2D1458, Problemlösning och programmering under press

Föreläsning 3: Strängmatchning

Datum: 2006-09-18
Skribenter: Mikael Eliasson, Joakim Eriksson och Mats Linander
Föreläsare: Mikael Goldmann

Denna föreläsning behandlar problemet strängmatchning. Den naiva ansatsen, ändliga automater och Knuth–Morris–Pratts algoritm gås genom och kortare översikter av några metoder för matchning av flera mönster i samma text ges.

1 Strängmatchning – naiv algoritm

Problemet är att givet en text $T[1...n]$ och ett mönster $P[1...m]$ hitta alla förekomster av P som delsträng till T. Den naiva algoritmen undersöker alla positioner i texten mellan 1 och $n-m$ för att se om mönstret börjar där. När en position är undersökt så skiftas mönstret ett steg åt höger i T. Tidskomplexiteten för algoritmen som kan ses nedan är $\Theta(mn)$.

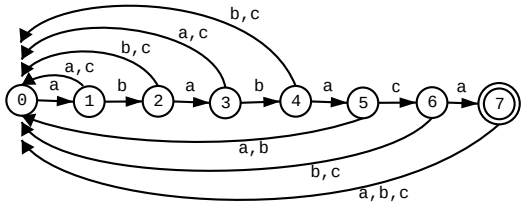
Algoritm 1: Strängmathcning - naiv algoritm
Input: Text $T[1...n]$, mönster $P[1...m]$.
Output: Positioner i T där P börjar.
STRING MATCHING($T[1...n]$, $P[1...m]$)

```
(1)   for j ← 1 to n − m
(2)       for i ← 1 to m
(3)           if P[i] ≠ T[i + j − 1] then break
(4)           if i ≥ m then OUTPUT j
```

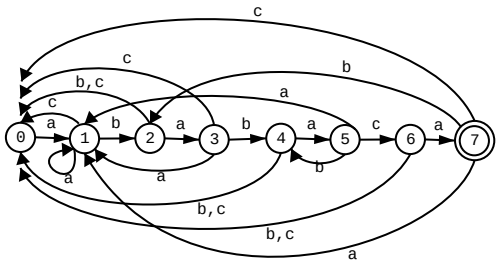
2 Strängmatchning – ändlig automat

En ändlig automat för strängmatchning är en riktad graf, vars noder och kanter kallas *tillstånd* respektive *övergångar*. Varje övergång i automaten är märkt med ett eller flera tecken ur alfabetet Σ. Automaten har ett *starttillstånd* och minst ett *accepterande tillstånd*.

Strängmatchning utförs genom att vandra i automaten, med början i starttillståndet, tills det accepterande tillståndet nås. I varje tillstånd väljs nästa tillstånd genom att ta ett nytt tecken x ur T och sedan låta den övergång som är märkt x ange nästa tillstånd. Betrakta följande automat som söker efter mönstret $P = ababaca$:



Man befinner sig i tillstånd q om de senaste q lästa tecknen i T matchar de första q tecknen i P. Som synes kan endast en förekomst av *ababaca* leda till ett accepterande tillstånd. Problem uppstår dock t ex om $T = ababababaca$, ty efter de första fem tecknen hamnar vi i starttillståndet och missar därför förekomsten av P med början i tecken 2 i T. Lösningen på detta är att lägga till övergångar enligt följande figur:



Automaten hamnar nu i det accepterande tillståndet precis om P förekommer i T. Eftersom varje tecken i T undersöks en enda gång och övergångarna kan göras i konstant tid så blir tidsåtgången $\Theta(n)$.

För att göra övergångar i konstant tid kan vi använda en övergångsmatris $\delta[q, x]$ = nästa tillstånd, där q är nuvarande tillstånd och x är indatatecken. Problem uppstår vid stora alfabeten, t ex unicode, då matrisen kan bli orimligt stor och tidskrävande att konstruera.

