Tehnica Greedy

Prezentare

Un algoritm *greedy* va funcționa asemănător modului de gândire al unui om lacom: va consuma elemente dintr-o anumită secvență, de fiecare dată luând acel element, ce se potrivește cel mai bine cu un anumit criteriu, fără a privi în perspectivă. Deși prima impresie ar fi că această abordare este una greșită, datorită conotației cuvântului lacom, totuși uneori algoritmii de acest tip pot duce la soluții simple și eficiente.

Un algoritm *greedy* este potrivit atunci când trebuie să luăm o serie de decizii, și anume pe cea mai convenientă la un moment dat. Această alegere este un optim local, iar speranța este ca, într-un final să fie obținută soluția globală, însă nu se întâmplă așa întotdeauna. Pentru un algoritm dat, va trebui să determinăm dacă soluția este sau nu optimă.

În cele mai multe situații avem:

- O mulțime de candidați (valori de ales, noduri in graf etc.)
- O funcție care verifică dacă o mulțime de candidați constituie o soluție posibilă, nu neapărat optimă, a problemei
- O funcție care verifică dacă o mulțime de candidați este fezabilă, adică dacă este posibil să completăm această mulțime astfel încât să obținem o soluție posibilă (nu neapărat optimă).
- O funcție de selecție ce indică într-un anumit moment care este cel mai potrivit dintre candidații nefolosiți.
- O funcție obiectiv care dă valoarea unei soluții (timpul necesar executării tuturor lucrărilor într-o anumită ordine), lungimea drumului pe care l-am găsit etc).

Iată un exemplu simplu ce ilustrează tehnica greedy. Atunci când facem cumpărături, și efectuăm plata, nu dorim să primim rest foarte mărunt. Soluția pentru a găsi cea mai bună soluție la plata restului (în monede) permite reformularea problemei, conform celor afirmate mai sus. Iată elementele problemei:

- Candidații: mulțimea inițială de monede de 1, 5, și 25 de unități și presupunem că avem o cantitate nelimitată din fiecare categorie
- Funcția ce verifică soluția posibilă: valoarea totală a unei astfel de mulțimi să fie exact valoarea ce trebuie returnată ca rest.
- O mulțime fezabilă: valoarea totală a unei astfel de mulțimi nu este mai mare decât suma de returnat ca rest.
- Funcția de selecție: se alege cea mai mare monedă din mulțimea de candidați. Motivul este acela că dorim să plătim folosind cât mai puține monede.
- Funcția obiectiv: numărul de monede folosite să fie cât mai mic.

Evident soluția optimă se poate găsi încercând toate combinările posibile de monede, și abordarea se mai numește brute *force*. Acest lucru necesită foarte mult timp. Din cele deduse mai sus rezultă următorul algoritm formulat într-un limbaj descriptiv:

1. $\boldsymbol{while}$ ($mai\ sunt\ monezi\ şi\ nu\ s-a\ găsit\ suma$) $\boldsymbol{do}$
2. $ia\ cea\ mai\ mare\ monedă\ din\ cele\ rămase$
3. $\boldsymbol{if}$ ($adăugarea\ monezii\ face\ ca\ suma\ să\ fie\ mai\ mare\ decât\ restul$) $\boldsymbol{then}$
4. $respinge\ moneda$
5. $\boldsymbol{else}$
6. $adaugă\ moneda\ la\ rest$
7. $\boldsymbol{if}$ ($restul\ este\ egal\ cu\ suma\ de\ returnat$) $\boldsymbol{then}$
8. $problema\ este\ rezolvată, s-a\ găsit\ suma$

Iată un mic exemplu pentru a demonstra acest algoritm. Se dă suma de returnat *36*. Se va prelua o monedă de 25 de unități, se verifică să fie mai mică sau egală cu 36 și este, ceea ce duce la adăugarea ei în soluția provizorie. Apoi se încearcă adăugarea unei monede de 25, ceea ce va depăși suma de 36 deci moneda va fi respinsă. Apoi se va încerca adăugarea unei monede de 5, suma devine 30, apoi încă o monedă de 5, suma devine 35 și încercarea adăugării încă unei monede de 5 eșuează. Apoi se încearcă adăugarea unei monede de 1 și suma este atinsă. Algoritmul se încheie.

Pe de altă parte dacă am introduce monezi de 18 sau de 12 algoritmul va găsi un număr mai mare decât 2 monede de 18 sau 3 de 12, ceea ce dovedește că atingerea numărului optim de monede nu este îndeplinit în toate cazurile, deci algoritmul nu funcționează pentru orice caz.

