Colegiul National de Informatica “Tudor Vianu”
Bucuresti

Lucrare de atestat

Automate finite – aplicatii in concursurile de informatica

profesor coordonator: Cherciu Rodica
autor: Filip Stefan-Alexandru
clasa: a XII-a D

mai 2008

Cuprins

1. Consideratii generale
2. Automate finite
    1. Potrivirea sirurilor
    2. Automate finite de potrivire
    3. Algoritmul Knuth-Morris-Pratt
    4. Potrivirea sirurilor cu mai multe modele
    5. Structura de TRIE
    6. Contruirea functiei de tranzitie
    7. Optimizarea constructiei functiei de tranzitie
3. Aplicatii in concursurile de informatica
    1. Cifru
    2. ADN
    3. ABC
    4. Strigat
    5. Dangerous Patterns
4. Prezentarea suportului digital
5. Bibliografie

Consideratii generale

Despre concursurile de informatica

Concursurile de informatica pentru elevi si studenti presupun rezolvarea unor probleme de natura algoritmica, concurentii fiind nevoiti sa scrie un program care sa proceseze eficient datele de intrare. Programele trebuie scrise intr-un limbaj de programare acceptat de comisia care organizeaza concursul.

Am spus ca un program trebuie sa fie eficient, prin asta intelegem sa ruleze in limita de timp si respectiv limita de memorie impuse de comisie pentru fiecare problema in parte.

Pentru a stabili corectitudinea programului, el este rulat pentru o serie de date de intrare si pentru fiecare din ele sunt analizate timpul de rulare, memoria folosita si corectitudinea datelor de iesire.

Exista multe site-uri cu arhive de probleme, fiecare din ele gazduind concursuri regulat. Le amintim pe cele mai importante:

- http://infoarena.ro
- http://acm.uva.es
- http://acm.sgu.ru
- http://acm.pku.cn
- http://ace.delos.com
- http://www.topcoder.com/tc

Un capitol important in informatica il reprezinta algoritmii si structurile de date pentru potrivirea sirurile de caractere. Astfel, in multe probleme de concurs necesita cunoasterea in detaliu a acestor algoritmi si structuri de date.

Lucrarea de fata analizeaza din punct de vedere teoretic automatele finite, in diversele forme in care pot fi utilizate, impreuna cu algoritmii care se aplica pe aceste structuri de date.

Automate finite

Un automat finit M este un cvintuplu <Q, q0, A, Σ, δ>, unde

- Q este o multie finita de stari
- q0 ϵ Q este starea de start
- A este o multime de stari de acceptare inclusa in Q
- Σ este un alfabet de intrare finit
- δ : Q × Σ → Q, numita functie de tranzitie a automatului M

Automatele finite se impart in doua categorii:

- Automate finite deterministice (DFA)
- Automate finite nondeterministice (NFA)

Un DFA este un automat finit pentru care fiecare tranzitie este unic determinata, in schimb ce un NFA accepta mai multe stari pentru fiecare tranzitie.

Desi cele doua tipuri de automate au definitii diferite, se poate arata ca in teorie sunt echivalente, si ca pentru fiecare DFA se poate construi un NFA si vice-versa.

Aplicatiile automatelor finite in practica se refera la inteligenta artificiala rudimentara si la potrivirea sirurilor.

Sa ne imaginam un model simplificat de aparat, si automatele finite mecanismul prin care este specificat limbajul lor. Singura lor memorie o reprezinta un numar finit de stari, din care nu pot iesi, functia de tranzitie asigurand trecerea intre stari in functie de comenzile primite. Drept exemplu putem lua automatul de bauturi racoritoare, starea lui se modifica atunci cand introducem monede sau cand apasam un buton, si o stare de acceptanta elibereaza o sticla din aparat.

Potrivirea Sirurilor

Mai intai sa analizam algoritmul rudimentar de potrivire al sirurilor, pentru un text T si un model P:

```
Potrivire-naiva-a-sirurilor(T, P)
   1. n ← lungime[T]
   2. m ← lungime[P]
   3. pentru s ← 0, n – m executa
   4.     daca P[1..m] = T[s+1..s+m] atunci
   5.         tipareste “Modelul apare cu deplasamentul”, s
```

Procedura are complexitatea $O(n^2)$. Se observa ca pentru cazul cel mai defovarabil in care sirul T este format din n caractere ε si modelul P este format din m caractere ε sunt valabile toate cele n-m+1 deplasamente. In acest caz ciclul implicit din linia 4 se va executa in θ(m). Astfel pentru cazul cel mai defavorabil m=n/2 rezulta o complexitate $O(n^2)$.

Un automat de potrivire a sirurilor are urmatoarea forma pentru un model P:

- Q este multimea formata din toate prefixele lui P; Q = { P_k | P_k prefix pt P }
- q0 este sirul vid; ø
- A este chiar modelul P
- Σ este format din toate literele alfabetului; in general A-Za-z
- δ este o functie de forma $\delta(q_k, \varepsilon) = \sigma(q_k\varepsilon)$, unde σ(X) este cel mai lung prefix la lui P care este sufix pentru X.

