Tìm kiếm chuỗi

I. Mở đầu

Dữ liệu trong máy tính được lưu trữ dưới rất nhiều dạng khác nhau, nhưng sử dụng chuỗi vẫn là một trong những cách rất phổ biến. Trên chuỗi các đơn vị dữ liệu không có ý nghĩa quan trọng bằng cách sắp xếp của chúng. Ta có thể thấy các dạng khác nhau của chuỗi như ở các file dữ liệu, trên biểu diễn của các gen, hay chính văn bản chúng ta đang đọc.

Một phép toán cơ bản trên chuỗi là đối sánh mẫu (pattern matching), bài toán yêu cầu ta tìm ra một hoặc nhiều vị trí xuất hiện của mẫu trên một văn bản.. Trong đó mẫu và văn bản là các chuỗi có độ dài N và M ($M \leq N$), tập các ký tự được dùng gọi là bảng chữ cái $\sum$, có số lượng là δ.

Việc đối sánh mẫu diễn ra với nhiều lần thử trên các đoạn khác nhau của văn bản. Trong đó *cửa sổ* là một chuỗi M ký tự liên tiếp trên văn bản. Mỗi lần thử chương trình sẽ kiểm tra sự giống nhau giữa mẫu với cửa số hiện thời. Tùy theo kết quả kiểm tra cửa sổ sẽ được dịch đi sang phải trên văn bản cho lần thử tiếp theo.

Trong trình bày này chúng ta sẽ quan tâm đến việc tìm kiếm tất cả các vị trí xuất hiện của mẫu trên một văn bản. Cài đặt sẽ dùng một hàm ra : Output để thông báo vị trí tìm thấy mẫu.

II. Thuật toán Brute Force

Có lẽ cái tên của thuật toán này đã nói lên tất cả (brute nghĩa là xúc vật, force nghĩa là sức mạnh). Thuật toán brute force thử kiểm tra tất cả các vị trí trên văn bản từ 1 cho đến n-m+1. Sau mỗi lần thử thuật toán brute force dịch mẫu sang phải một ký tự cho đến khi kiểm tra hết văn bản.

Thuật toán brute force không cần công việc chuẩn bị cũng như các mảng phụ cho quá trình tìm kiếm. Độ phức tạp tính toán của thuật toán này là $O(n*m)$

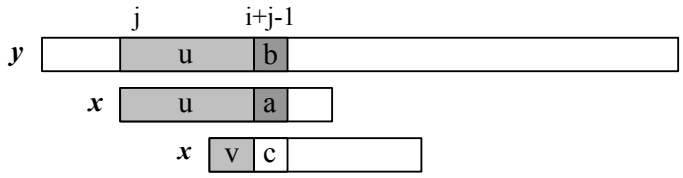
```
function IsMatch(const X: string; m: integer;
                 const Y: string; p: integer): boolean;
var
  i: integer;
begin
  IsMatch := false;
  Dec(p);
  for i := 1 to m do
    if X[i] <> Y[p + i] then Exit;
  IsMatch := true;
end;

procedure BF(const X: string; m: integer;
             const Y: string; n: integer);
var
  i: integer;
begin
  for i := 1 to n - m + 1 do
    if IsMatch(X, m, Y, i) then
      Output(i);   { Thông báo tìm thấy mẫu tại vị trí i của văn bản }
end;
```

III. Thuật toán Knuth-Morris-Pratt

Thuật toán Knuth-Morris-Pratt là thuật toán có độ phức tạp tuyến tính đầu tiên được phát hiện ra, nó dựa trên thuật toán brute force với ý tưởng lợi dụng lại những thông tin của lần thử trước cho lần sau. Trong thuật toán brute force vì chỉ dịch cửa sổ đi một ký tự nên có đến m-1 ký tự của cửa sổ mới là những ký tự của cửa sổ vừa xét. Trong đó có thể có rất nhiều ký tự đã được so sánh giống với mẫu và bây giờ lại nằm trên cửa sổ mới nhưng được dịch đi về vị trí so sánh với mẫu. Việc xử lý những ký tự này có thể được tính toán trước rồi lưu lại kết quả. Nhờ đó lần thử sau có thể dịch đi được nhiều hơn một ký tự, và giảm số ký tự phải so sánh lại.

