

Semillero de Programación

Algoritmo de Knuth-Morris-Pratt

Ana Echavarría Juan Francisco Cardona

Universidad EAFIT

26 de abril de 2013

Contenido

- 1 Problemas semana anterior
 - Problema A - Numbering Roads
 - Problema B - Ubiquitous Religions
 - Problema C - Dark roads
- 2 String Matching
- 3 Algoritmo de Knuth-Morris-Pratt (KMP)
- 4 Tarea

Contenido

- 1 Problemas semana anterior
 - Problema A - Numbering Roads
 - Problema B - Ubiquitous Religions
 - Problema C - Dark roads

Problema A - Numbering Roads

- Hay que representar r calles con n números y 26 letras
- El número de calles que se tienen que representar con letras son $f = r - n$
- El número mínimo de letras que hay que usar es $\left\lceil \frac{f}{n} \right\rceil$
- Si ese número es mayor que las 26 letras disponibles, es imposible.

Implementación

```
1  int main(){
2      int roads, numbers;
3      int cases = 1;
4      while(cin >> roads >> numbers){
5          if (roads == 0 and numbers == 0) break;
6          int remaining = roads - numbers;
7          int ans = (remaining + numbers - 1) /  numbers;
8          if (ans <= 26){
9              printf("Case %d: %d\n", cases, ans);
10         }else{
11             printf("Case %d: impossible\n", cases);
12         }
13         cases++;
14     }
15     return 0;
16 }
```

Problema B - Ubiquitous Religions

- Utilizar Union-Find para representar los diferentes conjuntos de religiones.
- Inicialmente se asume que todas las personas tienen una religión diferente.
- Cada que dos personas tengan la misma religión se unen los conjuntos de esas dos personas.
- La respuesta es contar el número de conjuntos diferentes.

Implementación I

```
1  const int MAXN = 50005;
2  int p[MAXN];
3  bool seen[MAXN];
4
5  int find(int u){
6      if (p[u] == u) return u;
7      return p[u] = find(p[u]);
8  }
9  void join(int u, int v){
10     p[find(u)] = find(v);
11 }
12
13 int main(){
14     int n, m;
15     int run = 1;
16     while (cin >> n >> m){
17         if (n == 0 and m == 0) break;
18         for (int i = 0; i <= n; ++i) p[i] = i;
```

Implementación II

```
19     for (int i = 0; i < m; ++i){
20         int u, v; cin >> u >> v;
21         join(u, v);
22     }
23
24     memset(seen, false, sizeof(seen));
25     int count = 0;
26     for (int u = 1; u <= n; ++u){
27         int parent = find(u);
28         if (!seen[parent]){
29             count++;
30             seen[parent] = true;
31         }
32     }
33     printf("Case %d: %d\n", run++, count);
34 }
35 return 0;
36 }
```


Problema C - Dark roads

- Hay que hallar el mínimo costo de mantener iluminadas las calles de manera que haya un trayecto iluminado entre cualquier par de ciudades.
- Este costo corresponde con el minimum spanning tree del grafo.
- Utilizar el algoritmo de Prim / Kruskal.
- La respuesta es el dinero ahorrado, es decir el costo total menos el costo hallado por el MST.

Implementación I

```
1  const int MAXN = 200005;
2  typedef pair <int, int> edge;
3  bool visited[MAXN];
4  vector <pair <int, int> > g[MAXN];
5
6  int prim(int n){
7      for (int i = 0; i <= n; ++i) visited[i] = false;
8      int total = 0;
9
10     priority_queue<edge, vector <edge>, greater<edge> > q;
11     q.push(edge(0, 0));
12     while (!q.empty()){
13         int u = q.top().second;
14         int w = q.top().first;
15         q.pop();
16         if (visited[u]) continue;
17
18
```

Implementación II

```
19     visited[u] = true;
20     total += w;
21     for (int i = 0; i < g[u].size(); ++i){
22         int v = g[u][i].first;
23         int next_w = g[u][i].second;
24         if (!visited[v]){
25             q.push(edge(next_w, v));
26         }
27     }
28 }
29 return total;
30 }
31 int main(){
32     int n, m;
33     while (cin >> n >> m){
34         if (n == 0 and m == 0) break;
35
36         for (int i = 0; i <= n; ++i) g[i].clear();
37
```

