HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

BÁO CÁO CHUYÊN ĐỀ THUẬT TOÁN TÌM KIẾM MẪU

Môn học : Chuyên đề công nghệ phần mềm

Giảng viên : Nguyễn Duy Phương

Sinh viên : Hàn Công Nhu

Mã sinh viên : B17DCCN481

Nội Dung

CHƯƠNG 1: TỔNG QUAN: SẮP XẾP LẠI CÁC THUẬT TOÁN

1.1 Giới thiệu vấn đề

- Đối sánh xâu (String matching) là một chủ đề quan trọng trong lĩnh vực xử lý văn bản. Các thuật toán đối sánh xâu được xem là những thành phần cơ sở được cài đặt cho các hệ thống thực tế đang tồn tại trong hầu hết các hệ điều hành. Hơn thế nữa, các thuật toán đối sánh xâu cung cấp các mô hình cho nhiều lĩnh vực khác nhau của khoa học máy tính: xử lý ảnh, xử lý ngôn ngữ tự nhiên, tin sinh học và thiết kế phần mềm.

- String-matching được hiểu là việc tìm một hoặc nhiều xâu mẫu (pattern) xuất hiện trong một văn bản (có thể là rất dài). Ký hiệu xâu mẫu hay xâu cần tìm là $X =(x_0, x_1,..,x_{m-1})$ có độ dài m. Văn bản $Y =(y_0, y_1,..,y_{n-1})$ có độ dài n. Cả hai xâu được xây dựng từ một tập hữu hạn các ký tự Alphabet ký hiệu là Σ với kích cỡ là σ. Như vậy một xâu nhị phân có độ dài n ứng dụng trong mật mã học cũng được xem là một mẫu. Một chuỗi các ký tự ABD độ dài m biểu diễn các chuỗi AND cũng là một mẫu.

Input:

- Xâu mẫu X =(x0, x1,.., xm), độ dài m.
- Văn bản Y =(y0, x1,.., yn), độ dài n.

Output:

- Tất cả vị trí xuất hiện của X trong Y.

1.2 Phân loại các thuật toán đối sánh mẫu

Thuật toán đối sánh mẫu đầu tiên được đề xuất là Brute-Force. Thuật toán xác định vị trí xuất hiện của X trong Y với thời gian O(m.n). Nhiều cải tiến khác nhau của thuật toán Brute-Force đã được đề xuất nhằm cải thiện tốc độ tìm kiếm mẫu. Ta có thể phân loại các thuật toán tìm kiếm mẫu thành các lớp:

- **Tìm kiếm mẫu từ bên trái qua bên phải**: Harrison Algorithm, Karp-Rabin Algorithm, Morris-Pratt Algorithm, Knuth- Morris-Pratt Algorithm, Forward Dawg Matching algorithm , Apostolico-Crochemore algorithm, Naive algorithm.

- **Tìm kiếm mẫu từ bên phải qua bên trái**: Boyer-Moore Algorithm , Turbo BM Algorithm, Colussi Algorithm, Sunday Algorithm, Reverse Factorand Algorithm, Turbo Reverse Factor, Zhu and Takaoka and Berry-Ravindran Algorithms.
- **Tìm kiếm mẫu từ một vị trí cụ thể**: Two Way Algorithm, Colussi Algorithm , Galil-Giancarlo Algorithm, Sunday's Optimal Mismatch Algorithm, Maximal Shift Algorithm, Skip Search, KMP Skip Search and Alpha Skip Search Algorithms.
- **Tím kiếm mẫu từ bất kì :** Horspool Algorithm, Boyer-Moore Algorithm, Smith Algorithm , Raita Algorithm.

CHƯƠNG 2: TÌM KIẾM MẪU TỪ TRÁI SANG PHẢI

2.1 Karp-Rabin Algorithm

a. Phát biểu thuật toán

Thuật toán Karp-Rabin sử dụng hàm băm để so sánh giá trị băm của chuỗi trước khi thực hiện so sánh chuỗi.Phương pháp này giúp tiết kiệm được thời gian so sánh, đặc biệt với các chuỗi tìm kiếm dài.

Input:

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

Output:

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

- TH1: ts = p thực hiện phép đối sánh chuỗi giữa T[s. s+m-1] và P[0..m-1]
- TH2: $ts \neq p$ nếu $s \leq m$ tính gán s= s+1 và tính tiếp giá trị băm ts

b. Đánh giá độ phức tạp của thuật toán

- sử dụng 1 hàm băm để tìm chuỗi con

- độ phưc tạp thời gian và không gian tiền xử lý O(m)

- độ phưc tạp thời gian và khoong gian xử lý tìm kiếm O(m+n)

c. Kiểm nghiệm

input:

x = GCAGAGAG

y = GCATCGCAGAGAGTATACAGTACG

hash(GCAGAGAG) = 17819 (băm theo mã ASCII)

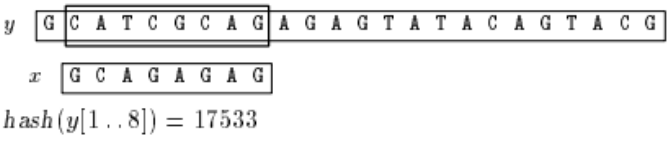
Luồng thuật toán

Searching phase

First attempt:

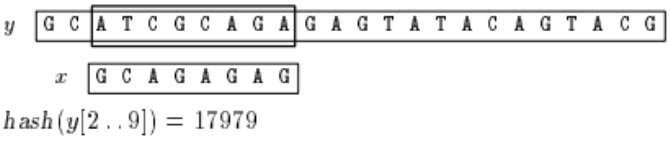
$hash(y[0\,..\,7]) = 17819$

Second attempt:



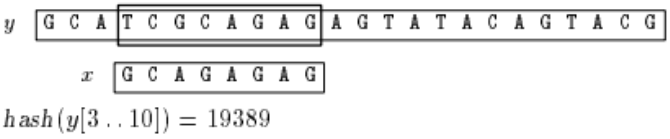
$hash(y[1\,..\,8]) = 17533$

Third attempt:



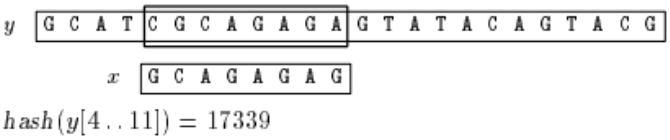
$hash(y[2\,..\,9]) = 17979$

Fourth attempt:



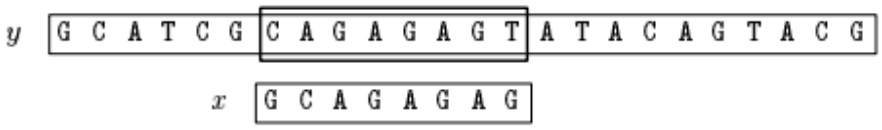
$hash(y[3\,..\,10]) = 19389$

Fifth attempt:



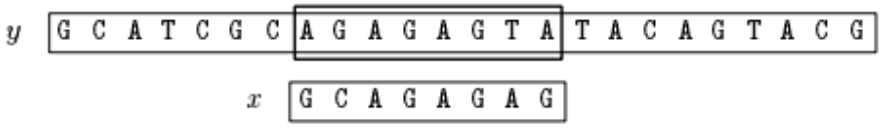
$hash(y[4\,..\,11]) = 17339$

Sixth attempt:



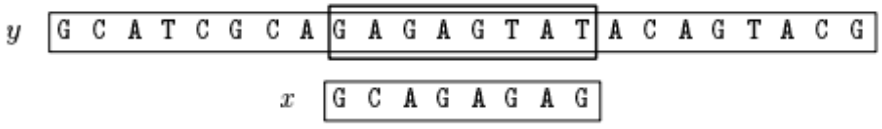
$hash(y[6\,..\,13]) = 17102$

Eighth attempt:



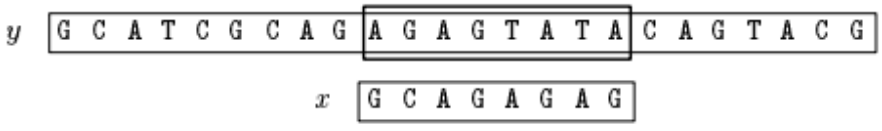
$hash(y[7\,..\,14]) = 17117$

Ninth attempt:



$hash(y[8\,..\,15]) = 17678$

Tenth attempt:



$hash(y[9\,..\,16]) = 17245$

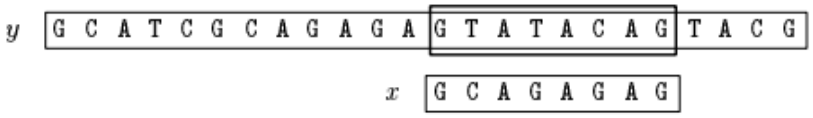
Eleventh attempt:

$hash(y[10\,..\,17]) = 17917$

Twelfth attempt:

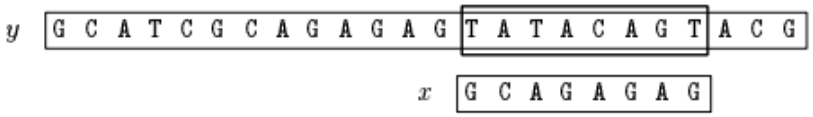
$hash(y[11\,..\,18]) = 17723$

Thirteenth attempt:



$hash(y[12\,..\,19]) = 18877$

Fourteenth attempt:



$hash(y[13\,..\,20]) = 19662$

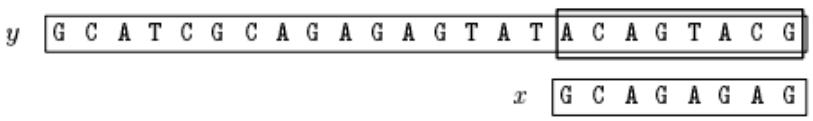
Fifteenth attempt:

$hash(y[14\,..\,21]) = 17885$

Sixteenth attempt:

$hash(y[15\,..\,22]) = 19197$

Seventeenth attempt:



$hash(y[16\,..\,23]) = 16961$

Kết luận: Mất 17 so sánh để so sánh hash chuỗi x với các đoạn của hash chuỗi y

d.Lập trình theo thuật toán

code C

```
#define REHASH(a, b, h) ((((h) - (a)*d) << 1) + (b))

void KR(char *x, int m, char *y, int n) {
   int d, hx, hy, i, j;

   /* Preprocessing */
   /* computes d = 2^(m-1) with
      the left-shift operator */
   for (d = i = 1; i < m; ++i)
      d = (d<<1);

   for (hy = hx = i = 0; i < m; ++i) {
      hx = ((hx<<1) + x[i]);
      hy = ((hy<<1) + y[i]);
   }

   /* Searching */
   j = 0;
   while (j <= n-m) {
      if (hx == hy && memcmp(x, y + j, m) == 0)
         OUTPUT(j);
      ++j;
      hy = REHASH(y[j], y[j + m], hy);
   }
}
```

2.2 Morris-Pratt Algorithm

a.Trình bày thuật toán

– Morris pratt là một phân tích chặt chẽ của thuật toán Brute force, đặc biệt là với phương pháp này đã sử dụng các thông tin thu thập đc trong quá trình quét văn bản.
– Nhược điểm của phương pháp Brute force là chúng ta phải tốn công so sánh lại những kí tự trước đó.
Input:
Xâu mẫu x=(x0,x1,…,xm-1) độ dài m
Xâu văn bản: y= (y0, y1,…, yn-1) độ dài n
Ouput: tất cả các vị trí của x trong y

b.Đánh giá độ phức tạp của thuật toán

- Giai đoạn tiền xử lí trong không gian O(m) và độ phức tạp thời gian
- Giai đoạn tìm kiếm có độ phức tạp thời gian O(m+n)
- Giai đoạn biểu diễn là 2n-1 phép so sánh văn bản chữ cái

- Giới hạn trễ là m

c. Kiểm nghiệm theo thuật toán

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$mpNext[i]$	−1	0	0	0	1	0	1	0	1

Searching phase

First attempt:

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

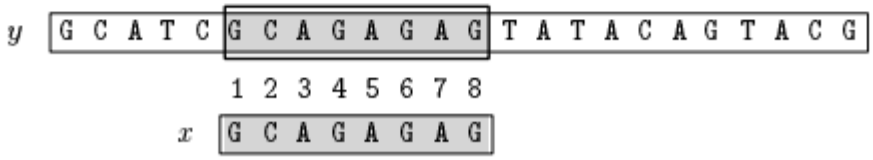
Second attempt:

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

Third attempt:

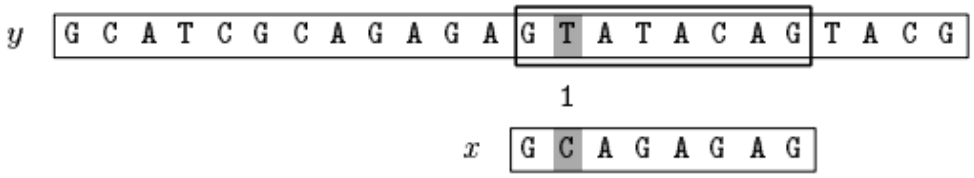
Shift by 1 ($i - mpNext[i] = 0 - -1$)

Fourth attempt:



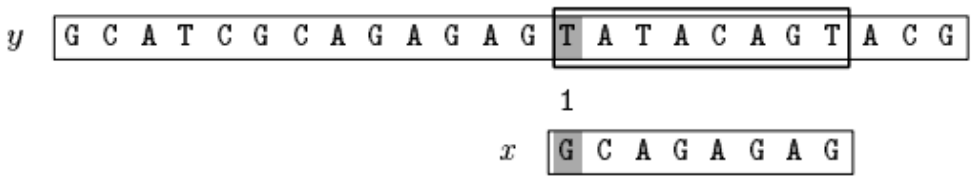
Shift by 7 ($i - mpNext[i] = 8 - 1$)

Fifth attempt:



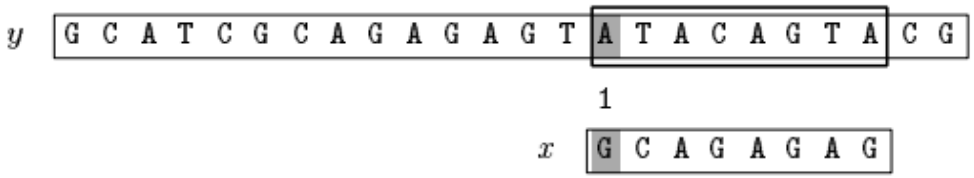
Shift by 1 $(i - mpNext[i] = 1 - 0)$

Sixth attempt:



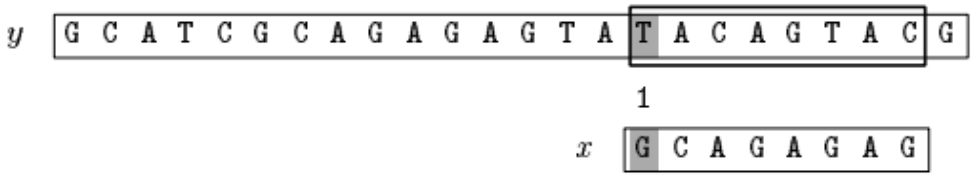
Shift by 1 $(i - mpNext[i] = 0 - -1)$

Seventh attempt:



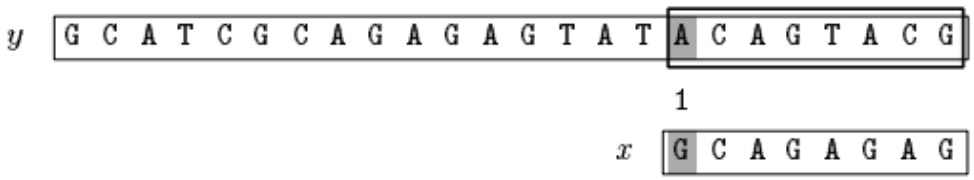
Shift by 1 $(i - mpNext[i] = 0 - -1)$

Eighth attempt:



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

Ninth attempt:



Kết luận : thuật toán sử dụng 19 phép so sánh các chữ trong văn bản

d. Lập trình theo thuật toán

```
void preMp(char *x, int m, int mpNext[]) {
   int i, j;

   i = 0;
   j = mpNext[0] = -1;
   while (i < m) {
      while (j > -1 && x[i] != x[j])
         j = mpNext[j];
      mpNext[++i] = ++j;
   }
}


void MP(char *x, int m, char *y, int n) {
   int i, j, mpNext[XSIZE];

   /* Preprocessing */
   preMp(x, m, mpNext);

   /* Searching */
   i = j = 0;
   while (j < n) {
      while (i > -1 && x[i] != y[j])
         i = mpNext[i];
      i++;
```

```
            j++;
            if (i >= m) {
               OUTPUT(j - i);
               i = mpNext[i];
            }
         }
      }
```

2.3 Knuth- Morris-Pratt Algorithm

a. Trình bày thuật toán

– Knuth Morris Pratt là một phân tích chặt chẽ của thuật toán Morris pratt.
– Bảng kmpNext giúp loại bỏ việc so sánh lại những kí tự đã so sánh trước đó trong phương pháp thông thường.
– Thực hiện: Dò từ trái sang phải cho tới khi gặp vị trí X và Y không giống nhau. Giả sử: X[i] != Y[j], xem giá trị kmpNext[i] tương ứng.
+) Nếu kmpNext[i] = -1 thì dịch chuỗi X lên một bước.
+) Nếu kmpNext[i] = m, >= 0 thì ta di chuyển X[m] trùng với i.
Input:
Xâu mẫu x=(x0,x1,…,xm-1) độ dài m
Xâu văn bản: y= (y0, y1,…, yn-1) độ dài n
Ouput: tất cả các vị trí của x trong y

b. Đánh giá độ phức tạp của thuật toán

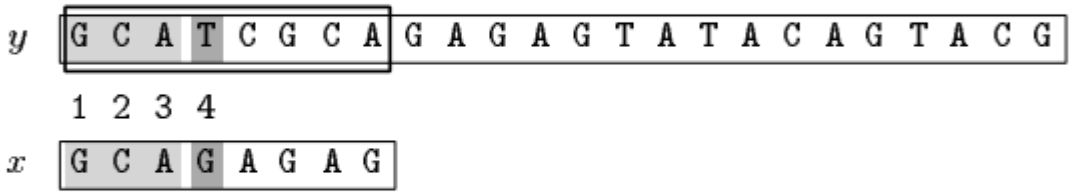
– Pha tiền xử lí có độ phức tạp về không gian và thời gian là O(m)
– Pha tìm kiếm có độ phức tạp về thời gian O(m+n)

c. Kiểm nghiệm thuật toán

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$kmpNext[i]$	-1	0	0	-1	1	-1	1	-1	1

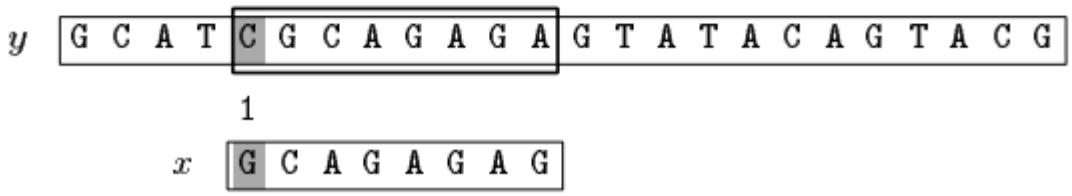
Searching phase

First attempt:



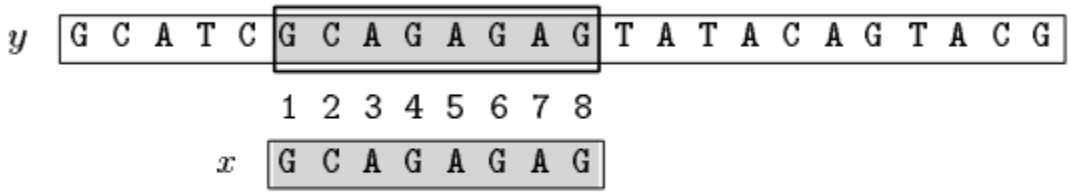
Shift by 4 ($i - kmpNext[i] = 3 - -1$)

Second attempt:



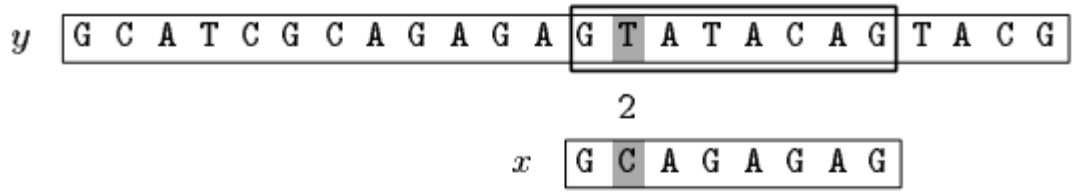
Shift by 1 ($i - kmpNext[i] = 0 - -1$)

Third attempt:



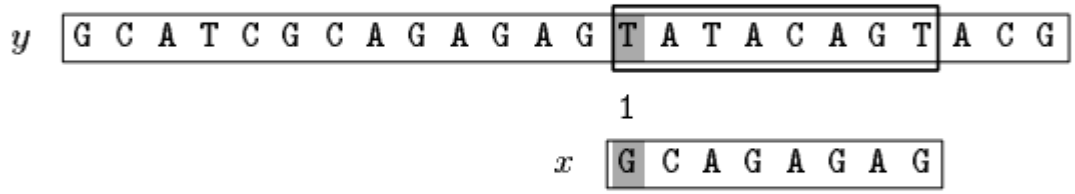
Shift by 7 ($i - kmpNext[i] = 8 - 1$)

Fourth attempt:



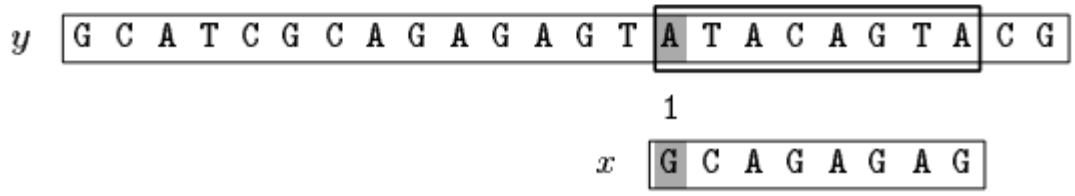
Shift by 1 ($i - kmpNext[i] = 1 - 0$)

Fifth attempt:



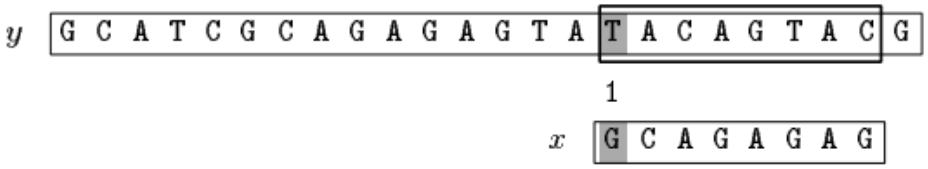
Shift by 1 ($i - kmpNext[i] = 0 - -1$)

Sixth attempt:



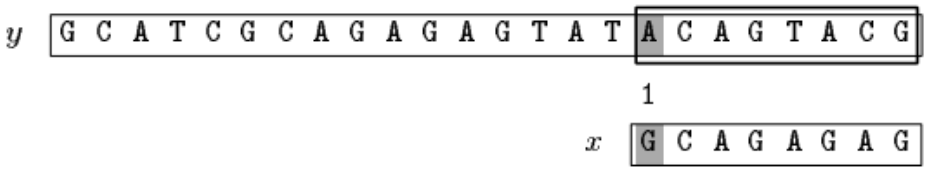
Shift by 1 ($i - kmpNext[i] = 0 - -1$)

Seventh attempt:



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

Eighth attempt:



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

The Knuth-Morris-Pratt algorithm performs 18 text character comparisons on the example.

d. Lập trình theo thuật toán

code java

```
public class KnuthMorrisPratt {
public static int[] preKMP(char[] x) {
int[] kmpNext = new int[x.length + 1];
int i = 0;
int j = -1;
kmpNext[0] = -1;
while (i < x.length - 1) {
while (j > -1 && x[i] != x[j]) {
j = kmpNext[j];
}
i++;
j++;
if (x[i] == x[j]) {
kmpNext[i] = kmpNext[j];
} else {
kmpNext[i] = j;
}
}

return kmpNext;
}
```

```
public static void search(char[] x, char[] y) {
int[] kmpNext = preKMP(x);
int i = 0;// the position of character in x
int m = 0;// the beginning of the current match in y
while (m <= y.length - x.length) {
if (x[i] == y[m + i]) {
i++;
if (i == x.length) {
System.out.print(m + " ");
m = m + i - kmpNext[i];
i = kmpNext[i];
}
} else {
m = m + i - kmpNext[i];
i = 0;
}

}
}

public static void main(String[] args) {
String x = "GCAGAGAG";
char[] X = x.toCharArray();
String y = "GCATCGCAGAGAGTATACAGTACG";
char[] Y = y.toCharArray();
System.out.print("Các vị trí xuất hiện của x trong y là: ");
search(X, Y);
}
}
```

2.4 Apostolico-Crochemore algorithm

a.Trình bày thuật toán

Đây là một thuật toán so sánh chuỗi tìm kiếm mẫu từ bên trái sang phải
Lấy l=0 nếu x là một tập của cùng một ký tự và l được tính bằng vị trí của ký tự đầu tiên khác x[0] trong các trường hợp khác.
Trong suốt thuật toán xét 3 trường hợp:
-Vị trí xét nằm trong khoảng 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-l]
Khởi tạo (l,0,0)
Với mỗi attempt xét bộ ba (i,j,k)
Việc xét bộ tiếp theo dựa vào vị trí của i

-i=l
o nếu x[i]=y[i+j] bộ tiếp theo (i+1,j,k)
o nếu x[i]<>y[i+j] bộ tiếp theo (l,j+1,max(0,k-1))
-l<i<m
o nếu x[i]=y[i+j] bộ tiếp theo (i+1,j,k)
o nếu x[i]<>y[i+j] thì có 2 trường hợp được xét đến dựa vào kmpNext
·kmpNext[i]<l : (l,i+j-kmpNext[i],max(0,kmpNext[i]))
·kmpNext[i]>l : (kmpNext[i],i+j-kmpNext[i],l)
-i=m
o k<l và x[k]=y[j+k]: (I,j,k+1)
o Nếu một trong hai trường hợp k<l và x[k]<>y[j+k] hoặc k=l. Nếu k=l thì thông báo một xuất hiện của x. Trong cả 2 trường hợp bộ tiếp theo được tính toán giống như trường hợp l<i<m
Thuật toán PreKmp: //thực hiện bước tiền xử lý
Input:
- Xâu mẫu X =(x0, x1,..,xm), độ dài m.
- Văn bản Y =(y0, y1,..,xn), độ dài n.
Output:
-Mọi vị trí xuất hiện của X trong Y.

b. Độ phức tạp của thuật toán

- Có pha tiền xử lý với độ phức tạp O(m)
- Độ phức tạp thuật toán là O(n)

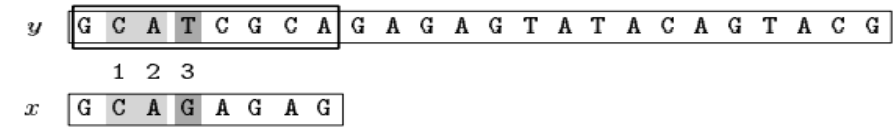
c. Kiểm nghiệm thuật toán

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$kmpNext[i]$	-1	0	0	-1	1	-1	1	-1	1

$\ell = 1$

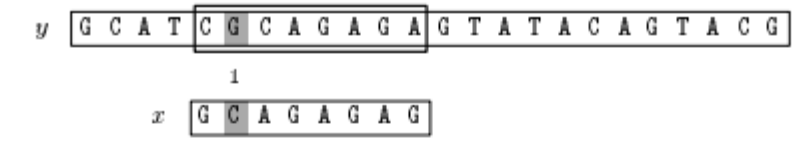
Searching phase

First attempt:



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

Second attempt:

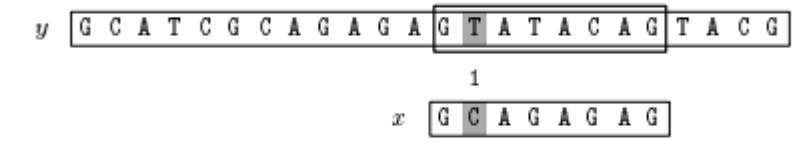


Shift by 1 ($i - kmpNext[i] = 1 - 0$)

Third attempt:

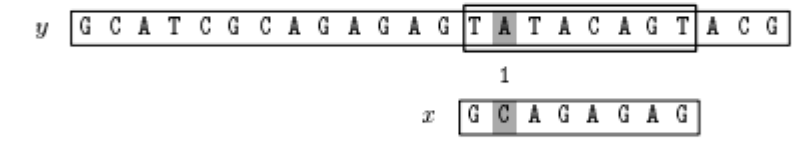
Shift by 7 ($i - kmpNext[i] = 8 - 1$)

Fourth attempt:



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

Fifth attempt:



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

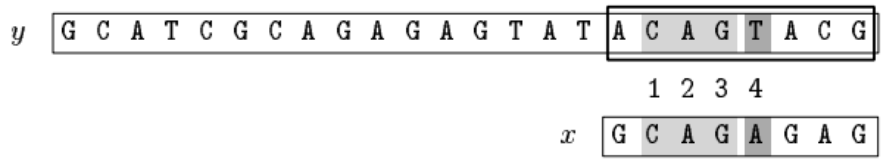
Sixth attempt:

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

Seventh attempt:

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

Eighth attempt:



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

Kết luật : thuật toán dùng đến 20 phép so sánh các chữ cái trong văn bản

d. Lập trình thuật toán

Formats: AXAMAC(X, m, Y, n);
Actions:
Bước 1(Tiền xử lý):
int i,j,k,ell;
preKmp(x, m, kmpNext); //Tiền xử lý với độ phức tạp O(m)
for(ell=1;x[ell-1]==x[ell];ell++);
if(ell==m) ell=0;
i=ell;
j=k=0;
30
Bước 2 (Lặp)
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)OUTPUT (j);
}
j=j+(i-kmpNext[i-1]);
if(i==ell) k=Max(0, k-1);
else if(kmpNext[i-1]<=ell){
k=Max(0, kmpNext[i-1]);
i=ell;
}else{
k=ell;
i=kmpNext[i-1];
}

}
EndActions.

2.5 Naive algorithm

a. Trình bày thuật toán

Tìm kiếm mẫu đơn giản là phương pháp đơn giản nhất trong số các thuật toán tìm kiếm mẫu khác. Nó kiểm tra tất cả các ký tự của chuỗi chính đối với mẫu. Thuật toán này rất hữu ích cho các văn bản nhỏ hơn. Nó không cần bất kỳ giai đoạn xử lý trước nào. Chúng ta có thể tìm thấy chuỗi con bằng cách kiểm tra chuỗi một lần. Nó cũng không chiếm thêm không gian để thực hiện các hoạt động.