Xét lần thử tại vị trí j, khi đó cửa sổ đang xét bao gồm các ký tự $y[j…j+m-1]$ giả sử sự khác biệt đầu tiên xảy ra giữa hai ký tự $x[i]$ và $y[j+i-1]$.
Khi đó $x[1…i]=y[j…i+j-1]=u$ và $a=x[i]\neq y[i+j]=b$. Với trường hợp này, dịch cửa sổ phải thỏa mãn v là phần đầu của xâu x khớp với phần đuôi của xâu u trên văn bản. Hơn nữa ký tự c ở ngay sau v trên mẫu phải khác với ký tự a. Trong những đoạn như v thoả mãn các tính chất trên ta chỉ quan tâm đến đoạn có độ dài lớn nhất.



Dịch cửa sổ sao cho *v* phải khớp với *u* và *c* ≠ *a*

Thuật toán Knuth-Morris-Pratt sử dụng mảng `Next[i]` để lưu trữ độ dài lớn nhất của xâu `v` trong trường hợp xâu *u=x[1…i-1]*. Mảng này có thể tính trước với chi phí về thời gian là ***O****(m)* (việc tính mảng `Next` thực chất là một bài toán qui hoạch động một chiều).

Thuật toán Knuth-Morris-Pratt có chi phí về thời gian là ***O****(m+n)* với nhiều nhất là 2n-1 lần số lần so sánh ký tự trong quá trình tìm kiếm.

```
procedure preKMP(const X: string; m: integer;
                 var Next: array of integer);
var
  i, j: integer;
begin
  i := 1;
  j := 0;
  Next[1] := 0;
  while (i <= m) do
  begin
    while (j > 0)and(X[i] <> X[j]) do j := Next[j];
    Inc(i);
    Inc(j);
    if X[i] = X[j] then Next[i] := Next[j] {v khớp với u và c≠a}
    else Next[i] := j;
  end;
end;

procedure KMP(const X: string; m: integer;
              const Y: string; n: integer);
var
  i, j: integer;
  Next: ^TIntArr; { TIntArr = array[0..maxM] of integer }
begin
  GetMem(Next, (m + 1)*SizeOf(Integer));
  preKMP(X, m, Next^);
  i := 1;
  j := 1;
  while (j <= n) do
  begin
    {dịch đi nếu không khớp}
    while (i > 0)and(X[i] <> Y[j]) do i := Next^[i];
    Inc(i);
    Inc(j);
    if i > m then
    begin
      Output(j - i + 1);
      i := Next^[i];
    end;
  end;
  FreeMem(Next, (m + 1)*SizeOf(Integer));
End;
```

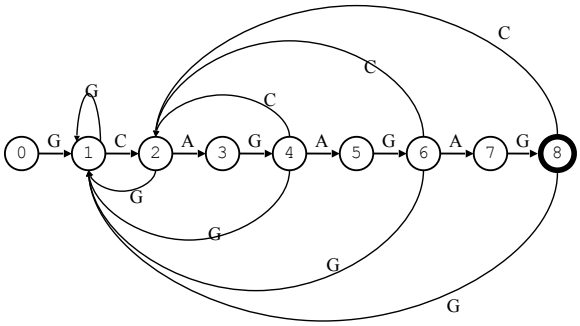
IV. Thuật toán Deterministic Finite Automaton (máy automat hữu hạn)

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

Thuật toán này có phần giống thuật toán Knuth-Morris-Pratt trong việc nhảy về trạng thái trước khi gặp một ký tự không khớp, nhưng thuật toán DFA có sự đánh giá chính xác hơn vì

việc xác định vị trí nhảy về dựa trên *ký tự* không khớp của văn bản (trong khi thuật toán KMP lùi về chỉ dựa trên vị trí không khớp).

