Tehnici de căutare în șiruri

Căutarea tuturor aparițiilor unui șablon într-un text este o problemă ce apare frecvent, mai ales într-un editor de texte. În mod obișnuit, un document este tipărit, și un cuvânt sau o succesiune de caractere este căutat la un moment dat de către utilizator. Alt exemplu este următorul: un utilizator primește un șir de lungime considerabilă din baza de date, și dorește să afle contextul în care apare un anume cuvânt. Algoritmii de căutare în șiruri pot fi folosiți și la cătarea de exemplu a unor șabloane în secvențe ADN.

Vom formula problema de căutare în șiruri după cum urmează. Presupunem că textul este un șir $T[1..n]$ de lungime n și că șablonul de căutat este un șir $P[1..m]$ de lungime m. Vom presupune mai departe că, elementele lui P și T sunt caractere ce aparțin alfabetului finit Σ.

De exemplu, putem avea $\Sigma = \{0,1\}$ sau $\Sigma = \{a, b, \dots, z\}$. Astfel șirurile sunt formate de obicei din caractere, astfel că le vom spune string-uri.

Vom spune că șablonul P apare *cu deplasamentul s* în textul T(sau, echivalent, că șablonul P apare la *începutul poziției* $s + 1$ în textul T) dacă $0 \leq s \leq n - m$ și $T[s + 1..s + m] = P[1..m]$ (altfel spus $T[s + j] = P[j]$ pentru $1 \leq j \leq m$). Dacă P apare cu deplasamentul s în T vom spune că s este un deplasament *valid*, altfel vom spune că s este un deplasament *nevalid*. Problema de căutare în șiruri este de a găsi toate deplasamentele valide pentru care P se găsește într-un text T. Figura 1 prezintă acest concept.

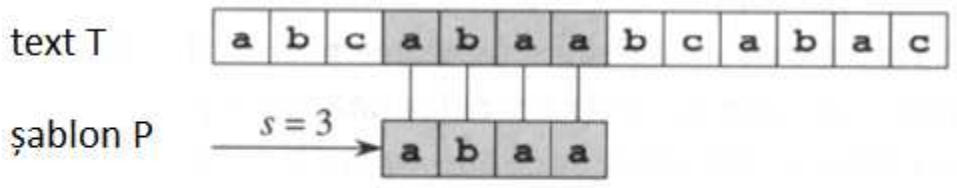


Figura 1. Problema căutării în string-uri. Scopul este de a găsi toate aparițiile șablonului $P = abaa$ în textul $T = abcabaabcabac$. șablonul apare doar la deplasamentul $s = 3$. Se zice că $s = 3$ este un deplasament valid. Cu gri sunt marcate caracterele care se potrivesc în ambele string-uri

În cele ce urmează vom studia câteva moduri de a rezolva această problemă. Vom începe cu algoritmul brute-force care are ordinul de timp $O((n-m+1)m)$. Apoi vom continua cu algoritmul Rabin și Karp ce are același ordin de timp însă funcționează mult mai bine pe cazul mediu și în practică. Apoi vom descrie algoritmul mai eficient Knuth-Morris-Pratt (sau KMP) cu un ordin de timp liniar și anume $O(n+m)$.

NOTAȚII ȘI TERMINOLOGII

Vom nota cu $\Sigma^*(sigma\ stelat)$ o mulțimea de șiruri de lungime finită, formate utilizând caractere din alfabetul Σ. Șirul de lungime zero sau șir gol, notat cu λ aparține de asemenea, lui Σ^*.

Lungimea șirului x este notată cu $|x|$. Concatenarea a două șiruri x și y, notată cu xy, are lungimea $|x|+|y|$ și constă din caracterele din x urmate de caracterele din y.

Vom spune că un șir w este un *prefix* al lui x, și se notează $w \sqsubset x$, dacă $x = wy$ pentru un șir $y \in \Sigma^*$. Dacă $w \sqsubset x$ atunci $|w| \leq |x|$. Asemănător, vom spune că string-ul w este un sufix al unui șir x, notat cu $w \sqsupset x$ dacă există $y \in \Sigma^*$ astfel ca $x = yw$. Șirul gol λ este atât sufix cât și un prefix pentru orice șir.

De exemplu, avem $ab \sqsubset abcca$ și $cca \sqsupset abcca$. Este bine să știm că pentru orice string-uri x și y și orice caracter a, avem $x \sqsupset y$ dacă și numai dacă $xa \sqsupset ya$. De asemenea relațiile $\sqsupset$ și $\sqsubset$ sunt tranzitive.

LEMA 1 – SUFFIX SUPRASCRIS

Fie x, y, z șiruri astfel încât $x \sqsupset z$ și $y \sqsupset z$. Dacă $|x| \leq |y|$ atunci $x \sqsupset y$. Dacă $|x| \geq |y|$ atunci y$\sqsupset x$. Dacă $|x| = |y|$ atunci $x = y$. Demonstrația are loc conform figurii 2, de mai jos.

