



TÌM KIẾM MẪU (Pattern Searching Algorithm)

NỘI DUNG:

- 6.1. Giới thiệu vấn đề
- 6.2. Thuật toán Brute-Force
- 6.3. Thuật toán Knuth-Morris-Partt
- 6.4. Thuật toán Boyer-More
- 6.5. Thuật toán Shift or
- 6.6. Thuật toán Automat hữu hạn
- 6.7. Cây hậu tố
- 6.8. CASE STUDY

6.1. 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 $X = (x_0, x_1, \dots, x_{m-1})$ có độ dài m . Văn bản $Y = (y_0, y_1, \dots, y_{n-1})$ 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 $X = (x_0, x_1, \dots, x_m)$, độ dài m .
- Văn bản $Y = (y_0, y_1, \dots, y_n)$, độ dài n .

Output:

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

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.
- **Tìm kiếm mẫu từ bất kỳ:** Horspool Algorithm, Boyer-Moore Algorithm, Smith Algorithm , Raita Algorithm.

Một số khái niệm và định nghĩa cơ bản về tìm kiếm mẫu:

Giả sử Alphabet là tập hợp (hoặc tập con) các mã ASCII. Một từ $w = (w_0, w_1, \dots, w_l)$ có độ dài l , $w_l = \text{null}$ giống như biểu diễn của ngôn ngữ C. Khi đó ta định nghĩa một số thuật ngữ sau:

- *Prefix (tiền tố)*. Từ u được gọi là tiền tố của từ w nếu tồn tại một từ v để $w = uv$ (v có thể là rỗng). Ví dụ: $u = \text{"AB"}$ là tiền tố của $w = \text{"ABCDEF"}$ và $u = \text{"com"}$ là tiền tố của $w = \text{"communication"}$.
- *Suffix (hậu tố)*. Từ v được gọi là hậu tố của từ w nếu tồn tại một từ u để $w = uv$ (u có thể là rỗng). Ví dụ: $v = \text{"EF"}$ là hậu tố của $w = \text{"ABCDEF"}$ và $v = \text{"tion"}$ là hậu tố của $w = \text{"communication"}$.
- *Factor (substring, subword)*. Một từ z được gọi là một xâu con, từ con hay nhân tố của từ w nếu tồn tại hai từ u, v (u, v có thể rỗng) sao cho $w = u z v$. Ví dụ từ $z = \text{"CD"}$ là factor của từ $w = \text{"ABCDEF"}$ và $z = \text{"muni"}$ là factor của $w = \text{"communication"}$.
- *Period (đoạn)*. Một số tự nhiên p được gọi là đoạn của từ w nếu với mọi i ($0 \leq i < m-1$) thì $w[i] = w[i+p]$. Giá trị đoạn nhỏ nhất của w được gọi là đoạn của w ký hiệu là $\text{pre}(w)$. Ví dụ $w = \text{"ABABCDEF"}$, khi đó tồn tại $p=2$.
- *Periodic (tuần hoàn)*. Từ w được gọi là tuần hoàn nếu đoạn của từ nhỏ hơn hoặc bằng $l/2$. Trường hợp ngược lại được gọi là không tuần hoàn. Ví dụ từ $w = \text{"ABAB"}$ là từ tuần hoàn. Từ $w = \text{"ABABCDEF"}$ là không tuần hoàn.

Một số khái niệm và định nghĩa cơ bản về tìm kiếm mẫu:

- *Basic word* (từ cơ sở). Từ w được gọi là từ cơ sở nếu nó không thể viết như lũy thừa của một từ khác. Không tồn tại z và k để $z^k = w$.
- *Boder word* (từ biên). Từ z được gọi là boder của w nếu tồn tại hai từ u, v sao cho $w = uz = zv$. Khi đó z vừa là tiền tố vừa là hậu tố của w . Trong tình huống này $|u| = |v|$ là một đoạn của w .
- *Reverse word* (Từ đảo). Từ đảo của từ w có độ dài l ký hiệu là $wR=(w_{r-1}, w_{r-2}, \dots, w_1, w_0)$.
- Deterministic Finite Automata (DFA). Một automat hữu hạn A là bộ bốn (Q, q_0, T, E) trong đó:
 - Q là tập hữu hạn các trạng thái.
 - q_0 là trạng thái khởi đầu.
 - T là tập con của Q là tập trạng thái dừng.
 - E là tập con của (Q, Σ, T) tập các chuyển dịch.