Sa detaliem putin functia σ(X). Aceasta nu ne transmite nici mai mult nici mai putin decat ce portiune a modelului P se potriveste pe ultimele caractere ale lui X.

Sa luam un exemplu: fie sirul P = aabbac, atunci

- σ(aaca) = a
- σ(aabbaa) = aa
- σ(aaaabb) = aabb
- σ(caabbac) = aabbac
- σ(htradv) = ø

Automate finite de potrivire

In continuare este prezentat algoritmul pentru potrivirea sirurilor folosind un astfel de automat:

```
Automat-finit-de-potrivire(T, δ, m)
   1. n ← lungime[T]
   2. q ← 0
   3. pentru i ← 1, n executa
   4.     q ← δ(Q, T[i])
   5.     daca q = P atunci
   6.         tipareste “Modelul apare cu deplasamentul”, i-m
```

Complexitatea acestui algoritm este O(n) daca se negrlijeaza comparatia din linia 5, care desi aparent are complexitatea O(m) se poate reduce usor la O(1). Problema adevarata este calcularea functiei δ. In continuare este prezentata functia naiva.

```
Calcul-functie-de-tranzitie(P, Σ)
   1. m ← lungime[P]
   2. pentru q ϵ Q executa
   3.     pentru a ϵ Σ executa
   4.         k ← m+1
   5.         repeta
   6.             k ← k-1
   7.         pana cand Pₖ prefix pt qa
   8.         δ(q, a) ← Pₖ
```

Aceasta procedura calculeaza simplu functia δ si are complexitatea $O(m^3|\Sigma|)$. Ciclul exterior contribuie cu $O(m|\Sigma|)$, sunt m prefixe posibile si comparatie necesita Θ(m) operatii. Acest algoritm poate fi imbunatatit pana la $O(m|\Sigma|)$ cum vom vedea mai tarziu.

Algoritmul Knuth-Morris-Pratt

Acest algoritm evita calcularea explicita a functiei de tranzitie si complexitatea este redusa proportional cu dimensiunea alfabetului de intrare Σ, folosind o functie auxiliara π precalculata pornind de la modelul P. Functia π poarta numele de functie prefix.

Vom porni de la algoritmul naiv de potrivire. Pentru un deplasament s, stim ca se potrivesc exact t caractere. Pentru cazul in care deplasamentul este s+1, este nevoie sa verificam doar unele din primele caractere ale modelului P, si anume acelea care sub forma unui prefix sunt egale cu sufixul lui P_t.

P_k sufix P_t

P_k prefix P_t

P[1..t] = T[s..s+t-1]

=> P[1..k] = T[s+1..s+k]

Functia π va retine pentru fiecare prefix al lui P exact care este lungimea celui mai lung prefix care este si sufix.

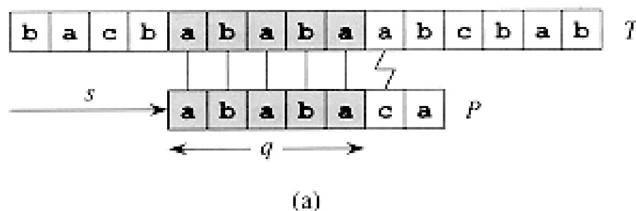
$\pi[q] = \max\{k : k < q \text{ si } P_k \text{ sufix pentru } P_q\}$

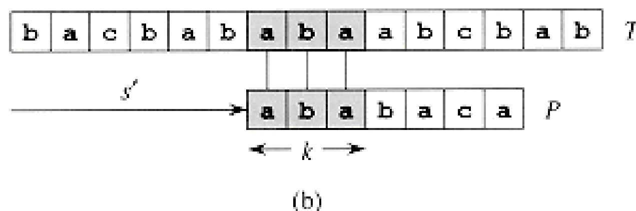
In continuare este prezentat in pseudocod algoritmul de potrivire al sirurilor Knuth-Morris-Pratt.

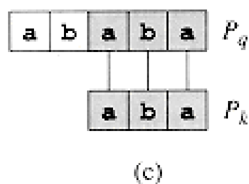
```
Potrivire-KMP(T, P)
   1. n ← lungime[T]
   2. m ← lungime[P]
   3. π ← Calcul-Functie-Prefix(P)
   4. q ← 0
   5. pentru i ← 1, n executa
   6.     cat timp q > 0 si P[q + 1] <> T[i] executa
   7.         q ← π[q]
   8.     daca P[q + 1] = T[i] atunci
   9.         q ← q + 1
   10.    daca q = m atunci
   11.        tipareste "Modelul apare cu deplasamentul" i - m
   12.        q ← π[q]

Calcul-Functie-Prefix(P)
   1. m ← lungime[P]
   2. π[1] ← 0
   3. k ← 0
   4. pentru q ← 2, m executa
   5.     cat timp k > 0 si P[k + 1] <> P[q] executa
   6.         k ← π[k]
   7.     daca P[k + 1] = P[q] atunci
   8.         k ← k + 1
   9.     π[q] ← k
   10.returneaza π
```

Folosind o analiza amortizata, putem demonstra ca timpul de exectutie pentru Calcul-Functie-Prefix este O(m). Asociem un potential k cu starea k curenta a algoritmului. Acest potential are valoarea initiala 0, initializat in linia 3. Linia 6 decrementeaza k ori de cate ori este executata, deoarece $\pi[k] < k$.

