Semillero de Programación

Algoritmo de Knuth-Morris-Pratt

Ana Echavarría Juan Francisco Cardona

Universidad EAFIT

26 de abril de 2013

Contenido

Contenido

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

```
int main(){
    int roads, numbers;
    int cases = 1;
    while(cin >> roads >> numbers){
        if (roads == 0 and numbers == 0) break;
        int remaining = roads - numbers;
        int ans = (remaining + numbers - 1) /  numbers;
        if (ans <= 26){
            printf("Case %d: %d\n", cases, ans);
        }else{
            printf("Case %d: impossible\n", cases);
        }
        cases++;
    }
    return 0;
}
```

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

```
const int MAXN = 50005;
int p[MAXN];
bool seen[MAXN];

int find(int u){
   if (p[u] == u) return u;
   return p[u] = find(p[u]);
}
void join(int u, int v){
   p[find(u)] = find(v);
}

int main(){
   int n, m;
   int run = 1;
   while (cin >> n >> m){
      if (n == 0 and m == 0) break;
      for (int i = 0; i <= n; ++i) p[i] = i;
```

Implementación II

```
        for (int i = 0; i < m; ++i){
            int u, v; cin >> u >> v;
            join(u, v);
        }

        memset(seen, false, sizeof(seen));
        int count = 0;
        for (int u = 1; u <= n; ++u){
            int parent = find(u);
            if (!seen[parent]){
                count++;
                seen[parent] = true;
            }
        }
        printf("Case %d: %d\n", run++, count);
    }
    return 0;
}
```

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

```
const int MAXN = 200005;
typedef pair <int, int> edge;
bool visited[MAXN];
vector <pair <int, int> > g[MAXN];

int prim(int n){
   for (int i = 0; i <= n; ++i) visited[i] = false;
   int total = 0;

   priority_queue<edge, vector <edge>, greater<edge> > q;
   q.push(edge(0, 0));
   while (!q.empty()){
      int u = q.top().second;
      int w = q.top().first;
      q.pop();
      if (visited[u]) continue;


```

Implementación II

```
        visited[u] = true;
        total += w;
        for (int i = 0; i < g[u].size(); ++i){
            int v = g[u][i].first;
            int next_w = g[u][i].second;
            if (!visited[v]){
                q.push(edge(next_w, v));
            }
        }
    }
    return total;
}
int main(){
    int n, m;
    while (cin >> n >> m){
        if (n == 0 and m == 0) break;

        for (int i = 0; i <= n; ++i) g[i].clear();

```

Implementación III

```
        int total_sum = 0;
        for (int i = 0; i < m; ++i){
           int x, y, c;
           cin >> x >> y >> c;
           total_sum += c;
           g[x].push_back(make_pair(y, c));
           g[y].push_back(make_pair(x, c));
        }

        cout << total_sum - prim(n) << endl;
     }
      return 0;
}
```

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

1. Para i desde 0 hasta $n-1$
2. encontrado = true
    3. Para j desde 0 hasta $m-1$
        4. Si $T[i+j] \neq P[j]$
            5. encontrado = false
            6. break
7. Si encontrado
    8. imprimir i

Algoritmo para String Matching

1. Para i desde 0 hasta $n-1$
2. encontrado = true
    3. Para j desde 0 hasta $m-1$
        4. Si $T[i+j] \neq P[j]$
            5. encontrado = false
            6. break
7. Si encontrado
    8. imprimir i

Complejidad

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

Contenido

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 caracter 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:

$$P[k] = T[i+k] \; \forall \, k \in [0 \ldots j) \qquad \text{y} \qquad P[j] \neq T[i+j]$$