Ngôn ngữ $L(A)$ đoán nhận bởi A được định nghĩa:

$$\{w \in \Sigma^* : \exists q_0, \dots, q_n, n = |w|, q_n \in T, \forall 0 \leq i < n, (q_i, w[i], q_{i+1}) \in \delta\}$$

6.2. Thuật toán Brute-Force

Đặc điểm:

- Không có pha tiền xử lý.
- Sử dụng không gian nhớ phụ hằng số.
- Quá trình so sánh thực hiện theo bất kỳ thứ tự nào.
- Độ phức tạp thuật toán là $O(n.m)$;

Thuật toán Brute-Force:

Input :

- Xâu mẫu $X = (x_0, x_1, \dots, x_m)$, độ dài m.
- Văn bản nguồn $Y = (y_1, y_2, \dots, y_n)$ độ dài n.

Output:

- Mọi vị trí xuất hiện của X trong Y.

Formats: Brute-Force(X, m, Y, n);

Actions:

```
for ( j = 0; j <= (n-m); j++) { //đuỵệt từ trái qua phải xâu X  
    for (i =0; i<m && X[i] == Y[i+j]; i++) ; //Kiểm tra mẫu  
    if (i>=m) OUTPUT (j);  
}
```

EndActions.

6.3. Thuật toán Knuth-Morris-Pratt

Đặc điểm:

- Thực hiện từ trái sang phải.
- Có pha tiền xử lý với độ phức tạp $O(m)$.
- Độ phức tạp thuật toán là $O(n + m)$;

Thuật toán PreKmp: //thực hiện bước tiền xử lý

Input :

- Xâu mẫu $X = (x_0, x_1, \dots, x_m)$, độ dài m .

Output: Mảng giá trị $kmpNext[]$.

Formats:

- $\text{PreKmp}(X, m, kmpNext);$

Actions:

```
i = 1; kmpNext[0] = 0; len = 0; //kmpNext[0] luôn là 0
while (i < m) {
    if (X[i] == X[len]) { //Nếu X[i] = X[len]
        len++; kmpNext[i] = len; i++;
    }
    else { // Nếu X[i] != X[len]
        if (len != 0) { len = kmpNext[len-1]; }
        else { kmpNext[i] = 0; i++; }
    }
}
```

EndActions.

Kiểm nghiệm PreKmp (X , m , $kmpNext$):

- $X[] = "ABABCABAB"$, $m = 9$.

i=?	($X[i] == X[len]$)?	Len =?	kmpNext[i]=?
		Len =0	kmpNext[0]=0
i=1	('B'=='A'): No	Len =0	kmpNext[1]=0
i=2	('A'=='A'): Yes	Len =1	kmpNext[2]=1
i=3	('B'=='B'): Yes	Len=2	kmpNext[3]=2
i=4	('C'=='A'): No	Len=0	kmpNext[4]=0
i=5	('A'=='A'): Yes	Len=1	kmpNext[5]=1
i=6	('B'=='B'): Yes	Len=2	kmpNext[6]=2
i=7	('A'=='A'): Yes	Len=3	kmpNext[6]=3
i=8	('B'=='B'): Yes	Len=4	kmpNext[6]=4
Kết luận: $kmpNext[] = \{0, 0, 1, 2, 0, 1, 2, 3, 4\}$.			

Kiểm nghiệm PreKmp (X , m , $kmpNext$) với $X[] = "AABAACAAABAA"$, $m = 11$.

