 4

G C A **G** A G A G

Shift by: 1 (brBc[G][A])

G **C** A T C G C A G A G A G T A T A C A G T A C G

1

**G** C A G A G A G

Shift by: 2 (brBc[A][G])

G C A **T** C G C A G A G A G T A T A C A G T A C G

1

**G** C A G A G A G

Shift by: 2 (brBc[A][G])

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4 5 6 7 8

G C A G A G A G

Shift by: 10 (brBc[T][A])

G C A T C G C A G A G A G **T** A T A C A G T A C G

1

**G** C A G A G A G

Shift by: 7 (brBc[G][0])

G C A T C G C A G A G A G T A T **A** C A G T A C G

1

**G** C A G A G A G

Shift by: 10 (brBc[0][0])

*3.4. Chương trình chạy*

|  |
| --- |
| **package** phaisangtrai.Berry\_Ravindran;  **public class** Demo\_Berry\_Ravindran {  **public static void** main(String[] args) {  **char**[] x = **"AAA"**.toCharArray();  **char**[] y = **"AAAAAAA"**.toCharArray();  *searchBR*(x, y);  }   **static void** preBrBc(**char**[] x, **int** m, **int**[][] brBc) {  **int** a, b, i;   **for** (a = 0; a < 256; ++a)  **for** (b = 0; b < 256; ++b)  brBc[a][b] = m + 2;  **for** (a = 0; a < 256; ++a)  brBc[a][x[0]] = m + 1;  **for** (i = 0; i < m - 1; ++i)  brBc[x[i]][x[i + 1]] = m - i;  **for** (a = 0; a < 256; ++a)  brBc[x[m - 1]][a] = 1;  }    **static void** searchBR(**char**[] x, **char**[] yOld) {  **int** m = x.**length**;  **int** n = yOld.**length**;  **char**[] y = **new char**[n + 1];  System.*arraycopy*(yOld, 0, y, 0, n);  **int** j;  **int**[][] brBc = **new int**[256][256];  *preBrBc*(x, m, brBc);  y[n] = **'\0'**;  j = 0;  **while** (j <= n - m - 1) {  **if** (**new** String(y).startsWith(**new** String(x), j))  System.***out***.println(**"index : "** + j);  j += brBc[y[j + m]][y[j + m + 1]];  }  } } |

## 4. Turbo-BM algorithm

*4.1. Đặc điểm*

- Đây là thuật toán phát triến từ thuật toán boyer – moore.

- Pha chuẩn bị có độ phức tạp O(m + ∂).

- Pha tìm kiếm có độ phức tạp O(n).

- Trong trường hợp xấu nhất thuật toán cần so sánh 2n kí tự.

- Thêm 1 các tính toán bước nhảy so với Boyer Moore: turbo shift

*4.2. Trình bày thuật toán*

Turbo- BM là thuật toán cải tiến từ thuật toán Boyer-moore. Nó yêu cầu mở rộng thêm quá trính tiền xử lý so với thuật toán boyer-moore cơ bản. Bao gồm việc nhớ một đoạn hậu tố của mẫu trùng với đoạn con của xâu y. Trong lần thử mẫu trước (chỉ trường hợp good-suffix ta mới xác định được đoạn u này. Trường hợp khác u = 0)

Kĩ thuật này có hai lợi thế sau :

- Nó có thể nhẩy qua đoạn xâu con đã biết trước.

- Nó có thể có một bước dịch khác gọi là turbo-shift.

Bước dịch turbo-shift tồn tại trong trường hợp tại vị trí thử hiện tại, hậu tố xâu mẫu đã matches nho hơn so với đoạn nhớ trước đó. Chúng ta có thể gọi u là đoạn nhớ trước đó và v là đoạn hậu tố của x đã match. Như vậy turbo-shift = |u| - |v|

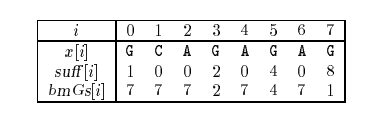
Trường hợp |v| < |u| , độ dài bad-character shift lớn hơn độ dài goodsuffix shift và turbo-shift thì bước dịch thực sự sẽ lơn hơn |u|+1. Lý do là bởi 2 kí từ c và d là khác nhau. Với bước dịch < |u| + 1. Nếu c khớp với kí tự trong v. thì d sẽ không khớp. ta luôn thấy có 1 cập kí tựkhác nhau trong mẫu x và xâu y.

*4.3. Kiểm nghiệm thuật toán*

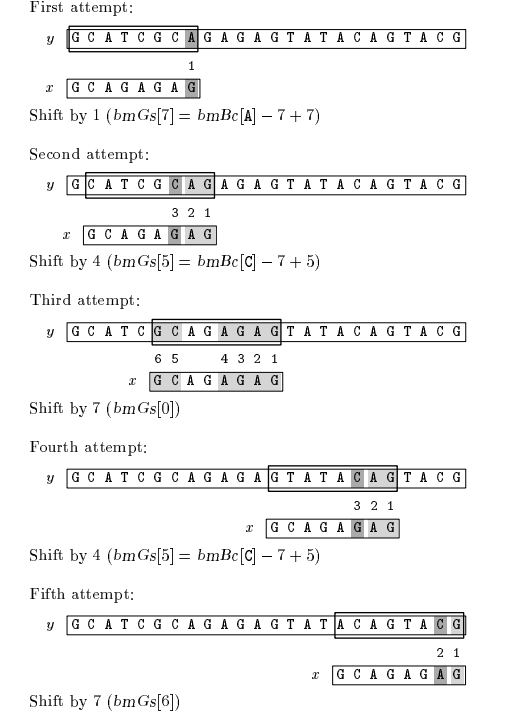
X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lý xác định:



Pha tìm kiếm thực hiện như sau:



*4.4. Chương trình chạy*

|  |
| --- |
| **package** phaisangtrai.Turbo\_BM;  **public class** Demo\_Turbo\_BM {  **public static void** main(String[] args) {  **char**[] x = **"AAA"**.toCharArray();  **char**[] y = **"AAAAAAA"**.toCharArray();  *searchTurboBM*(x, y);  }   **static void** preBmBc(**char**[] x, **int** m, **int** bmBc[]) {  **int** i;   **for** (i = 0; i < 256; ++i)  bmBc[i] = m;  **for** (i = 0; i < m - 1; ++i)  bmBc[x[i]] = m - i - 1;  }   **static void** preBmGs(**char**[] x, **int** m, **int** bmGs[]) {  **int** i, j;  **int**[] suff = **new int**[256];    **int** f = 0, g, h;  suff[m - 1] = m;  g = m - 1;  **for** (h = m - 2; h >= 0; --h) {  **if** (h > g && suff[h + m - 1 - f] < h - g)  suff[h] = suff[h + m - 1 - f];  **else** {  **if** (h < g)  g = h;  f = h;  **while** (g >= 0 && x[g] == x[g + m - 1 - f])  --g;  suff[h] = f - g;  }  }    **for** (i = 0; i < m; ++i)  bmGs[i] = m;  j = 0;  **for** (i = m - 1; i >= 0; --i)  **if** (suff[i] == i + 1)  **for** (; j < m - 1 - i; ++j)  **if** (bmGs[j] == m)  bmGs[j] = m - 1 - i;  **for** (i = 0; i <= m - 2; ++i)  bmGs[m - 1 - suff[i]] = m - 1 - i;  }   **static void** searchTurboBM(**char**[] x, **char**[] y) {  **int** m = x.**length**;  **int** n = y.**length**;  **int** bcShift, i, j, shift, u, v, turboShift;  **int**[] bmGs = **new int**[256];  **int**[] bmBc = **new int**[256];   *preBmGs*(x, m, bmGs);  *preBmBc*(x, m, bmBc);   j = u = 0;  shift = m;  **while** (j <= n - m) {  i = m - 1;  **while** (i >= 0 && x[i] == y[i + j]) {  --i;  **if** (u != 0 && i == m - 1 - shift)  i -= u;  }  **if** (i < 0) {  System.***out***.println(**"index : "** + j);  shift = bmGs[0];  u = m - shift;  } **else** {  v = m - 1 - i;  turboShift = u - v;  bcShift = bmBc[y[i + j]] - m + 1 + i;  shift = Math.*max*(turboShift, bcShift);  shift = Math.*max*(shift, bmGs[i]);  **if** (shift == bmGs[i])  u = Math.*min*(m - shift, v);  **else** {  **if** (turboShift < bcShift)  shift = Math.*min*(shift, u + 1);  u = 0;  }  }  j += shift;  }  } } |

## 5. Tuned boyer-moore algorithm

*5.1. Đặc điểm*

- Đây là thuật toán đơn giản hóa từ thuật toán Boyer – Moore.

- Dễ cài đặt.

- Thuật toán có độ phức tạp bình phương trong trường hợp xấu nhất. Nhưng thường chạy rất nhanh trong thực tế.

- Kiểm tra kí tự cuối cùng trước khi thực hiện kiểm tra xâu.

*5.2. Trình bày thuật toán*

Tuned- Boyer – moore là cài đặt đơn giản của thuật toán boyer – moore. Chi phí cho thuật toán string- matching thường phần nhiều là việc kiểm tra xem xâu mẫu cố bằng xâu tại vị trí cửa số đang xét hay không.

Để tránh việc phải so sánh nhiều lần. chúng ta có thể thực hiện nhiều bước dịch hơn trước khi thực sự so sánh xâu.

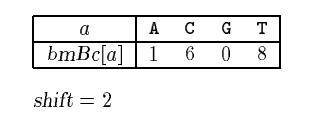
Thuật toán này sẽ sử dụng hàm bad-character xác định bước dịch. Và tìm x[m-1] trong y cho tới khi nào tìm được. Yêu cầu lưu giá trịbmBc[x[m-1]] vào biến shift và đặt lại giá trị bmBc[x[m-1]] = 0. Khi ta tìm được vị trí x[m-1] trong y, thì bước dịch tiếp theo sẽ là shift.

*5.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha chuẩn bị xác định:



Pha tìm kiếm thực hiện như sau:

First attempt

G **C** A T C G C **AG** A G A G T A T A C A G T A C G

1

G C A G A G A G

2

G C A G A G A **G**

3

G C A G A G A **G**

4

G C A G A G A **G**

5

**G** C A G A G A G

Shift by: 2 (shift)

Second attempt

G C A **T** C G C A G A **G** A G T A T A C A G T A C G

1

G C A G A G A **G**

2

**G** C A G A G A G

Shift by: 2 (shift)

Third attempt

G C A T C **G C A G A G A G** T A T A C A G T A C G

1

G C A G A G A **G**

2 3 4 5 6 7 8

**G C A G A G A** G

Shift by: 2 (shift)

Fourth attempt

G C A T C G C A G A G A G T **A T A** C A G T A C **G**

1

G C A G A G A **G**

2

G C A G A G A **G**

3

G C A G A G A **G**

4

G C A G A G A **G**

5

**G** C A G A G A G

Shift by: 2 (shift)

*5.4. Chương trình chạy*

|  |
| --- |
| **package** phaisangtrai.Tuned\_BM;  **public class** Demo\_Tuned\_BM {  **public static void** main(String[] args) {  **char**[] x = **"ABABCABAB"**.toCharArray();  **char**[] y = **"ABABDABACDABABCABAB"**.toCharArray();  *searchTunedBM*(x, y);  }   **static void** preBmBc(**char**[] x, **int** m, **int** bmBc[]) {  **int** i;  **for** (i = 0; i < 256; ++i)  bmBc[i] = m;  **for** (i = 0; i < m - 1; ++i)  bmBc[x[i]] = m - i - 1;  }   **static void** searchTunedBM(**char**[] x, **char**[] yOld) {  **int** m = x.**length**;  **int** n = yOld.**length**;  **int** j, k, shift;  **int**[] bmBc = **new int**[256];  *preBmBc*(x, m, bmBc);  shift = bmBc[x[m - 1]];  bmBc[x[m - 1]] = 0;  **char**[] y = **new char**[n + m - 1];  System.*arraycopy*(yOld, 0, y, 0, n);  j = 0;  **while** (j < n) {  k = bmBc[y[j + m - 1]];  **while** (k != 0) {  j += k;  k = (j + m - 1) < y.**length** ? bmBc[y[j + m - 1]] : 0;  }  **if** (**new** String(y).startsWith(**new** String(x), j))  System.***out***.println(**"index : "** + j);  j += shift;  }  } } |

## 6. Apostolico-giancarlo algorithm

*6.1. Đặc điểm*

- Đây là thuật toán cải tiến từ thuật toàn Boyer – moore.

- Pha xử lý có độ phực tạp O(m + ∂ ) không gian và thời gian.