Pornind de la problema de mai sus, și mai exact de la pașii folosiți pentru a rezolva algoritmul, putem generaliza rezolvarea problemei astfel:

$FUNCTION\ GREEDY(C)$

1. $\triangleright\ C\ este\ mulțimea\ candidaților$
2. $S \leftarrow \emptyset \quad unde\ S\ este\ mulțimea\ în\ care\ reținem\ soluția$
3. $\boldsymbol{while\ not}\ solutie(S)\boldsymbol{and}\ C \neq \emptyset\ \boldsymbol{do}$
4. $\quad x \leftarrow un\ element\ din\ C$ care minimalizează $select(x)$
5. $\quad C \leftarrow C \backslash \{x\}$
6. $\quad \boldsymbol{if}\ fezabil(S \cup \{x\})\ \boldsymbol{then}\ S \leftarrow S \cup \{x\}$
7. $\boldsymbol{if}\ soluție(S)\ \boldsymbol{then\ return}\ S$
8. $\boldsymbol{else\ return}$ "$nu\ există\ soluție$"

Este de înțeles acum de ce un astfel de algoritm se numește lacom: la fiecare pas, procedura alege cel mai bun candidat la momentul respectiv, fără să îi pese de viitor și fără a se răzgândi (a reveni asupra deciziei). Dacă un candidat este inclus în soluție el va rămâne acolo. Dacă în schimb, acel candidat este exclus, el nu va mai putea niciodată fi reconsiderat.

Asemenea unui întreprinzător rudimentar, care urmărește câștigul imediat, în dauna celui din perspectivă, un algoritm greedy acționează simplist. Totuși, ca și în afaceri, o astfel de metodă se dovedește utilă, tocmai datorită simplității ei.

Funcția $select$ este de obicei derivată din funcția obiectiv, uneori aceste funcții sunt identice.

Minimizarea timpului mediu de așteptare

Să presupunem că un frizer are mai mulți clienți pentru diverse tratamente (tuns simplu, tuns cu șampon, permanent, vopsit). Tratamentele nu vor dura la fel, dar stilistul știe cât va lua fiecare. Un scop al problemei, este ca frizerul să programeze clienții astfel ca să minimizeze timpul de așteptare în salon (atât cât stau cât și când sunt serviți). Timpul total petrecut în salon se numește *timp în sistem*. Problema minimizării timpului de așteptare are multe aplicații: putem permite accesul la disc în funcție de utilizator. Astfel utilizatorii vor aștepta cât mai puțin pentru a citi un fișier.

O altă problemă apare când un client va avea nevoi de același timp pentru a completa o sarcină dar are un anumit *deadline* ceea ce înseamnă uneori ca trebuie să înceapă mai devreme, sau să fie servit la timp. Scopul este evident, de a minimiza timpul pentru a maximiza profitul. Vom discuta și acest aspect.

Să presupunem că există trei sarcini care trebuie îndeplinite și timpii pentru le a îndeplini sunt:

$$t_1 = 5, \quad t_2 = 10 \ \textit{și} \ \ t_3 = 4$$

Unitățile de timp nu sunt relevante. Dacă ordonăm cronologic cele trei sarcini obținem următoarea secvențe de acțiuni:

Sarcină	Timp în sistem
1	5(timp de servire)
2	5(timp așteptare după 1) + 10(timp de servire)
3	5(timp așteptare după 1) + 10(timp de așteptare după 2) +4(timp de servire)

Timpul de total de așteptare este: $5 + (5 + 10) + \ (5 + 10 + 4) = 39$

Aceeași metodă de calcul va duce la următoarele posibilități de a servi cele trei cereri:

Mod de a servi	Timpul total in sistem
[1, 2, 3]	5+(5+10)+(5+10+4) = 39
[1, 3, 2]	5+(5+4)+(5+4+10) = 33
[2, 1, 3]	10+(10+5)+(10+5+4) = 44
[2, 3, 1]	10+(10+4)+(10+4+5) = 43
[3, 1, 2]	4+(4+5)+(4+5+10) = 32
[3, 2, 1]	4+(4+10)+(4+10+5) = 37

Se observă că cel mai bun mod de a servi cererile este **[3, 1, 2]** cu un timp total de *32*.

Există clar un algoritm care poate verifica toate permutările posibile. Ordinul de timp al acestui algoritm este însă factorial. Această soluție, numită și brute force poate fi înlocuită cu un algoritm de tip greedy mult mai simplist și mai eficient. Se observă că soluția cea mai bună conține timpii sortați crescător. Intuitiv, un algoritm pe care îl dezvoltăm va încerca să servească sarcina care durează cel mai puțin:

$PROCEDURE\ OPTIMAl - TIME$

1. $sortează\ sarcinile\ după\ timp\ în\ ordine\ crescătoare$
2. $\boldsymbol{while}\ (instanța\ nu\ este\ rezolvată)\ \boldsymbol{do}$
3. $\quad programează\ următoarea\ sarcină$
4. $\quad \boldsymbol{if}\ (nu\ mai\ există\ sarcini)\ \boldsymbol{then}$
5. $\quad\quad instanța\ este\ rezolvată$

Evident că algoritmul poate fi adaptat după sensul problemei, și în cazul de față putem înlocui pasul 3 cu adăugarea timpului sarcinii următoare la timpul total.

Complexitatea acestui algoritm este : $O(n \log n)$.

Teorema 4.1

Singurul mod de a aranja astfel ca timpul total să fie minim, este acela în care sarcinile sunt ordonate crescător după timpul de servire.