$i=?$	$(X[i]== X[len])?$	$Len =?$	$kmpNext[i]=?$
		Len =0	$kmpNext[0]=0$
$i=1$	('A'=='A'): Yes	Len =1	$kmpNext[1]=1$
$i=2$	('B'=='A'): No	Len =0	$kmpNext[2]=$ chưa xác định
$i=2$	('B'=='A'): No	Len=0	$kmpNext[2]=0$
$i=3$	('A'=='A'): Yes	Len=1	$kmpNext[3]=1$
$i=4$	('A'=='A'): Yes	Len=2	$kmpNext[4]=2$
$i=5$	('C'=='B'): No	Len=1	$kmpNext[5]=$ chưa xác định
$i=5$	('C'=='A'): No	Len=0	$kmpNext[5]=$ chưa xác định
$i=5$	('C'=='A'): No	Len=0	$kmpNext[5]=0$
$i=6$	('A'=='A'): Yes	Len =1	$kmpNext[6]=1$
$i=7$	('A'=='A'): Yes	Len =2	$kmpNext[7]=2$
$i=8$	('B'=='B'): Yes	Len=3	$kmpNext[8] = 3$
$i=9$	('A'=='A'): Yes	Len=4	$kmpNext[9] = 4$
$i=10$	('A'=='A'): Yes	Len=5	$kmpNext[10] = 5$
Kết luận: $kmpNext = \{0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5\}$			

Thuật toán Knuth-Morris-Partt:

Input :

- Xâu mẫu X =(x₀, x₁,..,x_m), độ dài m.
- Văn bản Y =(y₀, y₁,..,x_n), độ dài n.

Output:

- Tất cả vị trí xuất hiện X trong Y.

Formats: Knuth-Morris-Partt(X, m, Y, n);

Actions:

Bước 1(Tiền xử lý):

```
preKmp(x, m, kmpNext); //Tiền xử lý với độ phức tạp O(m)
```

Bước 2 (Lặp):

```
i = 0; j = 0;
```

```
while (i < n) {
```

```
    if ( X[j] == Y[i] ) { i++; j++; }
```

```
    if ( i == m ) {
```

```
        < Tìm thấy mẫu ở vị trí i-j>;
```

```
        j = kmpNext[j-1];
```

```
}
```

```
    else if (i < n && X[j] != Y[i] ) {
```

```
        if (j != 0) j = kmpNext[ j-1];
```

```
        else i = i + 1;
```

```
}
```

```
}
```

EndActions.

Kiểm nghiệm Knuth-Moriss-Patt (X, m, Y, n):

- X[] = “ABABCABAB”, m = 9.
- Y[] = “ABABDABACDABABCABAB”, n = 19

Bước 1 (Tiền xử lý). Thực hiện Prekmp(X, m, kmpNext) ta nhận được:

$$\text{kmpNext[]} = \{ 0, 0, 1, 2, 0, 1, 2, 3, 4 \}$$

Bước 2 (Lặp):

(X[i]==Y[i])?	(J ==9)?	I=? J=?	
(X[0]==Y[0]): Yes	No	i=1, j=1	
(X[1]==Y[1]): Yes	No	i=2, j=2	
(X[2]==Y[2]): Yes	No	i=3, j=3	
(X[3]==Y[3]): Yes	No	i=4, j=4	
(X[4]==Y[4]): No	No	i=4, j=2	
(X[2]==Y[4]): No	No	i=4, j=0	
(X[0]==Y[4]): No	No	i=5, j=0	
(X[0]==Y[5]): Yes	No	i=6, j=1	
(X[1]==Y[6]): Yes	No	i=7, j=2	
.....			

6.4. Thuật toán Boyer-More

Đặc điểm:

- Thực hiện từ phải sang trái.
- Pha tiền xử lý có độ phức tạp $O(m + \sigma)$.
- Pha tìm kiếm có Độ phức tạp thuật toán là $O(n.m)$;

Thuật toán Boyer-More:

Input :

- Xâu mẫu $X = (x_0, x_1, \dots, x_m)$, độ dài m .
- Văn bản $Y = (y_1, y_2, \dots, y_n)$ độ dài n .

Output:

- Đưa ra mọi vị trí xuất hiện của X trong Y .

Formats: $k = \text{Boyer-More}(X, m, Y, n);$

Actions:

Bước 1 (Tiền xử lý):

- Xây dựng tập hậu tố tốt của X : $\text{bmGs}[]$.
- Xây dựng tập ký tự không tốt của X : $\text{bmBc}[]$.

Bước 2 (Tìm kiếm):

- Tìm kiếm dựa vào $\text{bmGs}[]$ và $\text{bmBc}[]$.

EndActions.

