Algoritmos sobre Strings

Ingaramo Gastón[1]

[1]Facultad de Matemática, Astronomía y Física
Universidad Nacional de Córdoba

Training Camp 2012

Contenido

Contenidos

El problema a solucionar

Problema: Calcular las posiciones de las distintas apariciones de una cadena S dentro de otra cadena T, asumiendo $|S| < |T|$.

Contenidos

Algoritmo de string matching exhaustivo

Primer intento

A continuación vemos un algoritmo fuerza bruta para calcular las distintas apariciones de una cadena S en otra cadena T, asumiendo $|S| < |T|$.

```
1  void find_match(char *S, char *T) {
2    for(int i = 0; i < strlen(T)-strlen(S)+1; ++i) {
3      bool match = true;
4      for(int j = 0; j < strlen(S) && match; ++j) {
5        if(S[j] != T[i+j]) match = false;
6      }
7      // match == true <==> S == T[i..(i+j)]
8    }
9  }
```

Orden: $O(|S| \text{ x } |T|)$

Contenidos

Borde de una cadena
Definición

Definición: El borde de una cadena es un sufijo que también es prefijo.

Ejemplos:

- "*a*" es borde de "*algoritmia*".
- "*aba*" es borde de "*ababa*".

Lema: El borde de un borde es un borde.

$$S = \text{ABABABABABA},\ B_1 = \text{ABABA},\ B_2 = \text{ABA},\ B_3 = \text{A}$$

Borde de una cadena

Algoritmo para calcular bordes.

Algoritmo

```
1  int border[MAXN];
2  void borde(const char *s) {
3    int i = 1, j = -1, n = strlen(s);
4    border[0] = -1;
5    while(i < n) {
6      while(j >= 0 && s[i] != s[j+1]) j = border[j];
7      if (s[i] == s[j+1]) j++;
8      border[i++] = j;
9    }
10 }
```

Orden: $O(|S|)$

Ejemplo:

```
      S = "A  B  R  A  C  A  D  A  B  R  A"
borde[] ={-1,-1,-1, 0, 1, 2, 1, 2, 1, 2, 3}
```

Borde de una cadena
Observaciones

- Al hacer string matching fuerza bruta, estamos desperdiciando información.
- Al matchear los primeros 10 caracteres de S en T, si el caracter 11 difiere en ambos, sabemos que la proxima posición potencialmente útil es la que continua el borde del prefijo de S de tamaño 10.
- Podemos diseñar un algoritmo que detecte ocurrencias de una cadea S en otra cadena T, utilizando solo el algoritmo de bordes.

Borde de una cadena

Algoritmo de string matching mejorado.

Algoritmo

```
1  void find_match(const char *s, const char *t) {
2    int i = 0, j = -1, ns = strlen(s), nt = strlen(t);
3    while(i < nt) {
4      while(j >= 0 && s[j+1] != t[i]) j = border[j];
5      if (s[j+1] == t[i]) j++;
6      i++;
7      if (j+1 == ns) j = border[j];   // match!
8    }
9  }
```

Orden: $O(|S|)$

Este algoritmo es también conocido como algoritmo de Morris-Pratt.

Contenidos

Knuth-Morris-Pratt

Mejorando el algoritmo de Bordes

Observacion: Cuando fallamos al intentar matchear un caracter c porque a continuación habia un caracter d, solo debemos intentar con el maximo borde que no continue con d.

Es asi como llegamos al algoritmo de Knuth-Morris-Pratt o **KMP**.

Este algoritmo mejora el peor caso, haciendolo $log_{\phi}(|S|)$, donde $\phi = \frac{1+\sqrt{5}}{2}$.

Si bien la complejidad amortizada de ambos es $O(|S|)$ este es el algoritmo mas famoso de string matching.

Knuth-Morris-Pratt

Algoritmo KMP

```
1  int KMP_next[MAXN];
2  int KMP_compute(const char *s) {
3    int i = 0, j = -1, n = strlen(s);
4    KMP_next[0] = -1;
5    while(i < n) {
6      while(j >= 0 && s[i] != s[j]) j = KMP_next[j];
7      i++; j++;
8      if (i == n || s[i] != s[j]) KMP_next[i] = j;
9      else KMP_next[i] = KMP_next[j];
10   }
11 }
12 int KMP(const char *s, const char *t) {
13   int i = 0, j = 0, ns = strlen(s), nt = strlen(t);
14   while(i < nt) {
15     while(j >= 0 && s[j] != t[i]) j = KMP_next[j];
16     i++; j++;
17     if (j == ns) j = KMP_next[j];  // match!
18   }
19 }
```

Contenidos

Suffix Array
Definición

Definición: Un suffix array es un arreglo que contiene los indices de los sufijos ordenados alfabeticamentes.

Ejemplo: sea SA un suffix array y dos enteros i, j.
Siempre que $i < j$ tendremos que S[SA[i]..N] < S[SA[j]..N].

Si tomamos todos los sufijos y los ordenamos lexicograficamente con la funcion sort, obtendremos un algotimo $O(n^2 x log(n))$.

Suffix Array
Algoritmo

Algoritmo

```
1 char s[MAXN];
2 int SA[MAXN];
3 void build_suffix_array(const char *s) {
4   for(int i = 0; i < strlen(s); i++) {
5     SA[i] = i;
6   }
7   sort(SA,SA+strlen(s),custom_cmp);
8 }
```

Orden: $O(|S|^2 log(|S|))$

Ejemplo:

```
  S =  "D  C  B  A  B  C  D"
SA = { 3, 2, 1, 0, 4, 5, 6}
```

Suffix Array
Observacion

Claramente el tiempo de ordenar el arreglo no puede ser mejorado, veamos que pasa con la función de comparación.