Demonstrație

Pentru $1 \leq i \leq n-1$ fie t_i timpul de servire pentru a *i*-lea sarcină programată în așa fel încât timpul total este minim. Va trebui să arătă, că este necesar ca sarcinile să fie ordonate după timpul lor de execuție. Vom demonstra acest lucru prin reducere la absurd. Dacă nu sunt ordonate după timpul lor de execuție, atunci există una și anume i unde $1 \leq i \leq n-1$, pentru care

$$t_i > t_{i+1}$$

Putem rearanja ordinea inițială, interschimbând aceste sarcini și anume i cu $i+1$. Realizând aceasta, am luat t_i unități de timp din sarcina $i+1$, care anterior erau calculate pentru timpul acesteia. Motivul pentru care scădem acest t_i este că acum $i+1$ nu mai așteaptă după i. Evident la sarcina i adunăm t_{i+1} pentru același motiv. Totuși timpul total din sistem nu se modifică. Dacă T este timpul total din sistem, atunci T' este timpul total după rearanjarea de mai sus. Atunci

$T' = T + t_{i+1} - t_i$.

Deoarece $t_i > t_{i+1}$ putem spune că $T' < T$ ceea ce contrazice afirmația că T este timpul optim.

Se poate deduce cum generalizăm algoritmul pentru ca acesta să trateze o problemă de programare multi-server. Fie *m* servere. Să se ordoneze aceste servere arbitrar. Să se ordoneze sarcinile după timp, în ordine crescătoare. Fie ca primul server să deservească prima sarcină, cu cel mai mic timp, și al *m*-lea server să deservească a *m*-a sarcină. Primul server va termina la un moment dat, primul, deoarece are si cel mai mic timp de executat. Apoi primul server va deservi a *(m+1)* sarcină. Apoi al doilea server va efectua a *(m+2)* sarcină. Schema de servire are loc după cum urmează:

$$Server\ 1\ deserverşte\ sarcina\ 1, (1+m), (1+2m), \dots$$

$$Server\ 2\ deserverşte\ sarcina\ 2, (2+m), (2+2m), \dots$$

…

$$Server\ i\ deserverşte\ sarcina\ i, (i+m), (i+2m), \dots$$

…

$$Server\ m\ deserverşte\ sarcina\ m, (m+m), (m+2m), \dots$$

Rezultă ordinea firească a servirii:

$$1, 2, \dots, m, (1+m), (2+m), \dots (m+m), (1+2m), \dots$$

Programarea servirii folosind termeni limită

În această problemă de programare, fiecare sarcină va avea nevoie de o unitate de timp pentru a termina și are un termen limită și un profit. Adică, dacă operația începe mai devreme de termenul limită, se obține un profit. Scopul este de a programa operația ca profitul să fie maxim. Nu toate sarcinile trebuie să fie executate. Iată un exemplu pentru a ilustra mai bine problema:

Fie următoarele sarcini, timpi limită, profituri:

Sarcină	Timp limită	Profit
1	2	30
2	1	35
3	2	25
4	1	40

Atunci când spunem că o sarcină 1 are un termen limită de 2, înseamnă că sarcina 1 poate începe la timpul 1 sau timpul 2. Nu există timp 0. Deoarece sarcina 2 are timpul limită 1, această sarcină poate porni doar la timpul 1. Iată posibilele profituri.

Programare	Profit total
[1, 3]	30+25 = 55
[2, 1]	35+30 = 65
[2, 3]	35+25 = 60
[3, 1]	25+30 = 55
[4, 1]	40+30 = 70
[4, 3]	40+25 = 65

Alte programări imposibile (de exemplu [1,2]), nu au mai fost listate. Pe de altă parte, programări ca [1,3] sunt posibile, deoarece sarcina 1 a pornit înaintea termenului limită, iar sarcina 3, a pornit exact la termenul limită. Se poate vedea că programarea [1,4] este optimă și are cel mai mare profit. Se poate modifica algoritmul de mai sus pentru a obține un algoritm mai bun decât cel de ordin factorial pentru a rezolva această problemă.

Interclasarea optimă a șirurilor ordonate

Să presupunem că avem două șiruri S_1, S_2 ordonate crescător și că dorim să obținem prin interclasarea lor, șirul ordonat crescător ce conține elementele din ambele șiruri. Dacă interclasarea are loc prin deplasarea elementelor din cele două șiruri, atunci evident numărul deplasărilor este $\#S_1 + \#S_2$.

Dacă generalizăm, considerăm n șiruri $S_1, S_2, \dots, S_n$ unde fiecare șir $S_i, 1 \leq i \leq n$ fiind format din q_i elemente ordonate crescător (vom denumi q_i $lungimea$ lui S_i). Ne propunem să obținem șirul S, ordonat crescător, unde S conține toate elementele din cele n șiruri. Vom realiza aceasta prin interclasări succesive de câte două șiruri. Problema constă în determinarea ordinii optime în care trebuie efectuare aceste interclasări, astfel ca numărul total de deplasări să fie minim. Exemplul de mai jos ne arată că problema nu este una banală.