Din definitie $\pi[k] >= 0$ pentru orice k, deci k nu poate deveni negativ. Singura linie care mai afecteaza valoarea lui k este linia 8 unde se incrementeaza k, cel mult o data, in timpul fiecarei iteratii a ciclului pentru, deoarece $k < q$ dupa intrarea in ciclul pentru si deoarece q este incrementat la fiecare iteratie a ciclului pentru, in orice moment $k < q$. Cum linia 8 incrementeaza o singura data functia de potential, costul amortizat al unei iteratii este O(1).







Exemplu pentru felul in care evolueaza deplasamentul in algoritmul KMP. (a) este detectata o divergenta; (b) din informatiile pe care le avem despre model, ne putem da seama ca deplasamentul s+1 este incorect, dar putem sa alegem deplasamentul s+2; (c) informatiile pe baza carora se poate alege deplasamentul s+2, P_k este sufix si prefix al lui P_q

Ca sa reluam, k creste maxim m unitati, cu cel mult o unitate pe iteratie, si scade atat cat sa nu fie negativ. Observati ca pentru o executie a liniei 8 (incrementare a lui k) ii corespunde numai o executie a liniei 6 (decrementare a lui k).

In concluzie procedura Calcul-Functie-Prefix are complexitatea O(m).

Asemanator se poate arata si pentru restul procedurii Potrivire-KMP ca are complexitatea O(n), analizand functia potentia a valorii q.

Asadar complexitatea finala a procedurii Potrivire-KMP este O(n + m). Cel mai important lucru pe care il putem observa este faptul ca nu depinde de dimensiunea alfabetului, ci doar de lungimea textului si modelului.

Problema potrivirii sirurilor cu mai multe modele

Algoritmul KMP este foarte eficient daca avem de cautat un singur model. Problema apare in momentul in care trebuie sa cautam un numar mai mare de modele, in acest caz complexitatea obtinuta folosind algoritmul KMP este O(w * n + M), unde w este numarul de cuvinte si M este lungimea tuturor celor w cuvinte.

O sa pornim de la un automat pentru un singur cuvant. In continuare este prezentat algoritmul de construire al unui automat finit de potrivire pentru un singur model.

```
Construire-Automat-Simplu(P)
   1. m ← lungime[P]
   2. initializeaza δ pe 0
   3. pentru i ← 0, m-1 executa
   4.     temp ← δ[i][ P[i+1] ]
   5.     δ[i][ P[i+1] ] ← i + 1
   6.     pentru ε ∈ Σ executa
   7.         daca P[temp + 1] = ε atunci
   8.             δ[i + 1][ε] ← temp + 1
   9.         altfel
   10.            δ[i + 1][ε] ← δ[temp][ε]
```

Mai intai sa explicam algoritmul. Sa presupunem ca ne aflam pe pozitia i in sir. Atunci pentru a = P[i + 1] prefixul cel mai lung este chiar i + 1, altfel ne vom raporta la cel mai lung prefix al lui P_i care este si sufix si este mai mic decat i, gasit in algoritm in variabila temp, deci temp < i tot timpul. Acum trebuie sa verificam relatia pe care o are acest prefix P_{temp} cu ε, unde ε este litera pentru care calculam functia δ din i. Daca P[temp+1] = ε atunci prefixul cel mai lung creste cu o unitate, daca P_{temp} sufix si prefix al lui P_i atunci P_{temp+1} va fi sufix si prefix al lui P_iε. Altfel, daca P[temp+1] <> ε, cel mai lung prefix care este si sufix al lui P_iε este δ[temp][ε], si asta rezulta chiar din definitia functiei δ. Singura grija este ca δ[temp][ε] sa fie calculat, dar cum temp < i si starile sunt calculate in mod crescator valoare lui δ[temp][ε] este deja calculata. Algoritmul prezentat mai sus prezinta si o ocolire pentru a pastra toate starile temporare in care se duce functia δ in absenta caracterului ε.

Este usor de observat ca algoritmul are complexitatea O(m|Σ|). Ciclul 4-10 este executat de m ori, iar pentru fiecare iteratie sunt analizate toate caracterele alfabetului Σ, cum pentru fiecare caracter sunt realizate numai o comparatie si o atribuire, complexitatea pentru liniile 7-10 este O(1). In concluzie complexitatea finala a algoritmului Construire-Automat-Simplu este O(m|Σ|).