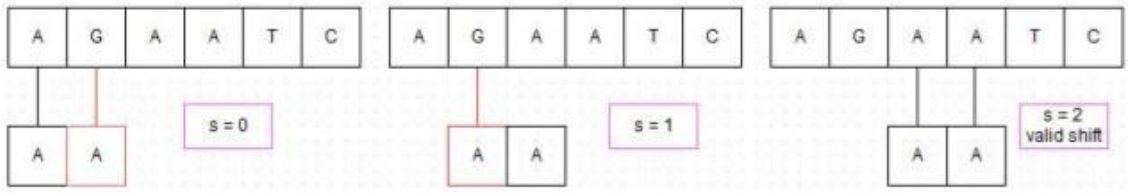
b. Độ phức tạp của thuật toán

Độ phức tạp về thời gian của phương pháp Naïve Pattern Search là O (m * n). M là kích thước của mẫu và n là kích thước của chuỗi chính.

Input - Văn bản và mẫu

Output - vị trí, nơi các mẫu hiển thị trong văn bản

c. kiểm nghiệm thuật toán



d. Lập trình thuật toán

code c++

```
#include<iostream>

using namespace std;


void naivePatternSearch(string mainString, string pattern, int
array[], int *index) {

   int patLen = pattern.size();

   int strLen = mainString.size();


   for(int i = 0; i<=(strLen - patLen); i++) {
```

```
      int j;
      for(j = 0; j<patLen; j++) {      //check for each character
of pattern if it is matched
         if(mainString[i+j] != pattern[j])
            break;
      }

      if(j == patLen) {     //the pattern is found
         (*index)++;
         array[(*index)] = i;
      }
   }
}

int main() {
   string mainString = "ABAAABCDBBABCDDEBCABC";
   string pattern = "ABC";
   int locArray[mainString.size()];
   int index = -1;
   naivePatternSearch(mainString, pattern, locArray, &index);

   for(int i = 0; i <= index; i++) {
      cout << "Pattern found at position: " << locArray[i]<<endl;
   }
}
```

CHƯƠNG 3: THUẬT TOÁN TÌM KIẾM MẪU TỪ PHẢI SANG TRÁI

3.1 Boyer Moore algorithm

a. 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.

Fig2: good-suffix shift, trường hợp u lại xuất hiện trong xNếu 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.

Fig3: good-suffix shift, trường hợp chỉ suffix của u xuất hiện trong 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] (các bạn thử nghĩ xem tại sao không phải là m-1)

Fig4: bad-character shift

- 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:

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

b. Độ phức tạp của thuật toán

- 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.

c. Kiểm nghiệm thuật toán

X = “GCAGAGAG”

Y =”GCATCGCAGAGAGTATACAGTACG”
Pha tiền xử lí:
bmBc and bmGs tables used by Boyer-Moore algorithm
Pha thực thi:
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
1
G C A G A G A G
Shift by: 1 (bmGs[7]=bmBc[A]-8+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
3 2 1
G C A G A G A G
Shift by: 4 (bmGs[5]=bmBc[C]-8+6
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
8 7 6 5 4 3 2 1
G C A G A G A G
Shift by: 7 (bmGs[0])
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
3 2 1
G C A G A G A G
Shift by: 4 (bmGs[5]=bmBc[C]-8+6)
Fifth 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
Shift by: 7 (bmGs[6])
Boyer Moore thực thi 17 lần so sánh kí tự trong ví dụ này.

d. Lập trình theo thuật toán

code c++

```
#include<iostream>
#include<algorithm>
#include<iomanip>
#include<cstdio>
#include<cstring>
#define For(i,a,b) for(long i = a;i<=b;i++)
using namespace std;
char x[100001],y[100001];
int m, n, ASIZE = 256,XSIZE;
void nhap(){
printf("Nhap x : "); gets(x); m = strlen(x); XSIZE = m;
printf("Nhap y : "); gets(y); n = strlen(y);
}
void preBmBc(char *x, int m, int bmBc[]) {
int i;
for (i = 0; i < ASIZE; ++i)
bmBc[i] = m;
for (i = 0; i < m - 1; ++i)
bmBc[x[i]] = m - i - 1;
}
void suffixes(char *x, int m, int *suff) {
int f, g, i;
suff[m - 1] = m;
g = m - 1;
for (i = m - 2; i >= 0; --i) {
if (i > g && suff[i + m - 1 - f] < i - g)
suff[i] = suff[i + m - 1 - f];
else {
```

```
if (i < g)
g = i;
f = i;
while (g >= 0 && x[g] == x[g + m - 1 - f])
--g;
suff[i] = f - g;
}
}
}
void preBmGs(char *x, int m, int bmGs[]) {
int i, j, suff[XSIZE];
suffixes(x, m, suff);
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;
}
void BM(char *x, int m, char *y, int n) {
int i, j, bmGs[XSIZE], bmBc[ASIZE];
/* Preprocessing */
preBmGs(x, m, bmGs);
preBmBc(x, m, bmBc);
/* Searching */
j = 0;
while (j <= n - m) {
for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);
if (i < 0) {
printf("position is %d\n",j);
j += bmGs[0];
}
else
j += max(bmGs[i], bmBc[y[i + j]] - m + 1 + i);
}
}
```

```
int main(){
nhap();
BM(x,m,y,n);
return 0;
}
```
Chạy test : Input : Nhập x : GCAGAGAG Nhập y :
GCATCGCAGAGAGTATACAGTACG Output: Position is 5

3.2 Zhu Kataoka algorithm

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

b. Độ phức tạp của thuật toán

- 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+σ2)

Pha thực thi có độ phức tạp là O(mxn)

c.Kiểm nghiệm thuật toán

X = “GCAGAGAG”
Y =”GCATCGCAGAGAGTATACAGTACG”
Pha tiền xử lí:
ztBc and bmGs tables used by Zhu-Takaoka algorithm.
Pha tìm kiếm thực hiện như sau :
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
1
G C A G A G A G
Shift by: 5 (ztBc[C][A])
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

8 7 6 5 4 3 2 1
G C A G A G A G
Shift by: 7 (bmGs[0])
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
3 2 1
G C A G A G A G
Shift by: 7 (bmGs[6])

d. Lập trình thuật toán

code c++

```
#include<iostream>
#include<algorithm>
#include<iomanip>
#include<cstdio>
#include<cstring>
#define For(i,a,b) for(long i = a;i<=b;i++)
using namespace std;
char x[100001],y[100001];
int m, n,ASIZE = 256, XSIZE;
void nhap(){
printf("Nhap x : "); gets(x); m = strlen(x); XSIZE = m;
printf("Nhap y : "); gets(y); n = strlen(y);
}
void suffixes(char *x, int m, int *suff) {
int f, g, i;
suff[m - 1] = m;
g = m - 1;
for (i = m - 2; i >= 0; --i) {
if (i > g && suff[i + m - 1 - f] < i - g)
suff[i] = suff[i + m - 1 - f];
else {
if (i < g)
g = i;
f = i;
while (g >= 0 && x[g] == x[g + m - 1 - f])
--g;
suff[i] = f - g;
}
}
}
```

```
void preBmGs(char *x, int m, int bmGs[]) {
int i, j, suff[XSIZE];
suffixes(x, m, suff);
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;
}
void preZtBc(char *x, int m, int ztBc[256][256]) {
int i, j;
for (i = 0; i < ASIZE; ++i)
for (j = 0; j < ASIZE; ++j)
ztBc[i][j] = m;
for (i = 0; i < ASIZE; ++i)
ztBc[i][x[0]] = m - 1;
for (i = 1; i < m - 1; ++i)
ztBc[x[i - 1]][x[i]] = m - 1 - i;
}
void ZT(char *x, int m, char *y, int n) {
int i, j, ztBc[256][256], bmGs[XSIZE];
/* Preprocessing */
preZtBc(x, m, ztBc);
preBmGs(x, m, bmGs);
/* Searching */
j = 0;
while (j <= n - m) {
i = m - 1;
while (i < m && x[i] == y[i + j])
--i;
if (i < 0) {
printf("position is %d\n",j);
j += bmGs[0];
}
else
```

```
j += max(bmGs[i],
ztBc[y[j + m - 2]][y[j + m - 1]]);
}
}
int main(){
nhap();
ZT(x,m,y,n);
return 0;
}
```

Chạy test :

Input :

Nhập x : GCAGAGAG

Nhập y : GCATCGCAGAGAGTATACAGTACG

Output: Position is 5

3.3 Colussi Algorithm

a. Trình bày thuật toán

Thiết kế của thuật toán Colussi tuân theo một phân tích chặt chẽ của thuật toán Knuth, Morris và Pratt.

Tập hợp các vị trí mẫu được chia thành hai tập con rời rạc. Sau đó, mỗi lần thử bao gồm hai giai đoạn:

trong giai đoạn đầu, các phép so sánh được thực hiện từ trái sang phải với các ký tự văn bản được căn chỉnh với vị trí mẫu mà giá trị của hàm kmpNext lớn hơn -1. Những vị trí này được gọi là lỗ hổng;

giai đoạn thứ hai bao gồm so sánh các vị trí còn lại (được gọi là lỗ) từ phải sang trái.

Chiến lược này có hai ưu điểm:

khi sự không khớp xảy ra trong giai đoạn đầu tiên, sau khi thay đổi thích hợp, không cần thiết phải so sánh các ký tự văn bản được căn chỉnh với các lỗ hổng so với trong lần thử trước đó;

khi sự không khớp xảy ra trong giai đoạn thứ hai, điều đó có nghĩa là một hậu tố của mẫu khớp với một yếu tố của văn bản, sau khi dịch chuyển tương ứng, một tiền tố của mẫu vẫn sẽ khớp với một yếu tố của văn bản, khi đó không cần thiết phải so sánh yếu tố này lần nữa.

b. Độ phức tạp của thuật toán

- Cải tiến thuật toán Knuth, Morris và Pratt;
- Phân vùng tập hợp các vị trí mẫu thành hai tập con rời rạc; các vị trí trong tập hợp đầu tiên được quét từ trái sang phải và khi không có sự sai lệch nào xảy ra, các vị trí của tập hợp con thứ hai được quét từ phải sang trái;
- Giai đoạn tiền xử lý theo thời gian và không gian phức tạp O (m);

- Pha tìm kiếm theo độ phức tạp thời gian O (n);
- Thực hiện so sánh ký tự văn bản 3 / 2n trong trường hợp xấu nhất.

c. Kiểm nghiệm thuật toán

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$kmpNext[i]$	-1	0	0	-1	1	-1	1	-1	1
$kmin[i]$	0	1	2	0	3	0	5	0	
$h[i]$	1	2	4	6	7	5	3	0	
$next[i]$	0	0	0	0	0	0	0	0	0
$shift[i]$	1	2	3	5	8	7	7	7	7
$hmax[i]$	0	1	2	4	4	6	6	8	8
$rmin[i]$	7	0	0	7	0	7	0	8	
$ndh0[i]$	0	0	1	2	2	3	3	4	

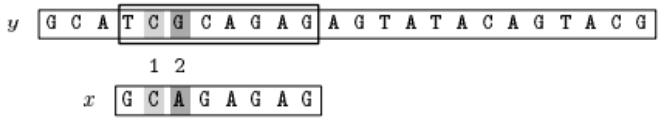
$nd = 3$

Searching phase

First attempt:

Shift by 3 ($shift[2]$)

Second attempt:

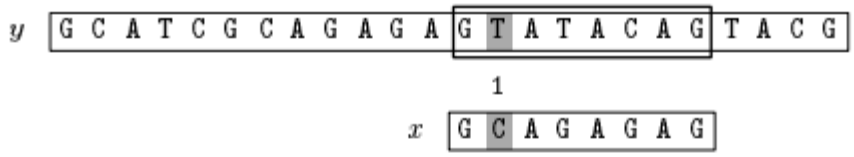


Shift by 2 ($shift[1]$)

Third attempt:

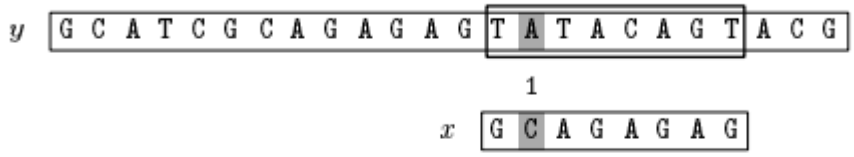
Shift by 7 ($shift[8]$)

Fourth attempt:



Shift by 1 ($shift[0]$)

Fifth attempt:



Shift by 1 ($shift[0]$)

Sixth attempt:

Shift by 1 ($shift[0]$)

Seventh attempt:

Shift by 1 ($shift[0]$)

Eighth attempt:

Shift by 3 ($shift[2]$)

Kết luận: thuật toán biểu diễn 20 so sánh ký tự văn bản

d. Lập trình thuật toán

code c

```
int preColussi(char *x, int m, int h[], int next[],
               int shift[]) {
   int i, k, nd, q, r, s;
   int hmax[XSIZE], kmin[XSIZE], nhd0[XSIZE], rmin[XSIZE];

   /* Computation of hmax */
   i = k = 1;
   do {
      while (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 */
   memset(kmin, 0, m*sizeof(int));
   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);
}

void COLUSSI(char *x, int m, char *y, int n) {
   int i, j, last, nd,
       h[XSIZE], next[XSIZE], shift[XSIZE];

   /* Processing */
   nd = preColussi(x, m, h, next, shift);

   /* Searching */
   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]) {
         OUTPUT(j);
         i = m;
      }
      if (i > nd)
         last = j + m - 1;
      j += shift[i];
      i = next[i];
   }
}
```

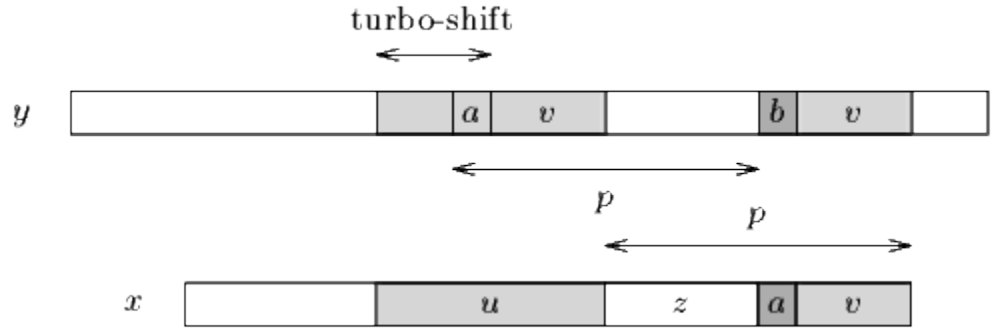
3.4 Turbo BM Algorithm
a. trình bày thuật toán

Thuật toán Turbo-BM là sự cải tiến của thuật toán Boyer-Moore. Nó không cần tiền xử lý thêm và chỉ yêu cầu một không gian bổ sung liên tục đối với thuật toán Boyer-Moore ban đầu. Nó bao gồm việc ghi nhớ yếu tố của văn bản khớp với một hậu tố của mẫu trong lần thử cuối cùng (và chỉ khi thực hiện chuyển đổi hậu tố tốt).

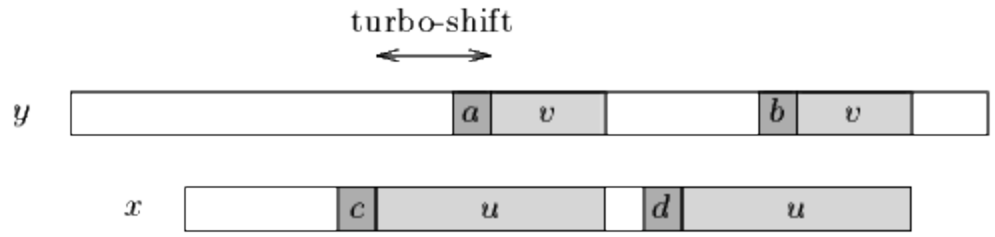
Kỹ thuật này có hai ưu điểm:

- có thể nhảy qua yếu tố này;
- nó có thể cho phép thực hiện chuyển đổi turbo.