Fie șirurile $S_1, S_2, S3$ de lungimi $q_1 = 30, q_2 = 20, q_3 = 10$. Dacă interclasăm pe S_1 cu S_2, iar rezultatul îl interclasăm cu S_3, numărul total de deplasări este $(30 + 20) + (50 + 10) = 110$.
Dacă interclasăm pe S_1 cu S_2, iar rezultatul în interclasăm cu S_3, numărul total de deplasări este $(10 + 20) + (30 + 30) = 90$. Dacă interclasăm pe S_1 cu S_3, iar rezultatul în interclasăm cu S_2, atunci numărul total de deplasări va fi $(10 + 30) + (40 + 20) = 100$.

După cum se poate vedea numărul minim de deplasări este 90.

Atașăm fiecărei strategii de interclasare un arbore binar în care valoarea fiecărui vârf este dată de lungimea șirului pe care îl reprezintă.

De exemplu, dacă șirurile $S_1, S_2, \dots, S_6$ au lungimile $q_1 = 30, q_2 = 10, q_3 = 20, q_4 = 30, q_1 = 50, q_6 = 10$, două dintre strategiile de interclasare sunt reprezentați prin arborii din figura de mai jos:

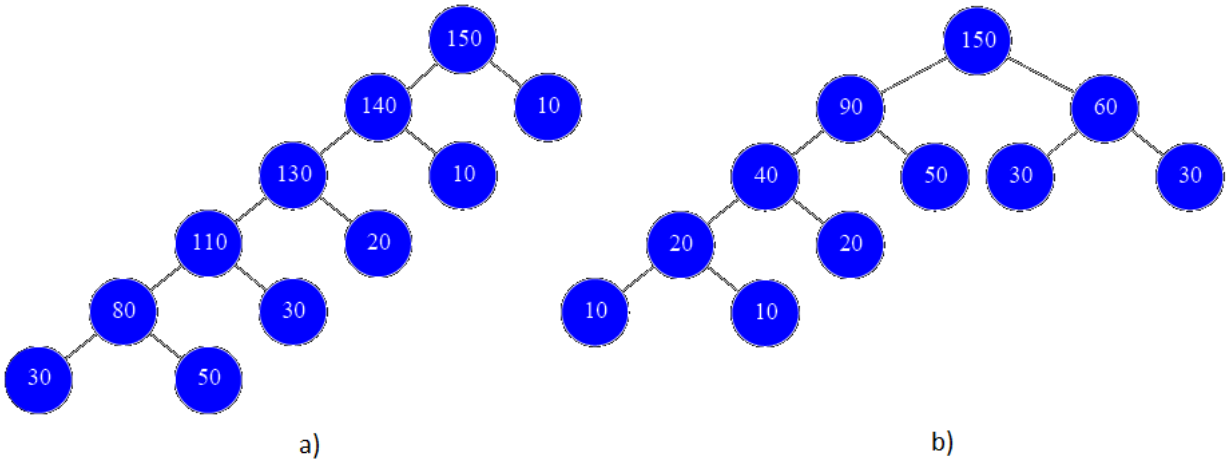


Figura 1. Reprezentarea strategiilor de interclasare

Observăm că fiecare arbore are 6 vârfuri terminale, corespunzând celor 6 șiruri inițiale și 5 vârfuri neterminale, adică cele 5 interclasări care definesc strategia. Numerotăm vârfurile astfel: vârful terminal i, $1 \leq i \leq 6$ va corespunde șirului S_i, iar vârfurile neterminale se numerotează de la 7 la 11 în ordinea obținerii lor, ca în figura 2.

Strategia greedy apare în figura 1b, și constă în a interclasa mereu cele mai scurte șiruri disponibile la momentul curent.

Interclasând șirurile $S_1, S_2, \ldots, S_n$ de lungimile $q_1, q_2, \ldots, q_n$, obținem pentru fiecare strategie câte un arbore binar cu n vârfuri terminale, numerotate de la 1 la n, și $n-1$ vârfuri neterminale, numerotate de la $n+1$ la $2n-1$. Definim pentru un arbore oarecare *A* de acest tip, *lungimea externă ponderată* ca fiind

$$L(A) = \sum_{i=1}^{n} a_i q_i$$

unde, a_i este adâncimea vârfului i. Se observă că numărul total de deplasări de elemente pentru strategia corespunzătoare lui *A* este chiar $L(A)$. Soluția optimă este deci arborele pentru care lungimea ponderată este minimă.

