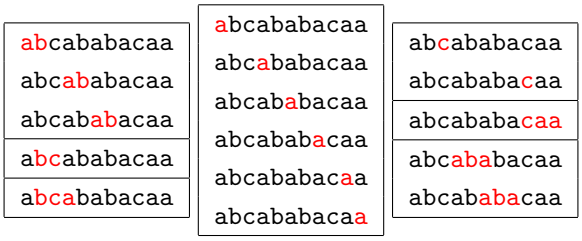
Algoritmul Aho-Corasick
Dinamici pe automate finite

Iulian Oleniuc

1 Motivație

Se dă un string s (numit *text*) și n string-uri $t_1, t_2, \ldots, t_n$ (numite *pattern*-uri). Să se determine, pentru fiecare pattern t_i, numărul său de apariții în s, notat $\alpha_s(t_i)$. Toate string-urile date sunt construite peste alfabetul Σ și au lungimea strict pozitivă.

De exemplu, $s =$ `abcababacaa` și $t_i \in \{$`a`, `ab`, `aba`, `bc`, `bca`, `c`, `caa`$\}$:



2 Automate finite deterministe

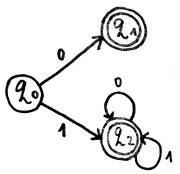
Pentru a înțelege cu adevărat ideea din spatele Algoritmului Aho-Corasick, trebuie mai întâi să înțelegem conceptul de *automat finit determinist* – un mecanism pentru recunoașterea limbajelor de diverse tipuri.

2.1 Definiție

Formal, un automat finit determinist este un tuplu $(Q, \Sigma, \delta, q_0, F)$, unde Q este mulțimea de stări, Σ este alfabetul de intrare, $\delta : Q \times \Sigma \rightsquigarrow Q$ este funcția de tranziție, q_0 este starea inițială, iar $F \subseteq Q$ este mulțimea stărilor finale.

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2\} \\
\Sigma &= \{\texttt{0}, \texttt{1}\} \\
\delta &= \{(q_0, \texttt{0}) \to q_1, (q_0, \texttt{1}) \to q_2\} \\
&\quad \cup \{(q_2, \texttt{0}) \to q_2, (q_2, \texttt{1}) \to q_2\} \\
q_0 &= q_0 \\
F &= \{q_1, q_2\}
\end{aligned}
$$

Pentru o intuiție mai bună, fiecărui automat îi putem asocia un (multi)graf orientat. Mulțimea sa de noduri este Q, iar pentru fiecare relație de forma $\delta(q_i, c) = q_j$ putem construi o muchie (q_i, q_j) cu eticheta c.



2.2 Algoritmul de recunoaștere

Rolul unui automat este să ne spună dacă un cuvânt peste alfabetul Σ, primit ca input, urmează un anumit pattern. Algoritmul pentru recunoașterea unui cuvânt s este foarte simplu: Se pornește din q_0 și se citește pe rând câte un caracter c din s. La fiecare pas, ne deplasăm din starea curentă urmând tranziția cu caracterul c. Adică, din q_i mergem în $\delta(q_i, c)$.

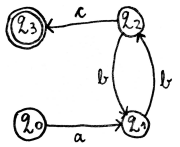
Dacă funcția δ nu este definită în (q_i, c), înseamnă că ne-am blocat în starea q_i înainte să procesăm tot input-ul, așa că *rejectăm* cuvântul. În schimb, dacă am ajuns la sfârșitul input-ului și ne aflăm într-o stare finală, *acceptăm* cuvântul și ne oprim.

accept	`10010`	$q_0 \xrightarrow{1} q_2 \xrightarrow{0} q_2 \xrightarrow{0} q_2 \xrightarrow{1} q_2 \xrightarrow{0} q_2 \in F$
reject	`011`	$q_0 \xrightarrow{0} q_1 \xrightarrow{1}$

Dacă un automat conține două tranziții din aceeași stare, cu aceeași literă, atunci acesta devine *nedeterminist*, pentru că, în timpul evaluării unui anumit cuvânt, nu va ști ce tranziție să urmeze din acea stare.

2.3 Exemple

Automatul de mai sus recunoaște cuvintele ce reprezintă numere scrise în baza 2. Întâi verifică cu ce cifră încep acestea. Dacă încep cu 0, avem grijă ca lungimea lor să nu depășească 1. În caz contrar, pot fi urmate de zero, una sau mai multe cifre, indiferent de valoarea lor.



