Tema: Precondicionamiento

asignatura: Métodos Algorítmicos de Resolución de Problemas, **profesor:** Ricardo Peña

Curso 2012/13

1. Introducción

⋆ La técnica de **precondicionamiento** consiste en emplear recursos de cómputo **antes** de resolver un problema, si con la información así obtenida se puede ahorrar tiempo en dicha resolución.

⋆ La suma del tiempo invertido en el pre-procesado inicial y el tiempo empleado en resolver el problema de modo más eficiente, ha de compensar con la alternativa de resolverlo sin precondicionamiento.

⋆ Con frecuencia, el problema se va resolver **repetidas** veces. Cuantas más veces se resuelva, más compensa emplear un tiempo inicial en pre-procesar la información.

⋆ Un ejemplo característico es el **ajuste de cadenas** donde, gracias al precondicionamiento, se han desarrollado algoritmos muy eficientes. El problema consiste en encontrar todas las apariciones de una cadena dentro de una cadena mayor. Este problema se presenta en los editores de texto, en las búsquedas dentro de una página HTML, en algoritmos de bio-informática que procesan cadenas de ADN, y en muchos otros contextos.

2. Algoritmo ingenuo de ajuste de cadenas

⋆ Suponemos un texto $T[1..n]$ de n caracteres y un patrón a buscar $P[1..m]$ de $m \leq n$ caracteres. Diremos que el patrón P aparece en T con **desplazamiento** s, $0 \leq s \leq n-m$, si $T[s+1..s+m] = P[1..m]$.

⋆ Suponemos un **alfabeto** Σ de caracteres y denotaremos por $|\Sigma|$ su tamaño. Similarmente, si $x \in \Sigma^*$ denotaremos por $|x|$ el número de caracteres de la cadena x.

⋆ Diremos que x es **prefijo** de y, denotado $x \sqsubseteq y$, si existe una cadena w tal que $y = xw$. Diremos que x es **sufijo** de y, denotado $x \sqsupseteq y$, si existe una cadena w tal que $y = wx$. La cadena vacía ϵ es prefijo y sufijo de cualquier cadena.

⋆ Abreviaremos $P[1..k]$ por P_k, por tanto, $P_m = P$ y $P_0 = \epsilon$. Seguiremos igual convenio para T. El problema del ajuste puede enunciarse entonces como encontrar todos los desplazamientos s que cumplen $P_m \sqsupseteq T_{s+m}$.

⋆ El siguiente algoritmo en pseudocódigo calcula este resultado, desplazando el patrón sobre el texto de uno en uno caracteres:

```
1 ajuste-ingenuo-de-cadenas (T[1..n], P[1..m])
2 for s = 0 to n-m
3   do if P[1..m]=T[s+1..s+m]
4         then print ("ajuste en desplazamiento", s)
```

⋆ El coste en el caso peor de este algoritmo está en $\Theta(nm)$. Considérese por ejemplo el caso en que $P =$ "$aa\ldots a$" y el texto T también consiste en todo aes.

⋆ Veremos que se puede conseguir un coste $\Theta(n)$. La clave está en aprovechar nuestro conocimiento previo sobre las características del patrón y sobre la parte de texto ya ajustado en cada momento.

3. El algoritmo de Rabin-Karp

⋆ Ideado por R. M. Karp y M. O. Rabin en 1981. Utiliza levemente la técnica de precondicionamiento. Se basa en simplificar la parte del algoritmo de ajuste que compara las dos cadenas de caracteres, la del texto y la del patrón, tratando de hacerla con un coste en $\Theta(1)$, siempre que sea posible.

⋆ Para ello considera que los caracteres, o en general los símbolos de Σ, son dígitos de un sistema de numeración en base $d = |\Sigma|$. A efectos de este apartado y con el fin de facilitar la comprensión, supondremos $\Sigma = \{0, 1, \ldots, 9\}$, de forma que el patrón $P[1..m]$ puede considerarse como si fuera un número en base 10 de m cifras. Por ejemplo, si $P =$"119", se interpreta como el número decimal 119.