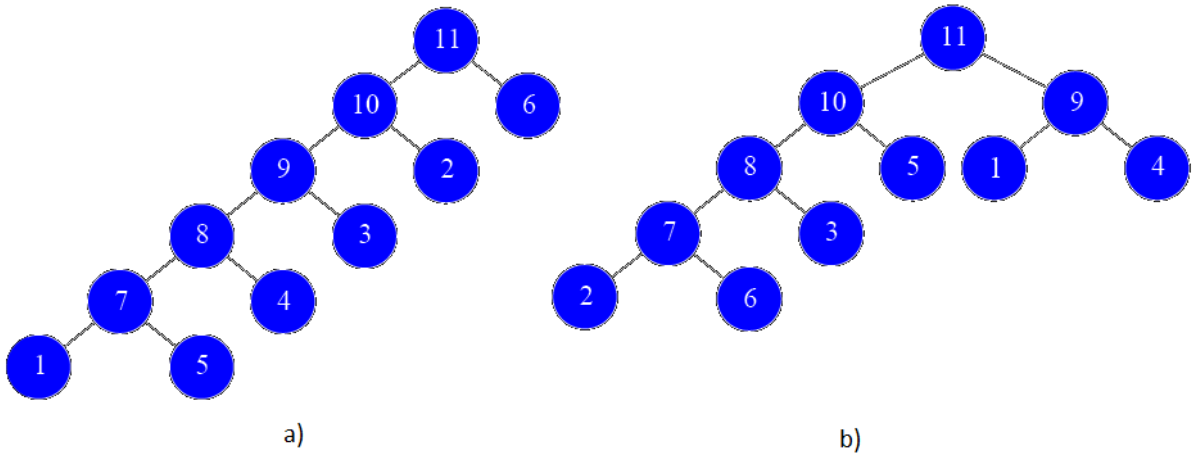


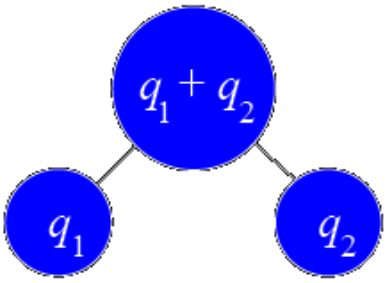
Figura 2. Numerotarea vârfurilor arborilor din figura 1.

Proprietatea 1

Prin metoda greedy se obține întotdeauna interclasarea optimă a n șiruri ordonate.

Demonstrație

Demonstrăm prin inducție. Pentru $n = 1$, proprietatea este verificată. Presupunem că proprietatea este adevărată pentru $n - 1$ șiruri. Fie A arborele strategiei greedy de interclasare a n șiruri de lungime $q_1 \leq q_2 \leq \cdots q_n$. Fie B un arbore cu lungimea externă ponderată minimă, corespunzător unei strategii optime de interclasarea a celor n șiruri. În subarborele A apare arborele



reprezentând prima interclasare făcută conform strategiei greedy. În arborele B, fie un vârf neterminal, de adâncime maximă. Ceio doi fii ai acestui nod, sunt q_j, q_k. Fie B' arborele obținut din B interschimbând vârfurile q_1 cu q_j respectiv, q_2 cu q_k. Evident că $L(B') \leq L(B)$. Deoarece B are lungimea externă ponderată minimă, rezultă că $L(B') = L(B)$. Eliminând din B' vârfurile q_1, q_2, obținem un arbore B'' cu $n - 1$ vârfuri terminale $q_1 + q_2, q_3, \dots, q_n$. Arborele B' are lungimea externă ponderată minimă și $L(B') = L(B'') + (q_1 + q_2)$. Rezultă că și B'' are lungimea externă ponderată

minimă. Atunci conform ipotezei, $L(B'') = L(A')$ unde A' este arborele strategiei greedy de interclasare a șirurilor de lungime $q_1 + q_2, q_3, \dots, q_n$. Cum A se obține din A' atașând nodului $q_1 + q_2$, fii q_1 și q_2, iar B' se obține în același mod din B'', rezultă că $L(A) = L(B') = L(B)$. Rezultă că proprietatea este adevărată pentru orice n.

La scrierea algoritmului care generează arborele strategiei greedy de interclasare, vom folosi un min-heap. Fiecare element al min-heap-ului este o pereche (q, i) unde i este numărul unui vârf din arborele strategiei de interclasare, iar q este lungimea șirului. Proprietatea de min-heap se referă la q. Algoritmul *interopt* va construi arborele strategiei greedy. Un vârf i al arborelui va fi memorat în trei locații diferite conținând:

$LU[i] =$ lungimea șirului reprezentat de vârf

$ST[i] =$ lungimea șirului corespondent fiului stâng

$DR[i] =$ lungimea șirului corespondent fiului drept

$PROCEDURE\ INTEROPT(Q[1..n])$

1. $\triangleright\ construieste\ arborele\ strategiei\ greedy\ de\ interclasare$
2. $\triangleright\ a\ sirurilor\ de\ lungimi\ Q[i] = q_i, 1 \le i \le n$
3. $H \leftarrow min - heap\ vid$
4. $\boldsymbol{for}\ i \leftarrow 1\ \boldsymbol{to}\ n\ \boldsymbol{do}$
5. $\quad (Q[i], i) \Rightarrow H \quad \triangleright\ insereaz\check{a}\ in\ min - heap$
6. $\quad LU[i] \leftarrow Q[i]$
7. $\quad ST[i] \leftarrow 0$
8. $\quad DR[i] \leftarrow 0$
9. $\boldsymbol{for}\ i \leftarrow n + 1\ \boldsymbol{to}\ 2n - 1\ \boldsymbol{do}$
10. $\quad (s, j) \Leftarrow H \quad \triangleright\ extrage\ r\check{a}d\check{a}cina\ lui\ H$
11. $\quad (r, k) \Leftarrow H \quad \triangleright\ extrage\ r\check{a}d\check{a}cina\ lui\ H$
12. $\quad ST[i] \leftarrow j$
13. $\quad DR[i] \leftarrow k$
14. $\quad LU[i] \leftarrow s + r$

Timpul total pentru acest algoritm este de $O(n \log n)$

Algoritmul lui Huffman

O altă aplicație a strategiei greedy și a arborilor binari cu lungime externă ponderată minimă este obținerea unei codificări cât mai compacte a unui text.