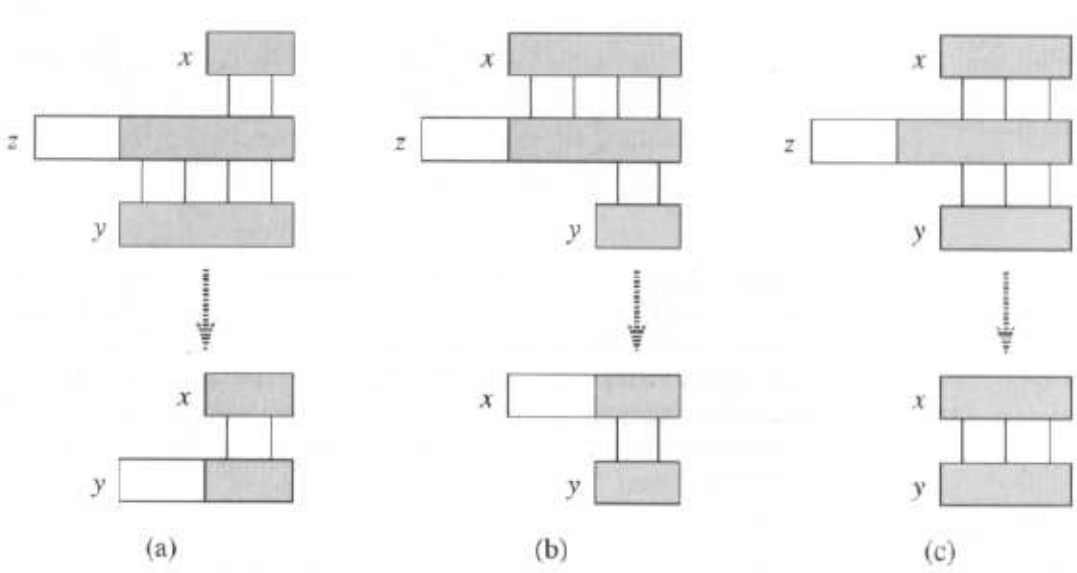


Figura 2. O demonstrație grafică a lemei 1. Presupunem că $x \sqsupset z$ și $y \sqsupset z$. Cele trei părți ale figurii demonstrează cele trei cazuri ale lemei. Liniile verticale leagă regiunile comune ale șirurilor. a) dacă $|x| \leq |y|$ atunci $x \sqsupset y$. b) dacă $|x| \geq |y|$ atunci y$\sqsupset x$ c) dacă $|x| = |y|$ atunci $x = y$.

Vom nota prefixul de k caractere $P[1..k]$ al șablonului $P[1..m]$ cu P_k. Astfel, $P_0 = \lambda$ și $P_m = P = P[1..m]$. Analog, vom nota prefixul format din k caractere, al lui T cu T_k. Folosind această notație, putem reformula problema căutării în șiruri în a găsi, de fapt, a tuturor deplasamentelor s în domeniul $0 \leq s \leq n - m$ astfel încât $P \sqsupset T_{s+m}$.

Algoritmul naiv de căutare în șir

Algoritmul naiv de căutare va găsi deplasamente valide folosind bucle ce testează condiția ca $P[1..m] = T[s + 1..s + m]$ pentru fiecare $n - m + 1$ valori posibile ale lui s.

$NAIVE\_STRING\_MATCHING(T, P)$

1. $n \leftarrow length[T]$
2. $m \leftarrow length[P]$
3. $\boldsymbol{for}\ s \leftarrow 0\ \boldsymbol{to}\ n - m\ \boldsymbol{do}$
4. $\boldsymbol{if}\ P[1..m] = T[s + 1..s + m]\ \boldsymbol{then}$
5. $print$ "$Șablonul\ a\ fost\ găsit\ la\ deplasamentul$" s

Acest algoritm poate fi interpretat grafic, ca un șablon ce alunecă de-a lungul textului, și verifică atunci când toate caracterele sale sunt și în text, după cum apare în figura 3. Bucla *for* corespunzătoare liniei 3, va testa fiecare deplasament explicit. Testul de pe linia 4 determină faptul că deplasamentul este valid sau nu. Aceasta înseamnă compararea fiecărui caracter (într-o buclă) din șablon cu textul inițial.

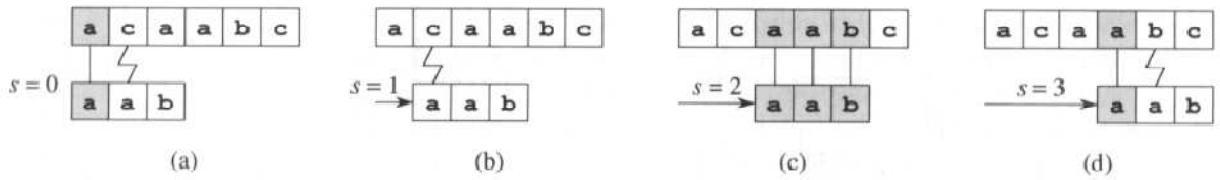


Figura 3. Operația de căutare cu algoritmul naiv pentru un șablon $P = aab$ în textul $T = acaabc$. Părțile a)-d) prezintă patru aliniamente succesive testate de algoritm. Cu gri sunt colorate caracterele ce se potrivesc. Cu linie în zigzag sunt marcate primele caractere ce diferă. Un cuvânt a fost găsit și anume la deplasamentul $s = 2$ (c).

Procedura $NAIVE\_STRING\_MATCHING(T, P)$ are ordinul de timp $O((n - m + 1)m)$. De exemplu dacă luăm în considerare string-ul a^n (adică un șir de a-uri) și șablonul a^m. Pentru fiecare din cele $n - m + 1$ valori posibile de poziționare a deplasamentului, căutarea de pe linia 4 va executa m instrucțiuni pentru a efectua validarea. De aceea, dacă $m = n/2$ algoritmul are un ordin $O(n^2)$.