⋆ En la fase de preprocesamiento, se calcula el entero p resultante de convertir el patrón P a su correspondiente número decimal, es decir $p = \sum_{i=1}^{m} P[i]10^{m-i}$.

⋆ Durante la fase de ajuste, para cada desplazamiento s, se calcula la conversión a decimal t_s del fragmento de texto $T[s+1..s+m]$. Es obvio que se produce ajuste en el desplazamiento s si y solo si $p = t_s$.

⋆ Aparentemente no hemos conseguido gran cosa, dado que la conversión del texto $T[s+1..s+m]$ a decimal necesita $\Theta(m)$ multiplicaciones y hay que hacerlo $\Theta(n)$ veces, y no se puede pensar que la comparación de dos números de m cifras, para un m arbitrario, puede hacerse en general en un tiempo $\Theta(1)$.

⋆ El algoritmo de Rabin-Karp da solución a ambos problemas. En cuanto al primero, cada conversión t_{s+1} puede calcularse con coste $\Theta(1)$ en función de la anterior:

$$t_{s+1} = 10 \times (t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

De esta forma, las $n - m + 1$ conversiones necesarias pueden calcularse con un coste en $\Theta(n)$.

⋆ En cuanto al segundo, se escoge un número primo q tal que $10q$ quepa en una palabra del computador, y tanto p como los valores t_s se calculan **módulo** q. Las comparaciones serían entonces de coste $\Theta(1)$. Las comparaciones sirven para descartar ajustes inválidos, pero no para determinar ajustes válidos, ya que:

$$p \not\equiv t_s \pmod{q} \Rightarrow p \neq t_s \quad \text{pero} \quad p \equiv t_s \pmod{q} \not\Rightarrow p = t_s$$

⋆ Si $p \equiv t_s \pmod{q}$ el algoritmo compara entonces el patrón sin convertir. En el caso peor, el coste sigue estando en $\Theta(nm)$, pero dependiendo de las características del patrón y del texto, muchos ajustes inválidos se pueden descartar con coste $\Theta(1)$. Para un $d = |\Sigma|$ arbitrario, tenemos:

```
1  ajuste-Rabin-Karp (T[1..n], P[1..m], d, q)
2    // fase de preprocesado O(m)
3  p   = 0
4  t_s = 0
5  h   = d^(m-1) mod q
6  for i = 1 to m
7    do p   = (d * p   + P[i]) mod q
8       t_s = (d * t_s + T[i]) mod q
9    // fase de ajuste O(nm)
10 for s = 0 to n-m
11   do if p = t_s
12         then if P[1..m]=T[s+1..s+m]
13                 then print ("ajuste en desplazamiento", s)
14      if s < n - m
15         then t_s = (d*(t_s - T[s+1]*h) + T[s+m+1]) mod q
```

4. Precondicionamiento con autómata

⋆ Este algoritmo aparece por primera vez en un libro de A. V. Aho, J. E. Hopcroft y J. D. Ulmann en 1974, aunque se desconoce su autor. La idea consiste en construir en la fase de preprocesado, y a partir del patrón, un **autómata finito determinista** (AFD) y usar durante la fase de ajuste dicho autómata, tanto para descartar ajustes inválidos como para decidir ajustes válidos.

⋆ Dado un patrón $P[1..m]$, definimos la **función sufijo** σ asociada a P del modo siguiente:

$$\sigma : \Sigma^* \to \{0 \dots m\}, \qquad \sigma(x) = \text{máx}\{k \mid P_k \sqsupseteq x\}$$

es decir $\sigma(x)$ devuelve la longitud del mayor prefijo de P que es sufijo de x.