Un principiu general de codificarea a unui șir de caractere este următorul: se măsoară frecvența de apariție a diferitelor caractere, și cele mai lungi coduri, celor mai puțin frecvente caractere, și cele mai lungi coduri, celor mai puțin frecvente caractere. Acest principiu stă, de exemplu, la baza codului Morse.

Pentru situația în care codificarea este binară, există o metodă elegantă pentru a obține codul respectiv. Această metodă, descoperită a fost descoperită de David Albert Huffman în 1953 și folosește o strategie greedy. Această strategie este folosită cu succes în compresii de date, pentru a codificarea datelor din fișiere de exemplu. Pentru a prezenta algoritmul vom avea nevoie de niște noțiuni preliminare.

Un mod convenient de a reprezenta un fișier este cel în care folosim cod binar. În astfel de cod, fiecare caracter este reprezentat de un șir binar unic, numit cuvânt codificat. Un cod binar de lungime fixă, poate reprezenta un caracter folosind un număr de biți. De exemplu iată mulțimea de caractere $\{a,b,c\}$ Putem folosi 2 biți pentru a codifica fiecare caracter astfel:

a:00 b:01 c:11

Dacă avem un fișier ce conține secvența

$$abababcbbbc$$

codificarea acestuia este:

000100011101010111

Putem obține o codificare mult mai eficientă folosind un cod binar de lungime variabilă. Un astfel de cod poate reprezenta diferite caractere folosind diferite numere de biți. în exemplul de față, putem nota unul din caractere ca 0, din moment ce b apare cel mai frecvent, ar fi suficient să codificăm acest caracter folosind 0. Totodată, a nu mai poate fi codificat ca 00, deoarece nu mai putem distinge între un a și doi de b succesivi. Dacă am codifica pe a cu 01, apare confuzia interpretării: primul 0 este un început de a sau este un b. Așa că o metodă de a codifica este:

a:10 b:0 c:11

Dat fiind această codificare, șirul de mai sus poate fi reprezentat ca:

1001001100011

Observăm că această codificare va avea nevoie de 13 biți pentru a reprezenta șirul, în comparație cu cea de mai sus care are nevoie de 18.

Coduri prefixate

Un tip aparte de coduri variabile ca lungime este codul prefixat. Într-u cod prefixat, nici un cuvânt codificat pentru un caracter are voie să constituie începutul unui cuvânt codificat al altui caracter. De exemplu, dacă 01 este cuvântul codificat pentru a atunci 011 nu are voi să fie cuvânt codificat pentru b. Codul de 13 biți de mai sus este un astfel de exemplu. Fiecare cod prefixat poate fi reprezentat ca un arbore binar a căror frunze sunt caracterele ce trebuie codificate. Arborele binar corespunzător codului de mai sus este:

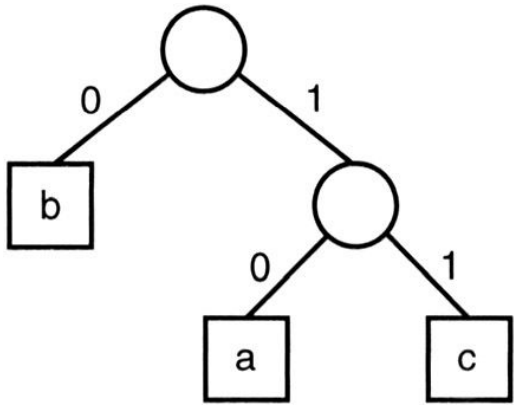


Figura 3. Arborele binar corespunzător codului a:10 b:0 c:11

Iată mai jos un alt exemplu:

Fie mulțimea de caractere $\{a,b,c,d,e,f\}$ și fiecare caracter apare în fișier de un număr de ori indicat în tabelul de mai jos. Acest tabel prezintă diferența între diversele coduri folosite.

Caracter	Frecvență	C1(Lungime fixă)	C2 (o codificare)	C3(Huffman)
a	16	000	10	00
b	5	001	11110	1110
c	12	010	1110	110
d	17	011	110	01
e	10	100	11111	1111
f	25	101	0	10

Se poate calcula numărul de biți folosiți pentru a codifica acest fișier:

$$Biți(C1) = 16(3) + 5(3) + 12(3) + 17(3) + 10(3) + 25(3) = 255$$

$$Biți(C2) = 16(2) + 5(5) + 12(4) + 17(3) + 10(5) + 25(1) = 231$$

$$Biți(C3) = 16(2) + 5(4) + 12(3) + 17(2) + 10(4) + 25(2) = 212$$

Se poate observa că $C2$ aduce o îmbunătățire, dar Huffman este mai bun decât acesta. În figura 4 mai jos, avem reprezentările arborilor ce conțin codificările $C2$ și $C3$.