Așa cum vom vedea algoritmul este ineficient deoarece informația aflată despre text, pentru o valoare s este ignorată total și această informație poate fi benefică. De exemplu, dacă $P = aaab$ și găsim că $s = 0$ este valid, atunci niciunul din deplasamentele $1,2,3$ nu sunt valide, pentru că oricum $T[4] = b$. Vom vedea în cele ce urmează cum să profităm de aceste informații.

Algoritmul Rabin-Karp

Rabin și Karp au propus un alt algoritm cu rezultate bune în practică și care se poate generaliza pentru alte tipuri de căutare, cum ar fi căutarea unui șablon bidimensional. Cel mai nefericit caz al acestui algoritm are un ordin de timp $O((n - m + 1)m)$ însă ordinul de timp al cazului mediu este mult mai bun. Se bazează pe noțiuni elementare de teoria numerelor, cum ar fi echivalența a două numere în urma aplicării operației de *mod* – adică restul împărțirii cu un al treilea număr.

De exemplu, să presupunem că $\Sigma = \{0,1,2, \dots ,9\}$ astfel încât fiecare caracter reprezintă o cifră. Într-un caz general, vom presupune că fiecare caracter este o cifră în baza d unde $d = |\Sigma|$. Putem vedea un șir de k caractere consecutive ca un număr în baza zece. De exemplu șirul "31415" corespunde numărului în zecimal 31415.

Dat fiind un șablon $P[1..m]$ vom nota cu p valoarea sa corespunzătoare în zecimal. Asemănător, pentru un text $T[1..n]$ vom nota cu t_s valoarea în zecimal a subșirului de lungime $T[s+1..s+m]$ pentru $s = 0,1, \dots , n-m$. Sigur, $t_s = p$ dacă și numai dacă $T[s+1..s+m] = P[1..m]$. Astfel s, este un deplasament valid doar dacă $t_s = p$. Dacă putem calcula p în timp $O(m)$ și toate valorile lui t_i în $O(n)$, vom putea determina toate deplasamentele valide în $O(n)$ comparând p cu fiecare t_s. Pentru moment nu vom trata cazul în care t_s sau p sunt numere mari.

Putem calcula p în $O(m)$ folosind regula Horner:

$$p = P[m] + 10\big(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1]) \dots)\big)$$

Valoarea t_0 poate fi calculată asemănător pentru $T[1..m]$ în timp $O(m)$.

Pentru a calcula valorile rămase, $t_1, t_2, \dots , t_{n-m}$ în timp $O(n-m)$, se poate observa că t_{s+1} poate fi calculat pe baza t_s în timp constant, deoarece:

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

De exemplu, dacă $m = 5$ și $t_s = 31415$, vom dori să ștergem cea mai din stânga cifră, și anume $T[s+1] = 3$ și să aducem prima cifra din dreapta, ce urmează în text și anume $T[s+5+1] = 2$ pentru a obține $t_{s+1} = 10(31415 - 10000 \cdot 3) + 2 = 14152$.

Dacă extragem $10^{m-1}T[s+1]$ vom șterge cifra aflată cel mai la stânga lui t_s și multiplicarea cu 10 introduce o nouă cifră în dreapta. Apoi adăugarea cu $T[s+m+1]$ va aduce pe poziția nouă din dreapta, următoarea cifră din text. Putem spune că aceste operații vor necesita un timp constant pentru execuția lor. De aceea p și $t_0, t_1, \dots , t_{n-m}$ poate fi calculat în timp $O(n+m)$ și putem afla toate deplasamentele valide în acest ordin de timp.

Singura problemă apare când numerele p și t_s sunt prea mari ca să se poată lucra cu ele în mod convenabil. Dacă P conține m caractere, atunci presupunerea că fiecare operație aritmetică pe p (care are m cifre) *necesită un timp constant,* nu mai este corectă. Există o soluție simplă pentru această problemă, după cum apare și în figura 4 și anume: se calculează restul împărțirii lui p și t_s la un număr ales q. Acest număr este ales pe baza a două condiții: să fie prim și numărul $10q$ să formeze un cuvânt, ce permite ca toate calculele să fie cât mai simple.