6.4. Thuật toán Boyer-More

```
Void BM(char *x, int m, char *y, int n) {
    int i, j, bmGs[XSIZE], bmBc[ASIZE];
    /* Bước tiền xử lý */
    preBmGs(x, m, bmGs); // xây dựng good suffix
    preBmBc(x, m, bmBc); // xây dựng bad character shift
    /* Bước tìm kiếm */
    j = 0;
    while (j <= n - m) {
        for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);
        if (i < 0) {
            cout<<"Vi tri:"<<(j)<<endl;
            j += bmGs[0];
        }
        else
            j += MAX(bmGs[i], bmBc[y[i + j]] - m + 1 + i);
    }
}
```

6.5. Thuật toán Rabin-Karp

Đặc điểm:

- Sử dụng hàm băm (hash function).
- Thực hiện pha tiền xử lý với độ phức tạp $O(m)$.
- Độ phức tạp thuật toán là $O(n + m)$;

Thuật toán Rabin-Karp:

Input :

- Xâu mẫu $X = (x_0, x_1, \dots, x_m)$, độ dài m .
- Văn bản $Y = (y_0, y_1, \dots, y_n)$, độ dài n .

Output:

- Tất cả vị trí của X trong Y .

Formats: Rabin-Karp(X, m, Y, n);

Actions:

```
#define REHASH(a, b, h) (((((h) - (a)*d) << 1) + (b)) //Hàm băm
int d, hx, hy, i, j;
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]);}
j = 0;
while (j <= n-m) {
    if (hx == hy && memcmp(x, y + j, m) == 0) OUT(j);
    hy = REHASH(y[j], y[j + m], hy);
    ++j;
}
```

EndActions.

6.6. Thuật toán Automat hữu hạn

Đặc điểm:

- Xây dựng Automat hữu hạn đoán nhận ngôn ngữ Σ^*x .
- Sử dụng không gian nhớ phụ $O(m\sigma)$.
- Thực hiện pha tiền xử lý với thời gian $O(m\sigma)$;
- Pha tìm kiếm có độ phức tạp tính toán $O(n)$.

Mô tả phương pháp:

Một Aotomat hữu hạn đoán nhận từ x là $A(x) = (Q, q_0, T, E)$ đoán nhận ngôn ngữ Σ^*x được định nghĩa như sau:

- Q là tập các tiền tố của x: $Q = \{\emptyset, x[0], x[0..1], \dots, x[0..m-2], x\}$.
- $q_0 = \emptyset$.
- $T = x$.
- Với mỗi $q \in Q$ (q là một tiền tố của x) và $a \in \Sigma$ thì $(q, a, qa) \in E$ khi và chỉ khi qa cũng là tiền tố của x. Trong trường hợp khác $(q, a, p) \in E$ chỉ khi p là hậu tố dài nhất của qa cũng là một tiền tố của x.
- Automat hữu hạn $A(x)$ xây dựng cần không gian nhớ $O(m\sigma)$ và thời gian $O(m + \sigma)$.

6.7. Thuật toán Shift-Or

Đặc điểm:

- Sử dụng các toán tử thao tác bít (Bitwise).
- Hiệu quả trong trường hợp độ dài mẫu nhỏ hơn một từ máy.
- Thực hiện pha tiền xử lý với thời gian $O(m + \sigma)$;
- Pha tìm kiếm có độ phức tạp tính toán $O(n)$.

Thuật toán Shift-Or:

Input :

- Xâu mẫu $X = (x_0, x_1, \dots, x_m)$, độ dài m .
- Văn bản $Y = (y_1, y_2, \dots, y_n)$ độ dài n .

Output:

- Đưa ra mọi vị trí xuất hiện của X trong Y .

Formats: $k = \text{Shift-Or}(X, m, Y, n);$

Actions:

Bước 1 (Tiền xử lý):

$\text{PreSo}(X, m, S);$ //chuyển X thành tập các số.

Bước 2 (Tìm kiếm):

$\text{SO}(X, n, Y, m);$ //Tìm kiếm mẫu X trong Y

EndActions.

