8. Mohó algoritmusok

Optimalizálási probléma megoldására szolgáló algoritmus gyakran olyan lépések sorozatából áll, ahol minden lépésben adott halmazból választhatunk. Sok optimalizálási probléma esetén a dinamikus programozási megoldás túl sok esetet vizsgál annak érdekében, hogy az optimális választást meghatározza. Ennél egyszerűbb, hatékonyabb algoritmus is létezik. A ***mohó algoritmus*** mindig az adott lépésben optimálisnak látszó választást teszi. Vagyis, a lokális optimumot választja abban a reményben, hogy ez globális optimumhoz fog majd vezetni. Olyan optimalizálási problémákkal foglalkozunk, amelyek megoldhatók mohó algoritmussal.

Mohó algoritmus nem mindig ad optimális megoldást, azonban sok probléma megoldható mohó algoritmussal. Először egy olyan egyszerű, de nem triviális problémát vizsgálunk, az esemény-kiválasztás problémáját, amelyre a mohó algoritmus hatékony megoldást ad. A mohó algoritmushoz úgy jutunk, hogy először dinamikus programozási megoldást adunk, aztán megmutatjuk, hogy a mohó választás mindig optimális megoldást eredményez. Ezután áttekintjük a mohó stratégia elemeit, ami mohó algoritmusok helyességének közvetlenebb bizonyítását teszi lehetővé.

8.1. Egy esemény-kiválasztási probléma

Az első probléma, amit vizsgálunk közös erőforrást igénylő, egymással versengő események ütemezése, azzal a céllal, hogy kiválasszunk egy maximális elemszámú, kölcsönösen kompatibilis eseményekből álló eseményhalmazt. Tegyük fel, hogy adott ***események*** egy $S = \{a_1, a_2, \ldots, a_n\}$ n elemű halmaza, amelyek egy közös erőforrást, például egy előadótermet kívánnak használni, amit egy időben csak egyik használhat. Minden a_i eseményhez adott az s_i ***kezdő időpont*** és az f_i ***befejező időpont*** , ahol $s_i < f_i$. Ha az a_i eseményt kiválasztjuk, akkor ez az esemény az $[s_i, f_i)$ félig nyitott időintervallumot foglalja le. Az a_i és a_j események ***kompatibilisek***, ha az $[s_i, f_i)$ és $[s_j, f_j)$ intervallumok nem fedik egymást (azaz a_i és a_j kompatibilisek, ha $s_i \geq f_j$ vagy $s_j \geq f_i$). Az esemény-kiválasztási probléma azt jelenti, hogy kiválasztandó kölcsönösen kompatibilis eseményeknek egy legnagyobb elemszámú halmaza. Például tekintsük azt az S eseményhalmazt, amelynek elemeit a befejezési idejük szerint nemcsökkenő sorrendbe rendeztünk.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

(Hamarosan látni fogjuk, hogy miért célszerű így rendezni az eseményeket.) Az $(\{a_3, a_9, a_1 1\}$ részhalmaz kölcsönösen kompatibilis eseményeket tartalmaz. Azonban nem maximális, mert az $\{a_1, a_4, a_8, a_1 1\}$ részhalmaz nagyobb elemszámú. A $\{a_1, a_4, a_8, a_1 1\}$ részhalmaz ténylegesen a legbővebb kölcsönösen kompatibilis események halmaza, és egy másik ilyen legnagyobb elemszámú részhalmaz az $\{a_2, a_4, a_9, a_1 1\}$ halmaz.

Ezt a feladatot több lépésben oldjuk meg. Dinamikus programozási megoldással kezdünk, amelyben két részprobléma optimális megoldását kombináljuk, hogy az eredeti probléma optimális megoldását kapjuk. Sok választási lehetőséget tekintünk, amikor meghatározzuk, hogy mely részproblémákból épül fel az optimális megoldás. Aztán megállapítjuk, hogy csak egy választást kell nézni – a mohó választást – és amikor a mohó választást tesszük, akkor az egyik részprobléma üres, tehát csak egy nem-üres részprobléma marad. Erre az észrevételre alapozva egy rekurzív mohó algoritmust fejlesztjük ki az esemény-kiválasztási feladat megoldására. Azzal tesszük teljessé a mohó algoritmus kifejlesztését, hogy a rekurzív algoritmust átalakítjuk iteratív algoritmussá. Lépéseknek a sorozata, amelyeken keresztülmegyünk ebben az alfejezetben egy kicsit bonyolultabb annál, mint amit általában alkalmazunk mohó algoritmusok kifejlesztésénél, de jól szemlélteti a dinamikus programozás és a mohó algoritmus viszonyát.

Az esemény-kiválasztási probléma optimális részproblémák szerkezete

Mint már mondtuk, esemény-kiválasztási feladat dinamikus programozási megoldásával indulunk. Mint a 15. fejezetben, az első lépésünk az, hogy megtaláljuk az optimális szerkezetet, és felépítsük a feladat optimális megoldást a részproblémák optimális megoldásaiból.

A dinamikus programozásnál már láttuk, hogy részproblémák alkalmas terét kell definiálnunk. Kezdjük azzal, hogy definiáljuk a következő halmazokat.

$$S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\},$$

tehát $S_{i,j}$ azokat az S-beli eseményeket tartalmazza, amelyek a_i befejeződése után kezdődhetnek, és befejeződnek a_j kezdete előtt. Valójában $S_{i,j}$ azokat az eseményeket tartalmazza, amelyek kompatibilisek mind a_i-vel, mind a_j-vel, és szintén kompatibilisek az összes olyan eseménnyel, amely nem később fejeződik be, mint amikor a_i befejeződik, és azokkal, amelyek a_j kezdeténél nem korábban kezdődnek. A teljes probléma kezeléséhez egészítsük ki az eseményhalmazt az a_0 és a_{n+1} eseményekkel, ahol $f_0 = 0$, $s_{n+1} = \infty$. Ekkor $S = S_{0,n+1}$, és a részproblémák indexeinek tartománya: $0 \leq i,\ j \leq n+1$.

Még tovább szűkíthetjük i és j tartományát a következőképpen. Tegyük fel, hogy az események a befejezésük szerint monoton nem-csökkenő sorrendbe rendezettek.

$$f_0 \leq f_1 \leq f_2 \leq \cdots \leq f_n < f_{n+1}. \tag{1}$$

Azt állítjuk, hogy $S_{i,j} = \emptyset$, valahányszor $i \leq j$. Miért? Tegyük fel, hogy van olyan $a_k \in S_{i,j}$ esemény, hogy $i > j$, azaz a_i hátrább van a sorrendben, mint a_j. Ekkor azt kapnánk, hogy $f_i \leq s_k < f_k \leq s_j < f_j$. Tehát $f_i < f_j$ lenne, ami ellentmond azon feltevésünknek, hogy a_i hátrább van a sorrendben, mint a_j. Azt kaptuk, hogy feltételezve, hogy az események a befejezésük szerint monoton nem-csökkenő sorrendbe rendezettek, az $S_{i,j}$, $0 \leq i < j \leq n+1$ részproblémák közül kell maximális elemszámú, kölcsönösen kompatibilis eseményhalmazt kiválasztani, tudva, hogy minden más $S_{i,j}$ halmaz üres.