- Pha tìm kiếm có độ phức tạp O(n).

- Trường hợp xấu nhất thuật toán cần 3/2n lần phép so sánh.

- Lưu lại những ký tự trùng khi dùng good suffix để bỏ qua khi so sánh nhằm tăng tốc độ so sánh

*6.2. Trình bày thuật toán*

Thuật toán Boyer – moore là thuật toán khó để phân tích bởi vì sau mỗi lần thử mẫu nó sẽ quen đi tất cả các kí tự đã đúng trước đó. Apostolico và Giancarlo đã thiết kế một thuật toán có thể nhớ được độ dài lớn nhất của hậu tố xâu mẫu x. Thông tin này sẽ được lưu vào một mảng skip.

Trong mỗi lần thứ tại vị trí j, nếu thuật toán đã so sánh đúng các kí tự y[i+j+1..j+m-1] .

Sẽ có trường hợp xảy ra :

- Trường hợp 1 : k > suff[i] && suff[i] = i+ 1 ; nghĩa là tồn tại x xuất hiện trong y tại vị trí j. bước dịch tiếp theo là per(x).

- Trường hợp 2 : k > suff[i] && suff[i] <= i ; nghĩa là xuất hiện vị trí sai giữa 2 kí tự x[i-suff[i]] and y[i+j-suff[i]] . bước dịch tiếp theo sẽ sửdụng bmBc[y[i+j-suff[i]]] and bmGs[i-suff[i]+1].

- Trường hợp 3 : k < suff[i] .nghĩa là vị trí sai xuật hiện tại x[i-k] and y[i+j-k] . bước dịch tiếp theo sẽ sử dụng bmBc[y[i+j-k] and bmGs[i-k+1].

- Trường hợp 4 : k = suff[i] ; nghĩa là ta đa có k phân tử khơp nhau. Ta sẽ nhẩy qua k phần tử này . tiếp tục so sánh ki tự tại y[i+j-k] and x[i-k]

Ta nhận thấy trong tất cả các trường hợp thông tin cần thiết duy nhất là độ dài hậu tố x xuất hiện tại vị trí i.

Aposltolico- Giancarlo sử dụng 2 cấu trúc dữ liệu :

\* Một mảng skip có thể cập nhập sau mỗi lần thử mẫu tại vị trí

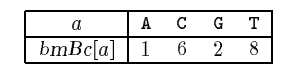
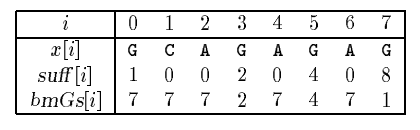
j:skip[j+m-1]=max{ k : x[m-k .. m-1]=y[j+m-k .. j+m-1]}

\* Một mảng suff thỏa mãn : for 1 i < msuff[i]=max{k : x[ik+1 .. i]=x[m-k .. m-1]}

*6.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lý xác định :

Pha tìm kiếm thực hiện như sau :

G C A T C G C **A** G A G A G T A T A C A G T A C G

1

G C A G A G A **G**

Shift by: 1 (bmGs[7]=bmBc[a]-8+8)

G C A T C G **C** A G A G A G T A T A C A G T A C G

3 2 1

G C A G A **G** A G

Shift by: 4 (bmGs[5]=bmBc[c]-8+6)

G C A T C G C A G A G A G T A T A C A G T A C G

6 5 4 3 2 1

G C A G A G A G

Shift by: 7 (bmGs[0])

G C A T C G C A G A G A G T A T A **C** A G T A C G

3 2 1

G C A G A **G** A G

Shift by: 4 (bmGs[5]=bmBc[c]-8+6)

G C A T C G C A G A G A G T A T A C A G T A **C** G

2 1

G C A G A G **A** G

Shift by: 7 (bmGs[6])

*6.4. Chương trình chạy*

|  |
| --- |
| **package** phaisangtrai.Apostolico\_Giancarlo;  **public class** Demo\_Apostolico\_Giancarlo {   **public static void** main(String[] args) {  **char**[] x = **"ABAB"**.toCharArray();  **char**[] y = **"ABABDABACDABABCABAB"**.toCharArray();  *searchAG*(x, y);  }   **static void** preBmBc(**char**[] x, **int** m, **int** bmBc[]) {  **int** i;   **for** (i = 0; i < 256; ++i)  bmBc[i] = m;  **for** (i = 0; i < m - 1; ++i)  bmBc[x[i]] = m - i - 1;  }   **static void** preBmGs(**char**[] x, **int** m, **int** bmGs[], **int**[] suff) {  **int** i, j;   **int** f = 0, g, h;  suff[m - 1] = m;  g = m - 1;  **for** (h = m - 2; h >= 0; --h) {  **if** (h > g && suff[h + m - 1 - f] < h - g)  suff[h] = suff[h + m - 1 - f];  **else** {  **if** (h < g)  g = h;  f = h;  **while** (g >= 0 && x[g] == x[g + m - 1 - f])  --g;  suff[h] = f - g;  }  }    **for** (i = 0; i < m; ++i)  bmGs[i] = m;  j = 0;  **for** (i = m - 1; i >= 0; --i)  **if** (suff[i] == i + 1)  **for** (; j < m - 1 - i; ++j)  **if** (bmGs[j] == m)  bmGs[j] = m - 1 - i;  **for** (i = 0; i <= m - 2; ++i)  bmGs[m - 1 - suff[i]] = m - 1 - i;  }   **static void** searchAG(**char**[] x, **char**[] y) {  **int** m = x.**length**;  **int** n = y.**length**;  **int**[] bmGs = **new int**[m];  **int**[] bmBc = **new int**[256];  **int** i, j, k, s, shift;  **int**[] skip = **new int**[m];  **int**[] suff = **new int**[m];   *preBmGs*(x, m, bmGs, suff);  *preBmBc*(x, m, bmBc);   **for** (**int** temp : suff) System.***out***.print(temp + **" "**);   j = 0;  **while** (j <= n - m) {  i = m - 1;  **while** (i >= 0) {  k = skip[i];  s = suff[i];  **if** (k > 0)  **if** (k > s) {  **if** (i + 1 == s)  i = (-1);  **else** i -= s;  **break**;  } **else** {  i -= k;  **if** (k < s)  **break**;  }  **else** {  **if** (x[i] == y[i + j])  --i;  **else  break**;  }  }  **if** (i < 0) {  System.***out***.println(**"index : "** + j);  skip[m - 1] = m;  shift = bmGs[0];  } **else** {  skip[m - 1] = m - 1 - i;  shift = Math.*max*(bmGs[i], bmBc[y[i + j]] - m + 1 + i);  }  j += shift;  **for** (**int** t = 0; t < m - shift; t++) skip[t] += shift;  }  } } |

## 7. Quick Search algorithm

*7.1. Đặc điểm*

- Là thuật toán đơn giản của Boyer Moore.