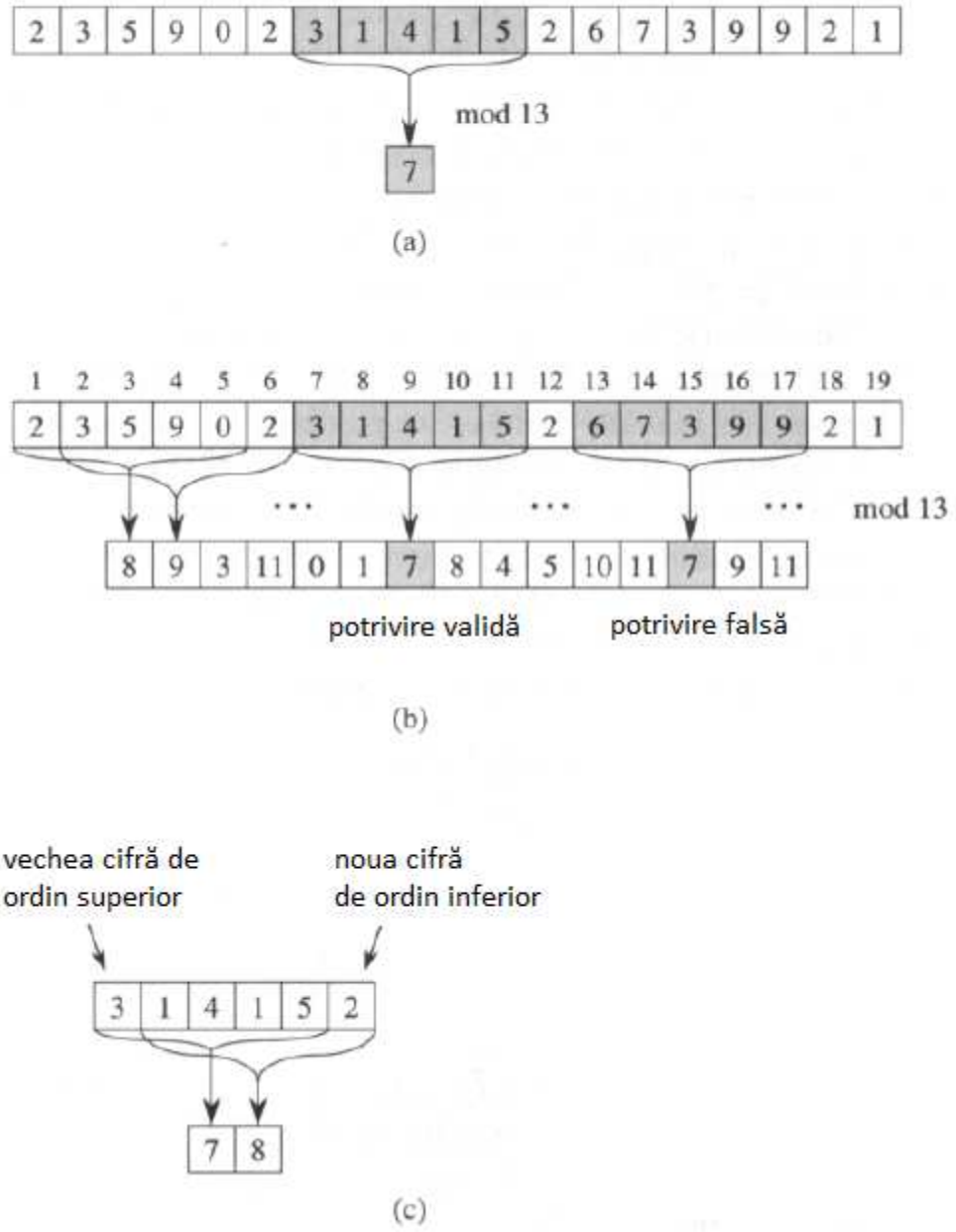


Figura 4. Algoritmul Rabin-Karp. Fiecare caracter este o cifră în baza zece, și vom calcula valorile prin restul împărțirii la 13. a) un text T de intrare. O fereastră de lungime 5 este marcată cui gri. Restul împărțirii numărului marcat la 13 este 7. b) același text cu diverse valori pentru diverse ferestre de lungimi 5 și resturile împărțirii acelor numere la 13. Presupunând șirul de căutat $P = 31415$, atunci căutăm ferestre pentru care numerele de 5 cifre dau restul împărțirii lor cu 13, 7. Avem două cazuri: primul valid iar al doilea fals. Deși restul împărțirii lui 67399 la 13 este 7, acel t_s este diferit de P. c) calculul unei valori corespunzătoare unei ferestre, având dată fereastra anterioară.

În general, cu un alfabet de d elemente, $\{0,1,\dots,d-1\}$, vom alege q astfel încât dq să încapă într-un cuvânt și ecuația de mai jos să aibă sens:

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1])\ mod\ q$$

unde $h \equiv d^{m-1}(mod\ q)$ este valoarea primei cifre de cel mai mare grad din fereastra de m cifre. În acest moment putem spune că $t_s \equiv p\ (mod\ q)$ nu implică $t_s = p$. Pe de altă parte dacă $t_s \not\equiv p\ (mod\ q)$ atunci cu siguranță avem că $t_s \neq p$ deci deplasamentul s este invalid. Vom folosi această condiție și anume $t_s \equiv p\ (mod\ q)$ ca un test pentru a elimina deplasamentele sigur nevalide. Totuși, ulterior orice deplasament pentru care $t_s \equiv p\ (mod\ q)$ va fi testat să vedem dacă $t_s = p$, în caz contrar denumindu-se o *potrivire falsă*. Acest test se poate face verificând $P[1..m] = T[s+1..s+m]$. Dacă q este suficient de mare, atunci aceste potriviri false nu vor fi prea frecvente.