3 Strängmatchning – Knuth–Morris–Pratt

Problemet med stora övergångsmatriser kan avhjälpas med Knuth–Morris–Pratt algoritm (KMP). Här använder vi oss av en prefixfunktion $\pi[1..m]$ sådan att $\pi[q]$ är längden av det längsta prefix av $P[1..q-1]$ som också är ett suffix av $P[1..m]$.

Algoritm 2: Strängmatchning – Knuth–Morris–Pratt
Input: Text $T[1...n]$, mönster $P[1...m]$.
Output: Positioner i T där P börjar.
KMP($T[1...n], P[1...m]$)
(1) $\pi \leftarrow$ PREKMP(P)
(2) $q \leftarrow 0$
(3) **for** $i \leftarrow 1$ **to** n
(4) **while** $q > 0$ and $P[q+1] \neq T[i]$
(5) $q \leftarrow \pi[q]$
(6) **if** $P[q+1] = T[i]$ **then** $q \leftarrow q+1$
(7) **if** $q = m$
(8) OUTPUT $i - m + 1$
(9) $q \leftarrow \pi[q]$

Algoritm 3: Prefixfunktionen π tillhörande KMP
Input: $P[1...m]$
Output: $\pi[1...m]$ så att
$\pi[q] = max\{k|k < q, P[1...k]$ suffix av $P[1...q]\}$
PREKMP($P[1...m]$)
(1) $\pi[1] \leftarrow 0$
(2) **for** $q \leftarrow 2$ **to** m
(3) $k \leftarrow \pi[q-1]$
(4) **while** $k > 0$ and $P[k+1] \neq P[q]$
(5) $k \leftarrow \pi[k]$
(6) **if** $P[k+1] = P[q]$ **then** $\pi[q] \leftarrow k+1$
(7) **else** $\pi[q] \leftarrow 0$

Då π kan förberäknas i tid $\Theta(m)$ blir tidsåtgången för KMP $\Theta(m+n)$. Eftersom $m < n$ rimligen gäller, så är algoritmen $\Theta(n)$ i tid.

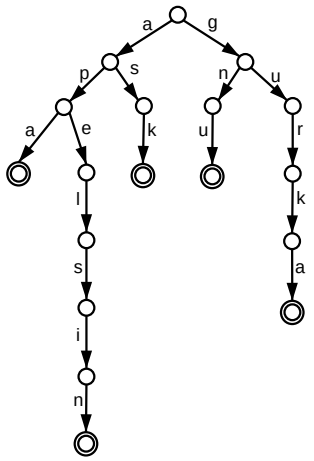
4 Översikter av algoritmer för matchning av flera mönster i samma text

Under föreläsningen gavs även ett par mer övergripande genomgångar av några algoritmer för samtidig matchning av flera mönster i samma text. Upprepad användning av algoritmer som KMP fungerar givetvis men bättre metoder finns att tillgå.

4.1 Trie

En trie är ett sökträd som representerar en mängd S av ord över alfabetet Σ. Trädets kanter är märkta med tecken ur Σ och dess noder är märkta med 1 eller 0. En nod märkt 1 svarar mot att märkningen av kanterna, från trädets rot ner till

den märkta noden, bildar ett ord i S. En nod märkt 0 betyder att motsvarande ord ej finns i S. Betrakta följande trie för $S = \{apa, apelsin, ask, gnu, gurka\}$:



Valet av datastruktur för representation av trädet beror på om låg minnesåtgång eller snabba sökningar skall prioriteras. Om hastighet är viktigast så kan det vara lämpligt att i varje nod hålla en array av storlek $|\Sigma|$ med pekare till barnnoderna. I varje nod kan då uppslagning av efterkommande nod göras snabbt, oavsett hur stort alfabetet är, så uppslag av ett m tecken långt ord går i tid $\Theta(m)$. Insättning av ett ord av längd m går i detta fall i tid $\Theta(m)$.