- Chỉ sử dụng dụng bad- character.

- Dễ để cài đặt.

- Pha cài đặt có độ phức tạp thuật toán là O(m+σ) và độ phức tạp bộ nhớ là O(σ).

- Pha thực thi có độ phức tạp là O(mxn).

- Rất nhanh trong thực tế với những mẫu ngắn và anphabets lớn.

*7.2. Trình bày thuật toán*

Thuật toán Quick Search chỉ sử dụng bảng bad charater. Giả sử trên cửa sổ dịch chuyển đang là đoạn văn bản y[j….j+m-1], độ dài bước dịch đồng đều là 1 bước. Vì vậy, vị trí tiếp theo có liên quan mật thiết tới số bước dịch chuyển. Quick Search sẽ quan tâm tới vị trí y[j+m].

qsBc[c]=min{i : 0 <= i <= m và x[m-i]=c} nếu c có trong x.

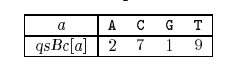
qsBc[c]=m+1 với trường hợp còn lại.

*7.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lí:



Pha tìm kiếm thực hiện:

G C A **T** C G C A G A G A G T A T A C A G T A C G

1 2 3 4

G C A **G** A G A G

Shift by: 1 (qsBc[G])

G **C** A T C G C A G A G A G T A T A C A G T A C G

1

**G** C A G A G A G

Shift by: 2 (qsBc[A])

G C A **T** C G C A G A G A G T A T A C A G T A C G

1

**G** C A G A G A G

Shift by: 2 (qsBc[A])

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4 5 6 7 8

G C A G A G A G

Shift by: 9 (qsBc[T])

G C A T C G C A G A G A G T **A**  T A C A G T A C G

1

**G**  C A G A G A G

Shift by: 7 (qsBc[C])

*7.4. Chương trình chạy*

|  |
| --- |
| **package** phaisangtrai.QuickSearch;  **public class** Demo\_Quick\_Search {   **public static void** main(String[] args) {  **char**[] x = **"AAA"**.toCharArray();  **char**[] y = **"AAAAAAA"**.toCharArray();  *searchQuickSearch*(x, y);  }   **public static int**[] preQsBc(**char**[] x) {  **int**[] bc = **new int**[255];  **int** m = x.**length**;  **for** (**int** i = 0; i < 255; i++) {  bc[i] = m + 1;  }  **for** (**int** i = 0; i < m; i++) {  bc[(**int**) x[i]] = m - i;  }  **return** bc;  }   **public static void** searchQuickSearch(**char**[] x, **char**[] y) {  **int** m = x.**length**;  **int** n = y.**length**;  **int**[] preBc = *preQsBc*(x);  System.***out***.println();  **int** i = 0;  **while** (i <= n - m) {  **if** (*cmp*(x, y, i)) {  System.***out***.println(**"index: "** + i);  }  i = i + preBc[y[i + m]];  }  }   **public static boolean** cmp(**char**[] x, **char**[] y, **int** y1) {  **for** (**char** c : x) {  **if** (c != y[y1++]) {  **return false**;  }  }  **return true**;  } } |

# V. Trình bày các thuật toán tìm kiếm mẫu từ vị trí cụ thể

## 1. Colussi algorithm

*1.1. Đặc điểm*

- Đây là thuật toán cái tiến từ thuật toán Knuth, Morris and Pratt.

- Mẫu tìm kiếm sẽ được chia thành 2 tập con. Tập thứ nhất sẽ được duyệt từ trái sang phải khi không gặp mismatch ta duyệt tập thứ hai từ trái qua phải.

- Pha xử lý có độ phức tạp là O(m).

- Pha tìm kiếm có độ phức tạp là O(n).

- Trong trường hợp xấu nhất quá trình thực thi sẽ mất 3/2 n lần so sánh kí tự.

*1.2. Trình bày thuật toán*

Thuật toán Colussi được thiết kế dựa theo những phân tích từ thuật toán knuth, Morris, và Pratt. Các vị trí kí tự trong mẫu được chia thành 2 tập riêng biệt. mỗi phép thử sẽ bao gồm 2 pha :

- Trong pha thứ nhất thực hiện so sánh từ trái sang phải lần lượt các kí tự tại những vị trí kmpNext > -1. Những vị trí này gọi là noholes.

- Pha thứ hai so sánh kí tự từ phải sang trái( các kí tự còn lại ) nhưng kí tự này có kmpNext = -1 và gọi là holes. Chiến lược này có 2 lợi thế:

\* Khi gặp mismath trong pha thứ nhất, sau khi thực hiện shift bước dịch ta không cần phải so sánh lại các kí tự tại những vị trí noholes phía trước.

\* Khi gặp mismath trong pha thứ hai có nghĩa là đoạn hậu tố đã matches tất cả kí tự định nghĩa biến :

For 0 <= i <= m-1:

kmin[i]= d>0 iff x[0 .. i-1-d]=x[d .. i-1] và x[i-d]x[i],

kmin[i] = 0 trong các trường hợp còn lại.

For 0 <= i < m:

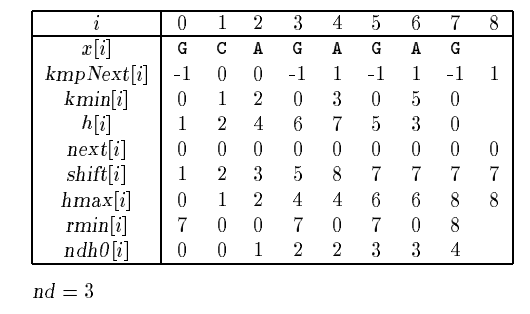
Nếu kmin[i] # 0 thì i là nohole ngược lại i là hole.