$RABIN - KARP - MATCHING(T, P, d, q)$

1. $n \leftarrow length[T]$
2. $m \leftarrow length[P]$
3. $h \leftarrow d^{m-1}\ mod\ q$
4. $p \leftarrow 0$
5. $t_0 \leftarrow 0$
6. $\boldsymbol{for}\ i \leftarrow 1\ \boldsymbol{to}\ m\ \boldsymbol{do}$
7. $\quad p \leftarrow (dp + P[i])\ mod\ q$
8. $\quad t_0 \leftarrow (dt_0 + T[i])\ mod\ q$
9. $\boldsymbol{for}\ s \leftarrow 0\ \boldsymbol{to}\ n - m - 1\ \boldsymbol{do}$
10. $\quad \boldsymbol{if}\ p = t_s\ \boldsymbol{then}$
11. $\quad\quad \boldsymbol{if}\ P[1..m] = T[s+1..s+m]\ \boldsymbol{then}\ "Deplasament\ valid\ la"\ s$
12. $\quad t_{s+1} \leftarrow (d(t_s - T[s+1]h) + T[s+m+1])\ mod\ q$

În procedura $RABIN - KARP - MATCHING$ toate caracterele sunt văzute ca și cifre în baza d. Linia 3 inițializează h cu valoarea cifrei de pe poziția cea mai din stânga dintr-o fereastră de m cifre. Liniile 4-8 servesc la calculul valorii $P[1..m]\ mod\ q$ și t_0 adică valoarea lui $T[1..m]\ mod\ q$. Bucla *for* de pe linia 9 iterează prin toate deplasamentele posibile s. Dacă $p = t_s$ avem o potrivire, verificăm dacă aceasta este validă adică $P[1..m] = T[s+1..s+m]$ și în acel caz afișăm deplasamentul.
Apoi trecem la următorul deplasament posibil și anume t_{s+1} calculat pe baza discuției de mai sus.

Ordinul de timp al acestui algoritm este $O((n-m+1)m)$ deoarece algoritmul va verifica explicit fiecare deplasament posibil valid. Dacă $P = a^m$ și $T = a^n$, atunci verificările vor fi făcute în timpul menționat, din moment ce fiecare $n - m + 1$ deplasamente sunt posibil valide. În multe aplicații însă vom găsi puține deplasament e valide poate $O(1)$, așa că uneori timpul poate fi în $O(n+m)$ plus timpul necesar procesării deplasamentelor false. În general, dacă avem un număr de v deplasamente valide și un număr de $O(n/q)$ deplasamente false atunci timpul total va fi $O(n) + O(m(v + n/q))$.

Căutarea folosind automate finite

Mulți algoritmi, printre care și cel ce urmează a fi prezentat, folosesc automate finite pentru a scana textele T și a găsi șabloanele P. Vom prezenta pe scurt conceptul de automat finit și cum este el folosit în această problemă. Ce este eficient la aceste automate este faptul că ele examinează fiecare caracter o *singură dată* iar timpul unei examinări este constant. De aceea timpul folosit de un automat este $O(n)$.

Automate finite

Un automat finit este un cvintuplu matematic $M(Q, q_0, A, \Sigma, \delta)$ unde

$Q - mulțime\ finită\ de\ stări$

$q_0 \in Q -\ starea\ de\ start$

$A \subseteq Q - mulțime\ de\ stări\ de\ acceptare$

$\Sigma - este\ alfabetul\ de\ intrare$

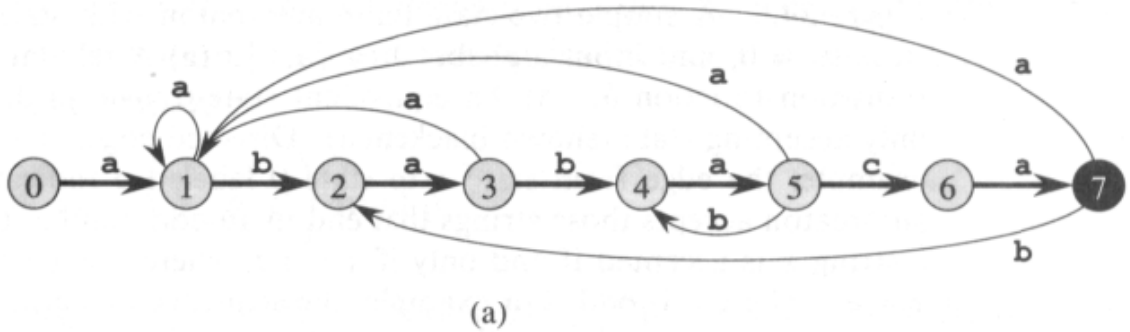
$\delta - funcție\ definită\ pe\ Q \times \Sigma\ cu\ valori\ in\ Q, denumită\ funcție\ de\ tranziție$

Un automat finit începe în starea q_0 și citește caracterele de la intrare unul câte unul. Dacă automatul se află în starea q și citește de la intrare caracterul a, va muta (face o tranziție) din starea q în starea $\delta(q, a)$. Dacă starea unui automat e q și acest q este membru al lui A, atunci se zice că mașina M a *acceptat* șirul, cel puțin până în acest moment. O intrare care nu este acceptată se zice *respinsă*.

Starea *finală* a unui automat, sau stările finale, sunt acelea pentru care $\phi(w)$ este o stare din M în care se ajunge după scanarea șirului w. Asta înseamnă că M acceptă șirul w doar dacă $\phi(w) \epsilon A$.

Folosirea automatelor în căutarea în șiruri

Există un automat pentru fiecare șablon P. Acesta trebuie construit pentru șablonul specific înainte de a fi folosit pentru căutarea în text. Figura 5 ilustrează construcția pentru un șablon $P = ababaca$. Vom presupune că P este un șablon fix.