Para cada $0 < k < j$, si $T[i+k \ldots i+j-1]$ 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 $P[0\ldots j-1] = T[i\ldots i+j-1]$ 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

  $$T[i+k\ldots i+j-1] = P[k\ldots j-1] \text{ es un prefijo de } P$$

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

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

Computando el borde de cada prefijo

Para cada prefijo de la cadena ababaca el borde es

$\epsilon \rightarrow \epsilon$
a $\rightarrow \epsilon$
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 $\rightarrow \epsilon$
ab $\rightarrow \epsilon$
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 $\rightarrow \epsilon$

ab $\rightarrow \epsilon$

aba $\rightarrow$ a

abab

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	
ababac	
ababaca	

Computando el borde de cada prefijo

Para cada prefijo de la cadena ababaca el borde es

$\epsilon \rightarrow \epsilon$

$a \rightarrow \epsilon$

$ab \rightarrow \epsilon$

$aba \rightarrow a$

$abab \rightarrow ab$

$ababa \rightarrow aba$

ababac

ababaca

Computando el borde de cada prefijo

Para cada prefijo de la cadena ababaca el borde es

$\epsilon \rightarrow \epsilon$

$a \rightarrow \epsilon$

$ab \rightarrow \epsilon$

$aba \rightarrow a$

$abab \rightarrow ab$

$ababa \rightarrow aba$

$ababac \rightarrow \epsilon$

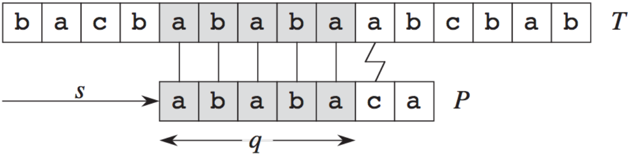
$ababaca$

Computando el borde de cada prefijo

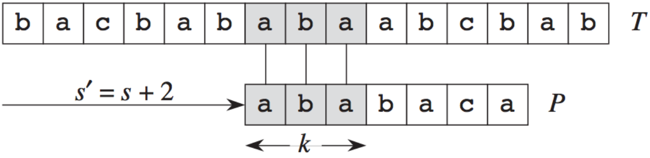
Para cada prefijo de la cadena ababaca el borde es

$$
\begin{aligned}
\epsilon &\rightarrow \epsilon \\
a &\rightarrow \epsilon \\
ab &\rightarrow \epsilon \\
aba &\rightarrow a \\
abab &\rightarrow ab \\
ababa &\rightarrow aba \\
ababac &\rightarrow \epsilon \\
ababaca &\rightarrow a
\end{aligned}
$$

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 `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
needle	a	b	a	c	a	b	a	c	a	b	a	d	a	b
border	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:
$s =$ abacaba, s termina en la posición 6 por lo que su borde es `border[6] = 3` $\rightarrow$ aba.
aba termina en la posición 2 por lo que su borde es `border[2] = border[border[6] - 1] = 1` $\rightarrow$ a.

¿Cómo calcular el borde de cada prefijo?

Supongamos que se tiene calculado `border[0]`, `border[1]`, ... `border[k-1]` y se quiere calcular `border[k]`.

Sabemos que `P[0...border[k-1]-1]` es el prefijo más largo que también es sufijo de `P[0...k-1]`. Para calcular `border[k]` hay dos posibilidades:

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

```
int m = needle.size();
vector<int> border(m);
border[0] = 0;

for (int i = 1; i < m; ++i) {
    border[i] = border[i - 1];
    // Mientras que el borde sea mayor que 0 y los caracteres no coincidan
    while (border[i] > 0 and needle[i] != needle[border[i]]) {
        border[i] = border[border[i] - 1];
    }
    // Si hubo coincidencia sumarle ese caracter a la longitud
    if (needle[i] == needle[border[i]]) border[i]++;
}
```

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.

1. Hacer seen 0 (el número de caracteres de P han coincidido)
2. Para i desde 0 hasta el tamaño de T
    3. Mientras que $seen > 0$ y $T[i] \neq P[seen]$
        4. Comparar desde el borde de $P[0 \dots seen - 1]$, es decir $seen = border[seen - 1]$
    5. Si $T[i] = P[seen]$
        6. $seen = seen + 1$
    7. Si seen $=$ tamaño de P
        8. Imprimir que hubo una aparición que termina en i
        9. $seen = seen[border[size(\mathrm{P}) - 1]]$

Ejemplo

Ejemplo

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

Implementación

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

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

Tarea

Tarea

Resolver los problemas de

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