Implementación III

```
38     int total_sum = 0;
39     for (int i = 0; i < m; ++i){
40         int x, y, c;
41         cin >> x >> y >> c;
42         total_sum += c;
43         g[x].push_back(make_pair(y, c));
44         g[y].push_back(make_pair(x, c));
45     }
46
47     cout << total_sum - prim(n) << endl;
48 }
49 return 0;
50 }
```

Contenido

2 String Matching

String Matching Problem

El “string mathching problem” o “needle in a haystack problem” se define así:

Entrada

Un string T (llamado texto o haystack) de tamaño n

Un string P (llamado patrón o needle) de tamaño m con $m \leq n$

Objetivo

Hallar todos los valores de i con $0 \leq i \leq n - m$ para los cuales

$$T[i + j] = P[j] \quad \forall j \in [0, m - 1]$$

Es decir, todas las posiciones en el string T donde ocurre la palabra P .

Algoritmo para String Matching

- ❶ Para i desde 0 hasta $n - 1$
- ❷ encontrado = true
- ❸ Para j desde 0 hasta $m - 1$
 - ❹ Si $T[i + j] \neq P[j]$
 - ❺ encontrado = false
 - ❻ break
- ❼ Si encontrado
- ❽ imprimir i

Algoritmo para String Matching

- ❶ Para i desde 0 hasta $n - 1$
- ❷ encontrado = true
- ❸ Para j desde 0 hasta $m - 1$
 - ❹ Si $T[i + j] \neq P[j]$
 - ❺ encontrado = false
 - ❻ break
- ❼ Si encontrado
- ❽ imprimir i

Complejidad

La complejidad de este algoritmo es $O(n \times m)$. ¿Será lo mejor que se puede lograr?

Contenido

3 Algoritmo de Knuth-Morris-Pratt (KMP)

Idea

- En el algoritmo anterior, cuando se define que la cadena P no ocurre en la posición i de T , se vuelve a empezar la comparación en la posición $i + 1$
- Sin embargo, este algoritmo no tiene en cuenta que si P no ocurrió en la posición i porque no coincidió el carácter j , esta información puede ser útil para evitar comparaciones innecesarias en la posición $i + 1$

Observación

Supongamos que los primeros j caracteres de P coincidieron con los caracteres de T cuando se compara desde la posición i , pero que el caracter $P[j]$ no coincide con el $T[i + j]$ esto es:

$$P[k] = T[i + k] \quad \forall k \in [0 \dots j) \quad \text{y} \quad P[j] \neq T[i + j]$$

Para cada $0 < k < j$, si $T[i + k \dots i + j - 1]$ no es un prefijo de P entonces P no puede ocurrir en la posición $i + k$.

En otras palabras, no puede haber una ocurrencia de P si en la posición $i + k$ si los caracteres de T que ya se compararon ($i + k$ al $i + j - 1$) no son un prefijo de P .

¿Cuál posición debe ser la siguiente en evaluarse?

- Sea $P[0 \dots j-1] = T[i \dots i+j-1]$ y $P[j] \neq T[i+j]$
- Se puede reanudar la búsqueda en la posición $i+k$ con el k más pequeño tal que $0 < k < j$ y

$T[i+k \dots i+j-1] = P[k \dots j-1]$ es un prefijo de P

- Pero $P[k \dots j-1]$ es un sufijo de $P[0 \dots j-1]$ luego la búsqueda se reanuda en la posición $i+k$ tal que $P[k \dots j-1]$ es el sufijo más largo de la cadena $P[0 \dots j-1]$ que también es sufijo de la misma.

Utilizando el borde

Definamos un borde de una cadena s como la cadena más larga que es a la vez prefijo y sufijo de s pero que es diferente de s .

La idea que tuvieron Knuth, Morris y Pratt para evitarse comparaciones dobles fue la de calcular el borde para cada prefijo del patrón / needle y usarlo para evitar comparaciones innecesarias.