Với xâu mẫu là GCAGAGAG ta có hệ automat sau



Với ví dụ ở hình trên ta có:

- Nếu đang ở trạng thái 2 gặp ký tự A trên văn bản sẽ chuyển sang trạng thái 3
- Nếu đang ở trạng thái 6 gặp ký tự C trên văn bản sẽ chuyển sang trạng thái 2
- Trạng thái 8 là trạng thái cuối cùng, nếu đạt được trạng thái này có nghĩa là đã tìm thất một xuất hiện của mẫu trên văn bản
- Trạng thái 0 là trạng thái mặc định(các liên kết không được biểu thị đều chỉ về trạng thái này), ví dụ ở nút 5 nếu gặp bất kỳ ký tự nào khác G thì đều chuyển về trạng thái 0

Việc xây dựng hệ automat khá đơn giản khi được cài đặt trên ma trận kề. Khi đó thuật toán có thời gian xử lý là $O(n)$ và thời gian và bộ nhớ để tạo ra hệ automat là $O(m*\delta)$ (tùy cách cài đặt)

Nhưng ta nhận thấy rằng trong DFA chỉ có nhiều nhất m cung thuật và m cung nghịch, vì vậy việc lưu trữ các cung không cần thiết phải lưu trên ma trận kề mà có thể dùng cấu trúc danh sách kề Forward Star để lưu trữ. Như vậy thời gian chuẩn bị và lượng bộ nhớ chỉ là $O(m)$. Tuy nhiên thời gian tìm kiếm có thể tăng lên một chút so với cách lưu ma trận kề.

Cài đặt dưới đây xin được dùng cách đơn giản(ma trận kề)

Đỗ Huy Hoàng

```
type
  TAut = array[0..maxM, 0..maxd] of integer;

procedure preAUT(const X: string; m: integer; var G: TAut);
var
  i, j, prefix, cur, c, newState: integer;
begin
  FillChar(G, SizeOf(G), 0);
  cur := 0;
  for i := 1 to m do
  begin
    prefix := G[cur, Ord(X[i])]; {x[1..prefix]=x[i-prefix+1..i]}
    newState := i;
    G[cur, Ord(X[i])] := newState;
    for c := 0 to maxd do  {copy prefix -> newState }
      G[newState, c] := G[prefix, c];
    cur := newState;
  end;
end;

procedure AUT(const X: string; m: integer;
              const Y: string; n: integer);
var
  G: ^TAut;
  state, i: integer;
begin
  New(G);
  preAUT(X, m, G^);
  state := 0;
  for i := 1 to n do
  begin
    state := G^[state, Ord(Y[i])]; {chuyển trạng thái}
    if state = m then Output(i - m + 1);
  end;
  Dispose(G);
end;
```

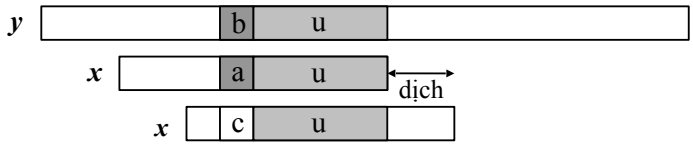
IV. Thuật toán Boyer-Moore

Thuật toán Boyer Moore là thuật toán có tìm kiếm chuỗi rất có hiệu quả trong thực tiễn, các dạng khác nhau của thuật toán này thường được cài đặt trong các chương trình soạn thảo văn bản.

Khác với thuật toán Knuth-Morris-Pratt (KMP), thuật toán Boyer-Moore kiểm tra các ký tự của mẫu từ phải sang trái và khi phát hiện sự khác nhau đầu tiên thuật toán sẽ tiến hành dịch cửa sổ đi Trong thuật toán này có hai cách dịch của sổ:

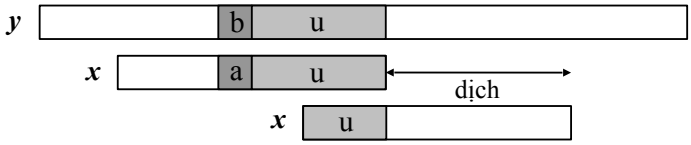
Cách thứ 1: gần giống như cách dịch trong thuật toán KMP, dịch sao cho những phần đã so sánh trong lần trước khớp với những phần giống nó trong lần sau.
Trong lần thử tại vị trí j, khi so sánh đến ký tự i trên mẫu thì phát hiện ra sự khác nhau, lúc đó $x[i+1…m]=y[i+j...j+m-1]=u$ và $a=x[i]\neq y[i+j-1]=b$ khi đó thuật toán sẽ

dịch cửa sổ sao cho đoạn `u=y[i+j…j+m-1]` giống với một đoạn mới trên mẫu (trong các phép dịch ta chọn phép dịch nhỏ nhất)