Az esemény-kiválasztási probléma részprobléma szerkezetének meghatározásához tekintsünk egy nem üres $S_{i,j}$ részproblémát, [1] és tegyük fel, hogy valamely a_k eleme a megoldásnak, azaz $f_i \leq s_k < f_k \leq s_j$. Az a_k eseményt használva két részproblémát kaphatunk, $S_{i,k}$-t (amely azon események halmaza, amelyek a_i befejezése után kezdődnek, és befejeződnek a_k kezdete előtt) és $S_{k,j}$-t (amely azon események halmaza, amelyek a_k befejezése után kezdődnek, és befejeződnek a_j kezdete előtt). Nyilvánvaló, hogy $S_{i,k}$ és $S_{k,j}$ részhalmaza az $S_{i,j}$ eseményhalmaznak. $S_{i,j}$ megoldását megkapjuk, ha az $S_{i,k}$ és $S_{k,j}$ megoldásának egyesítéséhez hozzávesszük az a_k eseményt. Tehát az $S_{i,j}$ megoldásának elemszámát kapjuk, ha az $S_{i,k}$ megoldásának elemszámához hozzáadjuk $S_{k,j}$ megoldásának elemszámát és még egyet (a_k miatt).

Az optimális részproblémák szerkezet a következő lesz. Tegyük fel, hogy $A_{i,j}$ egy optimális megoldása az $S_{i,j}$ részproblémának és $a_k \in A_{i,j}$. Ekkor az $A_{i,k}$ megoldás optimális megoldása kell legyen az $S_{i,k}$ részproblémának, és az $A_{k,j}$ megoldás optimális megoldása kell legyen az $S_{k,j}$ részproblémának. A szokásos kivágás-beillesztés módszer alkalmazható a bizonyításhoz. Ha lenne olyan $A'_{i,k}$ megoldása $S_{i,k}$-nak, amely több eseményt tartalmazna, mint $A_{i,k}$, akkor $A_{i,j}$-ben $A_{i,k}$ helyett $A'_{i,k}$-t véve $S_{i,j}$-nek egy olyan megoldását kapnánk, amely több eseményt tartalmazna, mint $A_{i,j}$. Mivel feltettük, hogy $A_{i,j}$ optimális, ezért ellentmondásra jutottunk. Hasonlóan, ha lenne olyan $A'_{k,j}$ megoldása $S_{k,j}$-nek, amely több eseményt tartalmazna, mint $A_{k,j}$, akkor $A_{i,j}$-ben $A_{k,j}$ helyett $A'_{k,j}$-t véve $S_{i,j}$-nek egy olyan megoldását kapnánk, amely több eseményt tartalmazna, mint $A_{i,j}$.

Most az optimális részproblémák szerkezet felhasználásával megmutatjuk, hogy az eredeti probléma optimális megoldása felépíthető a részproblémák optimális megoldásaiból. Láttuk, hogy egy nem üres $S_{i,j}$ részprobléma minden megoldása tartalmaz valamely a_k eseményt, és minden optimális megoldás tartalmazza az $S_{i,k}$ és $S_{k,j}$ részproblémák optimális megoldását. Tehát felépíthetünk egy maximális elemszámú, kölcsönösen kompatibilis eseményeket tartalmazó megoldását az $S_{i,j}$ részproblémának úgy, hogy két részproblémára bontjuk (a $S_{i,k}$ és $S_{k,j}$ részproblémák maximális elemszámú megoldás megkeresésével), a megkeressük két részprobléma maximális elemszámú, kölcsönösen kompatibilis események tartalmazó $A_{i,k}$ és $A_{k,j}$ megoldását, aztán az alábbi formában megalkotjuk a kölcsönösen kompatibilis eseményekből álló $A_{i,j}$ maximális elemszámú megoldást.

$$A_{i,j} = A_{i,k} \cup \{a_k\} \cup A_{k,j}. \tag{2}$$

Az eredeti probléma optimális megoldását $S_{0,n+1}$ megoldása adja.

Rekurzív megoldás

A dinamikus programozási megoldás kifejlesztésének második lépéseként rekurzív módon definiáljuk az optimális megoldás értékét. Az esemény-kiválasztási probléma esetén legyen $c[i,j]$ az $S_{i,j}$ részprobléma maximális elemszámú, kölcsönösen kompatibilis eseményeket tartalmazó részhalmaz elemszáma. Az tudjuk, hogy $c[i,j] = 0$, ha $S_{i,j} = \emptyset$, és $c[i,j] = 0$, ha $i > j$.

Tekintsünk egy $S_{i,j}$ nem üres részhalmazt. Amint láttuk, ha a_k benne van az $S_{i,j}$ egy maximális elemszámú, kölcsönösen kompatibilis eseményeket tartalmazó részhalmazában, akkor az $S_{i,k}$ és $S_{k,j}$ részproblémák egy maximális elemszámú, kölcsönösen kompatibilis eseményeket tartalmazó részhalmazait használhatjuk. A 2. egyenlőséget felhasználva kapjuk a következő rekurzív összefüggést.

$$c[i,j] = c[i,k] + c[k,j] + 1.$$

Ez a rekurzív egyenlet feltételezi, hogy ismerjük a k értéket, de ez nem így van. Összesen $j-i-1$ lehetséges értéket vehet fel k, nevezetesen $k = i+1,\ldots,j-1$. Mivel $S_{i,j}$ a maximális elemszámú részhalmaza valamelyik k-ra előáll, ezért ellenőrizzük az összes lehetséges értékre, hogy a legjobbat kiválasszuk. Tehát $c[i,j]$ teljes rekurzív alakja a következő lesz.

$$c[i,j] = \begin{cases} 0 & \text{ha } S_{i,j} = \emptyset \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{i,j}}} \{c[i,k] + c[k,j] + 1\} & \text{ha } S_{i,j} \neq \emptyset \end{cases}. \tag{3}$$

[1] Az $S_{i,j}$ halmazra néha azt mondjuk, hogy részprobléma és nem események halmaza. A szövegkörnyezetből mindig világos lesz, hogy ha $S_{i,j}$-re hivatkozunk, akkor mint események halmazát értjük, avagy egy részproblémát, amelynek a bemenete ez a halmaz.