⋆ El AFD asociado a P, $M = (\Sigma, Q, q_0, \delta, F)$ se define del modo siguiente:

$$\begin{array}{lcl} Q & = & \{0, \dots, m\} \\ q_0 & = & 0 \\ F & = & \{m\} \\ \delta(q, a) & = & \sigma(P_q a) \end{array}$$

⋆ La idea es hacer evolucionar el autómata M con cada carácter a encontrado en el texto. Si M está en el estado q, ello significará que el prefijo P_q es sufijo del texto en curso T_i. Si $q = m$, entonces todo el patrón es sufijo del texto y se decide que hay ajuste. Si $q < m$ y el siguiente carácter $a = T[i+1]$ ajusta con $P[q+1]$, entonces $\delta(q, a) = q + 1$ porque P_{q+1} pasa a ser el mayor prefijo de P que es sufijo del texto en curso T_{i+1}. Si a no ajusta con $P[q+1]$, entonces aprovechamos nuestro conocimiento del texto ajustado hasta P_q y el conocimiento de que puede existir un prefijo $P_k,\ k < q$, que sea sufijo de $P_q a$, para pasar al estado k. La razón es que P_k sería el mayor prefijo de P que es sufijo de T_{i+1}. De ahí la definición $\delta(q, a) = \sigma(P_q a)$.

⋆ La fase ajuste del algoritmo, de coste $\Theta(n)$, queda así:

```
1 ajuste-con-automata (T[1..n], m, delta)
2   // fase de ajuste O(n)
3 q = 0
4 for i = 1 to n
5   do q = delta(q, T[i])
6      if q = m
7        then print ("ajuste en desplazamiento", i-m)
```

⋆ Obsérvese que todo el conocimiento sobre el patrón está encapsulado en la función δ de transición del autómata. Si llamamos $\hat{\delta}$ a la función δ extendida a cadenas de Σ^*, el invariante del bucle es simplemente $\hat{\delta}(q_0, T_{i-1}) = \sigma(T_{i-1})$.

⋆ La fase de preproceso construye el AFD M a partir del patrón P. El siguiente algoritmo lo hace:

```
1 preproceso-con-automata (P[1..m], Sigma, out delta)
2 for q = 0 to m
3   do for cada a en Sigma
4        do k = min (q+1,m)
5           while P[1..k] no es sufijo de P[1..q]a
6             do k = k - 1
7           delta (q,a) = k
```

⋆ El algoritmo no puede tener un coste menor de $\Theta(m|\Sigma|)$ puesto que para cada estado q ha de analizar todos los símbolos de Σ. El que se presenta aquí tiene un coste en el caso peor en $\Theta(m^3|\Sigma|)$, pero se puede optimizar hasta conseguir un coste $\Theta(m|\Sigma|)$.

5. El algoritmo de Knuth-Morris-Pratt

⋆ Ideado por D. E. Knuth, J. H. Morris y V. R. Pratt en 1977, comparte un coste en $\Theta(n)$ con el algoritmo basado en autómata, pero el tiempo de preprocesado se reduce a $\Theta(m)$.

⋆ La idea clave del preprocesado es calcular una función $\pi : \{1 \dots m\} \to \{0 \dots m-1\}$ a partir del patrón $P[1..m]$, que encapsule todo el conocimiento sobre el mismo que se necesitará en la fase de ajuste, y que juega el mismo papel que δ en el algoritmo basado en autómata. Su definición es:

$$\pi(q) = \text{máx}\{k \mid k < q, P_k \sqsupseteq P_q\}$$

Es decir, calcula la longitud del prefijo más largo de P, excluyendo a P_q, que es sufijo de P_q.

⋆ Supongamos que en un cierto desplazamiento s de la fase de ajuste, el texto T_i ajusta con P_q pero el carácter siguiente no ajusta, es decir $P[q+1] \neq T[i+1]$. Entonces $\pi(q) = k$ nos informa de que $P[1 \dots k] = P[q-k+1 \dots q]$ y por tanto igual también a $T[s+q-k+1 \dots s+q]$. Si desplazamos el patrón hacia la derecha hasta un desplazamiento $s' > s$ tal que $s' + k = s + q$ entonces $P_k \sqsupseteq T_i$. Podemos proseguir entonces el ajuste con $P[k+1]$ y $T[i+1]$. Mas aún, debido a que k es máximo, $s' - s = q - k$ es el menor desplazamiento posible a la derecha que tiene posibilidades de éxito.