(a)

intrare

stare	a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
stare $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

Figura 5. a) o diagramă pentru un automat ce acceptă toate șirurile ce se termină în $ababaca$. Starea 0 este starea de start și starea 7 (cea înnegrită) este starea de acceptare. O muchie de la starea i la starea j etichetată cu a se reprezintă astfel $\delta(i, a) = j$. Muchiile care merg *către dreapta* corespund deplasamentelor valide, iar cele care merg *către stânga*, unor deplasamente nereușite. Dacă o stare i nu are nici un arc cu eticheta a ce iese din ea, atunci $\delta(i, a) = 0$. b) funcția de tranziție δ și șablonul P. caracterele care se potrivesc sunt marcate cu gri. c) operarea automatului pe un text $T = abababacaba$. Sub fiecare caracter din $T[i]$ se află starea automatului după ce a procesat prefixul T_i. Se găsește un singur șablon, ce se termină pe poziția 9.

Pentru a specifica automatul corespunzător unui șablon vom defini o funcție auxiliară σ, numită funcție sufix, corespunzătoare lui P. Funcția σ este o corespondență din Σ^* în $\{0,1,\dots,m\}$ astfel că $\sigma(x)$ este lungimea celui mai lung prefix al lui P care este sufix al lui x:

$$\sigma(m) = max\{k: P_k \sqsupset x\}$$

Funcția σ este bine definită din moment ce șirul gol $P_0 = \lambda$ este un sufix pentru orice string. Ca exemplu, pentru $P = ab$ avem $\sigma(\lambda) = 0$, $\sigma(ccaca) = 1$ sau $\sigma(ccab) = 2$.

Pentru un șablon de lungime m avem $\sigma(x) = m$ dacă și numai dacă $P \sqsupset x$.

Vom defini un automat de căutare a unui șablon dat $P[1..m]$:

- Mulțimea Q este $\{0,1,\dots,m\}$. Starea de start q_0 este starea 0 și starea m este singura stare de acceptare.
- Funcția de tranziție δ este definită de ecuația următoare pentru orice stare q și un caracter oarecare a:

$$\delta(q,a) = \sigma(P_q a)$$

Iată un mod intuitiv de a defini această ecuație:

$$\phi(T_i) = \sigma(T_i) \qquad (1)$$

Cu alte cuvinte după scanarea primelor i caractere ale textului T, mașina se va afla în starea $\phi(T_i) = q$, unde $q = \sigma(T_i)$ este lungimea celui mai mare sufix al lui T_i care este și prefix pentru șablonul P. Dacă următorul caracter scanat este $T[i+1] = a$, atunci mașina va face o tranziție către starea $\sigma(T_{i+1}) = \sigma(T_i)a$. Dar $\sigma(T_i a) = \sigma(P_q a)$. Adică pentru a calcula cel mai lung sufix al lui $T_i a$ care este și prefix al lui P, putem calcula cel mai lung sufix al lui $P_q a$ care este prefix al lui P. la fiecare stare mașina va trebui să știe doar lungimea celui mai lung prefix al lui P care este sufix al ceea ce a fost citit până acum.

De exemplu pentru automatul din figura de mai sus, avem $\delta(5,b) = 4$. Aceasta înseamnă că automatul citește un b în starea $q = 5$, apoi $P_q b = ababab$ și cel mai lung prefix al lui P care este sufix al lui $ababab$ este $P_4 = abab$.

Pentru a clarifica operația de căutare în șir, vom prezenta un program ce simulează comportamentul unui automat.

Pentru orice automat, pentru un șablon de lungime m, mulțimea de stări Q este $\{0,1,\dots,m\}$, starea de start este 0 și singura stare de acceptare este m.

$FINITE-AUTOMATON-MATCHING(T,\delta,m)$

1. $n \leftarrow length[T]$
2. $q \leftarrow 0$
3. **for** $i \leftarrow 1$ **to** n **do**
4. $\quad q \leftarrow \delta(q,T[i])$
5. $\quad$ **if** $q = m$ **then** $s \leftarrow i - m$
6. $\quad\quad$ *Șablon găsit la deplasamentul* s

Bucla *for* simplă din cadrul acestei proceduri implică faptul că timpul este $O(n)$. Totuși nu se include timpul necesar calculului tranziției δ. Vom calcula ceva mai târziu timpul corect.

Fie textul de intrare $T[1..n]$. Automatul se află în starea $\sigma(T_i)$ după scanarea caracterului $T[i]$. Din moment ce $\sigma(T[i]) = m$ dacă și numai dacă $P \sqsupset T_i$, mașina este în starea de acceptare, doar dacă șablonul P tocmai a fost scanat. Demonstrația se face prin inducție și se bazează pe ecuația (1).

Calculul funcției de tranziție

Procedura următoare calculează funcția δ pentru un șablon dat

$COMPUTE-TRANSITION-FUNCTION(P,\Sigma)$

1. $m \leftarrow length[P]$
2. **for** $q \leftarrow 0$ *to* m *do*
3. $\quad$ **for** *each caracter* $a \in \Sigma$ **do**
4. $\quad\quad k \leftarrow \min(m+1, q+2)$
5. $\quad\quad$ **repeat** $k \leftarrow k-1$
6. $\quad\quad$ **until** $P_k \sqsupset P_q a$
7. $\quad\quad \delta(q,a) \leftarrow k$
8. **return** δ

Procedura calculează $\delta(q,a)$ conform definiției. Buclele corespunzătoare liniilor 2 și 3 vor considera toate stările q și caracterele a și se caută cer mai mare k *pentru care* $P_k \sqsupset P_q a$.