*1.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lý xác định:



Pha tìm kiếm:

G C A T **C** G C A G A G A G T A T A C A G T A C G

1 2 3

G C A G **A** G A G

Shift by: 3 (shift[2])

G C A T C **G** C A G A G A G T A T A C A G T A C G

1 2

G C **A** G A G A G

Shift by: 2 (shift[1])

G C A T C G C A G A G A G T A T A C A G T A C G

8 1 2 7 3 6 4 5

G C A G A G A G

Shift by: 7 (shift[8])

G C A T C G C A G A G A G **T** A T A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (shift[0])

G C A T C G C A G A G A G T **A** T A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (shift[0])

G C A T C G C A G A G A G T A **T** A C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (shift[0])

G C A T C G C A G A G A G T A T **A** C A G T A C G

1

G **C** A G A G A G

Shift by: 1 (shift[0])

G C A T C G C A G A G A G T A T A C A G **T** A C G

1 2 3

G C A G **A** G A G

Shift by: 3 (shift[2])

*1.4. Chương trình chạy*

|  |
| --- |
| **package** cuthe.Colussi;  **public class** Demo\_Colussi {   **public static void** main(String[] args) {  **char**[] x = **"ABABCABAB"**.toCharArray();  **char**[] y = **"ABABDABACDABABCABAB"**.toCharArray();  *searchCOLUSSI*(x, y);  }   **static int** preColussi(**char** []x, **int** m, **int**[] h, **int**[] next,  **int**[] shift) {  **int** i, k, nd, q, r = 0, s;  **int**[] hmax = **new int**[m+1];  **int**[] kmin = **new int**[m+1];  **int**[] nhd0 = **new int**[m+1];  **int**[] rmin = **new int**[m+1];   */\* Computation of hmax \*/* i = k = 1;  **do** {  **while** (i<m&&x[i] == x[i - k])  i++;  hmax[k] = i;  q = k + 1;  **while** (hmax[q - k] + k < i) {  hmax[q] = hmax[q - k] + k;  q++;  }  k = q;  **if** (k == i + 1)  i = k;  } **while** (k <= m);   */\* Computation of kmin \*/* **for** (i = m; i >= 1; --i)  **if** (hmax[i] < m)  kmin[hmax[i]] = i;   */\* Computation of rmin \*/* **for** (i = m - 1; i >= 0; --i) {  **if** (hmax[i + 1] == m)  r = i + 1;  **if** (kmin[i] == 0)  rmin[i] = r;  **else** rmin[i] = 0;  }   */\* Computation of h \*/* s = -1;  r = m;  **for** (i = 0; i < m; ++i)  **if** (kmin[i] == 0)  h[--r] = i;  **else** h[++s] = i;  nd = s;   */\* Computation of shift \*/* **for** (i = 0; i <= nd; ++i)  shift[i] = kmin[h[i]];  **for** (i = nd + 1; i < m; ++i)  shift[i] = rmin[h[i]];  shift[m] = rmin[0];   */\* Computation of nhd0 \*/* s = 0;  **for** (i = 0; i < m; ++i) {  nhd0[i] = s;  **if** (kmin[i] > 0)  ++s;  }    */\* Computation of next \*/* **for** (i = 0; i <= nd; ++i)  next[i] = nhd0[h[i] - kmin[h[i]]];  **for** (i = nd + 1; i < m; ++i)  next[i] = nhd0[m - rmin[h[i]]];  next[m] = nhd0[m - rmin[h[m - 1]]];   **return**(nd);  }   **static void** searchCOLUSSI(**char**[] x, **char**[] y) {  **int** m = x.**length**;  **int** n = y.**length**;  **int**[] h = **new int**[m+1];  **int**[] next = **new int**[m+1];  **int**[] shift = **new int**[m+1];  **int** i, j, last, nd;   nd = *preColussi*(x, m, h, next, shift);   i = j = 0;  last = -1;  **while** (j <= n - m) {  **while** (i < m && last < j + h[i] &&  x[h[i]] == y[j + h[i]])  i++;  **if** (i >= m || last >= j + h[i]) {  System.***out***.println(**"index : "** + j);  i = m;  }  **if** (i > nd)  last = j + m - 1;  j += shift[i];  i = next[i];  }  } } |

## 2. Skip Search Algorithm

*2.1. Đặc điểm*

- Sử dụng thùng chứa (bucket) các vị trí xuất hiện của kí tự trong xâu mẫu.

- Pha xử lý có độ phức tạp thời gian và không gian chứa O(m+ ∂).

- Pha tìm kiếm có độ phức tạp thời gian O(mn).

*2.2. Trình bày thuật toán*

Với mỗi kí tự trong bảng chữ cái, một thùng chứa (bucket) sẽ chứa tất cả các vị trí xuất hiện của kí tự đó trong xâu mẫu x. khi một kí tự xuất hiện k lần trong mẫu. bucket sẽ lưu k vị trí của kí tự đó. Khi mà xâu y chứa it kí tự hơn trong bản chữ cái thì sẽ có nhiều bucket rỗng. Quá trình xử lý của thuật toán Skip Search bao gồm việc tính các buckets cho tất cả các kí tự trong bảng chữ cái: for c in z[c] = {i : 0<= i <= m-1 và x[i] = c}

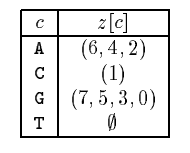
Thuật toán Skip Search có độ phức tạp bình phương trong trường hợp tồi nhất. Nhưng cũng có trường hợp là O(n).

*2.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lý xác định:



Pha tìm kiếm:

First attempt

G **C** A T C G C A G A G A G T A T A C A G T A C G

2 1

**G** C A G A G A G

G C A **T** C G C A G A G A G T A T A C A G T A C G

1

**G** C A G A G A G

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4 5 6 7 8

G C A G A G A G

Shift by: 8

Second attempt

G C A T C G C A G A G A G **T** A T A C A G T A C G

1

Shift by: 8

Third attempt

G C A T C G C A G A G A G T A T **A** C A G T A C G

2 1

**G** C A G A G A G

*2.4. Chương trình chạy*

|  |
| --- |
| **package** cuthe.Skip;  **import** java.util.ArrayList; **import** java.util.HashMap;  **public class** Skip {  **public static** HashMap<String, ArrayList<Integer>> preSkip(**char**[] x) {  HashMap<String, ArrayList<Integer>> re = **new** HashMap<>();  **for** (**int** i = x.**length** - 1; i >= 0; i--) {  ArrayList<Integer> temp;  **if** (re.containsKey(x[i] + **""**)) {  temp = re.get(x[i] + **""**);  } **else** {  temp = **new** ArrayList<>();  }  temp.add(i);  re.put(x[i] + **""**, temp);  }  **return** re;  }   **public static boolean** cmp(**char**[] x, **char**[] y, **int** y1) {  **for** (**char** c : x) {  **if** (c != y[y1++]) {  **return false**;  }  }  **return true**;  }   **public static void** main(String[] args) {  **char**[] x = **"GCAGAGAG"**.toCharArray();  **char**[] y = **"GCATCGCAGAGAGTATACAGTACG "**.toCharArray();  HashMap<String, ArrayList<Integer>> map = *preSkip*(x);  **int** m = x.**length**;  **int** n = y.**length**;  **for** (**int** j = m - 1; j < n; j += m) {  ArrayList<Integer> temp = map.get(y[j] + **""**);  **if** (temp != **null**) {  **for** (**int** i : temp) {  j -= i;  **if** (*cmp*(x, y, j)) {  System.***out***.println(**"Các vị trí xuất hiện trong văn bản của xâu mẫu là: "** + j);  }  j += i;  }  }  }  } } |

## 3. Alpha Skip Search Algorithm

*3.1. Đặc điểm*

- Cải tiến thuật toán Skip Search.

- Sử dụng các thùng vị trí bằng log(m) của mẫu

- Giai đoạn tiền xử lý trong độ phức tạp thời gian và không gian O (m)

*3.2. Trình bày thuật toán*

Giai đoạn tiền xử lý của thuật toán Alpha Skip Search bao gồm xây dựng bộ ba T (x) của tất cả các yếu tố có độ dài ell = logigmam xảy ra trong từ x. Các lá của T (x) đại diện cho tất cả các yếu tố về độ dài ell của x. Sau đó, có một thùng cho mỗi lá của T (x) trong đó được lưu trữ danh sách các vị trí trong đó yếu tố, liên quan đến lá, xảy ra trong x.

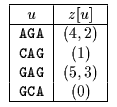
Thời gian trường hợp xấu nhất của giai đoạn tiền xử lý này là tuyến tính nếu kích thước bảng chữ cái được coi là một hằng số.

Giai đoạn tìm kiếm bao gồm việc xem xét các nhóm của các yếu tố văn bản y [j .. j + ell-1] cho tất cả j = k. (M-ell + 1) -1 với số nguyên k trong khoảng y [1, lfloor (n-ell) / mrfloor].

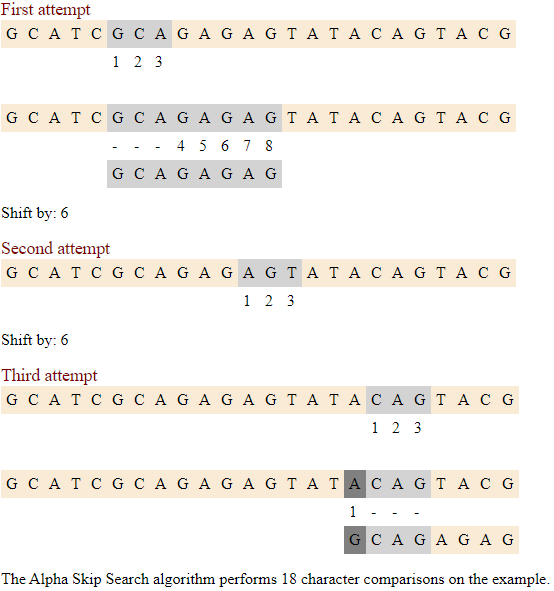
Độ phức tạp thời gian trường hợp xấu nhất của pha tìm kiếm là bậc hai nhưng số lượng so sánh ký tự văn bản dự kiến là O (logigma (m). (N / (m-logigma (m)))).

*3.3. Kiểm nghiệm thuật toán*

- Pha tiền xử lý:



- Pha thực thi:



*3.4. Chương trình chạy*

|  |
| --- |
| **package** cuthe.AlphaSkip;  **import** java.util.ArrayList; **import** java.util.HashMap;  **public class** AlphaSkip {  **public static** HashMap<String, ArrayList<Integer>> preAlphaSkip(**char**[] x, **int** k) {  HashMap<String, ArrayList<Integer>> re = **new** HashMap<>();  **for** (**int** i = x.**length** - 1; i >= k - 1; i--) {  StringBuilder tem = **new** StringBuilder();  **for** (**int** j = i - k + 1; j <= i; j++) {  tem.append(x[j]);  }  ArrayList<Integer> temp;  **if** (re.containsKey(tem + **""**)) {  temp = re.get(tem + **""**);  } **else** {  temp = **new** ArrayList<>();  }  temp.add(i - k + 1);  re.put(tem + **""**, temp);  }  **return** re;  }    **public static boolean** cmp(**char**[] x, **char**[] y, **int** y1) {  **for** (**char** c : x) {  **if** (c != y[y1++]) {  **return false**;  }  }  **return true**;  }    **public static void** main(String[] args) {  **char**[] x = **"CAGAGAGT"**.toCharArray();  **char**[] y = **"GCATCGCAGAGAGTATACAGTACG "**.toCharArray();  **int** k = 3;  HashMap<String, ArrayList<Integer>> map = *preAlphaSkip*(x, k);  **for** (String i : map.keySet()) {  System.***out***.print(i + **" : "**);  **for** (**int** j : map.get(i)) {  System.***out***.print(j + **" "**);  }  System.***out***.println();  }  **int** m = x.**length**;  **int** n = y.**length**;  **for** (**int** j = m - 1; j < n; j += m - k + 1) {  StringBuilder tem = **new** StringBuilder();  **for** (**int** l = j - k + 1; l <= j; l++) {  tem.append(y[l]);  }  ArrayList<Integer> temp = map.get(tem.toString());  **if** (temp != **null**) {  **for** (**int** i : temp) {  **if** (*cmp*(x, y, j - i - k + 1)) {  System.***out***.println(**"Các vị trí xuất hiện trong văn bản của xâu mẫu là: "** + (j - i - k + 1));  }  }  }  }  } } |

# VI. Trình bày các thuật toán tìm kiếm mẫu từ vị trí bất kỳ

## 1. Horspool algorithm

*1.1. Đặc điểm*

- Là thuật toán đơn giản hơn của Boyer Moore.

- Sử dụng dụng bad – character.

- Dễ để cài đặt.

- Pha cài đặt có độ phức tạp thuật toán là O(m+σ) và độ phức tạp bộ nhớ là O(σ).

- Pha thực thi có độ phức tạp là O(mxn).