⋆ Suponiendo precalculada la función π, el algoritmo de la fase de ajuste es:

```
1  ajuste-Knuth-Morris-Pratt (T[1..n], P[1..m], pi[1..m])
2  q = 0
3  for i = 1 to n
4    do while q>0 && P[q+1] != T[i]
5         do q = pi[q]
6       if P[q+1] = T[i]
7         then q = q + 1
8       if q = m
9         then print ("ajuste en desplazamiento", i-m)
10             q = pi[q]
```

⋆ El algoritmo de la fase de preprocesado es:

```
1  preprocesado-Knuth-Morris-Pratt (P[1..m], out pi[1..m])
2  pi[1] = 0
3  k     = 0
4  for q = 2 to m
5    do while k>0 && P[k+1] != P[q]
6         do k = pi[k]
7       if P[k+1] = P[q]
8         then k = k + 1
9       pi[q] = k
```

5.1. Corrección del algoritmo KMP

⋆ Tal como se ha definido π, es obvio que para todo $q \in \{1 \dots m\}$ se cumple $0 \leq \pi(q) < q$. En particular, $\pi(1) = 0$. Definimos la iteración cero o más veces de π del modo habitual: $\pi^0(q) = q$ y para $i \geq 0$, $\pi^{i+1}(q) = \pi(\pi^i(q))$. El siguiente conjunto recolecta todas las iteraciones no nulas de π:

$$\pi^+(q) = \{\pi^i(q) \mid i \geq 1,\ \pi^{i-1}(q) \neq 0\}$$

⋆ Veamos ahora que $\pi^+(q)$ recorre **todos** los prefijos de P que son sufijos de P_q, excluido este último, es decir:

$$\pi^+(q) = \{k \mid k < q,\ P_k \sqsupseteq P_q\} \tag{1}$$

Por inducción sobre i, tenemos que si $k = \pi^i(q)$, entonces $P_k \sqsupseteq P_q$. Si $i = 1$, es cierto por definición de $\pi(q)$. Si $i > 1$ y $P_{\pi^i(q)} \sqsupseteq P_q$, entonces $P_{\pi^{i+1}(q)} \sqsupseteq P_{\pi^i(q)}$ y la propiedad se obtiene por transitividad de $<$ y de $\sqsupseteq$. Por tanto, tenemos $\pi^+(q) \subseteq \{k \mid k < q,\ P_k \sqsupseteq P_q\}$. La inclusión contraria se obtiene por contradicción: supongamos que existe uno o más valores k tales que $k < q$, $P_k \sqsupseteq P_q$, y $k \notin \pi^+(q)$. Sea j el mayor de todos ellos. Sea $j' = \text{mín}\{k \mid k \in \pi^+(q),\ k > j\}$. Este mínimo está definido pues al menos $j < \pi(q) \in \pi^+(q)$, por definición de $\pi(q)$. Entonces j satisface exactamente la definición de $\pi(j')$. Por tanto, $j = \pi(j') \in \pi^+(q)$ en contradicción con la hipótesis. Luego $\pi^+(q) \supseteq \{k \mid k < q,\ P_k \sqsupseteq P_q\}$.

⋆ Otra propiedad que vamos a necesitar es la siguiente:

$$\text{Si } \pi \text{ es la función prefijo de } P[1 \ldots m] \text{ y } \pi(q) > 0, \text{ entonces } \pi(q) - 1 \in \pi^+(q-1) \qquad (2)$$

En efecto, si $r = \pi(q) > 0$, entonces $r < q$ y $P_r \sqsupseteq P_q$. Además, $r - 1 < q - 1$ y, eliminando el último carácter de P_q, tenemos $P_{r-1} \sqsupseteq P_{q-1}$. Por tanto, $r - 1 \in \pi^+(q-1)$.