Timpul acestei funcții este $O(m^3\Sigma)$ ceea ce înseamnă că nu este foarte eficientă pentru șabloane de dimensiune mare. Vom vedea în continuare un algoritm care eficientizează această căutare.

Algoritmul Knuth-Morris-Pratt

Vom prezenta în cele ce urmează, un algoritm în timp linear dezvoltat de autorii al căror nume poartă. Algoritmul lor are un ordin de timp $O(n+m)$ și se bazează pe o funcție auxiliară $\pi[1..m]$ ce este anterior calculată într-un timp de $O(m)$. Șirul π permite calculul unei funcții de tranziție pe loc. Pentru orice stare $q = 0,1,\dots,m$ și orice caracter $a \in \Sigma$, valoarea $\pi[q]$ conține o informație independentă de a și este necesară la calculul lui $\delta(q,a)$. Din moment ce șirul π are doar m elemente, unde δ are $O(m|\Sigma|)$ elemente, vom scoate acel Σ prin calculul preprocesat al lui π.

Funcția prefix pentru un șablon

Funcția prefix pentru un șablon încapsulează informația acumulată, despre cum șablonul în sine se potrivește cu el însuși pe anumite ferestre. Această informație poate fi folosită pentru a evita testele inutile din algoritmul naiv, sau evitarea calculului anticipat δ.

Să consideră operațiile din algoritmul naiv. Figura 6 prezintă un deplasament s pentru un șablon $P = ababaca$ căutat în textul T. De exemplu, $q = 5$ caractere s-au potrivit, dar al 6-lea caracter nu se potrivește. Informația că q caractere s-au potrivit ne permite localizarea acestor caractere, și a spune ce deplasamente sunt invalide. De exemplu deplasamentul $s+1$ este sigur invalid pentru a prima caracter al șablonului a va fi aliniat cu un caracter din text care se potrivește cu al doilea caracter din șablon și anume b. Deplasamentul $s+2$, însă, aliniază primele trei caractere ce trebuie să se potrivească.

Se pune următoarea întrebare: dat fiind $P[1..q]$ ce se potrivește cu $T[s+1..s+q]$, care este cel mai mic deplasament $s' > s$ astfel încât $P[1..k] = T[s'+1..s'+k]$ unde $s'+k = s+q$?

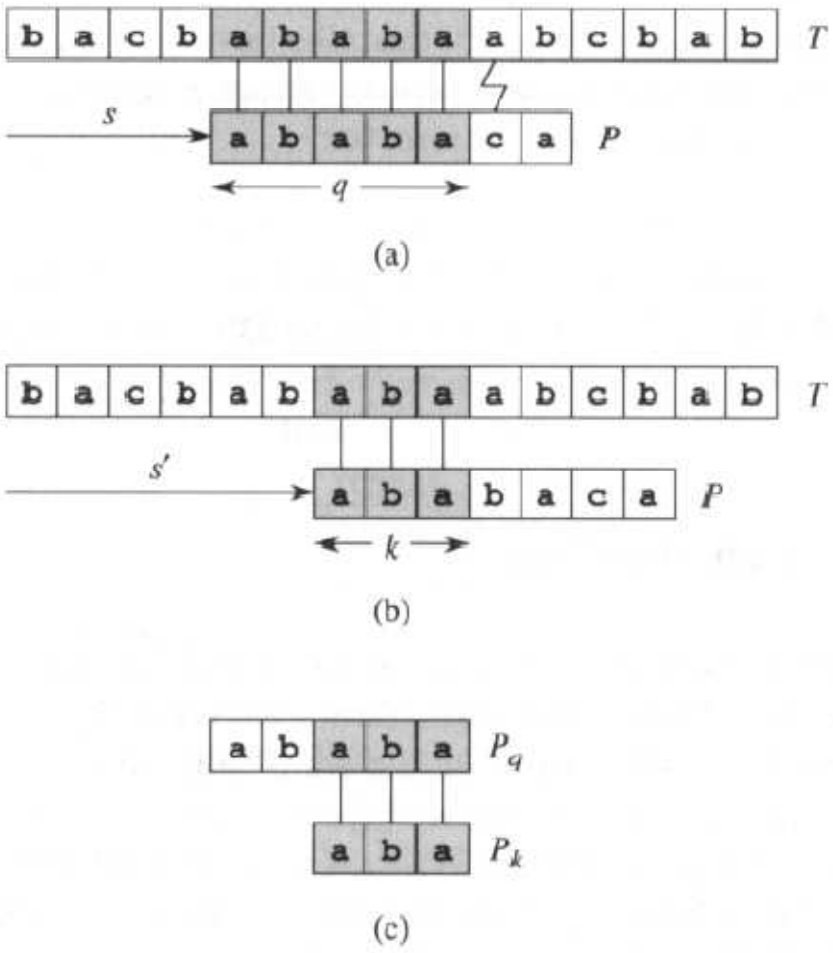


Figura 6. Funcția prefix π. a) Șablonul $P = ababaca$ este aliniat cu un text T astfel ca primele $q = 5$ caractere să fie egale. b) folosind caracterele colorate cu gri, putem deduce că un deplasament $s + 1$ este invalid, dar un deplasament de $s + 2$ este potențial valid. c) deducția făcută la punctul b) poate veni din studiul șablonului. Cel mai lung prefix al lui P care este și sufix al lui P este aba. Astfel $\pi[5] = 3$.