Một turbo-shift có thể xảy ra nếu trong lần thử hiện tại, hậu tố của mẫu khớp với văn bản ngắn hơn hậu tố được ghi nhớ từ lần thử trước. Trong trường hợp này, chúng ta hãy gọi u là thừa số được nhớ và v là hậu tố được so khớp trong lần thử hiện tại sao cho uzv là hậu tố của x. Gọi a và b là các ký tự gây ra sự không khớp trong lần thử hiện tại trong mẫu và văn bản tương ứng. Khi đó av là hậu tố của x, và do đó của u vì | v | <| u |. Hai ký tự a và b xuất hiện ở khoảng cách p trong văn bản và là hậu tố x có độ dài | uzv | có chu kỳ độ dài p = | zv | vì u là một đường viền của uzv, do đó nó không thể trùng lặp cả hai lần xuất hiện của hai ký tự a và b khác nhau, ở khoảng cách p, trong văn bản. Sự dịch chuyển nhỏ nhất có thể có độ dài | u | - | v |, chúng ta gọi là sự dịch chuyển turbo.



Vẫn trong trường hợp | v | <| u | nếu độ dài của dịch chuyển ký tự xấu lớn hơn độ dài của dịch chuyển hậu tố tốt và độ dài của dịch chuyển tăng áp thì độ dài của dịch chuyển thực tế phải lớn hơn hoặc bằng | u | +1. Thật vậy, trong trường hợp này, hai ký tự c và d khác nhau vì chúng ta đã giả định rằng lần dịch chuyển trước đó là sự thay đổi hậu tố tốt.



Khi đó, một sự thay đổi lớn hơn sự dịch chuyển turbo nhưng nhỏ hơn | u | +1 sẽ căn chỉnh c và d với cùng một ký tự trong câu v. Vì vậy, nếu trong trường hợp này, độ dài của sự thay đổi thực tế ít nhất phải bằng | u | +1 .

Giai đoạn tiền xử lý có thể được thực hiện với độ phức tạp về thời gian và không gian O (m + sigma). Giai đoạn tìm kiếm có độ phức tạp thời gian O (n). Số lượng so sánh ký tự văn bản được thực hiện bởi thuật toán Turbo-BM được giới hạn bởi 2n.

b. Độ phức tạp của thuật toán.

- biến thể của Boyer-Moore;
- không cần xử lý trước bổ sung đối với thuật toán Boyer-Moore;
- không gian bổ sung liên tục cần thiết đối với thuật toán Boyer-Moore;
- giai đoạn tiền xử lý theo thời gian và không gian phức tạp O (m + sigma);
- pha tìm kiếm theo độ phức tạp thời gian O (n);
- 2n so sánh ký tự văn bản trong trường hợp xấu nhất.

c. Kiểm nghiệm thuật toán

a	A	C	G	T
$bmBc[a]$	1	6	2	8

i	0	1	2	3	4	5	6	7
$x[i]$	G	C	A	G	A	G	A	G
$suff[i]$	1	0	0	2	0	4	0	8
$bmGs[i]$	7	7	7	2	7	4	7	1

Searching phase

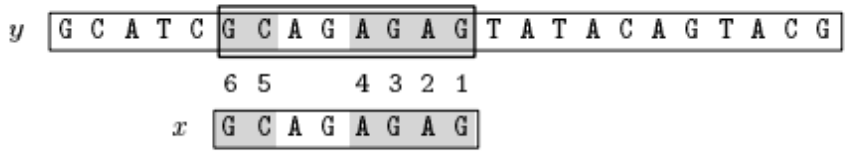
First attempt:

Shift by 1 ($bmGs[7] = bmBc[\mathtt{A}] - 7 + 7$)

Second attempt:

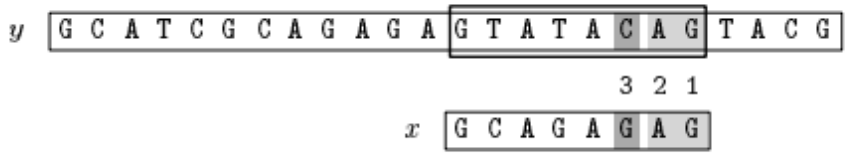
Shift by 4 ($bmGs[5] = bmBc[\mathtt{C}] - 7 + 5$)

Third attempt:



Shift by 7 ($bmGs[0]$)

Fourth attempt:



Shift by 4 ($bmGs[5] = bmBc[\mathtt{C}] - 7 + 5$)

Fifth attempt:

Shift by 7 ($bmGs[6]$)

Kết luận: thuật toán biểu diễn 15 so sánh đoạn văn bản

d. Lập trình thuật toán

```
void TBM(char *x, int m, char *y, int n) {
   int bcShift, i, j, shift, u, v, turboShift,
       bmGs[XSIZE], bmBc[ASIZE];

   /* Preprocessing */
   preBmGs(x, m, bmGs);
   preBmBc(x, m, bmBc);

   /* Searching */
   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) {
         OUTPUT(j);
         shift = bmGs[0];
         u = m - shift;
      }
      else {
         v = m - 1 - i;
         turboShift = u - v;
         bcShift = bmBc[y[i + j]] - m + 1 + i;
         shift = MAX(turboShift, bcShift);
         shift = MAX(shift, bmGs[i]);
         if (shift == bmGs[i])
            u = MIN(m - shift, v);
         else {
           if (turboShift < bcShift)
              shift = MAX(shift, u + 1);
           u = 0;
         }
      }
      j += shift;
   }
}
```

3.5 Turbo Reverse Factor

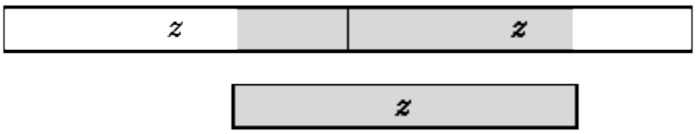
a. Trình bày thuật toán

Có thể làm cho thuật toán Nhân tố ngược trở nên tuyến tính. Trên thực tế, nó đủ để nhớ tiền tố u của x được khớp trong lần thử cuối cùng. Sau đó, trong quá trình cố gắng hiện tại khi đến đầu bên phải của u, có thể dễ dàng chứng minh rằng chỉ cần

đọc lại nhiều nhất là nửa ngoài cùng bên phải của u là đủ. Điều này được thực hiện bởi thuật toán Turbo Reverse Factor.

Nếu một từ z là một thừa số của một từ w, chúng ta định nghĩa disp (z, w) độ dời của z theo w là số nguyên nhỏ nhất d> 0 sao cho w [md- | z | -1 .. md] = z .

Tình huống chung của thuật toán Turbo Reverse Factor là khi tiền tố u được tìm thấy trong văn bản trong lần thử cuối cùng và đối với lần thử hiện tại, thuật toán cố gắng khớp với yếu tố v có độ dài m- | u | trong văn bản ngay bên phải của u. Nếu v không phải là nhân tố của x thì sự dịch chuyển được tính như trong thuật toán Nhân tố ngược. Nếu v là hậu tố của x thì x đã được tìm thấy. Nếu v không phải là hậu tố mà là hệ số của x thì chỉ cần quét lại min (per (u), | u | / 2) ký tự ngoài cùng bên phải của u là đủ. Nếu u là tuần hoàn (tức là per (u) leq | u | / 2) thì gọi z là hậu tố của u có độ dài per (u). Theo định nghĩa của chu kỳ z là một từ xoay chiều và khi đó sự chồng chéo như trong Hình 23.1 là không thể.



Vì vậy, z chỉ có thể xảy ra trong u ở khoảng cách bội số của per (u), điều này ngụ ý rằng hậu tố thích hợp nhỏ nhất của uv là tiền tố của x có độ dài bằng | uv | -disp (zv, x) = m-disp (zv, x). Do đó độ dài của dịch chuyển để thực hiện là disp (zv, x).

Nếu u không tuần hoàn (per (u)> | u | / 2), rõ ràng là x không thể tái xuất hiện ở phần bên trái của u có độ dài per (u). Sau đó, đủ để quét phần bên phải của u có độ dài | u | -per (u) <| u | / 2 để tìm một chuyển tiếp không xác định trong automaton.

Hàm disp được thực hiện trực tiếp trong automaton S (x) mà không làm thay đổi độ phức tạp của cấu trúc của nó.

Giai đoạn tiền xử lý bao gồm xây dựng tự động hóa hậu tố của X^R. Nó có thể được thực hiện với độ phức tạp thời gian O (m).

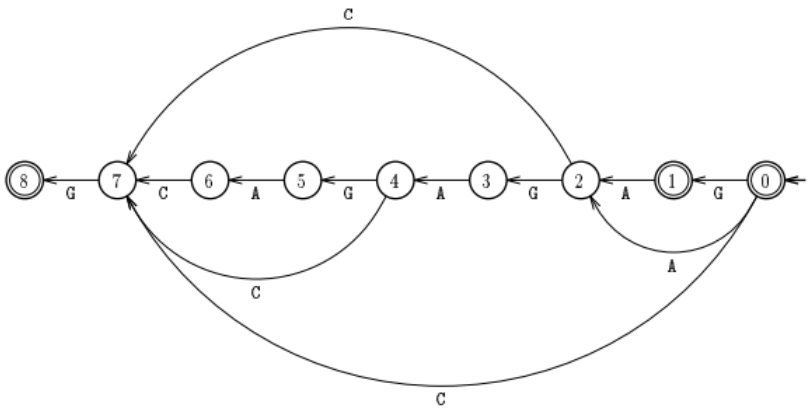
Giai đoạn tìm kiếm có độ phức tạp thời gian O (n). Turbo Reverse Factor thực hiện nhiều nhất 2n lần kiểm tra các ký tự văn bản và nó cũng tối ưu trong việc kiểm tra O (n.logsigma (m) / m) thực hiện trung bình đối với các ký tự văn bản ở mức trung bình đạt đến giới hạn tốt nhất được hiển thị bởi Yao vào năm 1979.

b. Độ phức tạp của thuật toán

- Tinh chỉnh của thuật toán Nhân tố ngược;
- Giai đoạn tiền xử lý theo thời gian và không gian phức tạp O (m);
- Pha tìm kiếm theo độ phức tạp thời gian O (n);
- Thực hiện kiểm tra 2n ký tự văn bản trong trường hợp xấu nhất;
- Tối ưu trong mức trung bình.

c.Kiểm nghiệm thuật toán

$\mathcal{L}(\mathcal{S}) = \{\texttt{GCAGAGAG}, \texttt{GCAGAGA}, \texttt{GCAGAG}, \texttt{GCAGA}, \texttt{GCAG}, \texttt{GCA}, \texttt{GC}, \texttt{G}, \varepsilon\}$



Searching phase

The initial state is 0
First attempt:

```
y  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 2
x  G C A G A G A G
```

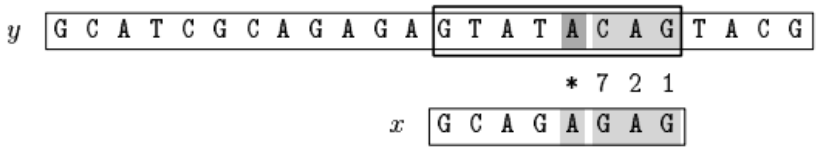
Shift by 5 (8-3)

Second attempt:

```
y  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
         x G C A G A G A G
```

Shift by 7 (8-1)

Third attempt:



Kết luận: biểu diễn 17 lần điều tra ký tự văn bản

d. Lập trình thuật toán

```
void TRF(char *x, int m, char *y, int n) {
   int period, i, j, shift, u, periodOfU, disp, init,
       state, mu, mpNext[XSIZE + 1];
   char *xR;
   Graph aut;

   /* Preprocessing */
   aut = newSuffixAutomaton(2*(m + 2), 2*(m + 2)*ASIZE);
   xR = reverse(x, m);
   buildSuffixAutomaton(xR, m, aut);
   init = getInitial(aut);
   preMp(x, m, mpNext);
   period = m - mpNext[m];
   i = 0;
   shift = m;

   /* Searching */
   j = 0;
   while (j <= n - m) {
      i = m - 1;
      state = init;
      u = m - 1 - shift;
      periodOfU = (shift != m ?
                   m - shift - mpNext[m - shift] : 0);
      shift = m;
      disp = 0;
      while (i > u &&
             getTarget(aut, state, y[i + j]) !=
             UNDEFINED) {
         disp += getShift(aut, state, y[i + j]);
         state = getTarget(aut, state, y[i + j]);
         if (isTerminal(aut, state))
            shift = i;
         --i;
```

```
        }
        if (i <= u)
           if (disp == 0) {
              OUTPUT(j);
              shift = period;
           }
           else {
              mu = (u + 1)/2;
              if (periodOfU <= mu) {
                 u -= periodOfU;
                 while (i > u &&
                        getTarget(aut, state, y[i + j]) !=
                        UNDEFINED) {
                    disp += getShift(aut, state, y[i + j]);
                    state = getTarget(aut, state, y[i + j]);
                    if (isTerminal(aut, state))
                       shift = i;
                    --i;
                 }
                 if (i <= u)
                    shift = disp;
              }
              else {
                 u = u - mu - 1;
                 while (i > u &&
                        getTarget(aut, state, y[i + j]) !=
                        UNDEFINED) {
                    disp += getShift(aut, state, y[i + j]);
                    state = getTarget(aut, state, y[i + j]);
                    if (isTerminal(aut, state))
                       shift = i;
                    --i;
                 }
              }
           }
        j += shift;
     }
  }
```

3.6 Reverse Factor algorithm

a. Trình bày thuật toán

Các thuật toán kiểu Boyer-Moore khớp với một số hậu tố của mẫu nhưng có thể khớp với một số tiền tố của mẫu bằng cách quét ký tự của cửa sổ từ phải sang trái và sau đó cải thiện độ dài của các dịch chuyển. Điều này có thể thực hiện được bằng cách sử dụng automaton hậu tố nhỏ nhất (còn được gọi là DAWG cho Đồ thị từ vòng có hướng dẫn) của mẫu đảo ngược. Thuật toán kết quả được gọi là thuật toán Nhân tố ngược.

Automaton hậu tố nhỏ nhất của một từ w là Automaton hữu hạn xác định S (w) = (Q, q0, T, E). Ngôn ngữ được chấp nhận bởi S (w) là L (S (w)) = {u in Sigma *:

tồn tại v trong Sigma * sao cho w = vu}. Giai đoạn tiền xử lý của thuật toán Reverse Factor bao gồm tính toán tự động hóa hậu tố nhỏ nhất cho mẫu đảo ngược X^R. Nó là tuyến tính theo thời gian và không gian theo chiều dài của mẫu.

Trong giai đoạn tìm kiếm, thuật toán Reverse Factor phân tích các ký tự của cửa sổ từ phải sang trái với tự động hóa S (X^R), bắt đầu với trạng thái q0. Nó sẽ tiếp tục cho đến khi không còn quá trình chuyển đổi nào được xác định cho ký tự hiện tại của cửa sổ từ trạng thái hiện tại của máy tự động. Tại thời điểm này, có thể dễ dàng biết được độ dài của tiền tố dài nhất của mẫu đã được khớp là bao nhiêu: nó tương ứng với độ dài của đường dẫn được thực hiện trong S (X^R) từ trạng thái bắt đầu q0 đến trạng thái cuối cùng gặp phải. Biết độ dài của tiền tố dài nhất này, việc tính toán dịch chuyển phải để thực hiện là điều không cần thiết.