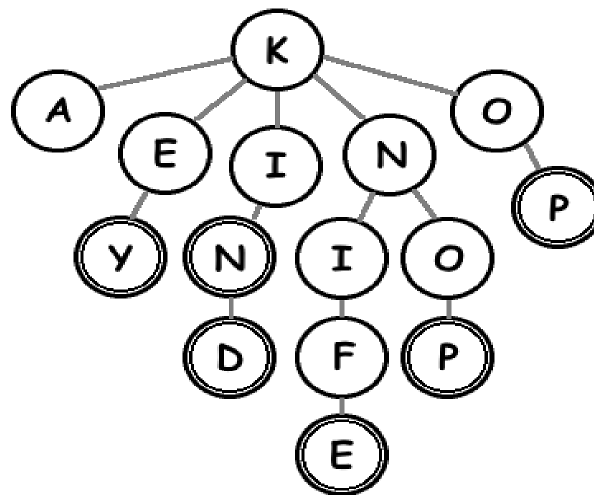
Structura de date TRIE

Inainte sa trecem la generalizarea automatului prezentat mai inainte pentru un numar oarecare de w modele, sa prezetam pe scurt structura de date numita TRIE (pronuntia fonetica este try). Un TRIE este un arbore ordonat pentru care fiecare nod reprezinta in general un sir de caractere. O muchie intre un parinte si un fiu este marcata cu un caracter iar fiul are stringul care corespunde valorii fiului este egal cu stringul care corespunde valorii parintelui, la care se adauga caracterul asociat muchiei.

u → v; char[u][v] = c

=> string[v] = string[u] + c

Un TRIE poate fi privit si ca un automat finit deterministic, unde functia δ este slab definita, mai exact δ[u][c] = v, unde v ϵ Q si nu unde c ϵ Σ.



Exemplu de TRIE, care are radacina caracterul K.

Sa prezentam acum algoritmul de construire al unui TRIE, unde P reprezinta sirul de modele.

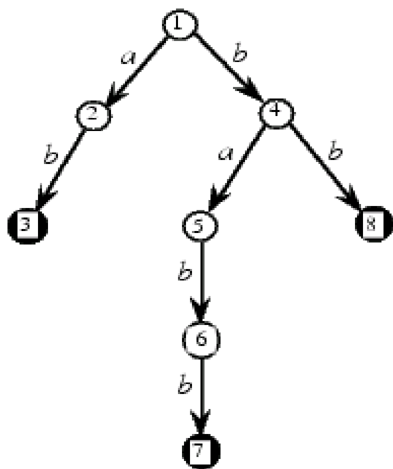
```
Construieste-Trie(P)
   1. w ← lungime[P]
   2. pentru i ← 1, w executa
   3.     m ← lungime[ P[i] ]
   4.     pos ← radacina
   5.     pentru j ← 1, m executa
   6.         Trie[ pos ][ P[i][j] ] ← Pj
   7.         pos ← Pj
   8.     adauta P la A
```

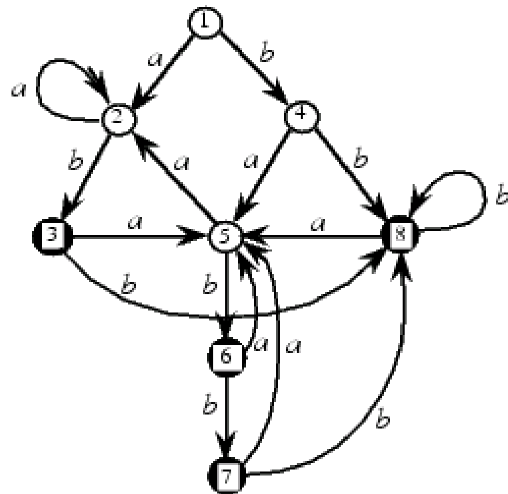
De mentionat ca R este radacina TRIE-ului, si ca A este multimea starilor de acceptare. Din cele ce le-am prezentat pana acum, nu ar trebui sa fie probleme in intelegerea pseudocodului. Practic Trie este doar portiunea din δ unde exista muchia cu eticheta c incidenta nodului u.

Construirea functiei de tranzitie

Partea dificila este construirea functiei δ unde nu exista muchia inainte.

```
Construieste–Automat(Trie, A)
   1. initializeaza δ cu radacina
   2. pentru x ∈ Trie, x <> radacina, in ordinea parcurgerii in latime
      executa
   3.      t ← parinte[x]
   4.      a ← characterul asociat muchiei catre parinte
   5.      temp ← δ[t][a]
   6.      δ[t][a] ← x
   7.      daca temp ∈ A atunci
   8.           adauga x la A
   9.      pentru ε ∈ Σ executa
   10.          daca Trie[temp][ε] este definit atunci
   11.               δ[x][ε] ← Trie[temp][ε]
   12.          altfel
   13.               δ[x][ε] ← δ[temp][ε]
   14.returneaza δ, A
```





In stanga este un TRIE, iar in dreapta este reprezentarea automatului corestpunzator lui.

Acest algoritm este asemanator celui care construieste un automat pentru un singur model, cu precizarea ca referintele la caracterul precedent s-au transformat in referinte catre parintele nodului, si cele la caracterele urmatoare in referinte catre fii din Trie. Putem sa privim Contruieste-Automat-Simplu ca un caz particular la procedurii Construieste-Automat pentru cazul in care fiecare nod al TRIEului are un singur fiu, inclusiv radacina.

Diferenta principala o constituie liniile 7-8 care se refera la starile de acceptanta ale automatului. Este usor sa aratam ca daca temp este o stare de acceptanta si temp este sufix al lui x, atunci si x este o stare de acceptanta.