Observacion: Dadas dos cadenas $s_1 = s_{11}s_{12}$ y $s_2 = s_{21}s_{22}$, con $|s_{ij}| = 2^k$, sabremos que
$s_1 < s_2$ **sii** $(s_{11} < s_{21}) \,||\, (s_{11} = s_{21}$ && $s_{12} < s_{22})$

Suffix Array
Observacion

Utilizando esta observación podemos calcular el suffix array en $k = log(n)$ pasos.

En cada paso haríamos un sort calculando la “posición final” de cada sufijo en la etapa $k + 1$ de longitud 2^{k+1}, utilizando para ello la información del paso anterior.

Suffix Array
Algoritmo

Pseudo codigo

```
1  n <- length(A)
2  for i <- 0, n - 1
3    P(0, i) <- position of Ai in the ordered array of A s characters
4    cnt <- 1
5    for k <- 1, [log2n] (ceil)
6      for i <- 0, n - 1
7        L(i) <- (P(k - 1, i) , P(k - 1, i + cnt) , i)
8      sort L
9      compute P(k, i) , i = 0, n - 1
10     cnt <- 2 * cnt
```

Orden: $O(|S|)$.

Lamentamos si estas viendo estas slides, el codigo fue mostrado en clase.

Suffix Array
LCP

LCP o Longest Common Prefix puede llegar a ser una resultarnos de utilidad.

SA	S = abazaba	LCP	
6	a	0	
4	aba	1	a
0	abazaba	3	aba
2	azaba	1	a
5	ba	0	
1	bazaba	2	ba
3	zaba	0	

Suffix Array
LCP

Notar que si queremos saber el LCP entre dos sufijos, podemos usar la matriz P para obtener matches de tamaño 2^k.

Algoritmo

```
1  int lcp(int x, int y) {
2    int k, ret = 0;
3    if (x == y) return N - x;
4    for (k = stp - 1; k >= 0 && x < N && y < N; k --)
5      if (P[k][x] == P[k][y])
6        x += 1 << k, y += 1 << k, ret += 1 << k;
7    return ret;
8  }
```

Orden: $O(log(|S|))$, existe una forma de calcularlo linealmente utilizando RMQ.

Suffix Array
Aplicaciones

- Minimal lexicographic rotation.
- Saber si una cadena esta contenida en otra en $O(|S| x log(|T|))$.
- Dadas n cadenas s_1, s_2, ... s_n, encontrar la subcadena común mas larga.

Contenidos

Trie
Introducción

Supongamos que tenemos un diccionario con 1000000 palabras, y nos hacen 100 queries para saber si una palabra está o nó en el diccionario.

Podemos resolverlo usando **SA** o **KMP**, pero eso sería demasiado complejo. Para estas tareas se utilizan los tries.

Un trie es un caso particular de un DAG en el cual las hojas son nodos finales que representan a las palabras del diccionario.

Trie
Problema

Si queremos encontrar todas las ocurrencias de un conjunto de palabras en otra, podríamos usar un trie para obtener un algoritmo de complejidad $O(|S|\text{x}|T|)$.

Pero podemos mejorar el trie para agregarle mas información que nos permita determinar donde continuar cada vez que fallamos, tal como lo hace **KMP**.

A ese algoritmo lo denominaremos Aho-Corasik o **AC**.

Contenidos

Aho-Corasik
Algoritmo

El algoritmo cuenta con varias funciones que veremos en parte.

Estructura

```
1  typedef struct {
2  bool final;
3  int next[DICTLEN];
4  int failure;
5  }node;
6
7  node trie[MAXN];
8  int triesz;
9
10 string alph;
11 int toDict(char a) { return alph.find(a); };
```

Aho-Corasik
Algoritmo

Construcción

```
1  void construct_trie(const vector<string> &P) {
2    triesz = 1;
3    for(int i = 0; i < P.size(); i++) {
4      int q = 0;  // comenzamos en el nodo inicial.
5      for(int j = 0; j < P[i].size(); j++) {
6        char w = toDict(P[i][j]);
7        // vamos insertando los nuevos nodos si hacen falta.
8        if(trie[q].next[w] == -1) trie[q].next[w] = ++triesz-1;
9        // avanzamos al siguiente nodo.
10       q = trie[q].next[w];
11     }
12     trie[q].final = true;
13   }
14 }
```

Aho-Corasik
Algoritmo

Función next

```
1 int next_state(int q, char w) {
2   if(q<0) return 0; // esto es por cuando calculo trie[0].failure (da -1 eso
        ).
3   if(trie[q].next[w] != -1) return trie[q].next[w];
4   if(trie[q].failure != -1) return next_state(trie[q].failure,w);
5   return 0;
6 }
```

Aho-Corasik
Algoritmo

Función compute failure

```
1  void compute_failure() {
2    queue<int> Q;   // vamos a procesar por niveles.
3    Q.push(0);
4    while(Q.size()) {
5      int q = Q.front(); Q.pop();
6      for(char w = 0; w < alph.size(); w++) if(trie[q].next[w] != -1) {
7        int p = trie[q].next[w];
8        trie[p].failure = next_state(trie[q].failure,w);
9        // propago el hecho de que es un nodo final.
10       if(trie[trie[p].failure].final)
11 trie[p].final = true;
12         Q.push(p);
13     }
14   }
15 }
```