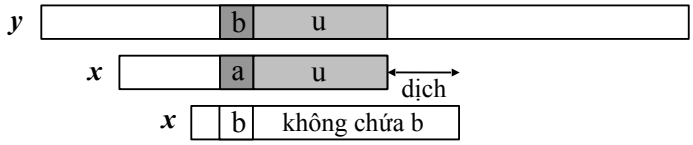
Dịch sao cho *u* xuất hiện lại và *c* ≠ *a*

Nếu không có một đoạn nguyên vẹn của u xuất hiện lại trong x, ta sẽ chọn sao cho phần đôi dài nhất của u xuất hiện trở lại ở đầu mẫu.



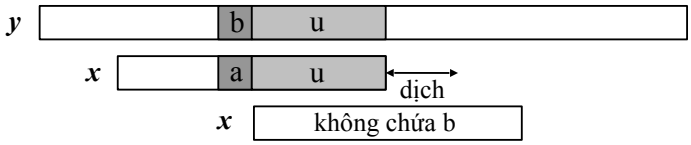
Dịch để một phần đôi của u xuất hiện lại trên x

Cách thứ 2: Coi ký tự đầu tiên không khớp trên văn bản là `b=y[i+j-1]` ta sẽ dịch sao cho có một ký tự giống `b` trên xâu mẫu khớp vào vị trí đó (nếu có nhiều vị trí xuất hiện `b` trên xâu mẫu ta chọn vị trí phải nhất)



Dịch để ký tự b ăn khớp với văn bản.

Nếu không có ký tự b nào xuất hiện trên mẫu ta sẽ dịch cửa sổ sao cho ký tự trái nhất của cửa sổ vào vị trí ngay sau ký tự `y[i+j-1]=b` để đảm bảo sự ăn khớp



Dịch khi b không xuất hiện trong *x*

Trong hai cách dịch thuật toán sẽ chọn cách dịch có lợi nhất.

Trong cài đặt ta dùng mảng `bmGs` để lưu cách dịch 1, mảng `bmBc` để lưu phép dịch thứ 2(ký tự không khớp). Việc tính toán mảng `bmBc` thực sự không có gì nhiều để bàn. Nhưng việc tính trước mảng `bmGs` khá phức tạp, ta không tính trực tiếp mảng này mà tính gián tiếp thông qua mảng suff. Có `suff[i]`=max{`k | x[i-k+1…i]=x[m-k+1…m]`}