Bước tiền xử lý: PreSo

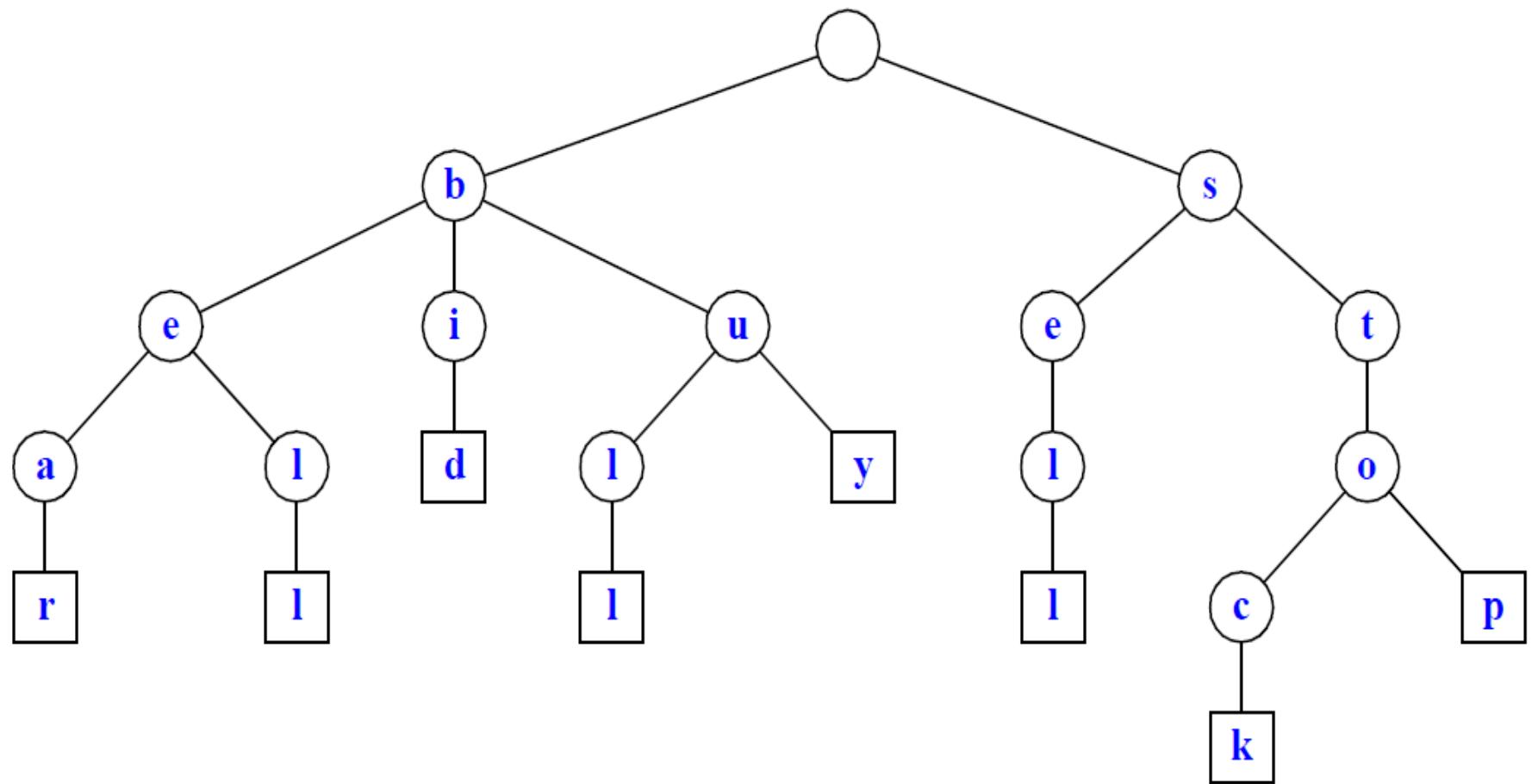
```
int preSo( char *x, int m, unsigned int S[]) {  
    unsigned int j, lim;  
    int i;  
    for (i = 0; i < ASIZE; ++i)  
        S[i] = ~0;  
    for (lim = i = 0, j = 1; i < m; ++i, j <<= 1) {  
        S[x[i]] &= ~j;  
        lim |= j;  
    }  
    lim = ~(lim>>1);  
    return(lim);  
}
```