⋆ Para $q \in \{2, 3, \ldots m\}$, definimos el siguiente conjunto:

$$E_{q-1} = \{k \mid k \in \pi^+(q-1),\ P[k+1] = P[q]\} \overset{(1)}{=} \{k \mid k < q-1,\ P_k \sqsupseteq P_{q-1},\ P[k+1] = P[q]\}$$

Es decir, es el conjunto de las longitudes de los prefijos de P que son sufijos de P_{q-1} y que pueden extenderse un carácter más allá para convertirse en sufijos de P_q.

⋆ La última propiedad que necesitamos es:

$$\pi(q) = \begin{cases} 0 & \text{si } E_{q-1} = \emptyset \\ 1 + \text{máx } E_{q-1} & \text{si } E_{q-1} \neq \emptyset \end{cases} \qquad (3)$$

En efecto si $E_{q-1} = \emptyset$, no hay modo de extender un sufijo P_k de P_{q-1} a un sufijo de P_q, por tanto es correcto $\pi(q) = 0$. Si $E_{q-1} \neq \emptyset$, todos los $k \in E_{q-1}$ satisfacen $k < q - 1$ y $P_{k+1} \sqsupseteq P_q$. Entonces, por definición de $\pi(q)$ tenemos $\pi(q) \geq 1 + \text{máx } E_{q-1}$. Por otro lado, podemos aplicar la propiedad (2) ya que $\pi(q) > 0$. Si $r = \pi(q) - 1$ entonces $r \in \pi^+(q-1)$. Por definición de $\pi(q)$ sabemos que $P[r+1] = P[q]$, por tanto $r \in E_{q-1}$, o sea $r \leq \text{máx } E_{q-1}$, o equivalentemente, $\pi(q) = r + 1 \leq 1 + \text{máx } E_{q-1}$. De ambas inecuaciones se deduce la propiedad.

⋆ Esta propiedad permite calcular $\pi(q)$ a partir de $\pi^+(q-1)$. Esa es precisamente la intención del algoritmo `preprocesado-Knuth-Morris-Pratt`. El invariante del bucle `for` es $k = \pi(q-1)$. Lo establecen inicialmente la líneas 2 y 3, junto con la asignación `q=2`, y lo restablecen en cada iteración la línea 9 y el incremento de q al final de la misma. Veamos que el valor k asignado a $\pi(q)$ en la línea 9 es correcto. En efecto, el bucle `while` de las líneas 5-6 recorre los valores k de $\pi^+(q-1)$ en sentido decreciente hasta encontrar uno que satisfaga $P[k+1] = P[q]$. Si lo encuentra, dicho k será el máximo de E_{q-1} y por la propiedad (3) ha de ser $\pi(q) = k+1$. Si no lo encuentra, entonces $E_{q-1} = \emptyset$ y entonces por (3) ha de ser $\pi(q) = 0$, tal como hace el algoritmo.

⋆ La fase de preprocesado puede considerarse legítimamente como un algoritmo de *programación dinámica*, ya que la función $\pi(q)$ se calcula recurriendo a la propia función $\pi(q')$ para valores $q' < q$. Estos valores se tabulan en el vector `pi[1..m]`, y este se rellena desde el caso base $\pi(1) = 0$ hacia los casos recursivos.

⋆ La corrección del algoritmo `ajuste-Knuth-Morris-Pratt` se deriva de la corrección del algoritmo `ajuste-con-autómata`. El invariante del bucle `for` del ajuste con autómata es $q = \hat{\delta}(q_0, T_{i-1})$, y por la definición de δ ello implica $P_q \sqsupseteq T_{i-1}$. El cuerpo del bucle obviamente mantiene dicho invariante mediante las asignaciones $q = \delta(q, T[i])$ y `i=i+1`.