A dinamikus programozási megoldás átalakítása mohó megoldássá

Ezen a ponton egyszerű gyakorlati feladat lehetne táblázat-kitöltős, dinamikus programozási algoritmus megírása a 3. rekurziós képlet alapján. Valóban, a

8.1. tétel. *Tekintsünk egy $S_{i,j}$ nem üres részproblémát, és legyen a_m a legkisebb befejezési idejű esemény $S_{i,j}$-ben.*

$$f_m = \min\{f_k : a_k \in S_{i,j}\}.$$

Ekkor

1. *a_m eleme $S_{i,j}$ valamely maximális elemszámú, kölcsönösen kompatibilis eseményekből álló részhalmazának.*

2. *Az $S_{i,m}$ részprobléma üres, tehát a_m választásával legfeljebb az $S_{m,j}$ nem üres.*

Bizonyítás. Először a második részt bizonyítjuk, mert az egyszerűbb. Tegyük fel, hogy $S_{i,m}$ nem üres, tehát van olyan a_k esemény, hogy $f_i \leq s_k < f_k \leq s_m < f_m$. Mivel a_k eleme $S_{i,j}$-nek, és befejezési ideje kisebb, mint a_m-é, ami ellentmond a_m választásának. Tehát azt kaptuk, hogy $S_{i,m}$ üres.

Az első rész bizonyításához tegyük fel, hogy $A_{i,j}$ egy maximális elemszámú, kölcsönösen kompatibilis eseményekből álló részhalmaza $S_{i,j}$-nek, és tekintsük $S_{i,j}$ elemeinek a befejezési idejük szerinti monoton nem-csökkenő felsorolását. Legyen a_k az első ebben a felsorolásban. Ha $a_k = a_m$, akkor készen vagyunk, mert megmutattuk, hogy a_m eleme $S_{i,j}$ valamely maximális elemszámú, kölcsönösen kompatibilis eseményeket tartalmazó részhalmazának. Ha $a_k \neq a_m$, akkor tekintsük az $A'_{i,j} = A_{i,j} - \{a_k\} \cup \{a_m\}$ részhalmazt. Az $A'_{i,j}$-beli események diszjunktak, mert $A_{i,j}$ elemei diszjunktak, és a_k az legkorábban befejeződő esemény $A_{i,j}$-ben, továbbá $f_m \leq f_k$. Mivel $A'_{i,j}$ ugyanannyi eseményt tartalmaz, mint $A_{i,j}$, ezért $A'_{i,j}$ is egy maximális elemszámú, kölcsönösen kompatibilis eseményeket tartalmazó részhalmaza $S_{i,j}$-nek, amely tartalmazza a_m-et. ■

Miért fontos az 1. tétel? Emlékeztetünk a dinamikus programozásra, amely szerint az optimális részproblémák szerkezetét az befolyásolja, hogy hány részproblémától függ az eredeti probléma, és hány választást kell végezni, hogy meghatározzuk, melyik részproblémát kell felhasználni. A dinamikus programozási megoldásunkban két részproblémát használunk az optimális megoldáshoz, és j-i-1 választást kell tenni az $S_{i,j}$ részprobléma megoldásához. Az 1. tétel jelentősen csökkenti mindkét értéket. Csak egy részprobléma kell az optimális megoldáshoz (a másik biztosan üres), és $S_{i,j}$ megoldása során csak egy választást kell nézni, ami az $S_{i,j}$ legkorábban befejeződő eseménye. Szerencsére könnyen meg tudjuk határozni ezt az eseményt.

Azon túl, hogy csökkentette a részproblémák és a választások számát, az 1. tétel más előnnyel is jár. Minden részproblémát felülről-lefelé haladó módon meg tudunk oldani, ellentétben a tipikus dinamikus programozási módszerrel, ahol alulról-felfelé kell haladni. Az $S_{i,j}$ részprobléma megoldását úgy kapjuk, vesszük $S_{i,j}$ legkorábban befejeződő a_m eseményét, és hozzávesszük az $S_{m,j}$ részprobléma egy optimális megoldásához. Mivel tudjuk, hogy a_m választásával $S_{m,j}$ optimális megoldása biztosan része $S_{i,j}$ egy optimális megoldásának, ezért nem kell megoldani $S_{m,j}$-t, $S_{i,j}$ megoldása előtt. $S_{i,j}$ -t úgy oldhatjuk meg, hogy kiválasztjuk a legkorábban befejeződő a_m eseményt $S_{i,j}$-ből, és aztán megoldjuk $S_{m,j}$ -t.

Jegyezzük meg azt is, hogy van séma a megoldandó részproblémákra. Az eredeti probléma az $S = S_{0,n+1}$. Tegyük fel, hogy az a_{m_1} eseményt választottuk, amely a legkorábban befejeződő eseménye $S_{0,n+1}$-nek. (Mivel az események befejezési idejük szerint monoton nem-csökkenő sorrendbe rendezettek, és $f_0 = 0$, így $m_1 = 1$.) A következő részproblémánk $S_{m_1,n+1}$ lesz. Tegyük fel, hogy a_{m_2}-t választottuk $S_{m_1,n+1}$-ből, amely a legkorábban befejeződő eseménye. (Nem feltétlenül teljesül, hogy $m_2 = 2$.) A következő részproblémánk $S_{m_2,n+1}$ lesz. Ezt folytatva látjuk, hogy minden részproblémánk $S_{m_i,n+1}$ alakú lesz, valamely m_i esemény-sorszámra. Más szóval, minden részproblémát a legkésőbb befejeződő esemény, és egy másik esemény sorszáma határoz meg, ahol az utóbbi részproblémáról-részproblémára változik.

A választandó eseményre is van sémánk. Mivel mindig $S_{m_i,n+1}$ -nek a legkorábban befejeződő eseményét választjuk, így a részproblémákhoz kiválasztott események sorozata a befejezési idő szerint szigorúan monoton növekvő lesz. Továbbá, minden eseményt csak egyszer kell vizsgálni, a befejezési idejük szerint monoton nem-csökkenő sorrendben.

Egy részprobléma megoldásához mindig azt az a_m eseményt választjuk ki, amely a legkorábban befejeződik, és legálisan

beosztható. Tehát a választás „mohó” abban az értelemben, hogy intuitíve a legnagyobb lehetőséget hagyja a fennmaradt események beosztására. Tehát az a mohó választás, amely maximalizálja a beosztásra fennmaradt időt.