Un alt exemplu este automatul de mai sus. Acesta recunoaște cuvintele de forma $\texttt{ab}^{2k-1}\texttt{c}$, cu $k \geq 1$, deoarece orice drum care duce la acceptarea input-ului are forma

$$q_0 \xrightarrow{\texttt{a}} \underbrace{q_1 \xrightarrow{\texttt{b}} q_2 \xrightarrow{\texttt{b}} q_1 \xrightarrow{\texttt{b}} q_2 \xrightarrow{\texttt{b}} \cdots \xrightarrow{\texttt{b}} q_1 \xrightarrow{\texttt{b}} q_2}_{\text{de } k \geq 1 \text{ ori } q_1 \xrightarrow{\texttt{b}} q_2} \xrightarrow{\texttt{c}} q_3.$$

3 Algoritmul Knuth-Morris-Pratt ($n = 1$)

Revenind la problema noastră de *string matching*, să analizăm cazul $n = 1$, adică cel în care trebuie să determinăm numărul de apariții ale unui *singur* pattern în textul dat. Acest caz se rezolvă în timp liniar folosind Algoritmul KMP, care a fost deja studiat. Însă îl vom recapitula, privindu-l dintr-o altă perspectivă – cea a automatelor finite.

3.1 Funcția π

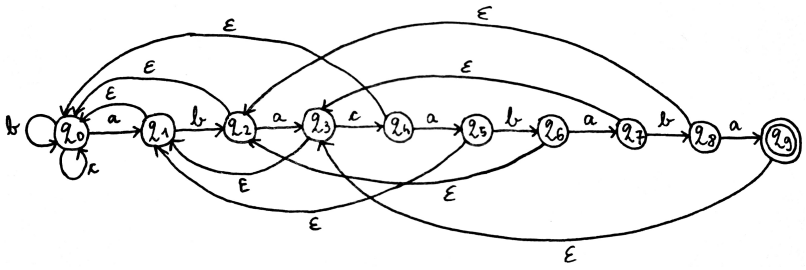
Algoritmul KMP se bazează pe funcția π, numită și funcția *prefix* sau *fail*. Astfel, $\pi(i)$ reprezintă lungimea maximă a unui sufix propriu al lui $t[0, i]$, care este totodată prefix al lui $t[0, i]$.

i	0	1	2	3	4	5	6	7	8
t	a	b	a	c	a	b	a	b	a
π	0	0	1	0	1	2	3	2	3

3.2 Automatul KMP

Ceea ce face Algoritmul KMP de fapt este să construiască un automat mai special, după cum urmează. Avem câte o stare q_i pentru fiecare poziție i din pattern, cu semnificația că, dacă ne aflăm în starea q_i, atunci am făcut *match* la primele i caractere din t. Tranzițiile inițiale sunt de forma $\delta(q_i, t[i]) = q_{i+1}$. Desigur, singura stare finală este $q_{|t|}$.

Restul tranzițiilor sunt generate de funcția π. Astfel, $\pi(i-1) = j$ se traduce prin $\delta(q_i, \epsilon) = q_j$, unde ϵ reprezintă string-ul vid. În timpul parsării input-ului, tranzițiile cu ϵ vor fi urmate doar atunci când nu avem tranziție cu caracterul curent din s! Evident, ϵ nu consumă niciun caracter din s.



Acum că fiecare stare q_i, cu $i > 0$, are câte o ϵ-tranziție, nu ne vom putea bloca niciodată în q_0 atunci când parsăm input-ul. Însă ne vom bloca în q_0 atunci când $s[i] \neq t[0]$. Pentru a rezolva asta, adăugăm tranzițiile $\delta(q_0, c) = q_0, \forall c \neq t[0]$.

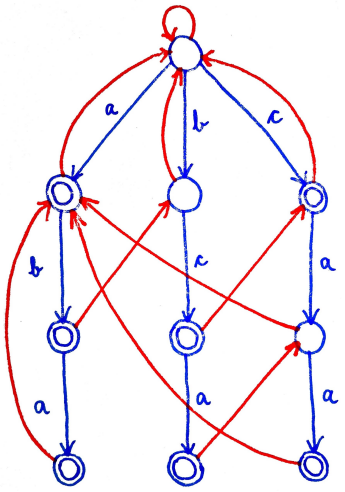
Ideea de bază a ϵ-tranzițiilor, și implicit a funcției π, este că acestea ne duc în starea care face match cu cel mai mare prefix al lui t. Astfel, chiar dacă trebuie să abandonăm un match aproape complet, măcar suntem siguri că nu vom rata niciun alt match care s-ar fi suprapus cu acesta.