Computando el borde de cada prefijo

Para cada prefijo de la cadena **ababaca** el borde es

ϵ

a

ab

aba

abab

ababa

ababac

ababaca

Computando el borde de cada prefijo

Para cada prefijo de la cadena **ababaca** el borde es

$\epsilon \rightarrow \epsilon$

a

ab

aba

abab

ababa

ababac

ababaca

Computando el borde de cada prefijo

Para cada prefijo de la cadena **ababaca** el borde es

ε → **ε**

a → **ε**

ab

aba

abab

ababa

ababac

ababaca

Computando el borde de cada prefijo

Para cada prefijo de la cadena **ababaca** el borde es

ε \rightarrow **ε**

a \rightarrow **ε**

ab \rightarrow **ε**

aba

abab

ababa

ababac

ababaca

Computando el borde de cada prefijo

Para cada prefijo de la cadena **ababaca** el borde es

ε → **ε**

a → **ε**

ab → **ε**

aba → **a**

abab

ababa

ababac

ababaca

Computando el borde de cada prefijo

Para cada prefijo de la cadena **ababaca** el borde es

ε → **ε**

a → **ε**

ab → **ε**

aba → **a**

abab → **ab**

ababa

ababac

ababaca

Computando el borde de cada prefijo

Para cada prefijo de la cadena **ababaca** el borde es

ϵ	$\rightarrow \epsilon$
a	$\rightarrow \epsilon$
ab	$\rightarrow \epsilon$
aba	$\rightarrow a$
abab	$\rightarrow ab$
ababa	$\rightarrow aba$
ababac	
ababaca	

Computando el borde de cada prefijo

Para cada prefijo de la cadena **ababaca** el borde es

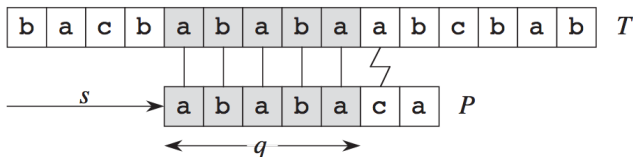
ϵ	$\rightarrow \epsilon$
a	$\rightarrow \epsilon$
ab	$\rightarrow \epsilon$
aba	$\rightarrow a$
abab	$\rightarrow ab$
ababa	$\rightarrow aba$
ababac	$\rightarrow \epsilon$
ababaca	

Computando el borde de cada prefijo

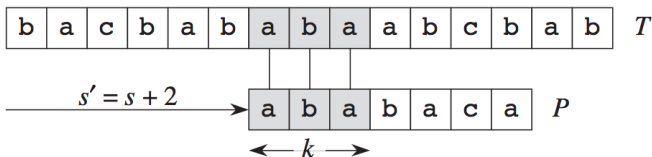
Para cada prefijo de la cadena **ababaca** el borde es

ϵ	$\rightarrow \epsilon$
a	$\rightarrow \epsilon$
ab	$\rightarrow \epsilon$
aba	$\rightarrow a$
abab	$\rightarrow ab$
ababa	$\rightarrow aba$
ababac	$\rightarrow \epsilon$
ababaca	$\rightarrow a$

Haciendo la comparación usando el borde



Comparando las cadenas, coincidieron los primeros 5 caracteres. El borde del prefijo de longitud 5 tiene longitud 3 por lo que la cadena se puede mover $5 - 3 = 2$ posiciones a la derecha y en esa posición 3 caracteres coinciden.



Comparación lineal

Utilizando el borde de cada prefijo se puede hacer una sola comparación para cada caracter del texto T , es decir que la comparación es lineal sobre la longitud del texto.

¿Cómo calcular el borde de cada prefijo?

Sea $\text{border}[i]$ la longitud del borde del prefijo de needle que termina en la posición i .

Para la cadena **abacabacabadab**

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
<i>needle</i>	a	b	a	c	a	b	a	c	a	b	a	d	a	b
<i>border</i>	0	0	1	0	1	2	3	4	5	6	7	0	1	2

Nótese que un borde de un borde de una cadena s es también un borde de s por ejemplo:

$s = \text{abacaba}$, s termina en la posición 6 por lo que su borde es $\text{border}[6] = 3 \rightarrow \text{aba}$.

aba termina en la posición 2 por lo que su borde es $\text{border}[2] = \text{border}[\text{border}[6] - 1] = 1 \rightarrow \text{a}$.