Mai ramane sa analizam timpul de executie al algoritmului. Ciclul pentru din linia 2 se executa pentru toate nodurile din Trie, care, conform procedurii Construieste-Trie este format din toate prefixele tuturor cuvintelor deci O(M), suma lungimilor tuturor modelelor. Linile 3-8 se executa in timp constant, iar ciclul pentru din linia 9 este direct proportional cu dimensiunea alfabetului, O(|Σ|), liniile 10-13, necesitand de asemenea timp constant la executie. In concluzie complexitatea procedurii Construieste-Automat este O(M|Σ|).

Optimizarea procedurii de constructie a automatului

In practica algoritmul de mai sus este foarte eficient, mai ales ca alfabetul impus este in general mic. In continuare vom prezenta o optimizare a algoritmului de construire a automatului.

Daca alfabetul este mare, comparativ cu lungimile sirurilor, atunci se poate opta pentru alte structuri de date pentru a pastra toate prefixele, diferite de TRIEuri, spre exemplu hash-uri, arbori echilibrati de cautare, siruri de prefixe sau cel mai eficient probabil arbori de prefixe.

```
Contruieste-Automat-opt(T, A)
   1. initializeaza π cu radacina
   2. seteaza π[radacina] nedefinit
   3. pentru x ϵ Trie, x <> radacina, in ordinea parcurgerii in latime
      executa
   4.     t ← parinte[x]
   5.     a ← characterul asociat muchiei catre parinte
   6.     z ←  π[t]
   7.     cat timp z definit si not(za ϵ T) executa
   8.         z ← π[z]
   9.     daca za ϵ T atunci
   10.        π[x] ← za
   11.    altfel
   12.        π[x] ← radacina
   13.    daca za ϵ A atunci
   14.        adauga x la A
   15.returneaza π, A
```

Algoritmul prezentat aici este o imbinare intre Construieste-Automat si Calcul-Functie-Prefix. De altfel, functia π are aceiasi semnificatie ca in Calcul-Functie-Prefix, cel mai lung prefix care este si sufix pentru un prefix ales x. Asemanator se demonstreaza ca ciclul 4-14 are costul amortizat O(1), rezulta deci ca toata procedura are complexitatea O(M), daca neglijam timpul necesar investigarii apartenentei unui prefix la T.

Aplicatii in concursurile de informatica

In concursurile de informatica mai rar se vor intalni probleme care sa necesite numai implementarea unui algoritm de potrivire a sirurilor. In probleme acesti algoritmi sunt de cele mai multe ori partea intermediara care face trecerea unui set de intrare intr-o problema de greedy, programare dinamica, backtracking sau teoria grafurilor.

Cifru. Problema propusa de Nistor Mot la proba de baraj pentru lotul national largit in anul 2006.

Enuntul problemei:
Copiii solarieni se joaca adesea trimitandu-si mesaje codificate. Pentru codificare ei folosesc un cifru bazat pe o permutare p a literelor alfabetului solarian si un numar natural d. Alfabetul solarian contine m litere foarte complicate, asa ca noi le vom reprezenta prin numere de la 1 la m.
Dat fiind un mesaj in limbaj solarian, reprezentat de noi ca o succesiune de n numere cuprinse intre 1 si m, c_1 c_2 ... c_n, codificarea mesajului se realizeaza astfel: se inlocuieste fiecare litera c_i cu $p(c_i)$, apoi sirul obtinut $p(c_1)$ $p(c_2)$ ... $p(c_n)$ se roteste spre dreapta, facand o permutare circulara cu d pozitii rezultand sirul $p(c_{n-d+1})$, ... ,$p(c_{n-1})$, $p(c_n)$, $p(c_1)$, $p(c_2)$... $p(c_{n-d})$.
De exemplu, pentru mesajul 2 1 3 3 2 1, permutarea p=(3 1 2) (adica p(1)=3, p(2)=1, p(3)=2) si d=2. Aplicand permutarea p vom obtine sirul 1 3 2 2 1 3, apoi rotind spre dreapta sirul cu doua pozitii obtinem codificarea 1 3 1 3 2 2.
Date fiind un mesaj necodificat si codificarea sa, determinati cifrul folosit (permutarea p si numarul d).
Restrictii: 1 < n < 100001; 1 < m < 10000; mesajul contine fiecare numar natural din intervalul [1, m] cel putin o data.

Algoritmul naiv probeaza pentru fiecare deplasament, daca exista o permutare care transforma sirul initial in permutarea data. Acest algoritm are complexitatea $O(n^2)$.

Pentru a rezolva problema in mod eficient trebuie sa observam ca dupa transformarea sirului folosind o permutare si un deplasament alese la intamplare, distantele dintre caractere nu se modifica. Deci nici distantele intre caracterele de acelasi fel, de pe pozitii consecutive nu se modifica (p_1 si p_2 pozitii consecutive daca $c_{p1} = c_{p2}$ si nu exista p_3 astfel incat $c_{p3} = c_{p1} = c_{p2}$ si $p_1 < p_3 < p_2$). Daca transformam sirurile de la intrare astfel:

TS[i] ← i – L[S[i]], unde L[c] reprezinta ultima aparitie a lui c inainte de i; daca c apare pentru prima oara pe pozitia i atunci
TS[i] ← i + N – U[S[i]], unde U[c] reprezinta ultima aparitie a lui c de la sfarsitul sirului S

Astfel problema se transforma in gasirea unui deplasament pentru care transformarea sirului initial si rotit este egal cu transformarea sirului final. Solutia acestei probleme este mai simpla decat pare. Sa notat transformarea sirului initial

TI si transformarea sirului final TF. Vom dubla sirul TI astfel incat TI[i] = TI[i+n]. Acum mai trebuie sa aflam deplasamentul pentru care modelul TF se gaseste in TI. Acesta se poate afla folosind algoritmul Knuth-Morris-Pratt. Permutarea este de aici usor de determinat. Se roteste sirul initial cu deplasamentul gasit si se face corelarea cu sirul final.