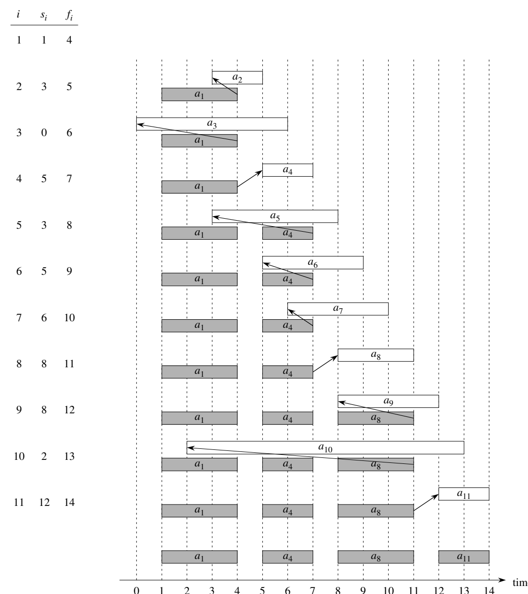
Rekurzív mohó algoritmus

Miután láttuk, hogyan adhatunk dinamikus programozási megoldás, amely felülről-lefelé haladó módszer, itt az ideje, hogy megadjunk egy tisztán mohó, alulról felfelé haladó módszerű algoritmust. A REKURZÍV-ESEMÉNY-KIVÁLASZTÓ eljárás közvetlenül kapható rekurzív megoldása a problémának. Ennek bemenő paraméterei az események kezdő és befejező időpontjait tartalmazó s és f tömb, továbbá a megoldandó $S_{i,n+1}$ részproblémát meghatározó i és n sorszám. (Az n paraméter az utolsó a_n esemény indexe, és nem az $n+1$ fiktív esemény, amely szintén eleme a részproblémának.) Az eljárás $S_{i,n+1}$ egy maximális elemszámú, kölcsönösen kompatibilis eseményeket tartalmazó részhalmazát adja eredményül. Feltételezzük, hogy az n bemeneti esemény befejezési

idő szerint monoton nem-csökkenő sorrendbe rendezett az 1. képletnek megfelelően. Ha a rendezettség nem teljesülne, akkor $O(n \log n)$ időben rendezhetjük őket. A kiindulási probléma megoldását a REKURZÍV-ESEMÉNY-KIVÁLASZTÓ$(s, f, 0, n)$ eljáráshívás adja.

REKURZÍV-ESEMÉNY-KIVÁLASZTÓ(s, f, i, n)

1 $m \leftarrow i+1$
2 **while** $m \leq n$ és $s_m < f_i$ ▷ $S_{i,n+1}$ első választható eseményét keressük
3 **do** $m \leftarrow m+1$
4 **if** $m < j$
5 **then return** $\{a_m\} \cup$ REKURZÍV-ESEMÉNY-KIVÁLASZTÓ(s, f, m, n)
6 **then return** $\emptyset$



1. ábra. A REKURZÍV-ESEMÉNY-KIVÁLASZTÓ algoritmus működése a korábban megadott 11 eseményre. Egy rekurzív hívás során vizsgált események két horizontális vonal között láthatóak. A fiktív a_0 esemény befejezési ideje 0, az első REKURZÍV-ESEMÉNY-KIVÁLASZTÓ$(s, f, 0, 11)$ eljáráshíváskor az a_1 esemény választódik ki. A már korábban kiválasztott események satírozottak, az éppen vizsgált esemény pedig fehér. Ha egy esemény kezdő időpontja előbb van, mint a legutoljára beválasztott esemény befejező időpontja (a közöttük meghúzott nyíl balra mutat), akkor azt elvetjük. Egyébként (ha a nyíl egyenesen felfelé, vagy jobbra mutat) beválasztjuk. Az utolsó REKURZÍV-ESEMÉNY-KIVÁLASZTÓ$(s, f, 11, 11)$ rekurzív hívás a $\emptyset$ értékkel tér vissza. Az eredményül kapjuk a kiválasztott események $\{a_1, a_4, a_8, a_11\}$ halmazát.

Az 1. ábra mutatja az algoritmus által végzett műveleteket. A REKURZÍV-ESEMÉNY-KIVÁLASZTÓ(s, f, m, n) egy adott meghívásakor a 2-3. sorokban a **while** ciklus megkeresi az $S_{i,n+1}$ első választható eseményét. A ciklus sorban az $a_{i+1}, a_{i+2}, \ldots, a_n$

eseményeket vizsgálja, amíg meg nem találja az első olyan a_m eseményt, amely kompatibilis a_i-vel, azaz $s_m \geq f_i$ teljesül. Ha a ciklus úgy ér véget, hogy talált ilyen eseményt, akkor az eljáráshívással befejeződik az 5. sorban végrehajtott **return** utasítással, ami visszaadja az $\{a_m\}$ és a REKURZÍV-ESEMÉNY-KIVÁLASZTÓ(s, f, m, n) rekurzív hívás által visszaadott halmazok egyesítését. Az utóbbi halmaz az $S_{m,n+1}$ részprobléma megoldása. A ciklus úgy is terminálhat, hogy a $m > n$ feltétel teljesül, amikor is nincs olyan esemény, amely kompatibilis lenne s_i-vel. Ebben az esetben $S_{i,n+1} = \emptyset$, és az eljárás az $\emptyset$ értéket adja vissza a 6. sorban.

Feltéve, hogy az események befejezési idejük szerint monoton nem-csökkenően rendezettek, a REKURZÍV-ESEMÉNY-KIVÁLASZTÓ$(s, f, 0, n)$ eljáráshívás futási ideje $\Theta(n)$. Ezt a következőképpen láthatjuk be. A rekurzív hívásokban minden eseményt pontosan egyszer vizsgálunk a **while** ciklus feltételvizsgálatakor a 2. sorban. Pontosabban, az a_k eseményt az utolsó olyan hívás vizsgálja, amelyre $i < k$.

Iteratív mohó algoritmus

A rekurzív eljárásunkat egyszerűen átalakíthatjuk iteratív algoritmussá. A REKURZÍV-ESEMÉNY-KIVÁLASZTÓ eljárás majdnem jobb-rekurzív, önmagát hívó rekurzív hívással végződik, amit követ egy egyesítés művelet. Jobb-rekurzív eljárás átalakítása iteratívvá általában egyszerű feladat, valójában több programozási nyelv fordítóprogramja ezt automatikusan elvégzi. Amint látjuk, a REKURZÍV-ESEMÉNY-KIVÁLASZTÓ eljárás minden $S_{i,n+1}$ részproblémára működik, tehát azokra, amelyek a legnagyobb befejezésű eseményeket tartalmazzák.

A MOHÓ-ESEMÉNY-KIVÁLASZTÓ eljárás egy iteratív változata a REKURZÍV-ESEMÉNY-KIVÁLASZTÓ eljárásnak. Ez ismét feltételezi, hogy a bemeneti események befejezési idejük szerint monoton nem-csökkenő sorrendbe rendezettek. Az eljárás az A változóban gyűjti össze a kiválasztott eseményeket, és ezt adja eredményül a végén.