Các mảng `bmGs` và `bmBc` có thể được tính toán trước trong thời gian tỉ lệ với $O(m+\delta)$. Thời gian tìm kiếm (độ phức tạp tính toán) của thuật toán Boyer-Moore là $O(m*n)$. Tuy nhiên với những bản chữ cái lớn thuật toán thực hiện rất nhanh. Trong trường hợp tốt chi phí thuật toán có thể xuống đến $O(n/m)$ là chi phí thấp nhất của các thuật toán tìm kiếm hiện đại có thể đạt được.

```
procedure preBmBc(const X: string; m: integer;
                  var bmBc: array of integer);
var
  i: integer;
begin
  for i := 0 to maxd - 1 do bmBc[i] := m;
  for i := 1 to m - 1 do bmBc[Ord(X[i])] := m - i;
end;

procedure suffixes(const X: string; m: integer;
                   var suff: array of integer);
var
  right, left, i: integer;
begin
  suff[m] := m;
  left := m;
  for i := m - 1 downto 1 do
    if (i > left)and(suff[i + m - right] < i - left) then
      suff[i] := suff[i + m - right]
    else
    begin
      if (i < left) then left := i;
      right := i;
      while (left >= 1)and(X[left] = X[left + m - right]) do
        Dec(left);
      suff[i] := right - left; {X[left…right] = X[m+left-right…m]}
    end;
end;

procedure preBmGs(const X: string; m: integer;
                  var bmGs: array of integer);
var
  i, j: integer;
  suff: ^TIntArr;
begin
  GetMem(suff, (m + 1)*SizeOf(Integer));
  suffixes(X, m, suff^);  {Tính mảng suff}
  for i := 1 to m do bmGs[i] := m;
  j := 0;
  for i := m downto 0 do
    if (i = 0)or(suff^[i] = i) then
      while (j < m - i) do
      begin
        {Nếu bmGs[j] chưa có giá trị thì điền vào}
        if bmGs[j] = m then bmGs[j] := m - i;
        Inc(j);
      end;
  for i := 1 to m - 1 do bmGs[m - suff^[i]] := m - i; {đảo lại}
  FreeMem(suff, (m + 1)*SizeOf(Integer));
end;

procedure BM(const X: string; m: integer;
             const Y: string; n: integer);
var
  i, j: integer;
  bmBc, bmGs: ^TIntArr;
begin
```

Thuật toán Boyer-Moore có thể đạt tới chi phí ***O****(n/m)* là nhờ có cách dịch thứ 2 "ký tự không khớp". Cách chuyển cửa sổ khi gặp "ký tự không khớp" cài đặt vừa đơn giản lại rất hiệu quả trong các bảng chữ cái lớn nên có nhiều thuật toán khác cũng đã lợi dụng các quét mẫu từ phải sang trái để sử dụng cách dịch này.

Tuy nhiên chi phí thuật toán của Boyer-Moore là ***O****(m*n)* vì cách dịch thứ nhất của thuật toán này không phân tích triệt để các thông tin của những lần thử trước, những đoạn đã so sánh rồi vẫn có thể bị so sánh lại. Có một vài thuật toán đã cải tiến cách dịch này để đưa đến chi phí tính toán của thuật toán Boyer-Moore là tuyến tính. (xin tham khảo thêm trong chương trình demo đi kèm)

VII. Thuật toán Karp-Rabin

Karp-Rabin bài toán tìm kiếm chuỗi không khác nhiều so với bài toán tìm kiếm chuẩn. Tại đây một hàm băm được dùng để tránh đi sự so sánh không cần thiết. Thay vì phải so sánh tất các vị trí của văn bản, ta chỉ cần so sánh những cửa sổ bao gồm những ký tự "có vẻ giống" mẫu.

Trong thuật toán này hàm băm phải thỏa mãn một số tính chất như phải dễ dàng tính được trên chuỗi, và đặc biệt công việc tính lại phải đơn giản để ít ảnh hưởng đến thời gian thực hiện của thuật toán. Và hàm băm được chọn ở đây là:

$$hash(w[i...i+m-1]) = h = (w[i]*d^{m-1} + w[i+1]*d^{m-2} + ... w[i+m-1]*d^{0}) \bmod q$$

Việc tính lại hàm băm sau khi dịch cửa sổ đi một ký tự chỉ đơn gian như sau:

$$h = ((h - w[i]*d^{m-1})*d + w[i+m]$$

Trong bài toán này ta có thể chọn d = 2 để tiện cho việc tính toán a*2 tương đương a shl 1. Và không chỉ thế ta chọn q = MaxLongint khi đó phép mod q không cần thiết phải thực hiện vì sự tràn số trong tính toán chính là một phép mod có tốc độ rất nhanh.