Complexitatea finala a algoritmului este O(n). Putem sa-l neglijam pe m in ceea ce priveste complexitatea pentru ca n este tot timpul mai mare decat m, deoarece mesajul contine fiecare numar de la 1 la m.

Exemplu de rezolvare a problemei: “cifru.cpp”.

ADN. Problema propusa de Cosmin Negruseri pe site-ul infoarena.ro.

Enuntul problemei:

O problema importanta in ultimii ani in biologie a fost gasirea secventei de ADN pentru om. Un lant ADN este format din doua spirale de molecule, fiecare molecula fiind denumita pe scurt A, G, C sau T. Deci un lant ADN poate fi reprezentat ca un sir de caractere din multimea A, G, C si T. Poblema cercetatorilor este ca ei nu pot gasi intreaga secventa ADN cu metode chimice sau biologice dar pot determina sectiuni din ea. Dupa ce au determinat o serie de sectiuni, un sir de molecule care are probabilitatea cea mai mare sa fie apropiat de secventa reala este sirul cel mai scurt de caractere care contine toate sectiunile determinate ca subsecvente!

Scrieti un program care sa-i ajute pe cercetatori sa determine cel mai scurt sir de caractere care contine toate sectiunile determinate ca subsecvente.

Precizari: $0 < N < 19$; Lungimea unui sir este mai mica de 30001.

Primul pas in rezolvarea problemei este eliminarea sirurilor care sunt incluse in alte subsiruri. Acest pas poate fi rezolvat folosind algoritmul lui Knuth, Morris si Pratt pentru oricare doua siruri date, complexitatea fiind $O(N^2*L)$, unde L reprezinta lungimea maxima a sirurilor.

In continuare vom construi un graf orientat G(V, E), unde multimea V a nodurilor corespunde sirurilor ramase. Vom lua acest graf complet si vom asocia fiecarei muchii o functie cost, care este egala cu lungimea sirului fara cel mai lung prefix al sirului care este si sufix al sirului corespunzator nodului din care pleaca muchia.

w(u, v) = lungime[v] – max(lungime[P_k]), P_k prefix al sirului v si sufix al sirului v

Calcularea valorilor functiei de cost se poate face folosind algoritmul KMP, si deci are complexitatea $O(N^2*L)$.

Vom adauga la graful construit un nod suplimentar S. Pentru acest nod valoarea functiei catre alt nod v, este egala cu lungimea sirului v; muchia de la v la S fie nu exista, fie are valoarea infinit.

w(S, v) = lungime[v]

Construind acest graf am redus problema din enunt la o problema clasica din teoria grafurilor si anume gasirea unui tur Hamiltonian pornind din nodul S.

Din pacate aceasta problema este NP (non-polinomiala). Avand aceasta in vedere se poate implementa un algoritm de backtracking pentru a gasi turul, avand complexitatea O(N!). Aceasta complexitate este insa prea mare pentru dimensiunea maxima a grafului, asa ca ne vom orienta catre un algoritm de programare dinamica. Vom defini o stare sub forma: ultimul nod inserat in drum si starea celorlalte noduri, de forma au fost sau nu inserate in drum. Avem maxim N * 2N astfel de stari si trecerea dintr-o stare in alta se face in O(N).

In concluzie complexitatea finala a solutiei este $O(N^2*(L+2^N))$.

Exemplu de rezolvare a problemei: “adn.cpp”.

ABC. Problema propusa de Mugurel Ionut Andreica.

Aceasta problema a fost propusa cu mai multe ocazii in care scopul nu era competitional ci propunea dezbaterea metodelor de rezolvare: Selectia lotului UPB pentru concursul ACM-ICPC, pregatirea lotului national de informatica, infoarena.ro.

Enuntul problemei:

Alfabetul limbii Makako este compus din numai 3 simboluri: a, b si c. Orice cuvant al acestei limbi este un sir format dintr-un numar finit de simboluri din alfabet (la fel ca in cele mai multe din limbile folosite in prezent). Totusi, nu orice insiruire de simboluri formeaza un cuvant cu sens. Conform dictionarului limbii Makako, numai anumite siruri de simboluri reprezinta cuvinte cu sens (in continuare, prin cuvant vom intelege unul dintre aceste siruri de simboluri ce au sens). O particularitate a limbii Makako este ca oricare doua cuvinte au exact aceeasi lungime.