- Số lượng so sánh trung bình cho một văn bản là trong khoảng 1/ σ và 2/( σ +1).

*1.2. Trình bày thuật toán*

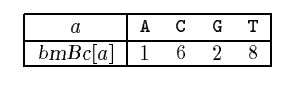
Quy tắc dịch bad- character được sử dụng trong Boyer Moore không có hiệu quả trong đoạn văn bản kí tự nhỏ, nhưng khi đoạn văn bản là lớn và phải so sánh với mẫu dài, nó thường trong trường hợp bảng bã ASCII và tìm kiếm thông thường trong soạn thảo văn bản, chúng rất h ữu dụng. Sử dụng nó như những kết quả riêng biệt lại rất hiệu quả.

*1.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lí:



Pha tìm kiếm thực hiện :

G C A T C G C **A** G A G A G T A T A C A G T A C G

1

G C A G A G A **G**

Shift by: 1 (bmBc[A])

G **C** A T C G C A G A G A G T A T A C A G T A C G

2 1

**G** C A G A G A G

Shift by: 2 (bmBc[G])

G C A **T** C G C A G A G A G T A T A C A G T A C G

2 1

**G** C A G A G A G

Shift by: 2 (bmBc[G])

G C A T C G C A G A G A G T A T A C A G T A C G

2 3 4 5 6 7 8 1

G C A G A G A G

Shift by: 2 (bmBc[G])

G C A T C G C A G A G A G T **A** T A C A G T A C G

1

G C A G A G A **G**

Shift by: 1 (bmBc[A])

G C A T C G C A G A G A G T A **T** A C A G T A C G

1

G C A G A G A **G**

Shift by: 8 (bmBc[T])

G C A T C G C A G A G A G T A T **A**  C A G T A C G

2 1

**G** C A G A G A G

Shift by: 2 (bmBc[G])

*1.4. Chương trình chạy*

|  |
| --- |
| **package** ngaunhien.Horspool;   **public class** Horspool {  **public static boolean** cmp(**char**[] x, **char**[] y, **int** y1) {  **for** (**char** c : x) {  **if** (c != y[y1++]) {  **return false**;  }  }  **return true**;  }   *//tao mang danh dau khoang cach tu tung tu ki tu cuoi cung cua xau mau den ki tu cuoi cung cua xau (khong tinh ki tu cuoi cung)  //vi du xau mau: GCAGAGAG  //mang danh dau: A(1) C(6) G(2) T(8)* **public static int**[] preBc(**char**[] x) {  **int**[] bc = **new int**[255];  **int** m = x.**length**;  **for** (**int** i = 0; i < 255; i++) {  bc[i] = m;  }  **for** (**int** i = 0; i < m - 1; i++) {  bc[(**int**) x[i]] = m - i - 1;  }   **return** bc;  }   **public static void** search(**char**[] x, **char**[] y) {  **int** m = x.**length**;  **int** n = y.**length**;  **int**[] preBc = *preBc*(x);  **int** i = 0;  **while** (i <= n - m) {  **char** c = y[i + m - 1];  **if** (*cmp*(x, y, i)) {  System.***out***.println(**"Các vị trí xuất hiện trong văn bản của xâu mẫu là: "** + i);  }  i = i + preBc[c];  }  }   **public static void** main(String[] args) {  **char**[] x = **"GCAGAGAG"**.toCharArray();  **char**[] y = **"GCATCGCAGAGAGTATACAGTACG"**.toCharArray();  *search*(x, y);  } } |

## 2. Smith algorithm

*2.1. Đặc điểm*

- Thuật toán Smith tính max của hàm dịch bad charater Horspool và hàm dịch bad charater Quick Search.

- Pha cài đặt có độ phức tạp thuật toán là O(m+σ) và độ phức tạp bộnhớ là O(σ).

- Pha thực thi có độ phức tạp là O(mxn).

*2.2. Trình bày thuật toán*

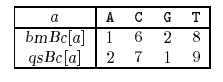
Smith nhận thấy rằng có thể tính toán được rằng số bước dịch với kí tự tiếp theo kí tự đầu mút bên phải của cửa sổ dịch chuyển đôi khi cho sốbước dịch ngắn hơn việc sử dụng đúng kí tự đầu mút bên phải cửa sổ ấy. Ông ấy khuyên nên chọn max giữa 2 giá trị.Pha tiền xử lí của thuật toán Smith bao gồm tính toán của hàm dịch chuyển các bad- character: preBmBc(X,m) , và hàm dịch chuyển các badcharacter trong Quick Search: preQsBc(X,m) Pha tìm kiếm có độ phức tạp về thời gian trong trường hợp xấu nhất là một hàm bậc 2.

*2.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lí tính toán được:



G C A **T** C G C A G A G A G T A T A C A G T A C G

1 2 3 4

G C A **G** A G A G

Shift by: 1 (bmBc[A]=qsBc[G])

G **C** A T C G C A G A G A G T A T A C A G T A C G

1

**G** C A G A G A G

Shift by: 2 (bmBc[G]=qsBc[A])

G C A **T**  C G C A G A G A G T A T A C A G T A C G

1

**G** C A G A G A G

Shift by: 2 (bmBc[G]=qsBc[A])

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4 5 6 7 8

G C A G A G A G

Shift by: 9 (qsBc[T])

G C A T C G C A G A G A G T **A** T A C A G T A C G

1

**G** C A G A G A G

Shift by: 7 (qsBc[C])

*2.4. Chương trình chạy*

|  |
| --- |
| **package** ngaunhien.Smith;  **public class** Smith {  **public static int**[] preQsBc(**char**[] x) {  **int**[] bc = **new int**[255];  **int** m = x.**length**;  **for** (**int** i = 0; i < 255; i++) {  bc[i] = m + 1;  }  **for** (**int** i = 0; i < m; i++) {  bc[(**int**) x[i]] = m - i;  }  **return** bc;  }   **public static int**[] preBc(**char**[] x) {  **int**[] bc = **new int**[255];  **int** m = x.**length**;  **for** (**int** i = 0; i < 255; i++) {  bc[i] = m;  }  **for** (**int** i = 0; i < m - 1; i++) {  bc[(**int**) x[i]] = m - i - 1;  }  **return** bc;  }   **public static void** search(**char**[] x, **char**[] y) {  **int** m = x.**length**;  **int** n = y.**length**;  **int**[] preBc = *preBc*(x);  **int**[] preQsBc = *preQsBc*(x);  **int** i = 0;  **while** (i <= n - m) {  **if** (*cmp*(x, y, i)) {  System.***out***.println(**"Các vị trí xuất hiện trong văn bản của xâu mẫu là: "** + i);  }  i += Math.*max*(preBc[y[i + m - 1]], preQsBc[y[i + m]]);  }  }   **public static boolean** cmp(**char**[] x, **char**[] y, **int** y1) {  **for** (**char** c : x) {  **if** (c != y[y1++]) {  **return false**;  }  }  **return true**;  }   **public static void** main(String[] args) {  **char**[] x = **"GCAGAGAG"**.toCharArray();  **char**[] y = **"GCATCGCAGAGAGTATACAGTACG "**.toCharArray();  *search*(x, y);  } } |