Om minnessnålhet prioriteras kan det vara mer lämpligt med dynamiska arrayer i noderna. Detta låter oss i varje nod lagra precis så många barnreferenser som behövs, istället för en barnreferens för varje tecken i alfabetet. Dessa arrayer hålls med fördel sorterade för snabbare uppslag. Skillnaden i tidsåtgång (och minnesåtgång) skiljer sig endast från det föregående fallet med en konsant faktor, så tidskomplexiteten är oförändrad.

4.2 Suffixträd

Om vi skall söka efter flera strängar P_i i T, så kan det löna sig att först konstruera ett suffixträd över T och sedan för varje P_i söka i suffixträdet. Själva suffixträdet är helt enkelt en trie innehållandes alla suffix till T. Om P_i är en delsträng till T så måste P_i vara prefix till något av T's suffix och kan därför i tid $\Theta(m)$ slås upp i suffixträdet. Själva konstruktionen görs naivt i tid $\Theta(n^2)$ men kan göras i tid $\Theta(n)$ (se litteratur för detaljer).

4.3 Suffixarray

Ett alternativ till att konstruera suffixträd är att använda en suffixarray, vilket är en array vars element är alla suffix till T (eller referenser till dessa) sorterade i lexikografisk ordning. På samma sätt som för suffixträdet kan förekomster av ett mönster P i T finnas genom att söka efter de element i arrayen vars prefix är P. Sökningen kan göras genom binärsökning i tid $O(m \log n)$ där m och n är mönstret respektive textens storlek. Suffixarrayen kan konstrueras i tid $\Theta(n)$, t ex genom att först bygga ett suffixträd och sedan söka genom detta i lexikografisk ordning. I

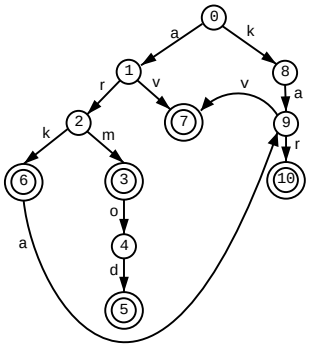
praktiken är det ofta, bland annat på grund av höga konstantfaktorer, bättre att använda andra metoder.

Den naiva ansatsen vore att sortera suffixen med en vanlig jämförelsebaserad algoritm. Eftersom suffixens längd är $O(n)$ blir då tidskomplexiteten $O(n^2 \log n)$. Genom att utnyttja att varje suffix till ett suffix förekommer som prefix till något annat suffix, kan vi relativt enkelt förbättra den naiva ansatsen.

Låt k gå från 0 till $\log n$ och sortera i varje steg suffixen med avseende på de första 2^k tecknen. I den första iterationen inspekterar vi varje suffix första tecken och sorterar efter dessa. I efterföljande iterationer utnyttjar vi att den första hälften av de 2^k tecknen som skall ses till redan är i ordning och att de återstående 2^{k-1} tecknen är prefix till något annat suffix och som sådant har ordnats i föregående iteration. Sorteringssteget kan reduceras till ett konstant antal jämförelser för varje suffixpar och kan alltså göras i tid $O(n \log n)$. Sorteringen utförs i alla de $\log n$ iterationerna, så tidskomplexiteten blir $O(n(\log n)^2)$.

4.4 Aho–Corasick

Aho–Corasicks algoritm är en generalisering av KMP som söker efter förekomster av flera strängar P_i samtidigt. Algoritmen konstruerar en trie innehållandes P_i, lägger till ett antal återlänkar och behandlar sökträdet som en ändlig automat med starttillstånd i roten och accepterande tillstånd i de noder som är märkta 1. Följande figur illustrerar hur en sådan automat, med en majoritet av övergångarna borttagna, skulle kunna se ut för $P_i \in \{\text{av,arm,ark,armod,kar}\}$.



I praktiken använder Aho–Corasicks algoritm, liksom KMP, inte den ändliga automaten direkt, utan konstruerar istället en generalisering av KMP:s prefixfunktion.