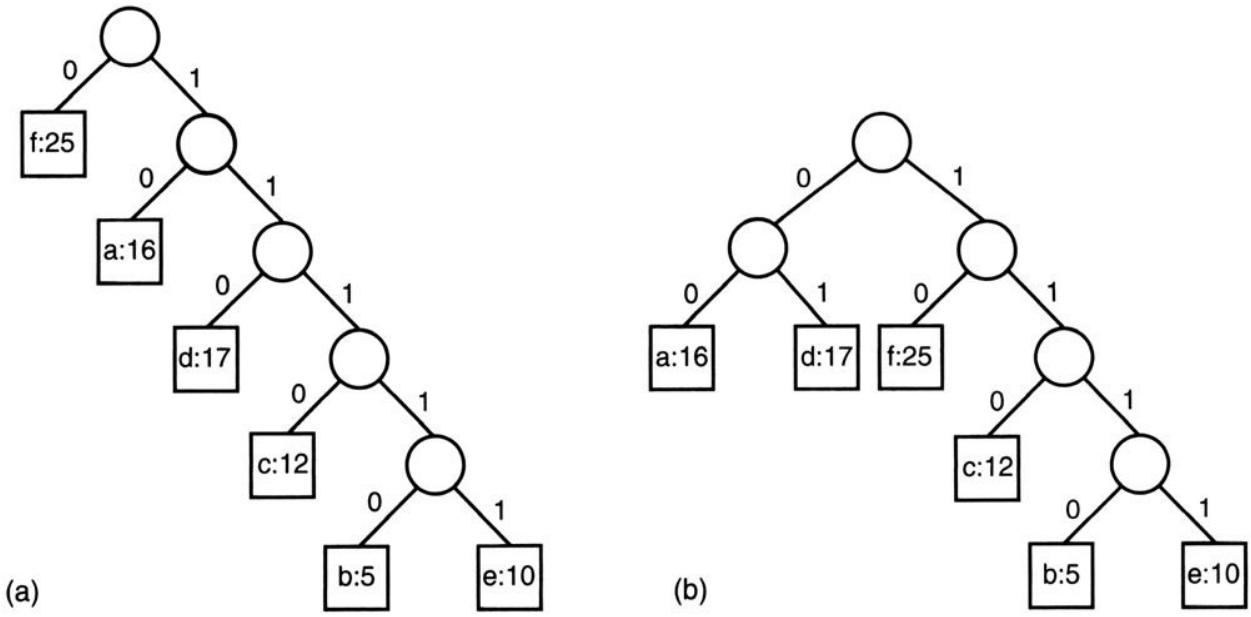


Figura 4. Reprezentarea codificărilor $C2\ a)$ și $C3–Huffman\ b)$

De aici se poate deduce următoarea formulă de calcul a numărului de biți necesari reprezentării acestor caractere, dat fiind un arbore binar A ca cele de mai sus:

$$biți(A) = \sum_{i=1}^{n} frecvența(v_i) \cdot adâncimea(v_i)$$

Unde v_i sunt caracterele din fișier.

Algoritmul de codificare Huffman

Algoritmul construiește un arbore A corespunzător codului optimal. Va începe de la un set de $|C|$ noduri și va efectua $|C|-1$ operații de compunere pentru a crea arborele final.

În pseudocodul de mai jos, vom presupune că C este o mulțime de n caractere și că fiecare caracter $c \in C$ este un obiect cu o frecvență de $f|c|$.

O coadă de priorități, Q, construită pe f, este folosită pentru a identifica cele două mai puțin frecvente obiecte pentru a le compune. Rezultatul compunerii celor două obiecte este un nou obiect a cărei frecvență este suma frecvențelor celor 2 obiecte. Q este implementată ca un arbore heap.

Iată mai jos algoritmul sub forma pseudocod.

$HUFFMAN(C)$

1. $n \leftarrow |C|$
2. $Q \leftarrow C$
3. $\boldsymbol{for}\ i \leftarrow 1\ \boldsymbol{to}\ n-1\ \boldsymbol{do}$
4. $\quad z \leftarrow ALLOCATE-NODE()$
5. $\quad x \leftarrow left[z] \leftarrow EXTRACT-MIN(Q)$
6. $\quad y \leftarrow right[z] \leftarrow EXTRACT-MIN(Q)$
7. $\quad f[z] \leftarrow f[x]+f[y]$
8. $\quad INSERT(Q,z)$
9. $\boldsymbol{return}\ EXTRACT-MIN(Q)$

Fie exemplul următor:

Caractere *a b c d e f*

Frecvențe *45 13 12 16 9 5*

Pentru exemplul nostru algoritmul va funcționa astfel. Din moment ce sunt 6 litere, mărimea inițială a cozii și a mulțimii va fi n=6, vor fi necesari 5 pași de combinare a frecvențelor. Cuvântul codificat pentru o literă este secvența de etichete de pe muchii parcursă de la rădăcină la literă.

Linia 2 inițializează coada de priorități Q cu literele din C. Bucla for din liniile 3-8 va repeta extragerea ultimelor noduri din heap-ul Q, ultimele având cele mai mici frecvențe, și va insera un nou nod (z) ce are ca valoare suma frecvențelor lor. Nodul z are pe x drept copil stânga iar pe y drept copil dreapta. După $n-1$ operații de acest fel, obținem un singur nod în Q, rădăcina arborelui ce conține codul, iar acest nod este returnat la linia 9.