Thuật toán Nhân tố ngược có độ phức tạp thời gian trong trường hợp xấu nhất bậc hai nhưng ở mức trung bình, nó là tối ưu. Nó thực hiện kiểm tra O (n.logsigma (m) / m) đối với các ký tự văn bản ở mức trung bình đạt đến giới hạn tốt nhất được hiển thị bởi Yao vào năm 1979.

b. Độ phức tạp thuật toán

sử dụng tự động hóa hậu tố của X^R;

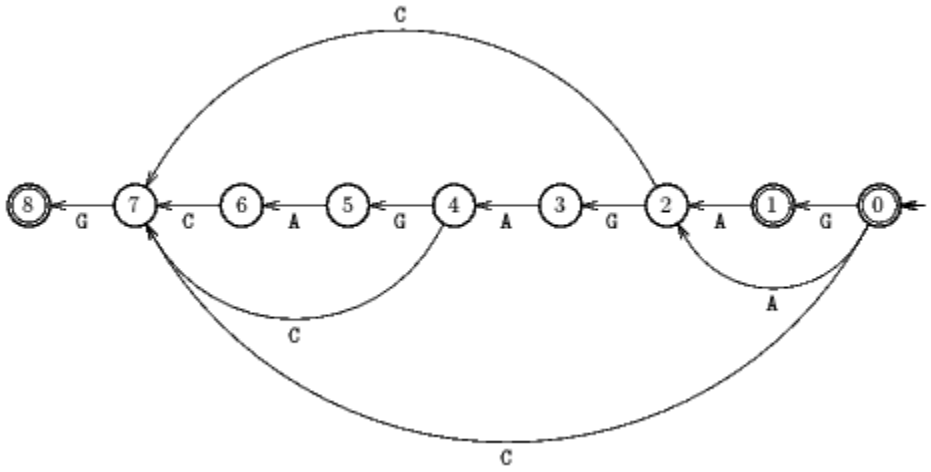
luyện tập nhanh cho các bài tập dài và bảng chữ cái nhỏ;

giai đoạn tiền xử lý theo thời gian và không gian phức tạp O (m);

pha tìm kiếm theo độ phức tạp thời gian O (mn);

tối ưu trong mức trung bình.

c. Kiểm nghiệm thuật toán



d. Lập trình thuật toán

```
void buildSuffixAutomaton(char *x, int m, Graph aut) {
   int i, art, init, last, p, q, r;
   char c;

   init = getInitial(aut);
   art = newVertex(aut);
   setSuffixLink(aut, init, art);
   last = init;
   for (i = 0; i < m; ++i) {
      c = x[i];
      p = last;
      q = newVertex(aut);
      setLength(aut, q, getLength(aut, p) + 1);
      setPosition(aut, q, getPosition(aut, p) + 1);
      while (p != init &&
             getTarget(aut, p, c) == UNDEFINED) {
         setTarget(aut, p, c, q);
         setShift(aut, p, c, getPosition(aut, q) -
                             getPosition(aut, p) - 1);
         p = getSuffixLink(aut, p);
      }
      if (getTarget(aut, p, c) == UNDEFINED) {
         setTarget(aut, init, c, q);
         setShift(aut, init, c,
                  getPosition(aut, q) -
                  getPosition(aut, init) - 1);
         setSuffixLink(aut, q, init);
      }
      else
         if (getLength(aut, p) + 1 ==
             getLength(aut, getTarget(aut, p, c)))
            setSuffixLink(aut, q, getTarget(aut, p, c));
         else {
            r = newVertex(aut);
            copyVertex(aut, r, getTarget(aut, p, c));
```

```
            setLength(aut, r, getLength(aut, p) + 1);
            setSuffixLink(aut, getTarget(aut, p, c), r);
            setSuffixLink(aut, q, r);
            while (p != art &&
                   getLength(aut, getTarget(aut, p, c)) >=
                   getLength(aut, r)) {
               setShift(aut, p, c,
                        getPosition(aut,
                                    getTarget(aut, p, c)) -
                        getPosition(aut, p) - 1);
               setTarget(aut, p, c, r);
               p = getSuffixLink(aut, p);
            }
         }
      last = q;
   }
   setTerminal(aut, last);
   while (last != init) {
      last = getSuffixLink(aut, last);
      setTerminal(aut, last);
   }
}


char *reverse(char *x, int m) {
   char *xR;
   int i;

   xR = (char *)malloc((m + 1)*sizeof(char));
   for (i = 0; i < m; ++i)
      xR[i] = x[m - 1 - i];
   xR[m] = '\0';
   return(xR);
}


int RF(char *x, int m, char *y, int n) {
   int i, j, shift, period, init, state;
   Graph aut;
   char *xR;

   /* Preprocessing */
   aut = newSuffixAutomaton(2*(m + 2), 2*(m + 2)*ASIZE);
   xR = reverse(x, m);
   buildSuffixAutomaton(xR, m, aut);
   init = getInitial(aut);
   period = m;

   /* Searching */
   j = 0;
   while (j <= n - m) {
      i = m - 1;
      state = init;
      shift = m;
      while (i + j >= 0 &&
             getTarget(aut, state, y[i + j]) !=
             UNDEFINED) {
```

```
            state = getTarget(aut, state, y[i + j]);
            if (isTerminal(aut, state)) {
               period = shift;
               shift = i;
            }
            --i;
         }
         if (i < 0) {
            OUTPUT(j);
            shift = period;
         }
         j += shift;
      }
   }
```

CHƯƠNG 4: THUẬT TOÁN TÌM KIẾM MẪU MỘT VỊ TRÍ CỤ THỂ

4.1 Skip Search Algorithm

a. 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 and 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).

b. Độ phức tạp thuật toán

- 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.

c. 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

d. Lập trình thuật toán

```
#include<iostream>
#include<algorithm>
#include<iomanip>
#include<cstdio>
#include<cstring>
#define For(i,a,b) for(long i = a;i<=b;i++)
using namespace std;
char x[100001],y[100001];
int m, n,ASIZE = 256;
void nhap(){
printf("Nhap x : "); gets(x); m = strlen(x);
printf("Nhap y : "); gets(y); n = strlen(y);
}
void preQsBc(char *x, int m, int qsBc[]) {
int i;
```

```
for (i = 0; i < ASIZE; ++i)
qsBc[i] = m + 1;
for (i = 0; i < m; ++i)
qsBc[x[i]] = m - i;
}
void QS(char *x, int m, char *y, int n) {
int j, qsBc[ASIZE];
/* Preprocessing */
preQsBc(x, m, qsBc);
/* Searching */
j = 0;
while (j <= n - m) {
if (memcmp(x, y + j, m) == 0)
printf("position is %d\n",j);
j += qsBc[y[j + m]]; /* shift */
}
}
int main(){
nhap();
QS(x,m,y,n);
return 0;
}
```

4.2 Two Way Algorithm

a. Trình bày thuật toán

Mẫu x được phân tích thành hai phần xell và xr sao cho x = xellxr. Sau đó, giai đoạn tìm kiếm của thuật toán Two Way bao gồm so sánh các ký tự của xr từ trái sang phải và sau đó, nếu không có sự trùng khớp nào xảy ra trong giai đoạn đầu tiên đó, sẽ so sánh các ký tự của xell từ phải sang trái trong giai đoạn thứ hai.

Giai đoạn tiền xử lý của thuật toán bao gồm việc chọn một xellxr phân tích nhân tử tốt.

Gọi (u, v) là một thừa số của x. Sự lặp lại trong (u, v) là một từ w sao cho hai thuộc tính sau giữ nguyên:

w là hậu tố của u hoặc u là hậu tố của w;

w là một tiền tố của v của v là một tiền tố của w.

Nói cách khác, w xảy ra ở cả hai phía của đường cắt giữa u và v với khả năng tràn ở cả hai phía. Độ dài của một lần lặp lại ở (u, v) được gọi là chu kỳ cục bộ và độ

dài của lần lặp lại nhỏ nhất trong (u, v) được gọi là chu kỳ cục bộ và được ký hiệu là r (u, v).

Mỗi thừa số (u, v) của có ít nhất một lần lặp lại. Có thể dễ dàng nhận thấy rằng 1 leq r (u, v) leq | x |

Một thừa số hóa (u, v) của x sao cho r (u, v) = per (x) được gọi là một thừa số hóa tới hạn của x.

Nếu (u, v) là một thừa số hóa tới hạn của x thì tại vị trí | u | trong x, chu kỳ toàn cầu và địa phương là như nhau. Thuật toán Two Way chọn thừa số hóa quan trọng (xell, xr) sao cho | xell | <per (x) và | xell | là tối thiểu.

Để tính toán thừa số quan trọng xell, xr của x, trước tiên chúng ta tính hậu tố z cực đại của x cho leq thứ tự và hậu tố cực đại z dấu ngã cho dấu ngã leq thứ tự ngược lại. Sau đó (xell, xr) được chọn sao cho. | xell | = max {| z |, | z dấu ngã |}

Giai đoạn tiền xử lý có thể được thực hiện trong thời gian O (m) và độ phức tạp không gian không đổi.

Giai đoạn tìm kiếm của thuật toán Two Way bao gồm đầu tiên so sánh ký tự của xr từ trái sang phải, sau đó là ký tự của xell từ phải sang trái.

Khi sự không khớp xảy ra khi quét ký tự thứ k của xr, thì việc dịch chuyển độ dài k được thực hiện.

Khi sự không khớp xảy ra khi quét xell hoặc khi tìm thấy sự xuất hiện của mẫu, thì việc dịch chuyển độ dài trên mỗi (x) được thực hiện.

Lược đồ như vậy dẫn đến thuật toán trường hợp xấu nhất bậc hai, điều này có thể tránh được bằng cách ghi nhớ tiền tố: khi thực hiện thay đổi độ dài trên mỗi (x) thì độ dài của tiền tố phù hợp của mẫu ở đầu cửa sổ (cụ thể là m- per (x)) sau khi ca làm việc được ghi nhớ để tránh quét lại trong lần thử tiếp theo.

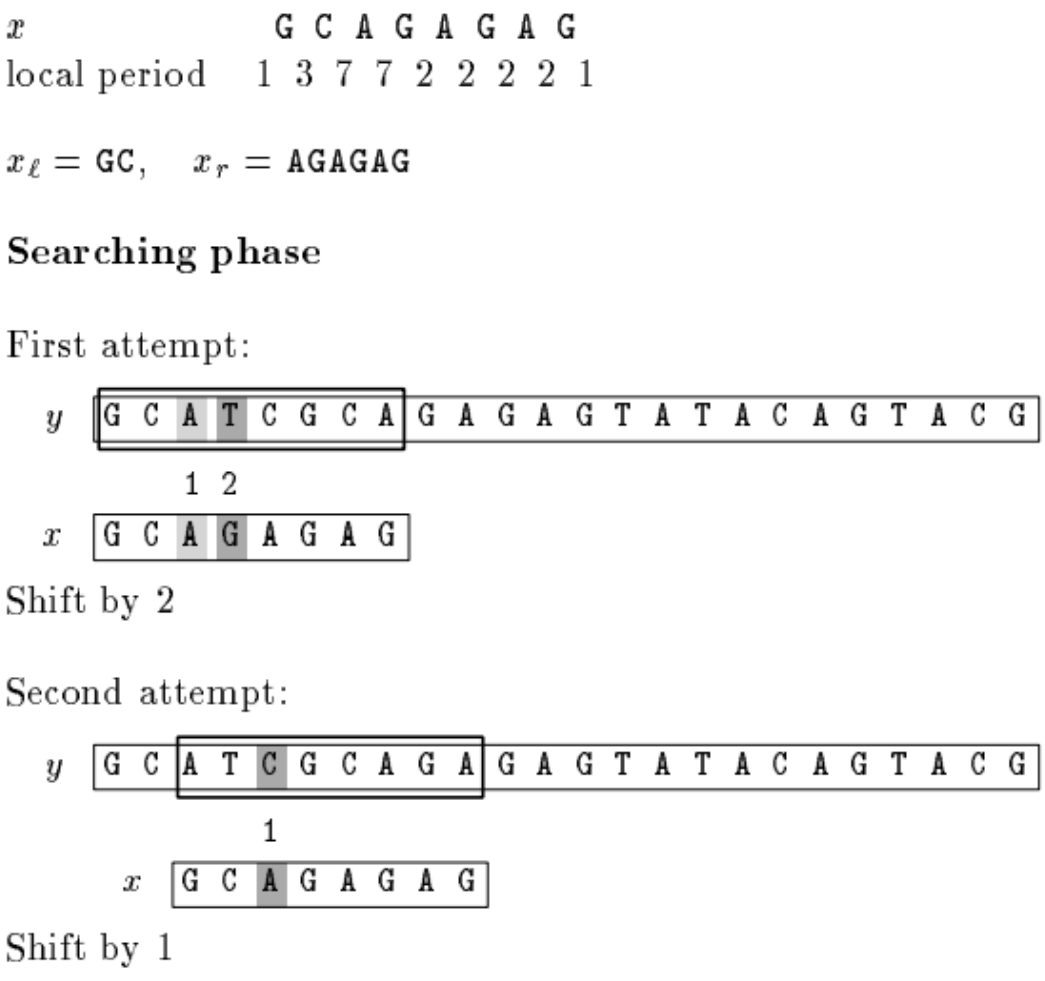
Giai đoạn tìm kiếm của thuật toán Two Way có thể được thực hiện với độ phức tạp thời gian là O (n).

Thuật toán Two Way thực hiện so sánh ký tự văn bản 2n-m trong trường hợp xấu nhất. Breslauer đã thiết kế một biến thể trên thuật toán Two Way thực hiện phép so sánh dưới 2n-m bằng cách sử dụng không gian không đổi.

b. Độ phức tạp thuật toán

- yêu cầu một bảng chữ cái có thứ tự;
- giai đoạn tiền xử lý theo thời gian O (m) và độ phức tạp không gian không đổi;
- độ phức tạp không gian không đổi cho giai đoạn tiền xử lý;
- giai đoạn tìm kiếm trong thời gian O (n);
- thực hiện so sánh ký tự văn bản 2n-m trong trường hợp xấu nhất.

c. Kiểm nghiệm thuật toán



Third attempt:

Shift by 1

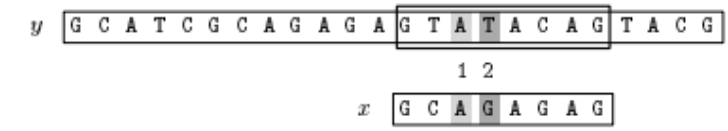
Fourth attempt:

Shift by 1

Fifth attempt:

Shift by 7

Sixth attempt:



Shift by 2

Seventh attempt:

Shift by 2

Eighth attempt:

Shift by 3

Kết luận: Thuật toán biểu diễn 20 so sánh ký tự văn bản

d. Lập trình thuật toán

```
/* Computing of the maximal suffix for <= */
int maxSuf(char *x, int m, int *p) {
   int ms, j, k;
   char a, b;

   ms = -1;
   j = 0;
   k = *p = 1;
   while (j + k < m) {
      a = x[j + k];
      b = x[ms + k];
      if (a < b) {
         j += k;
         k = 1;
         *p = j - ms;
      }
      else
         if (a == b)
            if (k != *p)
               ++k;
            else {
               j += *p;
               k = 1;
            }
         else { /* a > b */
            ms = j;
            j = ms + 1;
            k = *p = 1;
         }
   }
   return(ms);
}

/* Computing of the maximal suffix for >= */
int maxSufTilde(char *x, int m, int *p) {
   int ms, j, k;
   char a, b;

   ms = -1;
   j = 0;
   k = *p = 1;
   while (j + k < m) {
      a = x[j + k];
      b = x[ms + k];
      if (a > b) {
         j += k;
         k = 1;
         *p = j - ms;
      }
      else
         if (a == b)
            if (k != *p)
               ++k;
            else {
```

```
                j += *p;
                k = 1;
            }
         else { /* a < b */
            ms = j;
            j = ms + 1;
            k = *p = 1;
         }
   }
   return(ms);
}


/* Two Way string matching algorithm. */
void TW(char *x, int m, char *y, int n) {
   int i, j, ell, memory, p, per, q;

   /* Preprocessing */
   i = maxSuf(x, m, &p);
   j = maxSufTilde(x, m, &q);
   if (i > j) {
      ell = i;
      per = p;
   }
   else {
      ell = j;
      per = q;
   }

   /* Searching */
   if (memcmp(x, x + per, ell + 1) == 0) {
      j = 0;
      memory = -1;
      while (j <= n - m) {
         i = MAX(ell, memory) + 1;
         while (i < m && x[i] == y[i + j])
            ++i;
         if (i >= m) {
            i = ell;
            while (i > memory && x[i] == y[i + j])
               --i;
            if (i <= memory)
               OUTPUT(j);
            j += per;
            memory = m - per - 1;
         }
         else {
            j += (i - ell);
            memory = -1;
         }
      }
   }
   else {
      per = MAX(ell + 1, m - ell - 1) + 1;
      j = 0;
      while (j <= n - m) {
         i = ell + 1;
```

```
            while (i < m && x[i] == y[i + j])
               ++i;
            if (i >= m) {
               i = ell;
               while (i >= 0 && x[i] == y[i + j])
                  --i;
               if (i < 0)
                  OUTPUT(j);
               j += per;
            }
            else
               j += (i - ell);
         }
      }
   }
```

4.3 Optimal Mismatch algorithm

a. Trình bày thuật toán

Sunday đã thiết kế một thuật toán trong đó các ký tự mẫu được quét từ ký tự ít thường xuyên nhất đến thường xuyên nhất. Làm như vậy, người ta có thể hy vọng rằng hầu hết các trường hợp không khớp và do đó có thể quét toàn bộ văn bản rất nhanh. Người ta cần biết tần số của từng ký tự trong bảng chữ cái.