Bước tìm kiếm: SO

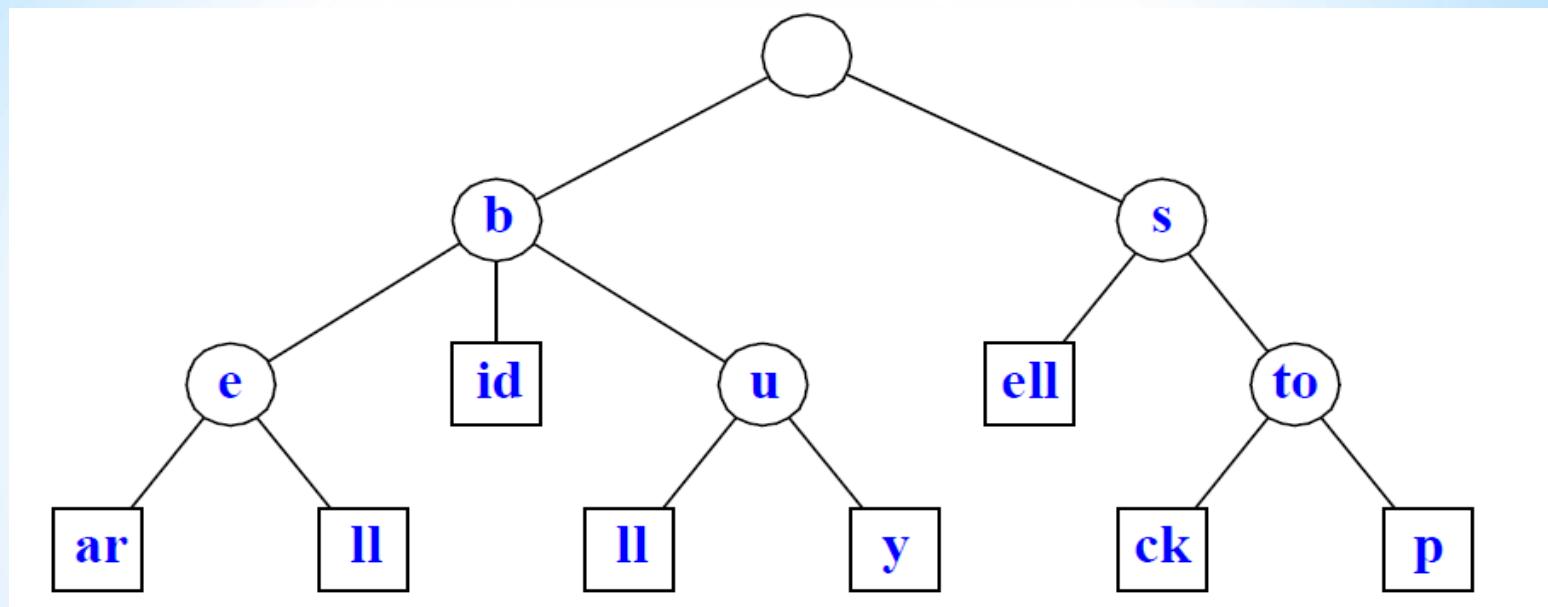
```
void SO(char *x, int m, char *y, int n) {
    unsigned int lim, state;
    unsigned int S[ASIZE];
    int j;
    if (m > WORD)
        cout<<"SO: Use pattern size <= word size";
    /* Preprocessing */
    lim = preSo(x, m, S);
    /* Searching */
    for (state = ~0, j = 0; j < n; ++j) {
        state = (state<<1) | S[y[j]];
        if (state < lim)
            cout<<"Vi tri:"<<(j - m + 1)<<endl;
    }
}
```

6.7. Cây hậu tố

Cây hậu tố: Một cây hậu tố của văn bản X là cây được nén cho tất cả các hậu tố của X. Ví dụ $X = \{\text{bear}, \text{bell}, \text{bid}, \text{bull}, \text{buy}, \text{sell}, \text{stock}, \text{stop}\}$. Khi đó cây hậu tố ban đầu của X được gọi là cây hậu tố chuẩn như dưới đây sau:



Cây hậu tố nén: Từ cây hậu tố chuẩn ta thực hiện nối các node đơn lẻ lại với nhau ta được một cây nén (compresses tree).