De curand s-a descoperit un text antic despre care se presupune ca ar fi scris intr-un dialect vechi al limbii Makako. Pentru a verifica aceasta ipoteza, oamenii de stiinta vor sa determine in ce pozitii din text se regasesc cuvinte din limba. Textul poate fi privit ca o insiruire de L simboluri din alfabetul limbii Makako, in care pozitiile simbolurilor sunt numerotate de la 1 la L. Daca un cuvant din limba se regaseste ca o insiruire continua de simboluri in cadrul textului, iar pozitia de inceput a acestuia este P, atunci P reprezinta o pozitie candidat. Oamenii de stiinta doresc sa determine numarul pozitiilor candidat din cadrul textului.

Sa presupunem ca dictionarul limbii Makako ar contine doar urmatoarele 3 cuvinte: bcc, aba si cba, iar textul antic descoperit ar fi cabababcba. La pozitiile 2 si 4 din text se regaseste cuvantul aba. La pozitia 7 se regaseste cuvantul cba. Cuvantul bcc nu se regaseste in text. Asadar, in text exista 3 pozitii candidat.

Precizari: Textul antic este alcatuit din cel mult 10 000 000 de caractere; Dictionarul limbii Makako va contine cel mult 50 000 de cuvinte; Cuvintele au o lungime de cel mult 20 de caractere.

Sa consideram numarul cuvintelor N, si lungimea maxima a cuvintelor M.

Folosind algoritmul naiv de potrivire a sirurilor complexitatea finala a solutiei va fi O(L*N*M).

Acesta complexitate poate fi imbunatatita, folosind algoritmul de potrivire al sirurilor Knuth-Morris-Pratt. In acest caz complexitatea scade la O(L*N), cum M << L.

Aceasta problema insa se potriveste ca o manusa automatelor finite de potrivire a sirurilor. Constructia automatului se face in O(N*M*|Σ|). In acest caz Σ = {a, b, c} si fiind atat de mic il putem considera constant, complexitatea fiind deci O(N*M). In continuare fiecare prefix al textului necesita timp constant pentru a fi analizat. Trecerea catre prefixul urmator se face cu ajutorul functiei δ, si verificarea apartenetei la starile de acceptare se poate face folosind un hash.

Complexitatea finala a algoritmului este O(N*M + L).

Exemplu de rezolvare a problemei: “abc.c”.

Strigat. Problema propusa de Adrian Diaconu pe site-ul infoarena.ro

Enuntul problemei:

La insistentele aghiotantului sau Arthur, Tick s-a decis sa renunte la strigatul sau de lupta "Lingura". Tick vrea ca strigatul sau sa fie cat mai inspaimantator pentru raufacatorii din oras. Tick stie ca exista M cuvinte care ii sperie pe raufacatori si care daca se vor afla in cadrul strigatului sau provoaca un anumit grad de spaima. Se cunoaste pentru fiecare cuvant gradul de spaima pe care il provoaca A_i. *Gradul total de spaima al strigatului va fi* $A_1*n_1 + A_2*n_2 + ... + A_M*n_M$, *unde* n_i *este numarul de aparitii al cunvatului i in cadrul strigatului. Atentie aparitiile cuvintelor se pot suprapune.*
Ajutati-l pe Tick sa isi gaseasca un strigat de lupta format din N caractere si care provoaca un grad maxim de spaima.
Precizari: $0 < N, M, lungime[cuvant] < 101; 0 < A_i < 1001$; *nu exista doua cuvinte identice.*

Vom construi un automat finit de potrivire a sirurilor care primeste ca modele cuvintele primite C_i. Vom nota in continuare cu L_i lungimea cuvantului C_i si L lungimea cuvantului cel mai lung.

$L_i \leftarrow lungime[C_i]$
$L \leftarrow max(L_i), i \leftarrow 1, N$

Construirea automatului sa presupunem ca are complexitatea O(M*L*|Σ|).

Tinand cont de limitele impuse de problema, automatul nu va depasi in dimensiune 10000 de noduri. Amintim ca un nod reprezinta un prefix al unui sau mai multe cuvinte. Tinand cont de aceste limite putem sa rezolvam problema folosind programarea dinamica. Informatiile care definesc o stare se refera la numarul de caractere folosite in construirea strigatului, cat si sufixul acestuia, care corespunde cu unul sau mai multe prefixe ale cuvintelor care provoaca spaima. Putem considera ca sirul vid este de asemeni un prefix, dar se poate demonstra usor ca acest caz nu este niciodata favorabil, si ca singura situatie in care are rost sa-l consideram este cand strigatul de lupta este la randul lui vid.

D[i][j] – valoarea maxima a spaimei pe care o poate transmite un strigat cu caractere si cu sufixul j (j ϵ Q; Q – multimea starilor unui automat)

Avand un sufixul j, putem sa adaugam la sfarsit un caracter oarecare, obtinand astfel un nou sufix. Spaima va creste cu spaima corespunzatoare noului prefix. O astfel de dinamica, in care starile viitore se construiesc din starile curente, se numeste dinamica in fata.

D[i+1][k] = max(D[i][j]) + suma(A_t), unde j, k ϵ Q, δ[j][c] = k, c ϵ Σ si C_t sufix al lui k

Valoarea strigatului maxim va fi max(D[N][j]), j ϵ Q. Mai ramane reconstituirea strigatului de valoare maxima. In acest sens pentru fiecare stare trebuie sa retinem si starea din care se obtine valoarea ei.