¿Cómo calcular el borde de cada prefijo?

Supongamos que se tiene calculado $\text{border}[0]$, $\text{border}[1]$,
... $\text{border}[k-1]$ y se quiere calcular $\text{border}[k]$.

Sabemos que $P[0 \dots \text{border}[k-1]-1]$ es el prefijo más largo
que también es sufijo de $P[0 \dots k-1]$. Para calcular $\text{border}[k]$
hay dos posibilidades:

- $P[\text{border}[k-1]] = P[k]$ en cuyo caso
 $\text{border}[k] = \text{border}[k-1] + 1$.
- $P[\text{border}[k-1]] \neq P[k]$ se debe buscar el siguiente
prefijo más grande de P que también sea un sufijo
 $P[0 \dots k-1]$. Este prefijo tiene la longitud de
 $\text{border}[\text{border}[k-1] - 1]$ — y ahora se deben comparar $P[k]$
con $P[\text{border}[\text{border}[k-1] - 1]]$.

Este proceso se hace iterativamente hasta que haya una
coincidencia de caracteres o hasta que se llegue a un prefijo
vacío.

Implementación

```
1  int m = needle.size();
2  vector<int> border(m);
3  border[0] = 0;
4
5  for (int i = 1; i < m; ++i) {
6      border[i] = border[i - 1];
7      // Mientras que el borde sea mayor que 0 y los caracteres no
        coincidan
8      while (border[i] > 0 and needle[i] != needle[border[i]]) {
9          border[i] = border[border[i] - 1];
10     }
11     // Si hubo coincidencia sumarle ese caracter a la longitud
12     if (needle[i] == needle[border[i]]) border[i]++;
13 }
```

Complejidad

Complejidad

En este algoritmo se cumple (aunque no es fácil de probar) que el while interno sólo se ejecuta máximo $m - 1$ veces en toda la ejecución del algoritmo.

Es por esto que la complejidad de hallar los bordes es en total $O(m)$.

Comparación

Como ya se mostró anteriormente, el arreglo de los bordes sirve para hacer la comparación rápidamente así.

- ❶ Hacer *seen* 0 (el número de caracteres de P han coincidido)
- ❷ Para i desde 0 hasta el tamaño de T
 - ❸ Mientras que $seen > 0$ y $T[i] \neq P[seen]$
 - ❹ Comparar desde el borde de $P[0 \dots seen - 1]$, es decir $seen = border[seen - 1]$
 - ❺ Si $T[i] = P[seen]$
 - ❻ $seen = seen + 1$
 - ❼ Si $seen = \text{tamaño de } P$
 - ❽ Imprimir que hubo una aparición que termina en i
 - ❾ $seen = seen[border[size(P) - 1]]$

Ejemplo

Ejemplo

Utilizar el algoritmo de comparación para hallar las ocurrencias de **ababa** en el texto **bacbabababacbb**.

Implementación

```
1  int n = haystack.size();
2  int seen = 0;
3  for (int i = 0; i < n; ++i){
4  // Buscar el borde ms grande cuyo siguiente elemento sea igual
   al caracter que se est mirando
5      while (seen > 0 and haystack[i] != needle[seen]) {
6          seen = border[seen - 1];
7      }
8      if (haystack[i] == needle[seen]) seen++;
9      // Si en total han coincidido m = tamao de needle
   caracteres, se hall la palabra
10     if (seen == m) {
11         printf("Needle occurs from %d to %d\n", i - m + 1, i);
12         seen = border[m - 1];
13     }
14 }
```

Complejidad

Complejidad de la comparación

En este algoritmo se cumple que el while interno sólo se ejecuta máximo $n - 1$ veces en toda la ejecución del algoritmo.

Es por esto que la complejidad de hacer las comparaciones es en total $O(m)$.

Complejidad de KMP

Hallar los bordes tiene una complejidad de $O(n)$ y hacer las comparaciones tiene una complejidad de $O(m)$.

Como en el problema se especificó que $n \leq m$, la complejidad total de KMP es $O(n)$.

Contenido

4 Tarea

Tarea

Tarea

Resolver los problemas de

<http://contests.factorcomun.org/contests/58>