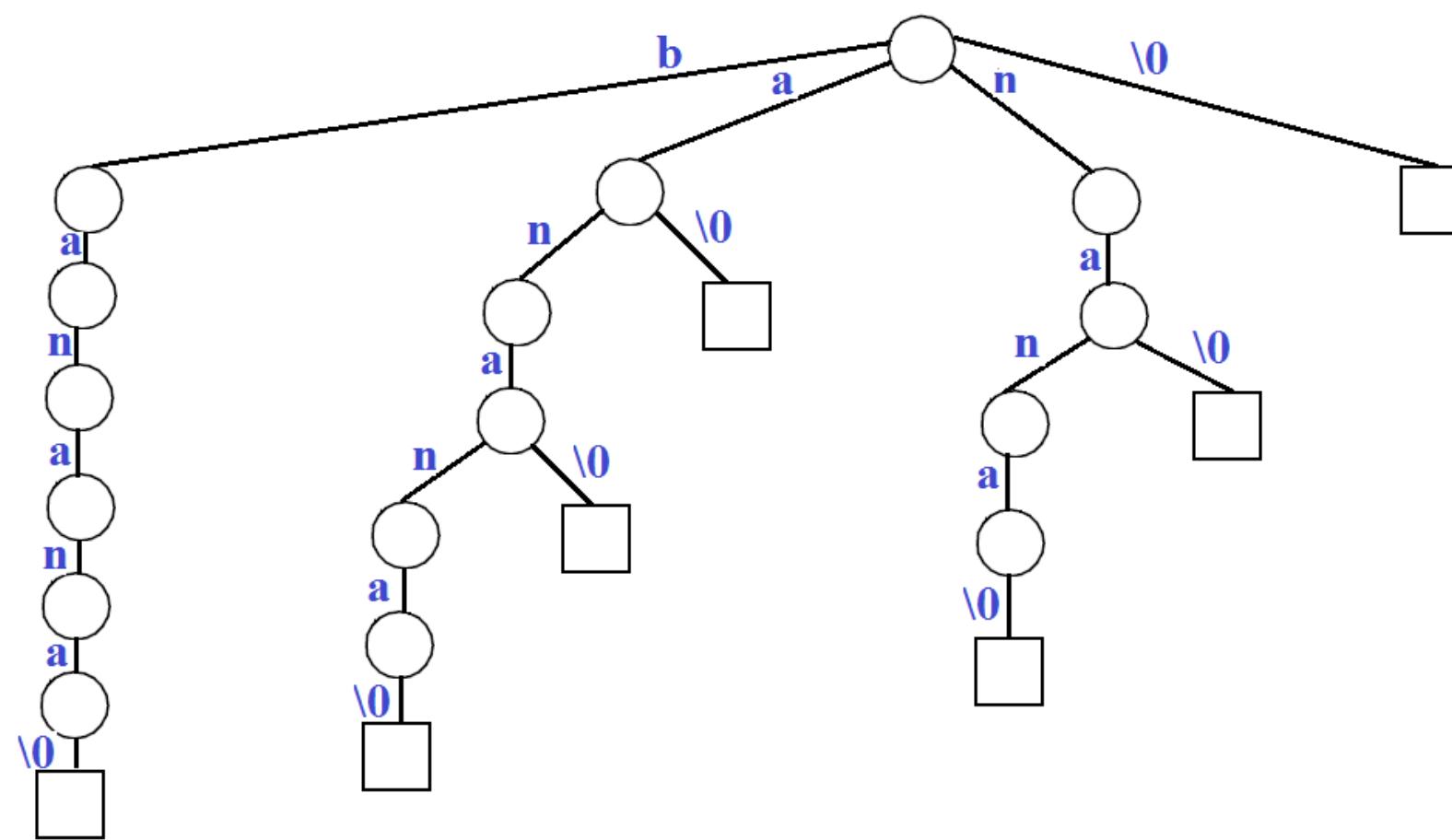
Phương pháp tạo cây hậu tố:

- Bước 1. Sinh ra tất cả các hậu tố của văn bản đã cho.
- Bước 2. Xem xét tất cả các hậu tố như một từ độc lập và xây dựng cây nén.