MOHÓ-ESEMÉNY-KIVÁLASZTÓ(s, f)

1 $n \leftarrow$ *hossz*$[s]$
2 $A \leftarrow \{a_1\}$
3 $i \leftarrow 1$
4 **for** $m \leftarrow 2$ **to** n
5   **do if** $s_m \geq f_i$
6   **then** $A \leftarrow A \cup \{a_m\}$
7             $i \leftarrow m$
8 **return** A

Az eljárás a következőképpen működik. Az i változó tartalmazza az A-ba legutoljára beválasztott esemény indexét, aminek az a_i esemény felel meg a rekurzív változatban. Mivel az eseményeket befejezési idejük szerinti monoton nem-csökkenő sorrendben vizsgáljuk, ezért f_i mindig a legnagyobb befejezési idejű esemény az A halmazban. Tehát

$$f_i = \max\{f_k : a_k \in A\} . \tag{4}$$

Az 2-3. sorban kiválasztjuk az a_1 eseményt, előkészítve ezzel az A halmazt, hogy egyedül az a_1 eseményt tartalmazza, az i változó pedig ezen esemény sorszámát veszi fel kezdetben. A ***for*** ciklus a 4-7. sorokban megkeresi a legkorábban befejeződő eseményt az $S_{i,n+1}$ halmazban. A ciklus egymás után vizsgálja az a_m eseményeket, és hozzáadja az A halmazhoz, ha kompatibilis az összes A-beli eseménnyel. Annak ellenőrzése, hogy a_m kompatibilis az összes A-ban lévő eseménnyel, a 4. egyenlőség miatt elegendő azt ellenőrizni (5. sor), hogy az s_m kezdő időpont nem korábbi, mint az A-ba legutoljára beválasztott esemény f_i befejező időpontja. Ha az a_m esemény kompatibilis, akkor a 6-7. sorokban hozzávesszük a_m-et A-hoz, és i felveszi az m értéket. A MOHÓ-ESEMÉNY-KIVÁLASZTÓ(s, f) eljáráshívás pontosan azt a halmazt adja, mint a REKURZÍV-ESEMÉNY-KIVÁLASZTÓ$(s, f, 0, n)$ hívás.

A MOHÓ-ESEMÉNY-KIVÁLASZTÓ algoritmus, csakúgy, mint a REKURZÍV-ESEMÉNY-KIVÁLASZTÓ $\Theta(n)$ időben megoldja n bemeneti eseményre a feladatot, feltéve, hogy az események kezdetben a befejezési idejük szerint monoton nem-csökkenő sorrendben vannak.

```
public int[] Kivalaszto(int[] s, int[] f){
   int n=s.length;
   int vege=f[0];
   int k=0;
   int[] Beoszt=new int[n];
   Beoszt[0]=0;
```

```
   for (int i=1; i<n; i++){
      if (vege<=s[i]){
         Beoszt[++k]=i;
         vege=f[i];
      }
   }

   return Beoszt;
}
```

A mohó stratégia elemei

A mohó algoritmus úgy alkotja meg a probléma optimális megoldását, hogy választások sorozatát hajtja végre. Az algoritmus során minden döntési pontban azt az esetet választja, amely az adott pillanatban optimálisnak látszik. Ez a heurisztikus stratégia nem mindig ad optimális megoldást, azonban néha igen, mint azt láttuk az esemény-kiválasztási probléma esetén. Ebben a szakaszban a mohó stratégia néhány általános tulajdonságát fogjuk megvizsgálni.

Az a módszer, amit követtünk mohó algoritmus kifejlesztésére, egy kicsit bonyolultabb az általános esetnél. A következő lépések sorozatán mentünk keresztül.

1. A probléma optimális szerkezetének meghatározása.
2. Rekurzív megoldás kifejlesztése.
3. Annak bizonyítása, hogy minden rekurzív lépésben az egyik optimális választás a mohó választás. Tehát mindig biztonságos a mohó választás.
4. Annak igazolása, hogy a mohó választás olyan részproblémákat eredményez, amelyek közül legfeljebb az egyik nem üres.
5. A mohó stratégiát megvalósító rekurzív algoritmus kifejlesztése.
6. A rekurzív algoritmus átalakítása iteratív algoritmussá.

Ezen lépéseken keresztülhaladva láttuk a mohó algoritmus dinamikus programozási alátámasztását. A gyakorlatban azonban általában egyszerűsítjük a fenti lépéseket mohó algoritmus tervezésekor. A részproblémák kifejlesztésekor arra figyelünk, hogy a mohó választás egyetlen részproblémát eredményezzen, amelynek optimális megoldását kell megadni. Például az esemény-kiválasztási feladatnál először olyan $S_{i,j}$ részproblémákat határoztunk meg, ahol i és j is változó érték lehetett. Ezután rájöttünk, hogy ha mindig mohó választást végzünk, akkor redukálhatjuk a részproblémákat $S_{i,n+1}$ alakúakra.

Másképpen kifejezve, az optimális részproblémák szerkezetét a mohó választás figyelembevételével alakíthattuk ki. Tehát elhagyhattuk a második indexet, és az $S_i = \{a_k \in S : f_i \leq s_k\}$ alakú részproblémákhoz jutottunk. Ezután bebizonyíthattuk, hogy a mohó választás (az első befejeződő a_m esemény S_i-ben), kombinálva S_m egy optimális megoldásával, az eredeti S_i probléma optimális megoldását adja. Általánosabban, mohó algoritmus tervezését az alábbi lépések végrehajtásával végezzük.

1. Fogalmazzuk meg az optimalizációs feladatot úgy, hogy minden egyes választás hatására egy megoldandó részprobléma keletkezzék.
2. Bizonyítsuk be, hogy mindig van olyan optimális megoldása az eredeti problémának, amely tartalmazza a mohó választást, tehát a mohó választás mindig biztonságos.
3. Mutassuk meg, hogy a mohó választással olyan részprobléma keletkezik, amelynek egy optimális megoldásához hozzávéve a mohó választást, az eredeti probléma egy optimális megoldását kapjuk.

Ezt a közvetlenebb módszert alkalmazzuk a fejezet hátralévő részében. Mindazonáltal, minden mohó algoritmushoz majdnem mindig van bonyolultabb dinamikus programozási megoldás.

Meg tudjuk-e mondani, hogy adott optimalizációs feladatnak van-e mohó algoritmusú megoldása? Erre nem tudunk általános választ adni, de a mohó-választási tulajdonság és az optimális részproblémák tulajdonság két kulcsfontosságú összetevő. Ha meg tudjuk mutatni, hogy a feladat rendelkezik e két tulajdonsággal, nagy eséllyel ki tudunk fejleszteni mohó algoritmusú megoldást.

Mohó-választási tulajdonság

Az első alkotóelem a ***mohó-választási tulajdonság***: globális optimális megoldás elérhető lokális optimum (mohó) választásával. Más szóval, amikor arról döntünk, hogy melyik választást tegyük, azt választjuk, amelyik az adott pillanatban a legjobbnak tűnik, nem törődve a részproblémák megoldásaival. Ez az a pont, ahol a mohó stratégia különbözik a dinamikus programozástól. Dinamikus programozás esetén minden lépésben választást hajtunk végre, de a választás függhet a részproblémák megoldásától. Következésképpen, a dinamikus programozási módszerrel a problémát alulról-felfelé haladó módon oldjuk meg, egyszerűbbtől összetettebb részproblémák felé haladva. A mohó algoritmus során az adott pillanatban legjobbnak tűnő választást hajtjuk végre, bármi is legyen az, és azután oldjuk meg a választás hatására fellépő részproblémát. A mohó algoritmus során végrehajtott választás függhet az addig elvégzett választásoktól, de nem függhet későbbi választásoktól, vagy részproblémák megoldásától. Tehát ellentétben a dinamikus programozással, amely a részproblémákat alulról-felfelé haladva oldja meg, a mohó stratégia általában felülről-lefelé halad, egymás után végrehajtva mohó választásokat, amellyel a problémát sorra kisebb méretűre redukálja.