R[i+1][k] = j, unde D[i+1][k] = D[i][j] + suma(A_t).

Complexitatea finala a algoritmului va fi O(N*M*L*|Σ|).

Exemplu de rezolvare a problemei: “strigat.cpp”.

Dangerous Pattern. Problema propusa de Xin Tao la *Online Contest of Christopher's Adventure*

Traducerea enuntului:

Serviciul FBI a primit informatii despre un nou atac terorist. Teroristii tin legatura prin reviste si ziare intr-un mod criptografic. O sa spunem ca un text contine un tipar periculoas daca S[1], S[2], ..., S[K] (S[i], i=1, K, este dat) apar in ordinea specificata si nu se suprapun.
De exemplu, K = 2, S[1] = 'aa', S[2] = 'ab', atunci textul 'aaab' contine un tipar periculos. 'aab' nu contine pentru ca secventele 'aa' [1, 2] si 'ab' [2, 3] se suprapun. Nici 'abaa' nu contine un tipar periculor pentru ca aparitiile nu sunt in ordinea specificata.
Cerinta este sa numeri numarul de tipare periculoase diferite dintr-un text. Spunem ca doua tipare sunt diferite daca exista cel putin S[i], astfel incat pozitiile pe care apare in cele doua tipare difera.
De exemplu, K = 2, S[1] = 'a', S[2] = 'b', textul 'aabb' contine 4 tipare periculoase: [1, 3], [2, 3], [1, 4], [2, 4].
Pentru ca rezultatul poate fi prea mare, se cere afisezi numai modulul impartirii la 28851.
Restrictii: Lungimile totale pentru S[i] nu depasesc 10000. Nu exista i si j, astfel incat S[i] si S[j] sa aibe acelasi sufix. Lungimea totala a textului nu depaseste 500000. In S[i] si in text apar numai caracterele mici ale alfabetului latin.

Sa notam cu M suma lungimilor sirurilor S[i] si cu N lungimea textului.

Primul pas in rezolvarea problemei consta in construirea unui automat finit cu modelele S[i]. Complexitatea acestui pas O(M*|Σ|), unde Σ={a, b, ..., z}.

Pentru a determina numarul de tipare periculoase vom folosi programarea dinamica. Vom defini o stare de forma:

D[i][j] – numarul de tipare periculoase gasite in primele i caractere ale textului, si folosind primele j modele.

D[i][j] = D[i-1][j], daca nu modelul j nu se termina pe pozitia i

D[i][j] = D[i-1][j] + D[(i-lungime[S[j]])][j-1], daca modelul j se termina pe pozitia i.

Problema este ca numarul de stari este foarte mare, numarul de siruri fiind de ordinul miilor in cazul cel mai defavorabil. Am putea sa tinem starile pentru ultimele M pozitii, dar memoria folosita ar fi $O(M^2)$ si complexitatea ar fi tot O(N*M).

Sa interpretam constrangerea conform careia nu exista doua modele cu acelasi sufix. Rezulta ca pentru un sufix dat exista cel mult un model care se poate potrivi. Deci pentru un prefix oarecare al textului dat, exista cel mult un model care poate fi sufix al sau.

Astfel numai o singura valoare a dinamicii se poate schimba pentru pozitia curenta i, fata de pozitia anterioara i-1. Vom pastra numai starile pentru pozitia curenta si vom transforma dinamica “inapoi” in dinamica “in fata”. Daca pentru pozitia curenta exista un model j care se termina atunci stim ca D[i+lungime[S[j+1]][j] va trebui crescut cu D[i][j-1]. In acest sens vom utiliza o structura auxiliara care tine minte ca D[i][j] trebuie crescut cu valoarea v. O astfel de structura poate fi un arbore echilibrat de cautare (caci inserarea nu se face neapar la sfarsit). Insa cel mai eficient din punctul de vedere al timpului si usurintei in implementare este un vector de liste “role-back”. Numele vine de la faptul ca ajuns la capatul lui, indicii pornesc de la inceput, asemanator unei liste circulare. Astfel V[i] va fi de fapt V[j], unde j este echivalentul lui i in clasa de resturi a dimensiunii vectorului V.

Complexitatea finala a algoritmului este O(M*|Σ| + N).

Exemplu de rezolvare a problemei: “pattern.cpp”.

Prezentarea suportului digital

In radacina CD-ului se afla fisierul atestat.pdf, care contine documentul in format electronic.

Pe langa acest fisier se gasesc 5 directoare, corespunzatoare fiecarei probleme prezentate: cifru, adn, abc, strigat si pattern.

Fiecare director contine 2 fisiere: sursa cu extensia cpp si textul problemei in format pdf. De asemeni va mai contine 2 directoare, input si output, reprezentant fisierele folosite pentru testarea corectitudinii solutiilor.

Bibliografie

[1] Introducere in Algoritmi. Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest

[2] Efficient Algorithms on Texts. M. Crochemore & W. Rytter

[3] http://infoarena.ro

[4] http://zhuzeyuan.hp.infoseek.co.jp/