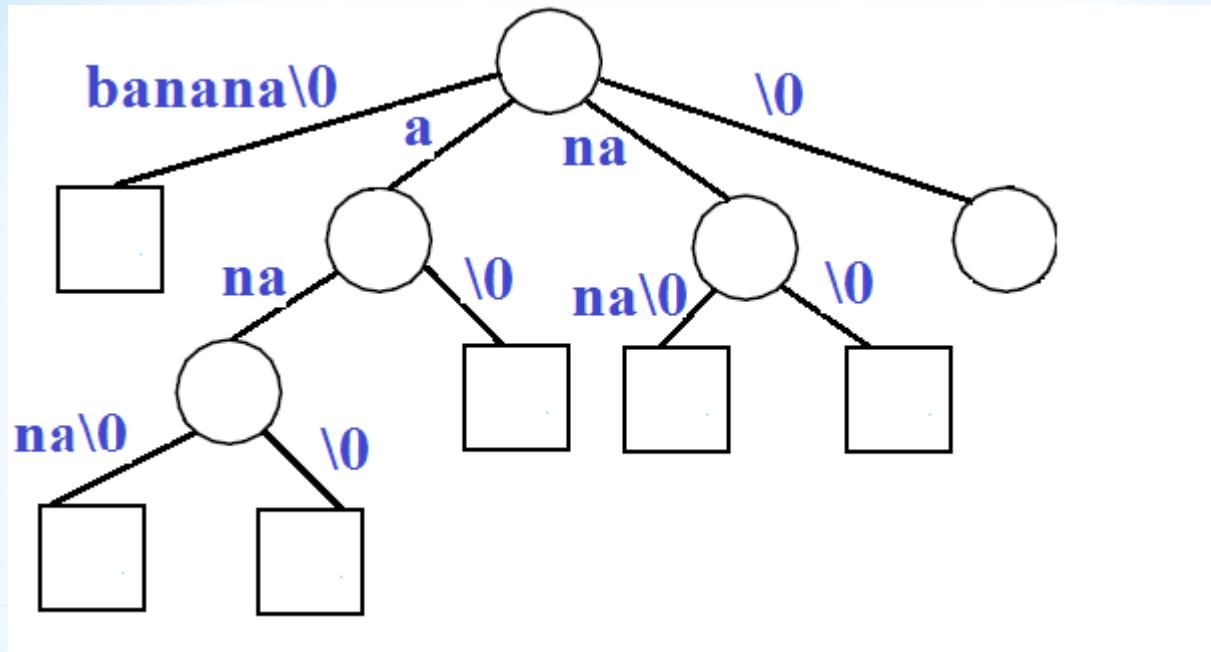
Ví dụ. Với X = “Banana” , khi đó thực hiện bước 1 ta nhận được các hậu tố:

Banana;	na;
anana;	a;
nana;	∅
ana;	

Cây hậu tố chuẩn được tạo ra như sau:



Bước 2. Kết hợp các node đơn để được cây hậu tố



CASE STUDY: Biểu diễn, đánh giá và cài đặt những thuật toán tìm kiếm mẫu dưới đây.

- 2.1. Giới thiệu vấn đề
- 2.2. Thuật toán Brute-Force
- 2.3. Thuật toán Knuth-Morris-Partt
- 2.4. Thuật toán Karp-Rabin
- 2.5. Thuật toán Shift or
- 2.6. Thuật toán Morris-Partt
- 2.7. Thuật toán Automat hữu hạn
- 2.8. Thuật toán Simon
- 2.9. Thuật toán Colussi
- 2.10. Thuật toán Galil-Giancarlo
- 2.11. Apostolico-Crochemore algorithm
- 2.12. Not So Naive algorithm
- 2.13. Turbo BM algorithm
- 2.14. Apostolico-Giancarlo algorithm
- 2.15. Reverse Colussi algorithm
- 2.16. Reverse Colussi algorithm
- 2.17. Horspool algorithm
- 2.18. Quick Search algorithm
- 2.19. Tuned Boyer-Moore algorithm
- 2.30. Zhu-Takaoka algorithm
- 2.31. Berry-Ravindran algorithm
- 2.32. Smith algorithm
- 2.33. Raita algorithm
- 2.34. Reverse Factor algorithm
- 2.35. Turbo Reverse Factor algorithm
- 2.36. Forward Dawg Matching algorithm
- 2.37. Backward Nondeterministic Dawg
- 2.38. Matching algorithm
- 2.39. Backward Oracle Matching algorithm
- 2.40 Galil-Seiferas algorithm
- 2.41. Two Way algorithm
- 2.42. String Matching on Ordered
- 2.43. Alphabets algorithm
- 2.44. Optimal Mismatch algorithm
- 2.45. Maximal Shift algorithm
- 2.46. Skip Search algorithm
- 2.47. KMP Skip Search algorithm
- 2.48. Alpha Skip Search algorithm