Természetesen bizonyítanunk kell, hogy a lépésenkénti mohó választásokkal globálisan optimális megoldáshoz jutunk, és ez az ami leleményességet igényel. Tipikusan, mint az 1. tétel esetén, a bizonyítás részproblémák globális optimális megoldását vizsgálja. Megmutatja, hogy az optimális megoldás módosítható úgy, hogy az a mohó választást tartalmazza, és hogy ez a választás redukálja a problémát hasonló, de kisebb méretű részproblémára.

A mohó-választási tulajdonság gyakran hatékonyságot eredményez a részprobléma választásával. Például az eseménykiválasztási feladatnál, feltételezve, hogy az események befejezési idejük szerint monoton nem-csökkenő sorrendbe rendezettek, minden eseményt csak egyszer kell vizsgálni. Gyakran az a helyzet, hogy a bemeneti adatokat alkalmasan elő-feldolgozva, vagy alkalmas adatszerkezetet használva (ami gyakran prioritási sor), a mohó választás gyorsan elvégezhető, és ezáltal hatékony algoritmust kapunk.

Optimális részproblémák tulajdonság

Egy probléma teljesíti az ***optimális részproblémák tulajdonságot***, ha az optimális megoldás felépíthető a részproblémák optimális megoldásából. Ez az alkotóelem kulcsfontosságú mind a dinamikus programozás, mind a mohó stratégia alkalmazhatóságának megállapításánál. Az optimális részproblémákra példaként emlékeztetünk arra, ahogy megmutattuk, hogy ha $S_{i,j}$ egy optimális megoldása tartalmazza az a_k eseményt, akkor az szükségképpen tartalmazza $S_{i,k}$ és $S_{k,j}$ egy optimális megoldását. Ezen optimális szerkezet alapján, ha tudjuk, hogy melyik a_k eseményt kell választani, akkor $S_{i,j}$ egy optimális megoldása megalkotható a_k, továbbá $S_{i,k}$ és $S_{k,j}$ egy optimális megoldásából. Az optimális részproblémák ezen tulajdonságát észre véve meg tudtuk adni a 3. rekurzív egyenletet, ami az optimális megoldás értékét adja meg.

Általában sokkal közvetlenebb alkalmazását használjuk az optimális részproblémák tulajdonságnak mohó algoritmus kifejlesztése során. Mint már említettük, szerencsénk van, amikor feltételezzük, hogy az eredeti probléma mohó választása megfelelő részproblémát eredményez. Csak azt kell belátni, hogy a részprobléma optimális megoldása, kombinálva a már elvégzett mohó választással, az eredeti probléma optimális megoldását adja. Ez a séma implicit módon használ részproblémák szerinti indukciót annak bizonyítására, hogy minden lépésben mohó választást végezve optimális megoldást kapunk.

Mohó stratégia vagy dinamikus programozás

Mivel az optimális részproblémák tulajdonságot kihasználjuk mind a mohó, mind a dinamikus programozási stratégiáknál, előfordulhat, hogy dinamikus programozási megoldást próbálunk adni akkor, amikor mohó megoldás is célravezető lenne, és fordítva, tévesen mohó megoldással próbálkozunk akkor, amikor valójában dinamikus programozási módszert kellene alkalmazni. A finom különbségek illusztrálására tekintsük a következő klasszikus optimalizálási probléma két változatát.

A 0-1 ***hátizsák feladat*** a következőt jelenti. Adott n darab tárgy, az i-edik tárgy használati értéke v_i, a súlya pedig w_i, ahol v_i és w_i egész számok. Kiválasztandó a tárgyaknak olyan részhalmaza, amelyek használati értékének összege a lehető legnagyobb, de a súlyuk összege nem nagyobb, mint a hátizsák W kapacitása, amely egész szám. Mely tárgyakat rakjuk a hátizsákba? (Ezt a problémát azért nevezzük 0-1 hátizsák feladatnak, mert minden tárgyat vagy beválasztunk, vagy elhagyunk, nem tehetjük meg, hogy egy tárgy töredékét, vagy többszörösét választjuk.)

A ***töredékes hátizsák feladat*** csak abban különbözik az előzőtől, hogy a tárgyak töredéke is választható, nem kell 0-1 bináris választást tenni. Úgy tekinthetjük, hogy 0-1 hátizsák feladat esetén a tárgyak arany tömbök, míg a töredékes hátizsák feladatnál aranyporból meríthetünk.

Mindkét hátizsák feladat teljesíti az optimális részproblémák tulajdonságot. A 0-1 feladat esetén tekintsünk egy olyan választást, amely a legnagyobb használati értéket adja, de a tárgyak összsúlya nem haladja meg a W értéket. Ha kivesszük a j-edik tárgyak a hátizsákból, akkor a bennmaradt tárgyak használati értéke a legnagyobb lesz azon feltétel mellett, hogy az összsúly nem nagyobb, mint $W - w_j$, és $n - 1$ tárgyból választhatunk, kizárva az eredeti tárgyak közül a j-ediket. A töredékes hátizsák feladatnál ha egy

	a	b	c	d	e	f
Gyakoriság (ezrekben)	45	13	12	16	9	5
Fix hosszú kódszó	000	001	010	011	100	101
Változó hosszú kódszó	0	101	100	111	1101	1100

2. ábra. Karakter kódolási probléma. Az adatállomány 100 000 karakterből áll, és csak az a–f karakterek fordulnak elő az állományban a feltüntetett gyakoriságokkal. Ha minden karaktert 3 bites kódszóval kódolunk, akkor 300 000 bitre van szükség. Az ábrán látható változó hosszú kódszavakat használva az állományt 224 000 bittel kódolhatjuk.

optimális választásból kiveszünk a j tárgyból w mennyiséget, akkor a megmaradt választás optimális lesz arra az esetre, amikor legfeljebb $W-w$ összsúlyt érhetünk el és a j-edik tárgyból legfeljebb w_j-w mennyiséget választhatunk.