4 Algoritmul Aho-Corasick ($n > 1$)

Primul pas în generalizarea Algoritmului KMP, pentru a putea face matching simultan la n pattern-uri, este să construim o trie din acestea. Motivul este că fiecare nod din trie reprezintă un prefix al (cel puțin) unui pattern t_i. Acum, stările din automat nu vor mai fi determinate de lungimea prefixului cu care avem match, ci de prefix în sine. Cu alte cuvinte, vom putea avea mai multe prefixe de aceeași lungime, adică pe același nivel al triei.

4.1 Suffix-Links

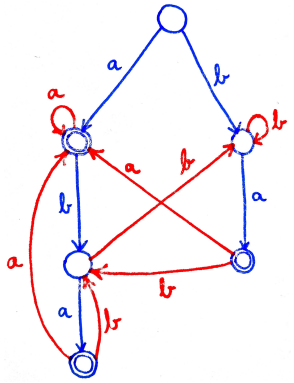
Rămâne să generalizăm funcția π. Vom păstra notația, însă numele va deveni *suffix-link*. Din nou, $\pi(q_i)$ ne va duce în starea q_j corespunzătoare celui mai lung sufix propriu al string-ului reprezentat de q_i, care este totodată prefix al acestuia.



Observăm că suffix-link-ul oricărei stări q_i ne duce pe un nivel al triei mai mic decât cel pe care se află q_i (sau egal dacă $i = 0$). Astfel, putem calcula funcția π exact ca în Algoritmul KMP, procesând nodurile triei în ordine crescătoare după nivel, adică în timpul unui BFS din rădăcină. Însă, această implementare este mai dificilă, și nu ne ajută prea mult în rezolvarea problemelor de mai jos, care necesită generarea unui automat *fără* ϵ-tranziții.

4.2 Automatul Aho-Corasick

Fie δ funcția de tranziție din automatul fără ϵ-tranziții. Din nou, tranzițiile de bază sunt date de muchiile triei, și rămâne să ne ocupăm de suffix-link-uri. Practic, $\delta(q_i, c)$, unde c produce *mismatch*, comprimă drumul obținut mergând pe suffix-link-uri din q_i până într-o stare ce are tranziție cu c, adică $\delta(q_i, c) = \delta(\pi(\pi(\cdots\pi(q_i)\cdots)), c)$.



Așadar, δ depinde de π, dar dacă ne gândim la modul în care se calculează funcția π în Algoritmul KMP, realizăm că și π depinde de δ. Mai precis, obținem recurențele

$$\pi(q) = \delta(\pi(q.p), q.c)$$
$$\delta(q, c) = \delta(\pi(q), c),$$

unde $q.p$ este părintele lui q în trie, iar $q.c$ reprezintă ultimul caracter al string-ului reprezentat de q. Desigur, avem și cazuri particulare:

- dacă q este rădăcina triei sau vreun fiu al acesteia, atunci $\pi(q) = q_0$;
- dacă în trie există din q muchie cu eticheta c către q', atunci $\delta(q, c) = q'$;
- dacă această muchie nu există, și în plus $q = q_0$, atunci $\delta(q, c) = q_0$.

Cum atât $\pi(q)$ cât și $\delta(q, c)$ depind de stări aflate mai sus în trie, remarcăm că recursia este finită. Prin urmare, aceste recurențe pot fi calculate eficient prin memoizare.

5 Aplicații

Prima aplicație la automatul Aho-Corasick este chiar problema din secțiunea introductivă. Celelalte probleme se bazează pe analiza drumului generat de automat în timpul parsării input-ului. Majoritatea sunt probleme de programare dinamică, astfel că recurențele lor depind în general de starea curentă, de poziția la care am ajuns în input și eventual de mulțimea de pattern-uri cu care am făcut match până acum.

Problema 1. Să se determine numărul de apariții ale fiecărui pattern t_i în textul s.

Trecem string-ul s prin automatul Aho-Corasick generat de pattern-uri. De fiecare dată când trecem printr-un nod q al triei, incrementăm valoarea $\alpha(q)$, deoarece tocmai am făcut match cu string-urile reținute în q. Însă acestea nu sunt singurele cu care se face de fapt match la poziția respectivă!