Giai đoạn tiền xử lý của thuật toán Không khớp tối ưu bao gồm sắp xếp các ký tự mẫu theo thứ tự giảm dần tần số của chúng và sau đó xây dựng chức năng dịch chuyển ký tự xấu của Tìm kiếm nhanh (xem chương Thuật toán tìm kiếm nhanh) và chức năng thay đổi hậu tố tốt được điều chỉnh để quét thứ tự của các ký tự mẫu. Nó có thể được thực hiện trong thời gian O (m2 + sigma) và độ phức tạp không gian O (m + sigma).

Giai đoạn tìm kiếm của thuật toán Không khớp Tối ưu có độ phức tạp thời gian là O (mn).

b. Độ phức tạp thuật toán

- biến thể của thuật toán Tìm kiếm nhanh;

- yêu cầu tần số của các ký tự;

- giai đoạn tiền xử lý theo thời gian O (m2 + sigma) và độ phức tạp không gian O (m + sigma);

- pha tìm kiếm theo độ phức tạp thời gian O (mn).

c. Kiểm nghiệm thuật toán

c	A	C	G	T
$freq[c]$	8	5	7	4
$qsBc[c]$	2	7	1	9

i	0	1	2	3	4	5	6	7
$x[i]$	G	C	A	G	A	G	A	G
$pat[i].loc$	1	7	5	3	0	6	4	2
$pat[i].c$	C	G	G	G	G	A	A	A

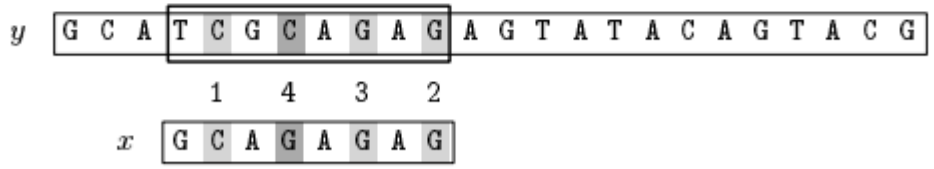
i	0	1	2	3	4	5	6	7	8
$adaptedGs[i]$	1	3	4	2	7	7	7	7	7

Searching phase

First attempt:

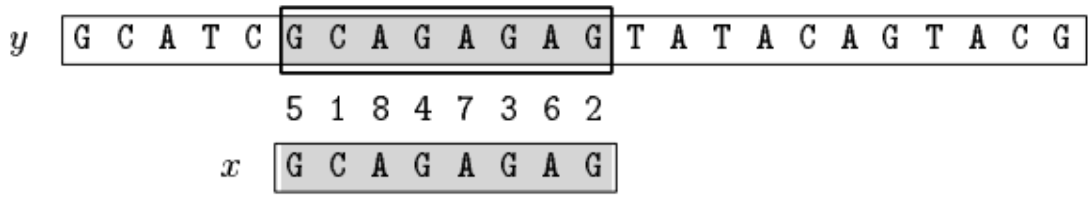
Shift by 3 ($adaptedGs[1]$)

Second attempt:



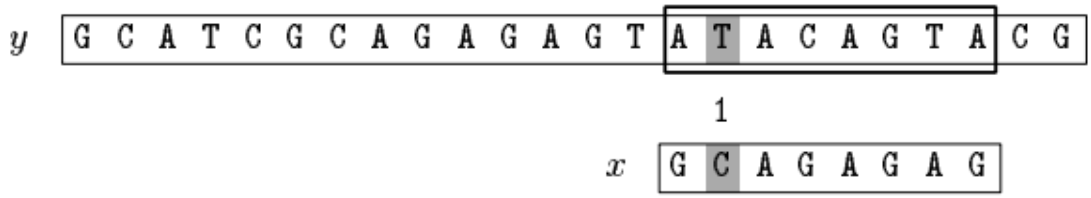
Shift by 2 ($qsBc[\mathtt{A}] = adaptedGs[3]$)

Fourth attempt:



Shift by 9 ($qsBc[\mathtt{T}]$)

Fifth attempt:



Shift by 7 ($qsBc[\mathtt{C}]$)

Kết luận: thuật toán biểu diễn 15 so sánh ký tự văn bản

d. Lập trình thuật toán

```
typedef struct patternScanOrder {
   int loc;
   char c;
} pattern;

int freq[ASIZE];

/* Construct an ordered pattern from a string. */
void orderPattern(char *x, int m, int (*pcmp)(),
                  pattern *pat) {
   int i;

   for (i = 0; i <= m; ++i) {
      pat[i].loc = i;
      pat[i].c = x[i];
```

```
   }
   qsort(pat, m, sizeof(pattern), pcmp);
}


/* Optimal Mismatch pattern comparison function. */
int optimalPcmp(pattern *pat1, pattern *pat2) {
   float fx;

   fx = freq[pat1->c] - freq[pat2->c];
   return(fx ? (fx > 0 ? 1 : -1) :
               (pat2->loc - pat1->loc));
}


/* Find the next leftward matching shift for
   the first ploc pattern elements after a
   current shift or lshift. */
int matchShift(char *x, int m, int ploc,
               int lshift, pattern *pat) {
   int i, j;

   for (; lshift < m; ++lshift) {
      i = ploc;
      while (--i >= 0) {
         if ((j = (pat[i].loc - lshift)) < 0)
            continue;
         if (pat[i].c != x[j])
            break;
      }
      if (i < 0)
         break;
   }
   return(lshift);
}


/* Constructs the good-suffix shift table
   from an ordered string. */
void preAdaptedGs(char *x, int m, int adaptedGs[],
                  pattern *pat) {
   int lshift, i, ploc;

   adaptedGs[0] = lshift = 1;
   for (ploc = 1; ploc <= m; ++ploc) {
      lshift = matchShift(x, m, ploc, lshift, pat);
      adaptedGs[ploc] = lshift;
   }
   for (ploc = 0; ploc <= m; ++ploc) {
      lshift = adaptedGs[ploc];
      while (lshift < m) {
         i = pat[ploc].loc - lshift;
         if (i < 0 || pat[ploc].c != x[i])
            break;
         ++lshift;
         lshift = matchShift(x, m, ploc, lshift, pat);
      }
```

```
      adaptedGs[ploc] = lshift;
   }
}


/* Optimal Mismatch string matching algorithm. */
void OM(char *x, int m, char *y, int n) {
   int i, j, adaptedGs[XSIZE], qsBc[ASIZE];
   pattern pat[XSIZE];

   /* Preprocessing */
   orderPattern(x, m, optimalPcmp, pat);
   preQsBc(x, m, qsBc);
   preAdaptedGs(x, m, adaptedGs, pat);

   /* Searching */
   j = 0;
   while (j <= n - m) {
      i = 0;
      while (i < m && pat[i].c == y[j + pat[i].loc])
         ++i;
      if (i >= m)
         OUTPUT(j);
      j += MAX(adaptedGs[i],qsBc[y[j + m]]);
   }
}
```

4.4 Maximal Shift Algorithm
a. Trình bày thuật toán

Sunday đã thiết kế một thuật toán trong đó các ký tự mẫu được quét từ một ký tự sẽ dẫn đến sự thay đổi lớn hơn sang ký tự sẽ dẫn đến sự thay đổi ngắn hơn. Làm như vậy người ta có thể hy vọng tối đa hóa độ dài của các ca thay đổi.

Giai đoạn tiền xử lý của thuật toán Maximal Shift bao gồm việc sắp xếp các ký tự mẫu theo thứ tự giảm dần của sự dịch chuyển của chúng và sau đó xây dựng chức năng dịch chuyển ký tự xấu Tìm kiếm nhanh (xem chương Thuật toán tìm kiếm nhanh) và chức năng dịch chuyển hậu tố tốt được điều chỉnh để quét thứ tự của các ký tự mẫu. Nó có thể được thực hiện trong thời gian O (m2 + sigma) và độ phức tạp không gian O (m + sigma).

Giai đoạn tìm kiếm của thuật toán Maximal Shift có độ phức tạp thời gian trong trường hợp xấu nhất bậc hai.

b. Độ phức tạp thuật toán

- biến thể của thuật toán Tìm kiếm nhanh;

- độ phức tạp thời gian trường hợp xấu nhất bậc hai;

- giai đoạn tiền xử lý theo thời gian O (m2 + sigma) và độ phức tạp không gian O (m + sigma);

- pha tìm kiếm theo độ phức tạp thời gian O (mn).

c. Kiểm nghiệm thuật toán

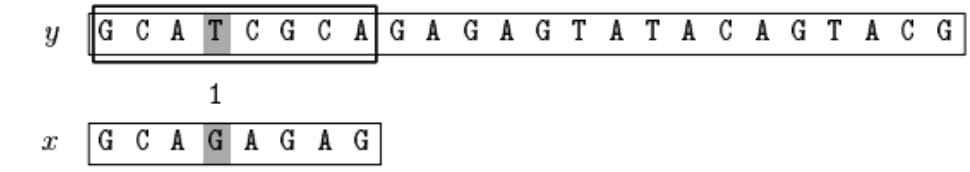
i	0	1	2	3	4	5	6	7
$x[i]$	G	C	A	G	A	G	A	G
$minShift[i]$	1	2	3	3	2	2	2	2
$pat[i].loc$	3	2	7	6	5	4	1	0
$pat[i].c$	G	A	G	A	G	A	C	G

c	A	C	G	T
$qsBc[c]$	2	7	1	9

i	0	1	2	3	4	5	6	7	8
$adaptedGs[i]$	1	3	3	7	4	7	7	7	7

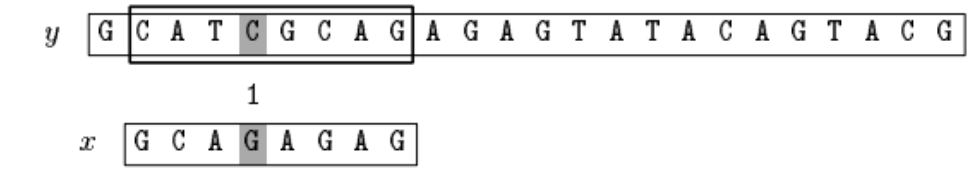
Searching phase

First attempt:



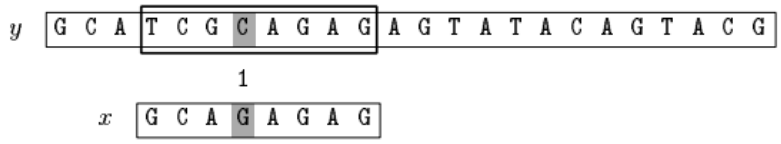
Shift by 1 ($qsBc[\mathtt{G}] = adaptedGs[0]$)

Second attempt:



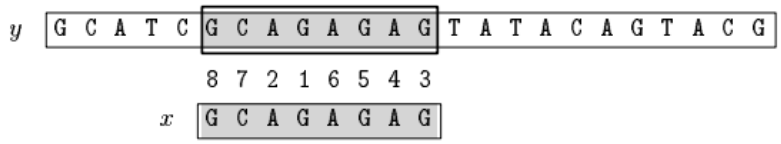
Shift by 2 ($qsBc[\mathtt{A}]$)

Third attempt:



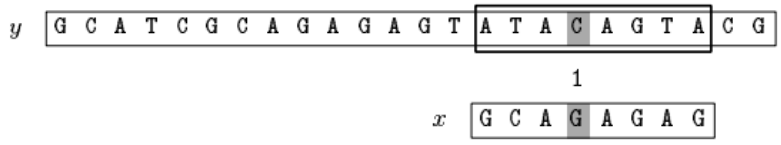
Shift by 2 ($qsBc$[A])

Fourth attempt:



Shift by 9 ($qsBc$[T])

Fifth attempt:



Shift by 7 ($qsBc$[C])

Kết luận: thuật toán biểu diễn 12 so sánh kí tự văn bản

d. Lập trình thuật toán

```
typedef struct patternScanOrder {
   int loc;
   char c;
} pattern;

int minShift[XSIZE];

/* Computation of the MinShift table values. */
void computeMinShift(char *x, int m) {
   int i, j;

   for (i = 0; i < m; ++i) {
      for (j = i - 1; j >= 0; --j)
          if (x[i] == x[j])
             break;
      minShift[i] = i - j;
   }
}


/* Maximal Shift pattern comparison function. */
int maxShiftPcmp(pattern *pat1, pattern *pat2) {
   int dsh;
```

```
   dsh = minShift[pat2->loc] - minShift[pat1->loc];
   return(dsh ? dsh : (pat2->loc - pat1->loc));
}


/* Maximal Shift string matching algorithm. */
void MS(char *x, int m, char *y, int n) {
   int i, j, qsBc[ASIZE], adaptedGs[XSIZE];
   pattern pat[XSIZE];

   /* Preprocessing */
   computeMinShift(x ,m);
   orderPattern(x, m, maxShiftPcmp, pat);
   preQsBc(x, m, qsBc);
   preAdaptedGs(x, m, adaptedGs, pat);

   /* Searching */
   j = 0;
   while (j <= n - m) {
      i = 0;
      while (i < m && pat[i].c == y[j + pat[i].loc])
         ++i;
      if (i >= m)
         OUTPUT(j);
      j += MAX(adaptedGs[i], qsBc[y[j + m]]);
   }
}
```

4.5 KMP Skip Search

a. Trình bày thuật toán

Có thể làm cho thuật toán Bỏ qua tìm kiếm (xem chương Thuật toán bỏ qua tìm kiếm) tuyến tính bằng cách sử dụng hai bảng dịch chuyển của Morris-Pratt (xem chương Morris và thuật toán Pratt) và Knuth-Morris-Pratt (xem chương Knuth, Morris và thuật toán Pratt) .