## 3. Raita algorithm

*3.1. Đặc điểm*

- Đầu tiên so sánh kí tự cuối cùng của xâu mẫu, sau đó là kí tự đầu tiên và kí tự giữa trước khi thực thi lệnh so sánh khác.

- Thực thi bước dịch giống thuật toán Horspool.

- Pha cài đặt có độ phức tạp thuật toán là O(m+σ) và độ phức tạp bộnhớ là O(σ).

- Pha thực thi có độ phức tạp là O(mxn).

*3.2. Trình bày thuật toán*

Raita thiết kế một thuật toán mà với mỗi lần duyệt, đầu tiên đem so sánh kí tự cuối cùng của mẫu với kí tự đầu mút bên phải của cửa sổ, sau đó nếu chúng trùng nhau, so sánh kí tự đầu tiên của mẫu với kí tự đầu mút bên trái của cửa sổ. Nếu chúng trùng nhau, tiếp tục so sánh kí tự giữa của mẫu với kí tự giữa của xâu văn bản đang trong cửa sổ duyệt. Nếu chúng vẫn trùng nhau, sẽ đem so sánh các kí tự khác trong xâu mẫu bắt đầu vị trí thứ 2 tới vị trí cuối cùng(so sánh lại kí tự ở giữa một lần nữa).

Raita thấy rằng thuật toán này sử dụng tốt trong những trường hợp: khi cần thực thi với mẫu trong văn bản tiếng anh. Pha tìm kiếm có độ phức tạp về thời gian trong trường hợp xấu nhất là một hàm bậc 2.

*3.3. Kiểm nghiệm thuật toán*

X = “GCAGAGAG”

Y = “GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lí:



Pha tìm kiếm thực hiện :

G C A T C G C **A** G A G A G T A T A C A G T A C G

1

G C A G A G A **G**

Shift by: 1 (bmBc[A])

G  **C** A T C G C A G A G A G T A T A C A G T A C G

2 1

**G** C A G A G A G

Shift by: 2 (bmBc[G])

G C A  **T** C G C A G A G A G T A T A C A G T A C G

2 1

**G** C A G A G A G

Shift by: 2 (bmBc[G])

G C A T C G C A G A G A G T A T A C A G T A C G

2 4 5 6 3 8 9 1

G C A G A G A G

Shift by: 2 (bmBc[G])

G C A T C G C A G A G A G T  **A** T A C A G T A C G

1

G C A G A G A **G**

Shift by: 1 (bmBc[A])

G C A T C G C A G A G A G T A **T** A C A G T A C G

1

G C A G A G A **G**

Shift by: 8 (bmBc[T])

G C A T C G C A G A G A G T A T **A** C A G T A C G

2 1

**G** C A G A G A G

Shift by: 2 (bmBc[G])

*3.4. Chương trình chạy*

|  |
| --- |
| **package** ngaunhien.Raita;  **public class** Raita {  *//ham so sanh 2 xau* **public static boolean** cmp(**char**[] x, **int** x1, **int** x2, **char**[] y, **int** y1) {  **for** (**int** i = x1; i <= x2; i++) {  **if** (x[i] != y[y1++]) {  **return false**;  }  }  **return true**;  }  *//tao mang danh dau khoang cach tu tung tu ki tu cuoi cung cua xau mau den ki tu cuoi cung cua xau (khong tinh ki tu cuoi cung)  //vi du xau mau: GCAGAGAG  //mang danh dau: A(1) C(6) G(2) T(8)* **public static int**[] preBc(**char**[] x) {  **int**[] bc = **new int**[255];  **int** m = x.**length**;  **for** (**int** i = 0; i < 255; i++) {  bc[i] = m;  }  **for** (**int** i = 0; i < m - 1; i++) {  bc[(**int**) x[i]] = m - i - 1;  }  **return** bc;  }   **public static void** search(**char**[] x, **char**[] y) {  **int** m = x.**length**;  **int** n = y.**length**;  **int**[] preBc = *preBc*(x);  **int** i = 0;  **while** (i <= n - m) {  **char** c = y[i + m - 1];  **char** first = x[0];  **char** middle = x[m / 2];  **char** last = x[m - 1];  **if** (first == y[i] && middle == y[i + m / 2] && last == y[i + m - 1] && *cmp*(x, 1, m - 2, y, i + 1)) {  System.***out***.println(**"Các vị trí xuất hiện trong văn bản của xâu mẫu là: "** + i);  }  i = i + preBc[c];  }  }   **public static void** main(String[] args) {  **char**[] x = **"GCAGAGAG"**.toCharArray();  **char**[] y = **"GCATCGCAGAGAGTATACAGTACG"**.toCharArray();  *search*(x, y);  } } |

**LỜI CẢM ƠN**

Em xin gửi lời cảm ơn chân thành tới thầy giáo Nguyễn Duy Phương đã tận tình hướng dẫn em trong suốt dự án giúp em hoàn thiện đề tài của mình và có cái nhìn sâu sắc hơn về công việc của mình trong tương lai.

Dù vậy, trong quá trình thực hiện đề tài em không thể không tránh khỏi sai sót và những mặt hạn chế mà em chưa làm được do kiến thức và kinh nghiệm thực tế còn hạn chế mong thầy bỏ qua và góp ý thêm cho em giúp em có một tiền đề vững chắc hơn để làm tốt hơn trong những báo cáo tiếp theo nói chung và đối với công việc tương lai sau khi tốt nghiệp nói riêng.

Lời cuối cùng, em xin kính chúc thầy có một sức khỏe dồi dào, niềm tin để tiếp tục thực hiện sứ mệnh cao đẹp của mình là truyền đạt kiến thức cho thế hệ mai sau.