De exemplu, dacă avem $s =$ `cabcabd`, $t_1 =$ `abcab` și $t_2 =$ `cab`, atunci la poziția 5 nu vom avea match doar cu t_1, ci și cu t_2. Practic, dacă avem match în q, vom avea și în $\pi(q)$, și în $\pi(\pi(q))$, și în $\pi(\pi(\pi(q)))$ etc. Prin urmare, după ce am parsat input-ul, vom parcurge nodurile triei în ordinea descrescătoare a nivelurilor, corectând valorile $\alpha(q_i)$ astfel: $\alpha(\pi(q)) \leftarrow \alpha(\pi(q)) + \alpha(q)$.

Este foarte important ca, în timpul parsării input-ului, să incrementăm valoarea $\alpha(q)$ pentru fiecare nod q prin care trecem – nu doar pentru frunze! Motivul este că, atunci când efectuăm actualizarea de mai sus, s-ar putea ca q să nu fie frunză, dar $\pi(q)$ să fie. Astfel, obținem niște match-uri pentru $\pi(q)$ pe care altfel le-am fi ratat. sursă

Problema 2. Să se determine string-ul cel mai mic din punct de vedere lexicografic s, de lungime k, care nu conține ca substring niciunul dintre pattern-urile t_i.

Ca string-ul s să nu conțină niciun pattern t_i, trebuie să ne asigurăm că drumul corespunzător parsării acestuia nu trece prin nicio frunză a triei. În acest sens, este suficient să eliminăm din automat stările frunză. Apoi, nu rămâne decât să mergem de k ori pe tranziția corespunzătoare celei mai mici litere disponibile, pornind din q_0. Concatenând etichetele muchiilor de pe acest drum, obținem string-ul s.

Problema 3. Să se determine cel mai scurt string s care conține ca substring fiecare dintre pattern-urile t_i.

În primul rând, trebuie să reținem în fiecare stare a automatului un *bitmask* cu toate pattern-urile cu care facem match când ajungem în starea respectivă. La fel ca în prima problemă, trebuie avut în vedere că acestea nu provin doar din acea stare, ci și din suffix-link-ul ei. Din acest motiv, după ce calculăm valoarea $\pi(q)$, îl calculăm și pe $\pi(\pi(q))$, pentru a putea actualiza masca astfel:

$$q.mask \leftarrow q.mask \texttt{ or } \pi(q).mask,$$

fiind siguri că valoarea $\pi(q).mask$, care depinde la rândul ei de $\pi(\pi(q))$, este calculată corect. Restul problemei constă în a efectua un BFS pe automat, ținând pe lângă starea curentă și un bitmask cu toate pattern-urile cu care am făcut match până acum.

Problema 4. Fiecărui pattern t_i îi este asociat un cost c_i. De asemenea, costul unui string s este definit drept $c(s) = \sum_{i=1}^{n} \alpha_s(t_i) \cdot c_i$. Să se determine costul maxim al unui string s de lungime k.

Precalculăm, tot în timpul construirii suffix-link-urilor, costul total cu care contribuie fiecare stare la $c(s)$. Restul se reduce la o simplă dinamică, unde dp$[i][q]$ reprezintă costul maxim obținut printr-un drum de lungime i care se termină în starea q. sursă

Problema 5. Să se determine câte string-uri de lungime k există, astfel încât acestea să conțină ca substring fiecare pattern t_i.

Notăm cu dp$[i][q][mask]$ numărul de drumuri de lungime i, care se termină în starea q, și conțin ca substring-uri pattern-urile din $mask$. sursă

Problema 6. Să se determine câte string-uri de lungime k există, astfel încât acestea să nu conțină ca substring niciun pattern t_i.

Eliminăm (conceptual) stările frunză din automat și calculăm dinamica dp$[i][q]$.

Problema 7. Să se determine numărul minim de caractere care trebuie șterse din s astfel încât noul string să nu conțină ca substring niciunul dintre pattern-urile t_i.

Notăm cu dp$[i][q]$ costul minim pentru a procesa primele i caractere din s, terminând în starea q. Fie $q' = \delta(q, s[i])$. Dacă în q' nu se face match la niciun pattern, atunci putem urma această tranziție și actualizăm dp$[i+1][q']$ folosind dp$[i][q]$. Altfel, suntem nevoiți să ștergem caracterul $s[i]$, așa că rămânem în q și actualizăm dp$[i+1][q]$ cu dp$[i][q] + 1$. sursă

6 Probleme