Đối với 1 leq i leq m, mpNext [i] bằng độ dài của đường viền dài nhất của x [0 .. i-1] và mpNext [0] = - 1.

Đối với 1 leq i <m, kmpNext [i] bằng độ dài của đường viền dài nhất của x [0 .. i-1] theo sau là một ký tự khác với x [i], kmpNext [0] = - 1 và kmpNext [m] = m-per (x).

Danh sách trong nhóm được lưu trữ rõ ràng trong danh sách bảng.

Giai đoạn tiền xử lý của thuật toán KmpSkip Search là ở độ phức tạp không gian và thời gian O (m + sigma).

Tình huống chung cho một nỗ lực trong giai đoạn tìm kiếm như sau ((xem hình 30.1):

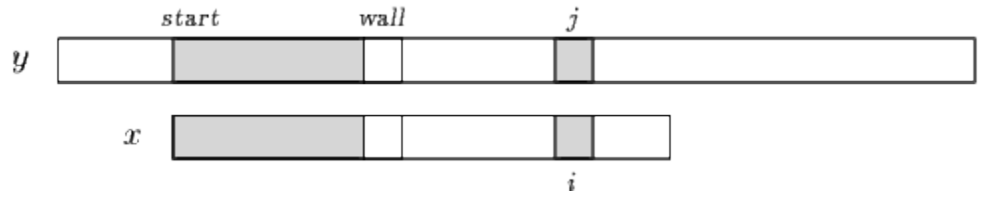
j là vị trí văn bản hiện tại;

x [i] = y [j];

start = j-i là vị trí bắt đầu có thể xảy ra của x trong y;

tường là vị trí văn bản được quét ngoài cùng bên phải;

x [0 .. wall-start-1] = y [start .. wall-1];



So sánh được thực hiện từ trái sang phải giữa x [wall-start .. m-1] và y [wall .. start + m-1] cho đến khi xảy ra sự không khớp hoặc toàn bộ so khớp. Gọi k geq wall-start là số nguyên nhỏ nhất sao cho x [k] neq y [start + k] hoặc k = m nếu sự xuất hiện của x bắt đầu tại vị trí bắt đầu trong y.

Khi đó wall nhận giá trị của start + k.

Sau đó, thuật toán KmpSkip tính toán hai dịch chuyển (hai vị trí bắt đầu mới): ca đầu tiên theo thuật toán bỏ qua (xem thuật toán AdvanceSkip để biết thêm chi tiết), điều này cho chúng ta vị trí bắt đầu bỏ qua Bắt đầu, ca thứ hai theo bảng dịch chuyển của Knuth- Morris-Pratt, cho chúng ta một vị trí bắt đầu khác kmpStart.

Một số trường hợp có thể phát sinh:

jumpStart <kmpStart sau đó một sự thay đổi theo thuật toán bỏ qua được áp dụng mang lại một giá trị mới cho bỏ quaStart và chúng ta phải so sánh lại lần nữa Bỏ qua Bắt đầu và kmpStart;

kmpStart <ignoreStart <wall sau đó một sự thay đổi theo bảng shift của Morris-Pratt được áp dụng. Điều này mang lại một giá trị mới cho kmpStart. Chúng ta phải so sánh một lần nữa Bỏ quaStart và kmpStart;

bỏ quaStart = kmpStart sau đó một lần thử khác có thể được thực hiện với start = skipStart;

kmpStart <wall <skipStart sau đó có thể thực hiện một lần thử khác với start = ignoreStart.

Giai đoạn tìm kiếm của thuật toán KmpSkip Search là trong thời gian O (n).

b. Độ phức tạp thuật toán

- cải tiến thuật toán Bỏ qua tìm kiếm;

- sử dụng nhóm các vị trí cho mỗi ký tự của bảng chữ cái;

- giai đoạn tiền xử lý theo thời gian và không gian phức tạp O (m + sigma);

- pha tìm kiếm theo độ phức tạp thời gian O (n).

c. Kiểm nghiệm thuật toán

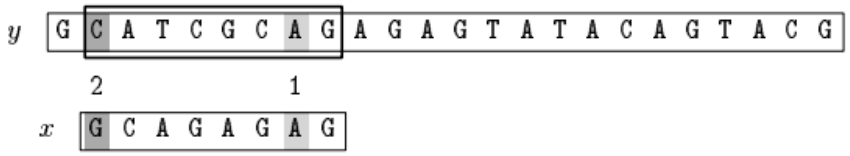
c	A	C	G	T
$z[c]$	6	1	7	-1

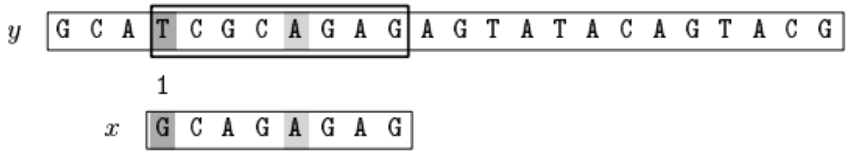
i	0	1	2	3	4	5	6	7
$list[i]$	-1	-1	-1	0	2	3	4	5

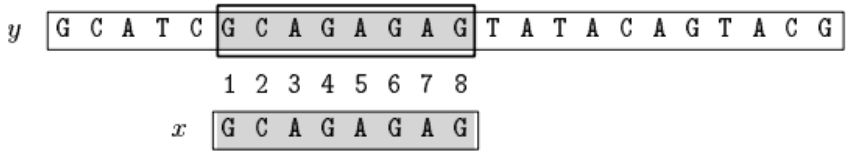
i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$mpNext[i]$	-1	0	0	0	1	0	1	0	1
$kmpNext[i]$	-1	0	0	-1	1	-1	1	-1	1

Searching phase

First attempt:

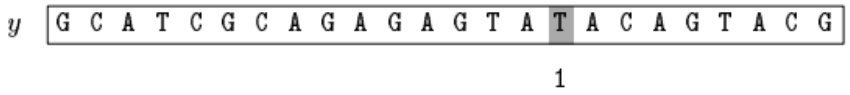






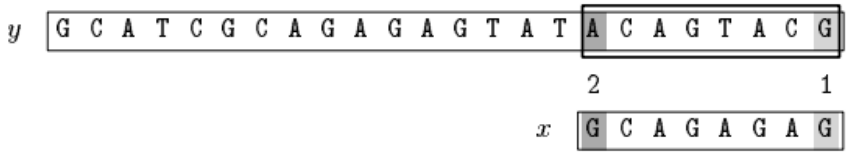
Shift by 8

Second attempt:



Shift by 8

Third attempt:



d. Lập trình thuật toán

```
int attempt(char *y, char *x, int m, int start, int wall) {
   int k;

   k = wall - start;
   while (k < m && x[k] == y[k + start])
      ++k;
   return(k);
}


void KMPSKIP(char *x, int m, char *y, int n) {
```

```
int i, j, k, kmpStart, per, start, wall;
int kmpNext[XSIZE], list[XSIZE], mpNext[XSIZE],
    z[ASIZE];

/* Preprocessing */
preMp(x, m, mpNext);
preKmp(x, m, kmpNext);
memset(z, -1, ASIZE*sizeof(int));
memset(list, -1, m*sizeof(int));
z[x[0]] = 0;
for (i = 1; i < m; ++i) {
   list[i] = z[x[i]];
   z[x[i]] = i;
}

/* Searching */
wall = 0;
per = m - kmpNext[m];
i = j = -1;
do {
   j += m;
} while (j < n && z[y[j]] < 0);
if (j >= n)
  return;
i = z[y[j]];
start = j - i;
while (start <= n - m) {
   if (start > wall)
      wall = start;
   k = attempt(y, x, m, start, wall);
   wall = start + k;
   if (k == m) {
      OUTPUT(start);
      i -= per;
   }
   else
      i = list[i];
   if (i < 0) {
      do {
         j += m;
      } while (j < n && z[y[j]] < 0);
      if (j >= n)
         return;
      i = z[y[j]];
   }
   kmpStart = start + k - kmpNext[k];
   k = kmpNext[k];
   start = j - i;
   while (start < kmpStart ||
          (kmpStart < start && start < wall)) {
      if (start < kmpStart) {
         i = list[i];
         if (i < 0) {
            do {
               j += m;
            } while (j < n && z[y[j]] < 0);
            if (j >= n)
```

```
                return;
             i = z[y[j]];
          }
          start = j - i;
       }
       else {
          kmpStart += (k - mpNext[k]);
          k = mpNext[k];
       }
    }
 }
```

4.6 Alpha Skip Search Algorithms.

a. Trình bày thuật toán

Giai đoạn tiền xử lý của thuật toán Alpha Skip Search bao gồm việc xây dựng một trie T (x) của tất cả các yếu tố có độ dài ell = logsigmam xuất hiện 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 mà nhân tố, được liên kết với lá, xuất hiện trong x.

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 nhóm các yếu tố văn bản y [j .. j + l-1] cho tất cả j = k. (M-l+ 1) -1 với số nguyên k trong khoảng y [1, lfloor (n-l) / mrfloor].

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

b. Độ phức tạp thuật toán

- cải tiến thuật toán Bỏ qua tìm kiếm;

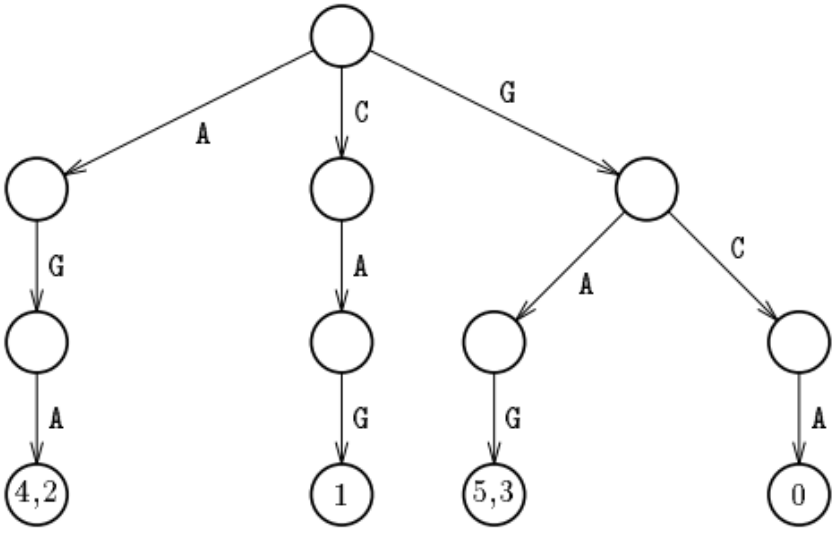
- sử dụng các nhóm vị trí cho mỗi hệ số của độ dài logigma (m) của mẫu;

- giai đoạn tiền xử lý theo thời gian và không gian phức tạp O (m);

- pha tìm kiếm theo độ phức tạp thời gian O (mn);

- O (logsigma (m). (N / (m-logsigma (m)))) so sánh ký tự văn bản được mong đợi.

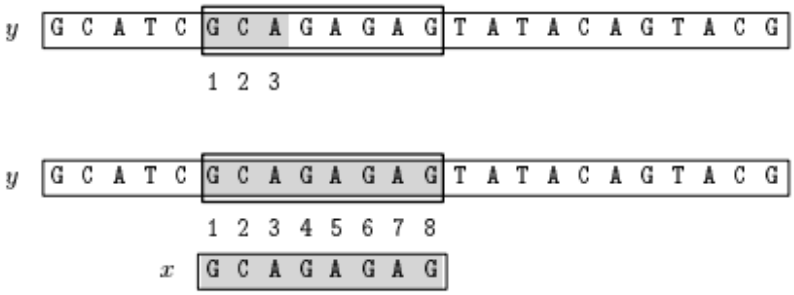
c. Kiểm nghiệm thuật toán


A
C
G
G
A
A
C
A
G
G
A
4,2
1
5,3
0

u	$z[u]$
AGA	(4, 2)
CAG	(1)
GAG	(5, 3)
GCA	(0)

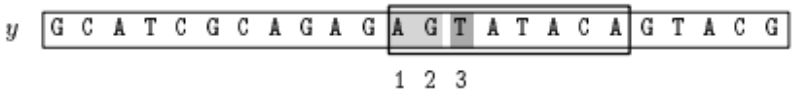
Searching phase

First attempt:



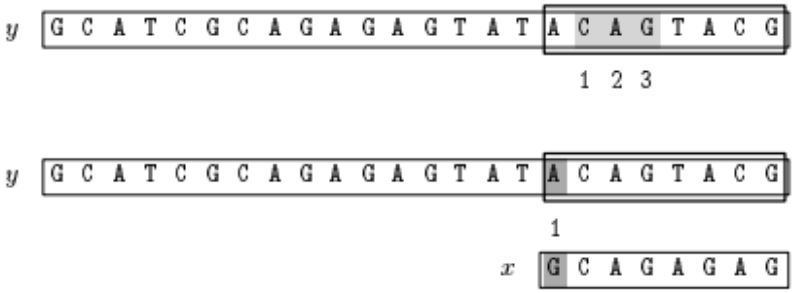
Shift by 6

Second attempt:



Shift by 6

Third attempt:



Kết luận: thuật toán biểu thị 18 sự kiểm tra về ký tự đoạn văn bản

d. Lập trình thuật toán

```
List *z;

#define getZ(i) z[(i)]

void setZ(int node, int i) {
   List cell;
```

```
   cell = (List)malloc(sizeof(struct _cell));
   if (cell == NULL)
      error("ALPHASKIP/setZ");
   cell->element = i;
   cell->next = z[node];
   z[node] = cell;
}


/* Create the transition labelled by the
    character c from node node.
   Maintain the suffix links accordingly. */
int addNode(Graph trie, int art, int node, char c) {
   int childNode, suffixNode, suffixChildNode;

   childNode = newVertex(trie);
   setTarget(trie, node, c, childNode);
   suffixNode = getSuffixLink(trie, node);
   if (suffixNode == art)
      setSuffixLink(trie, childNode, node);
   else {
      suffixChildNode = getTarget(trie, suffixNode, c);
      if (suffixChildNode == UNDEFINED)
         suffixChildNode = addNode(trie, art,
                                   suffixNode, c);
      setSuffixLink(trie, childNode, suffixChildNode);
   }
   return(childNode);
}


void ALPHASKIP(char *x, int m, char *y, int n, int a) {
   int b, i, j, k, logM, temp, shift, size, pos;
   int art, childNode, node, root, lastNode;
   List current;
   Graph trie;

   logM = 0;
   temp = m;
   while (temp > a) {
      ++logM;
      temp /= a;
   }
   if (logM == 0) logM = 1;


   /* Preprocessing */
   size = 2 + (2*m - logM + 1)*logM;
   trie = newTrie(size, size*ASIZE);
   z = (List *)calloc(size, sizeof(List));
   if (z == NULL)
      error("ALPHASKIP");

   root = getInitial(trie);
   art = newVertex(trie);
   setSuffixLink(trie, root, art);
   node = newVertex(trie);
```

```
   setTarget(trie, root, x[0], node);
   setSuffixLink(trie, node, root);
   for (i = 1; i < logM; ++i)
      node = addNode(trie, art, node, x[i]);
   pos = 0;
   setZ(node, pos);
   pos++;
   for (i = logM; i < m - 1; ++i) {
      node = getSuffixLink(trie, node);
      childNode = getTarget(trie, node, x[i]);
      if (childNode == UNDEFINED)
         node = addNode(trie, art, node, x[i]);
      else
         node = childNode;
      setZ(node, pos);
      pos++;
   }
   node = getSuffixLink(trie, node);
   childNode = getTarget(trie, node, x[i]);
   if (childNode == UNDEFINED) {
      lastNode = newVertex(trie);
      setTarget(trie, node, x[m - 1], lastNode);
      node = lastNode;
   }
   else
      node = childNode;
   setZ(node, pos);

   /* Searching */
   shift = m - logM + 1;
   for (j = m + 1 - logM; j < n - logM; j += shift) {
      node = root;
      for (k = 0; node != UNDEFINED && k < logM; ++k)
         node = getTarget(trie, node, y[j + k]);
      if (node != UNDEFINED)
         for (current = getZ(node);
              current != NULL;
              current = current->next) {
            b = j - current->element;
            if (x[0] == y[b] &&
                memcmp(x + 1, y + b + 1, m - 1) == 0)
               OUTPUT(b);
         }
   }
   free(z);
}
```