Việc chuẩn bị trong thuật toán Karp-Rabin có độ phức tạp ***O****(m)*. Tuy vậy thời gian tìm kiếm lại tỉ lệ với ***O****(m*n)* vì có thể có nhiều trường hợp hàm băm của chúng ta bị lừa và không phát huy tác dụng. Nhưng đó chỉ là những trường hợp đặc biệt, thời gian tính toán của thuật toán KR trong thực tế thường tỉ lệ với ***O****(n+m)*. Hơn nữa thuật toán KR có thể dễ dàng mở rộng cho các mẫu, văn bản dạng 2 chiều, do đó khiến cho nó trở nên hữu ích hơn so với các thuật toán còn lại trong việc xử lý ảnh.

```
procedure KR(const X: string; m: integer;
             const Y: string; n: integer);
var
  dM, hx, hy: longint;
  i, j: integer;
begin
  {$Q-}   { Disable arithmetic overflow checking }
  dM := 1;
  for i := 1 to m - 1 do dM := dM shl 1;
  hx := 0;
  hy := 0;
  for i := 1 to m do
  begin
    hx := (hx shl 1) + Ord(X[i]);
    hy := (hy shl 1) + Ord(Y[i]);
  end;
  j := 1;
  while j <= n - m do
  begin
    if hx = hy then
      if IsMatch(X, m, Y, j) then Output(j);
      {hàm IsMatch trong phần BruteForce}
    hy := ((hy - Ord(Y[j])*dM) shl 1) + Ord(Y[j + m]); {Rehash}
    Inc(j);
  end;
  if hx = hy then
    if IsMatch(X, m, Y, j) then Output(j);
end;
```

VIII. Cách thuật toán khác

Một số thuật toán nêu trên chưa phải là tất cả các thuật toán tìm kiếm chuỗi hiện có. Nhưng chúng đã đại diện cho đa số các tư tưởng dùng để giải bài toán tìm kiếm chuỗi.

Các thuật toán so sánh mẫu lần lượt từ trái sang phải thường là các dạng cải tiến (và cải lùi) của thuật toán Knuth-Morris-Pratt và thuật toán sử dụng Automat như: Forward Dawg Matching, Apostolico-Crochemore, Not So Naive, …

Các thuật toán so sánh mẫu từ phải sang trái đều là các dạng của thuật toán Boyer-Moore. Phải nói lại rằng thuật toán BM là thuật toán tìm kiếm rất hiệu quả trên thực tế nhưng độ phức tạp tính toán lý thuyết lại là $O(m*n)$. Chính vì vậy những cải tiến của thuật toán này cho độ phức tạp tính toán lý thuyết tốt như: thuật toán Apostolico-Giancarlo đánh dấu lại những ký tự đã so sánh rồi để khỏi bị so sánh lặp lại, thuật toán Turbo-BM đánh giá chặt chẽ hơn các thông tin trước để có thể dịch được xa hơn và ít bị lặp, … Còn có một số cải tiến khác của thuật toán BM không làm giảm độ phức tạp lý thuyết mà dựa trên kinh nghiệm để có tốc độ tìm kiếm nhanh hơn trong thực tế. Ngoài ra, một số thuật toán kết hợp quá trình tìm kiếm của BM vào hệ thống Automat mong đạt kết quả tốt hơn.

Các thuật toán so sánh mẫu theo thứ tự đặc biệt

- Thuật toán Galil-Seiferas và Crochemore-Perrin chúng chia mẫu thành hai đoạn, đầu tiên kiểm tra đoạn ở bên phải rồi mới kiểm tra đoạn bên trái với chiều từ trái sang phải.
- Thuật toán Colussi và Galil-Giancarlo lại chia mẫu thành hai tập và tiến hành tìm kiếm trên mỗi tập với một chiều khác nhau.
- Thuật toán Optimal Mismatch và Maximal Shift sắp xếp thứ tự mẫu dựa vào mật độ của ký tự và khoảng dịch được.
- Thuật toán Skip Search, KMP Skip Search và Alpha Skip Search dựa sự phân bố các ký tự để quyết đinh vị trí bắt đầu của mẫu trên văn bản.

Các thuật toán so sánh mẫu theo thứ tự bất kỳ những thuật toán này có thể tiến hành so sánh mẫu với cửa sổ theo một thứ thự ngẫu nhiên. Những thuật toán này đều có cài đặt rất đơn giản và thường sử dụng chiêu ký tự không khớp của thuật toán Boyer-Moore. Có lẽ loại thuật toán này dựa trên ý tưởng càng so sánh loạn càng khó kiếm test chết ☺

ĐHH