Analiza ordinului de timp al algoritmului lui Huffman, presupune că Q este implementată ca un arbore binar. Pentru un set C de n caractere, inițializarea din linia 2 poate fi realizată în timp $O(n)$ folosind metoda MAKE-HEAP din cursul 3. Bucla for poate fi executată de n–1 ori, dar o operație pe heap necesită un timp $O(\log(n))$, deci bucla va produce un timp $O(n\log(n))$. De aceea acesta va fi timpul total al algoritmului Huffman.

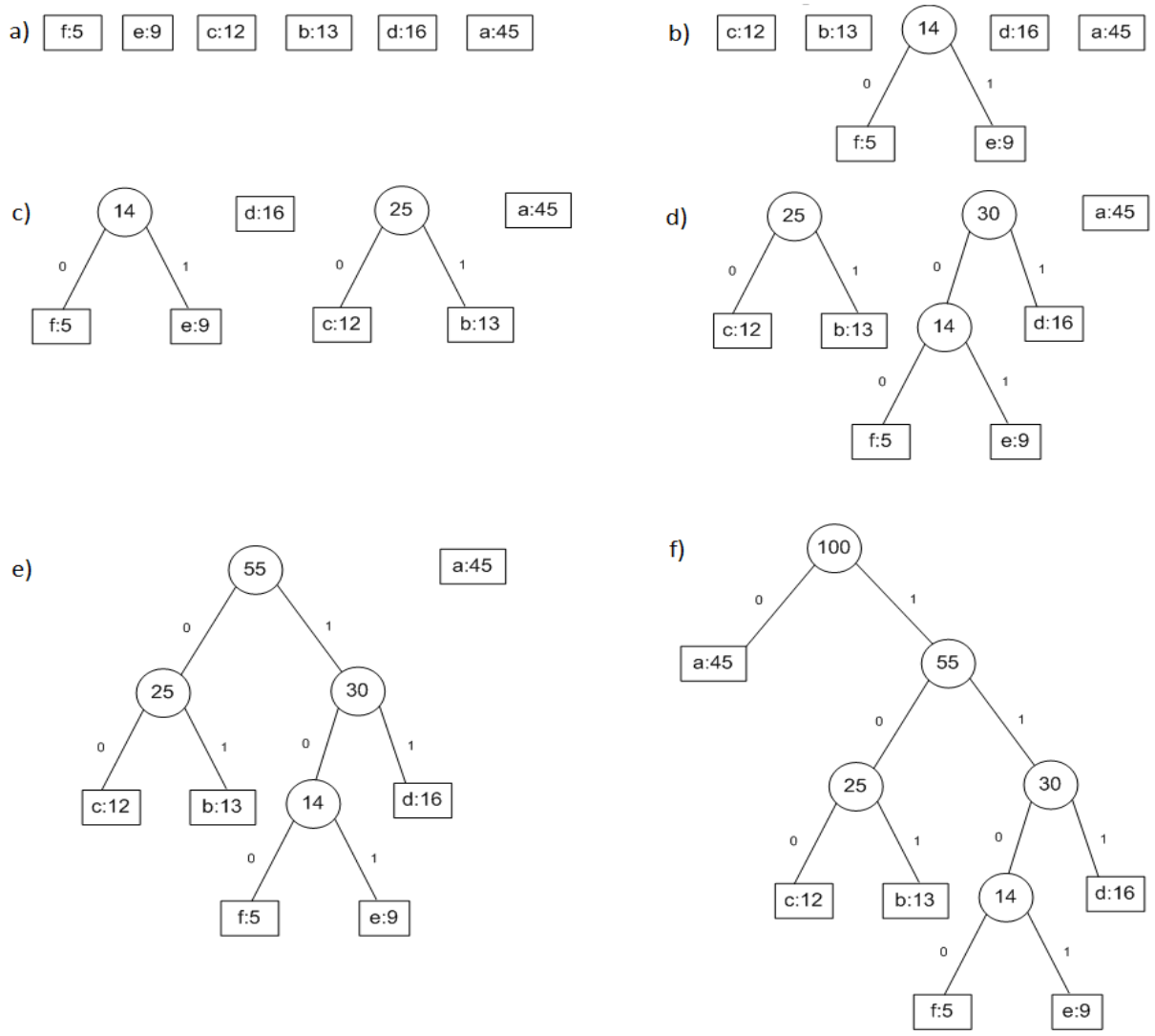


Figura 5. Pașii algoritmului pentru frecvențele de mai sus.

Fiecare parte prezintă conținutul cozii sortate după frecvență. la fiecare pas, două noduri cu frecvențele cele mai joase sunt unite. Frunzele sunt prezentate ca dreptunghiuri, iar nodurile interne sunt cercuri ce conțin suma frecvențelor copiilor. Eticheta unui copil stânga se va pune 0 iar eticheta unui copil dreapta va fi 1. Codul fiecărui caracter se obține parcurgând arborele final de la rădăcină la frunza ce conține litera.