CHƯƠNG 5: THUẬT TOÁN TÌM KIẾM MẪU TỪ BẤT KÌ

5.1 Horspool algorithm

a. 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.

b. Độ phức tạp của thuật toán

- 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/và 2/(+1)

c. Kiểm nghiệm thuật toán

X = “GCAGAGAG”
Y =”GCATCGCAGAGAGTATACAGTACG”
Pha tiền xử lí:
bảng bmBc của Horspool algorithm
Pha tìm kiếm thực hiện :
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
1
G C A G A G A G
Shift by: 1 (bmBc[A])
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
2 1
G C A G A G A G
Shift by: 2 (bmBc[G])
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
Shift by: 2 (bmBc[G])
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
2 3 4 5 6 7 8 1

G C A G A G A G
Shift by: 2 (bmBc[G])
Fifth 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
Shift by: 1 (bmBc[A])
Sixth 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
Shift by: 8 (bmBc[T])
Seventh 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
Shift by: 2 (bmBc[G])

Kết luận: cần 17 lượt so sánh cho ví dụ này

d. Lập trình thuật toán

```
#include<iostream>
#include<algorithm>
#include<iomanip>
#include<cstdio>
#include<cstring>
#define For(i,a,b) for(long i = a;i<=b;i++)
using namespace std;
char x[100001],y[100001];
int m, n,ASIZE = 256;
void nhap(){
printf("Nhap x : "); gets(x); m = strlen(x);
printf("Nhap y : "); gets(y); n = strlen(y);
}
void preBmBc(char *x, int m, int bmBc[]) {
int i;
for (i = 0; i < ASIZE; ++i)
bmBc[i] = m;
for (i = 0; i < m - 1; ++i)
bmBc[x[i]] = m - i - 1;
}
```

```
void HORSPOOL(char *x, int m, char *y, int n) {
int j, bmBc[ASIZE];
char c;
/* Preprocessing */
preBmBc(x, m, bmBc);
/* Searching */
j = 0;
while (j <= n - m) {
c = y[j + m - 1];
if (x[m - 1] == c && memcmp(x, y + j, m - 1) == 0)
printf("position is %d\n",j);
j += bmBc[c];
}
}
int main(){
nhap();
HORSPOOL(x,m,y,n);
return 0;
}
```

5.2 Quick Search algorithm

a. 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 and x[m-i]=c} if c có trong x hoặc bằng m+1 với trường hợp còn lại.

b. Độ phức tạp thuật toán

- 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.

c. Kiểm nghiệm thuật toán

X = “GCAGAGAG”

Y =”GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lí:

bảng qsBc sử dụng Quick Search algorithm

Pha tìm kiếm thực hiện :

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
1 2 3 4
G C A G A G A G
Shift by: 1 (qsBc[G])
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
Shift by: 2 (qsBc[A])
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
Shift by: 2 (qsBc[A])
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 2 3 4 5 6 7 8
G C A G A G A G
Shift by: 9 (qsBc[T])
Fifth 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
Shift by: 7 (qsBc[C])
Kết luận: thuật toán thực hiện 17 phép so sánh các kí tự văn bản

d. Lập trình thuật toán

```
#include<iostream>
#include<algorithm>
#include<iomanip>
#include<cstdio>
#include<cstring>
#define For(i,a,b) for(long i = a;i<=b;i++)
using namespace std;
char x[100001],y[100001];
int m, n,ASIZE = 256;
void nhap(){
printf("Nhap x : "); gets(x); m = strlen(x);
printf("Nhap y : "); gets(y); n = strlen(y);
}
void preQsBc(char *x, int m, int qsBc[]) {
```

```
int i;
for (i = 0; i < ASIZE; ++i)
qsBc[i] = m + 1;
for (i = 0; i < m; ++i)
qsBc[x[i]] = m - i;
}
void QS(char *x, int m, char *y, int n) {
int j, qsBc[ASIZE];
/* Preprocessing */
preQsBc(x, m, qsBc);
/* Searching */
j = 0;
while (j <= n - m) {
if (memcmp(x, y + j, m) == 0)
printf("position is %d\n",j);
j += qsBc[y[j + m]]; /* shift */
}
}
int main(){
nhap();
QS(x,m,y,n);
return 0;
}
```

Chạy test : Input : Nhập x : GCAGAGAG Nhập y : GCATCGCAGAGAGTATACAGTACG Output: Position is 5.

5.3 Smith Algorithm

a. Trình bày thuật toán

Smith nhận thấy rằng việc tính toán sự thay đổi với ký tự văn bản ngay bên cạnh ký tự văn bản ngoài cùng bên phải của cửa sổ đôi khi cho phép dịch chuyển ngắn hơn so với việc sử dụng ký tự văn bản ngoài cùng bên phải của cửa sổ.
Ông khuyên sau đó nên lấy giá trị tối đa giữa hai giá trị.

Giai đoạn tiền xử lý của thuật toán Smith bao gồm tính toán hàm dịch chuyển ký tự xấu (xem chương thuật toán Boyer-Moore) và hàm dịch chuyển ký tự xấu Tìm kiếm nhanh (xem chương thuật toán tìm kiếm nhanh).

Giai đoạn tiền xử lý ở thời gian O (m + sigma) và độ phức tạp không gian O (sigma).
Giai đoạn tìm kiếm của thuật toán Smith có độ phức tạp thời gian trong trường hợp xấu nhất bậc hai.

b. Độ phức tạp thuật toán

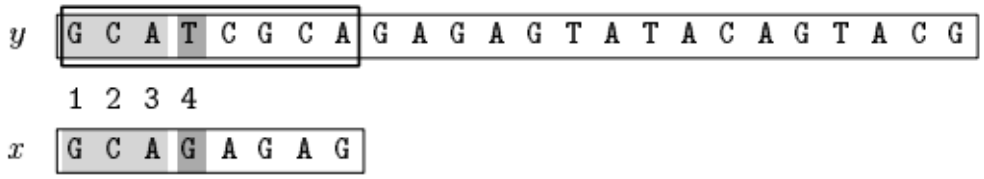
- sử dụng tối đa chức năng thay đổi ký tự xấu của Horspool và chức năng thay đổi - - ký tự xấu của Tìm kiếm nhanh;
- giai đoạn tiền xử lý theo thời gian O (m + sigma) và độ phức tạp không gian O (sigma);
- pha tìm kiếm theo độ phức tạp thời gian O (mn).

c. Kiểm nghiệm thuật toán

a	A	C	G	T
$bmBc[a]$	1	6	2	8
$qsBc[a]$	2	7	1	9

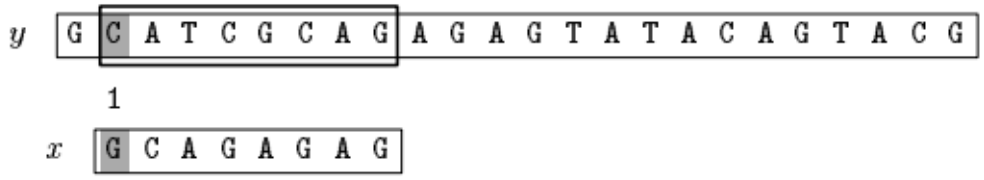
Searching phase

First attempt:



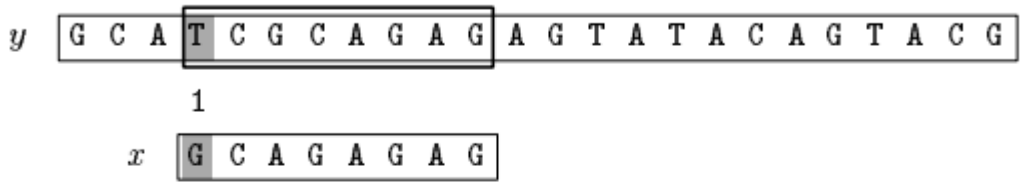
Shift by 1 ($bmBc[\mathtt{A}] = qsBc[\mathtt{G}]$)

Second attempt:



Shift by 2 ($bmBc[\mathtt{G}] = qsBc[\mathtt{A}]$)

Third attempt:

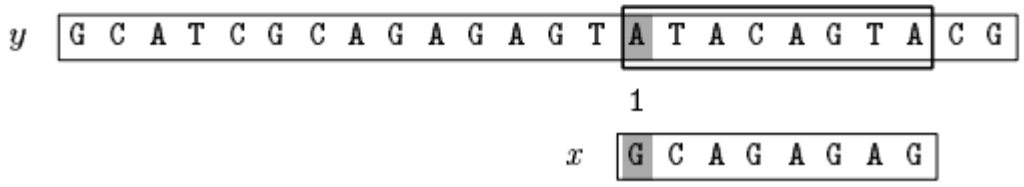


Shift by 2 ($bmBc[\texttt{G}] = qsBc[\texttt{A}]$)

Fourth attempt:

Shift by 9 ($qsBc[\texttt{T}]$)

Fifth attempt:



Shift by 7 ($qsBc[\texttt{C}]$)

Kết luận: Thuật toán sử dụng 15 phép so sánh ký tự văn bản

d. Lập trình thuật toán

```
void SMITH(char *x, int m, char *y, int n) {
   int j, bmBc[ASIZE], qsBc[ASIZE];

   /* Preprocessing */
   preBmBc(x, m, bmBc);
   preQsBc(x, m, qsBc);

   /* Searching */
   j = 0;
   while (j<= n - m) {
      if (memcmp(x, y + j, m) == 0)
         OUTPUT(j);
      j += MAX(bmBc[y[j + m - 1]], qsBc[y[j + m]]);
   }
}
```

5.4 Raita Algorithm

a. Trình bày thuật toán

Raita đã thiết kế một thuật toán mà ở mỗi lần thử đầu tiên sẽ so sánh ký tự cuối cùng của mẫu với ký tự văn bản ngoài cùng bên phải của cửa sổ, sau đó nếu chúng khớp, nó sẽ so sánh ký tự đầu tiên của mẫu với ký tự văn bản ngoài cùng bên trái của cửa sổ, sau đó nếu chúng khớp nó so sánh ký tự giữa của mẫu với ký tự giữa văn bản của cửa sổ. Và cuối cùng nếu họ khớp nó thực sự so sánh các ký tự khác từ ký tự thứ hai đến ký tự cuối cùng trừ một, có thể so sánh một lần nữa ký tự giữa.

Raita quan sát thấy rằng thuật toán của nó có một hành vi tốt trong thực tế khi tìm kiếm các mẫu trong văn bản tiếng Anh và cho rằng hiệu suất này là do sự tồn tại của các phụ thuộc ký tự.
Smith đã thực hiện thêm một số thí nghiệm và kết luận rằng hiện tượng này có thể là do hiệu ứng của trình biên dịch.

Giai đoạn tiền xử lý của thuật toán Raita bao gồm tính toán hàm dịch chuyển ký tự xấu (xem chương Boyer-Moore). Nó có thể được thực hiện trong thời gian O (m + sigma) và độ phức tạp không gian O (sigma).

Giai đoạn tìm kiếm của thuật toán Raita có độ phức tạp thời gian trong trường hợp xấu nhất bậc hai.

b. Độ phức tạp thuật toán

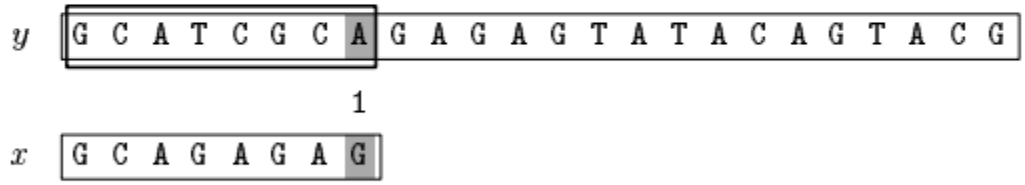
- đầu tiên so sánh ký tự mẫu cuối cùng, sau đó là ký tự đầu tiên và cuối cùng là ký tự ở giữa trước khi thực sự so sánh các ký tự khác;
- thực hiện các thay đổi như thuật toán Horspool;
- giai đoạn tiền xử lý theo thời gian O (m + sigma) và độ phức tạp không gian O (sigma);
- pha tìm kiếm theo độ phức tạp thời gian O (mn).

c. Kiểm nghiệm thuật toán

a	A	C	G	T
$bmBc[a]$	1	6	2	8

Searching phase

First attempt:



Shift by 1 ($bmBc$[A])

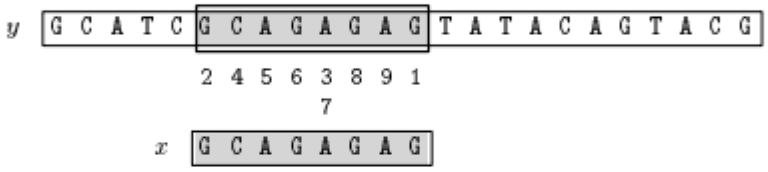
Second attempt:

Shift by 2 ($bmBc[\texttt{G}]$)

Third attempt:

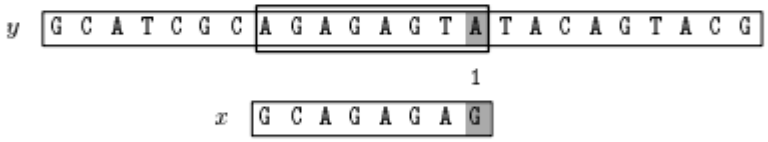
Shift by 2 ($bmBc[\texttt{G}]$)

Fourth attempt:



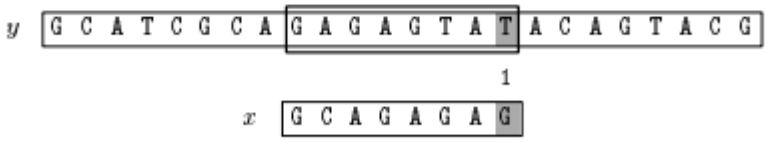
Shift by 2 ($bmBc[\texttt{G}]$)

Fifth attempt:



Shift by 1 ($bmBc[\texttt{A}]$)

Sixth attempt:



Shift by 8 ($bmBc[\texttt{T}]$)

Seventh attempt:

Shift by 2 ($bmBc[\texttt{G}]$)

Kết luận: Thuật toán sử dụng 18 phép so sánh ký tự văn bản

d. Lập trình thuật toán

```
void RAITA(char *x, int m, char *y, int n) {
   int j, bmBc[ASIZE];
   char c, firstCh, *secondCh, middleCh, lastCh;

   /* Preprocessing */
   preBmBc(x, m, bmBc);
   firstCh = x[0];
   secondCh = x + 1;
   middleCh = x[m/2];
   lastCh = x[m - 1];

   /* Searching */
   j = 0;
   while (j <= n - m) {
      c = y[j + m - 1];
      if (lastCh == c && middleCh == y[j + m/2] &&
          firstCh == y[j] &&
          memcmp(secondCh, y + j + 1, m - 2) == 0)
         OUTPUT(j);
      j += bmBc[c];
   }
}
```