⋆ El invariante del bucle `for` de `ajuste-Knuth-Morris-Pratt` es ligeramente distinto:

$$(\hat{\delta}(q_0, T_{i-1}) \neq m \wedge q = \hat{\delta}(q_0, T_{i-1})) \vee (\hat{\delta}(q_0, T_{i-1}) = m \wedge q = \pi(m))$$

La razón de esta complicación es la asignación de la línea 10, que cambia el valor de q cuando $q = m$. Con ello se evita acceder al elemento inválido $P[m+1]$ en la línea 4. Por la forma en que está definida δ, es fácil demostrar la igualdad $\delta(m, a) = \delta(\pi(m), a)$ para todo carácter $a \in \Sigma$. Por tanto, si llamamos q' al valor de q antes de ejecutar la línea 4, basta con demostrar que las asignaciones hechas a q en las líneas 4-7 de `ajuste-Knuth-Morris-Pratt`, son equivalentes a la asignación $q = \delta(q', T[i])$.

- ⋆ El bucle `while` de las líneas 4-5 puede acabar con una de dos condiciones:

  $P[q+1] = T[i]$ En este caso, q va recorriendo $\pi^+(q') = \{k \mid k < q',\ P_k \sqsupseteq P_{q'}\}$ en sentido decreciente. Sale con el valor $q = \text{máx } E_{q'}$, y por la propiedad (3), la definición de π, y la de σ, tenemos $\delta(q', T[i]) = q + 1$, lo cual coincide con el valor asignado a q en la línea 7.

  $P[q+1] \neq T[i]$ En este caso, $q = 0$ y $\pi^+(q') = \emptyset$. Ello significa $\sigma(T_i) = 0$, y por tanto $\delta(q', T[i]) = 0$. El algoritmo `ajuste-Knuth-Morris-Pratt` no pasa en ese caso por la línea 7 y deja inalterado el valor $q = 0$.

- ⋆ El algoritmo KMP pasa entonces por los mismos estados que el algoritmo basado en autómata, y su corrección de deriva de la corrección de aquel. La diferencia principal es que el uso de la función precalculada π nos evita tener que calcular el autómata de modo explícito.

- ⋆ Un invariante más sencillo del bucle `for`, que no hace referencia al autómata, es simplemente $P_q \sqsupseteq T_{i-1}$ y P_q es el mayor prefijo de P, distinto de P_m, que satisface dicha propiedad.

5.2. Coste del algoritmo KMP

- ⋆ El coste **amortizado** de una iteración del bucle `for` del algoritmo `preprocesado-Knuth-Morris-Pratt` está en $\Theta(1)$. Para demostrarlo, usamos la función de potencial $\Phi = k$. Nótese que $k = 0$ inicialmente y que $k \geq 0$ todo el tiempo, por lo que el bucle acaba con un potencial no menor que el inicial.

- ⋆ Supongamos una iteración en la que la línea 6 no se ejecuta (el bucle `while` no itera) y sí lo hace la línea 8. El coste amortizado en este caso es: $\hat{c} = c + \Delta(\Phi) \in \Theta(1 + 1) = \Theta(1)$.

- ⋆ Si la línea 6 se ejecuta una o más veces, en cada iteración k decrece al menos en uno. El coste real se "paga" en este caso con el decrecimiento de Φ. Por tanto, el coste amortizado del bucle `while` es nulo, y el de la iteración del bucle `for` sigue estando en $\Theta(1)$.

- ⋆ El coste total de la fase de preprocesado está pues en $\Theta(m)$.

- ⋆ El razonamiento para el algoritmo `ajuste-Knuth-Morris-Pratt` es similar, salvo que en este caso definimos $\Phi = q$. El coste de está fase está pues en $\Theta(n)$.

Lecturas complementarias

Estas notas están basadas en el Cap. 32 de [1], donde hay abundantes ilustraciones y más explicaciones con las que el lector puede completar lo resumido aquí.

Referencias

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.