Acest deplasament s' este primul deplasament mai mare decât s care nu este necesar invalid, datorită informației pe care o avem. În cel mai bun caz avem $s' = s + q$ și deplasamentele $s + 1, s + 2, \dots, s + q - 1$ sunt eliminate.

Oricum, la noul deplasament s' nu trebuie să mai calculăm primele k caractere din P cu acele caractere corespondente din T din moment ce știm sigur că $P[1..k] = T[s' + 1..s' + k]$. Această egalitate poate fi reformulată, prin căutarea celui mai mare $k < q$ astfel ca $P_k \sqsupset P_q$. Când găsim acest k, $s' = s + q - k$ este un potențial deplasament.

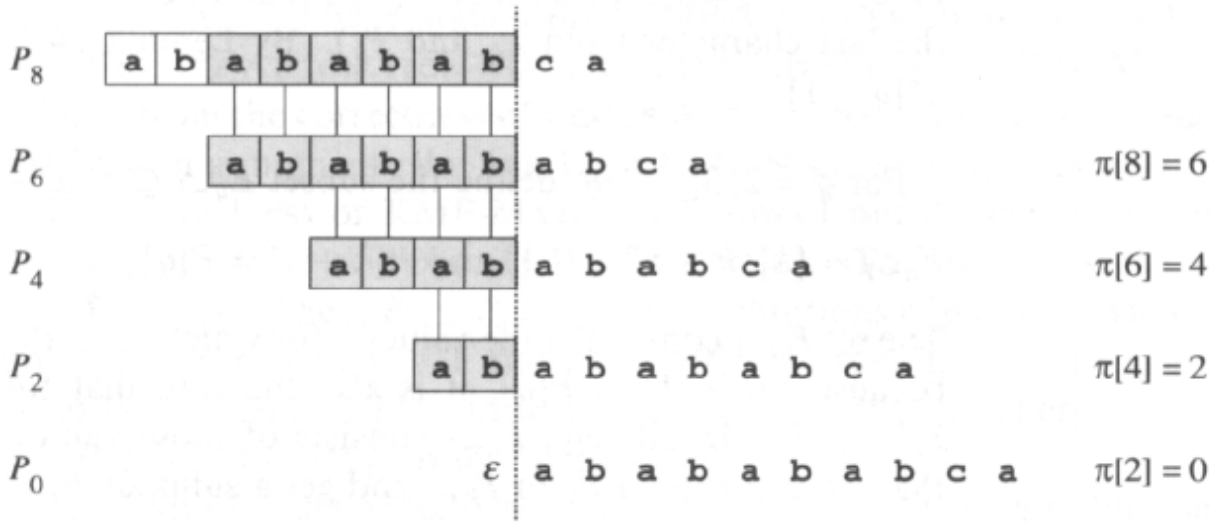
Vom exprima funcția prefix pentru un șablon P, $\pi: \{1,2, \dots, m\} \to \{0,1, \dots, m-1\}$ astfel că:

$$\pi[q] = max\{k: k < q \text{ și } i\ P_k \sqsupset P_q\}$$

Cu alte cuvinte $\pi[q]$ este lungimea celui mai lung prefix al lui P care este un sufix al lui P_q. În figura

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

(a)



(b)

Figura 7. a) funcția π pentru șablonul $P = abababacba$. Obținem $\pi^* = \{8,6,4,2,0\}$. b) vom parcurge șablonul astfel ca prefixul P_k să fie sufix al lui P_8. Aceasta are loc pentru $k = 6,4,2,0$.

$KMP - MATCHING(T, P)$

1. $n \leftarrow length[T]$
2. $m \leftarrow length[P]$
3. $\pi \leftarrow COMPUTE - PREFIX - FUNCTION(P)$
4. $q \leftarrow 0$
5. $\boldsymbol{for}\ i \leftarrow 1\ \boldsymbol{to}\ n\ \boldsymbol{do}$
6. $\quad \boldsymbol{while}\ q > 0\ and\ P[q+1] \neq T[i]\ \boldsymbol{do}$
7. $\quad\quad q \leftarrow \pi[q]$
8. $\quad \boldsymbol{if}\ P[q+1] = T[i]\ \boldsymbol{then}\ q \leftarrow q+1$
9. $\quad \boldsymbol{if}\ q = m\ \boldsymbol{then}\ print\ "Deplasament\ la"\ i - m$
10. $\quad\quad\quad\quad q \leftarrow \pi[q]$

$COMPUTE - PREFIX - FUNCTION(P)$

1. $m \leftarrow length[P]$
2. $\pi[1] \leftarrow 0$
3. $k \leftarrow 0$
4. $\boldsymbol{for}\ q \leftarrow 2\ \boldsymbol{to}\ m\ \boldsymbol{do}$
5. $\quad \boldsymbol{while}\ k > 0\ and\ P[k+1] \neq P[q]\ \boldsymbol{do}$
6. $\quad\quad k \leftarrow \pi[k]$
7. $\quad\quad \boldsymbol{if}\ P[k+1] = P[q]\ \boldsymbol{then}\ k \leftarrow k+1$
8. $\quad\quad \pi[q] \leftarrow k$
9. $return\ \pi$