Bár a két feladat hasonló, a töredékes hátizsák feladat megoldható mohó stratégiával, a 0-1 feladat azonban nem. A töredékes feladat megoldásához előbb számítsuk ki minden tárgyra a v_i/w_i használati érték per súly hányadost. A mohó stratégiát követve először a legnagyobb hányadosú tárgyból választunk amennyit csak lehet. Ha elfogyott, de még nem telt meg a hátizsák, akkor a következő legnagyobb hányadosú tárgyból választunk amennyit csak lehet, és így tovább, amíg a hátizsák meg nem telik. Mivel a tárgyakat az érték per súly hányados szerint kell rendeznie, a mohó algoritmus futási ideje $O(n\lg n)$ lesz. Annak bemutatására, hogy a mohó stratégia nem működik a 0-1 hátizsák feladatra, tekintsük a következő esetet.

i	1	2	3
w_i	10	20	30
v_i	60	100	120

Három tárgyunk van, és a hátizsák mérete 50 egységnyi. Az 1. tárgy súlya 10, használati értéke 60, a 2. tárgy súlya 20, használati értéke 100, a 3. tárgy súlya 30, használati értéke pedig 120 egység. Tehát az 1. tárgy érték per súly hányadosa 6, a 2. tárgyé 2, a 3. tárgyé pedig 4. Így a mohó stratégia először az 1. tárgyat választaná. Azonban az optimális megoldásban a 2. és a 3. tárgy szerepel, kihagyva az 1. tárgyat. Mindkét választás, amelyben az 1. tárgy szerepel nem optimális.

A megfelelő töredékes feladatra azonban a mohó stratégia, amely először az 1. tárgyat választja, optimális megoldást ad. A 0-1 feladat esetén az 1. tárgy választása nem vezet optimális megoldáshoz, mert ezután nem tudjuk telerakni a hátizsákot, és az üresen maradt rész csökkenti a hátizsák lehetséges érték per súly hányadost. A 0-1 feladatnál amikor egy tárgy beválasztásáról döntünk, akkor előbb össze kell hasonlítani annak a két részproblémának a megoldását, amely a tárgy beválasztásával, illetve kihagyásával adódik. Az így megfogalmazott probléma sok, egymást átfedő részproblémát eredményez, ami a dinamikus programozást fémjelzi. Valóban, a 0-1 feladat megoldható dinamikus programozási módszerrel.

Huffman-kód

A Huffman-kód széles körben használt és nagyon hatékony módszer adatállományok tömörítésére. Az elérhető megtakarítás 20%-tól 90%-ig terjedhet, a tömörítendő adatállomány sajátosságainak függvényében. A kódolandó adatállományt karaktersorozatnak tekintjük. A Huffman féle mohó algoritmus egy táblázatot használ az egyes karakterek előfordulási gyakoriságára, hogy meghatározza, hogyan lehet a karaktereket optimálisan ábrázolni bináris jelsorozattal.

Tegyük fel, hogy egy 100 000 karaktert tartalmazó adatállományt akarunk tömörítetten tárolni. Tudjuk, hogy az egyes karakterek előfordulási gyakorisága megfelel a 2. ábrán látható táblázatnak. Vagyis, hat különböző karakter fordul elő az állományban, és az a karakter 45 000-szer fordul elő az állományban.

Sokféleképpen ábrázolható egy ilyen típusú információ halmaz. Mi ***bináris karakterkód*** (vagy röviden ***kód***) tervezésének problémáját vizsgáljuk, amikor is minden karaktert egy bináris jelsorozattal ábrázolunk. Ha ***fix hosszú kódot*** használunk, akkor 3 bitre van szükség a hatféle karakter kódolására: $a=000, b=001, \ldots, f=101$. Ez a módszer 300 000 bitet igényel a teljes állomány kódolására. Csinálhatjuk jobban is? A ***változó hosszú kód*** alkalmazása tekintélyes megtakarítást eredményez, ha gyakori karaktereknek rövid, ritkán előforduló karaktereknek hosszabb kódszavat feleltetünk meg. A 2. ábra egy ilyen kódolást mutat: itt az egybites 0 kód az a karaktert ábrázolja, a négybites 1100 kód pedig az f karakter kódja. Ez a kódolás

$$(45\cdot 1+13\cdot 3+12\cdot 3+16\cdot 3+9\cdot 4+5\cdot 4)\cdot 1000=224\,000$$

bitet igényel az állomány tárolására, ami hozzávetőleg 25% megtakarítást eredményez. Valójában ez optimális kódolást jelent, mint majd látni fogjuk.

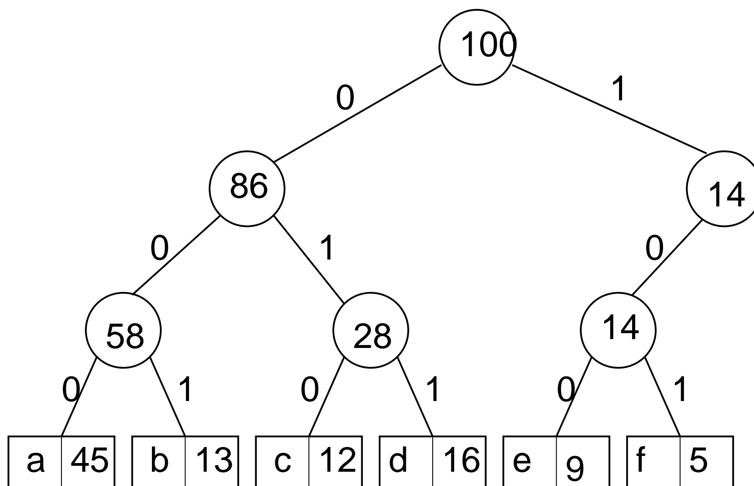
8.2. Prefix-kódok

A továbbiakban csak olyan kódszavakat tekintünk, amelyekre igaz, hogy egyik sem kezdőszelete a másiknak. Az ilyen kódolást ***prefix-kódnak*** nevezzük. [2] Megmutatható (bár mi ezt nem tesszük meg), hogy karakterkóddal elérhető optimális adattömörítés mindig megadható prefix-kóddal is, így az általánosság megszorítása nélkül elegendő prefix-kódokat tekinteni.

A prefix-kódok előnyösek, mert egyszerűsítik a kódolást (tömörítést) és a dekódolást. A kódolás minden bináris karakterkódra egyszerű: csak egymás után kell írni az egyes karakterek bináris kódját. Például a 3. ábrán adott változó hosszú karakterkód esetén az abc három karaktert tartalmazó állomány kódja $0 \cdot 101 \cdot 100 = 0101100$, ahol a „·" pont az egymásután írás művelet (konkatenáció) jele.

A dekódolás is meglehetősen egyszerű prefix-kód esetén. Mivel nincs olyan kódszó, amely kezdőszelete lenne egy másiknak, így egyértelmű, hogy a kódolt állomány melyik kódszóval kezdődik. Egyszerűen megállapítjuk, hogy a kódolt állomány melyik kódszóval kezdődik, aztán helyettesítjük ezt azzal a karakterrel, amelynek ez a kódja, és ezt az eljárást addig végezzük, amíg a kódolt állományon végig nem értünk. A példánkat tekintve, a 001011101 jelsorozat egyértelműen bontható fel a $0 \cdot 0 \cdot 101 \cdot 1101$ kódszavak sorozatára, tehát a dekódolás az $aabe$ sorozatot eredményezi.

A dekódolási eljáráshoz szükség van a prefix-kód olyan alkalmas ábrázolására, amely lehetővé teszi, hogy a kódszót könnyen azonosítani tudjuk. Az olyan bináris fa, amelynek levelei a kódolandó karakterek, egy ilyen alkalmas ábrázolás. Ekkor egy karakter kódját a fa gyökerétől az adott karakterig vezető út ábrázolja, a 0 azt jelenti, hogy balra megyünk, az 1 pedig, hogy jobbra megyünk az úton a fában. A 3. ábra a példánkban szereplő két kódot ábrázolja. Vegyük észre, hogy ezek a fák nem bináris keresőfák, a levelek nem rendezetten találhatók, a belső csúcsok pedig nem tartalmaznak karakter kulcsokat. Egy adatállomány optimális



3. ábra. A 2. ábrán adott kódolásokhoz tartozó bináris fák. Minden levél címkeként tartalmazza a a kódolandó karaktert és annak előfordulási gyakoriságát. A belső csúcsok az adott gyökerű részfában található gyakoriságok összegét tartalmazzák. A fix hosszú kódhoz tartozó fa; $a = 000, \dots, f = 101$.

kódját mindig *teljes* bináris fa

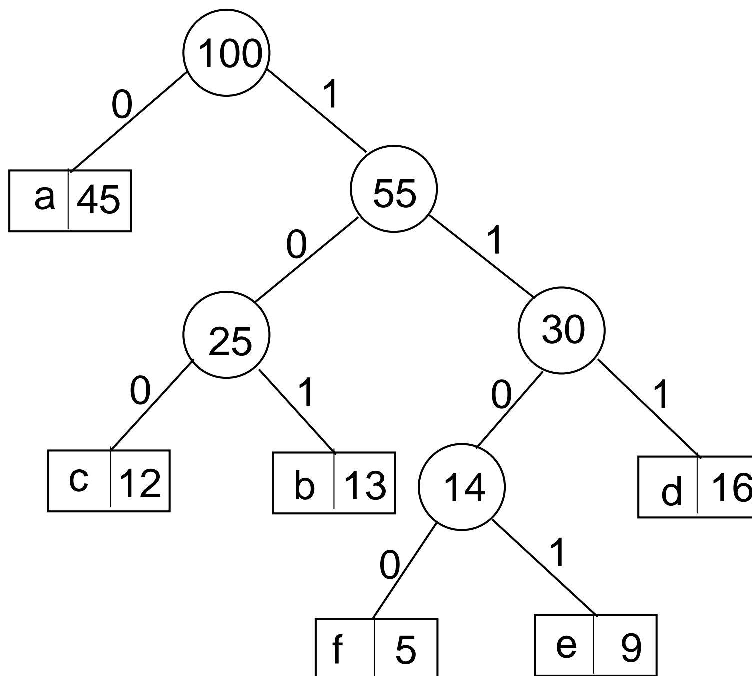
ábrázolja, tehát olyan fa, amelyben minden nem levél csúcsnak két gyereke van. A példánkban szereplő fix hosszú kód nem optimális, mert a 3. ábrán látható fája nem teljes bináris fa: van olyan kódszó, amely 10 -lal kezdődik, de nincs olyan, amely 11 -gyel kezdődne. Mivel a továbbiakban szorítkozhatunk teljes bináris fákra, azt mondhatjuk, hogy ha C az az ábécé, amelynek elemei a kódolandó karakterek, akkor az optimális prefix-kód fájának pontosan $|C|$ levele és pontosan $|C| - 1$ belső csúcsa van.

Ha adott egy prefix-kód T fája, akkor egyszerű kiszámítani, hogy az adatállomány kódolásához hány bit szükséges. A C ábécé minden c karakterére jelölje $f(c)$ a c karakter előfordulási gyakoriságát az állományban, $d_T(c)$ pedig jelölje a c-t tartalmazó levél mélységét a T fában. Vegyük észre, hogy $d_T(c)$ megegyezik a c karakter kódjának hosszával. A kódoláshoz szükséges bitek száma ekkor

$$B(T) = \sum_{c \in C} f(c) d_T(c) \tag{5}$$

és ezt az értéket a T fa költségének nevezzük.

[2] A „prefix-mentes" elnevezés helyesebb lenne, de a „prefix-kód" általánosan használt az irodalomban.



4. ábra. Az optimális prefix-kódhoz tartozó fa; $a = 0, b = 101, \ldots, f = 1100$.

8.3. Huffman-kód szerkesztése

Huffman találta ki azt a mohó algoritmust, amely optimális prefix-kódot készít, amit ***Huffman-kódnak*** nevezünk. A 2. szakasz megállapításait figyelembe véve az algoritmus helyességének bizonyítása a mohó-választási és az optimális részproblémák tulajdonságon alapszik. Ahelyett, hogy a kód kifejlesztése előtt bebizonyítanánk e két tulajdonság teljesülését, először a kódot adjuk meg. Ezt azért tesszük, hogy világosan lássuk, az algoritmus hogyan használja a mohó választást.

A következő, pszeudokód formájában adott algoritmusban feltételezzük, hogy C a karakterek n elemű halmaza, és minden $c \in C$ karakterhez adott annak $f[c]$ gyakorisága. Az algoritmus alulról-felfelé haladva építi fel azt a T fát, amely az optimális kód fája. Az algoritmus úgy indul, hogy kezdetben $|C|$ számú csúcs van, amelyek mindegyike levél, majd $|C| - 1$ számú „összevonás” végrehajtásával alakítja ki a végső fát. Az f-szerint kulcsolt Q prioritási sort használjuk az összevonandó két legkisebb gyakoriságú elem azonosítására. Két elem összevonásának eredménye egy új elem, amelynek gyakorisága a két összevont elem gyakoriságának összege.

HUFFMAN(C)

1 $n \leftarrow |C|$
2 $Q \leftarrow C$
3 **for** $i \leftarrow 1$ **to** $n-1$
4   **do** új z csúcs létesítése
5   *bal*$[z] \leftarrow x \leftarrow$ KIVESZ-MIN(Q)
6   *jobb*$[z] \leftarrow y \leftarrow$ KIVESZ-MIN(Q)
7         $f[z] \leftarrow f[x] + f[y]$
8   BESZÚR(Q, z)
9 **return** KIVESZ-MIN(Q)

Az algoritmus megvalósítása

A Huffman-kód fája $n+n-1 = 2n-1$ pontot tartalmazó rendezett bináris fa. Azonosítsuk a fa pontjait az $\{1, \ldots, 2n-1\}$ számokkal. A fát adjuk meg azzal az

$$Apa : \{1, \ldots 2n-1\} \to \mathbb{Z}$$

függvénnyel, amelyre

$$Apa(i) = \begin{cases} -j & \text{ha } i \text{ bal fia } j \text{ -nek} \\ j & \text{ha } i \text{ jobb fia } j \text{ -nek} \\ 0 & \text{ha } i \text{ a gyökér} \end{cases}$$

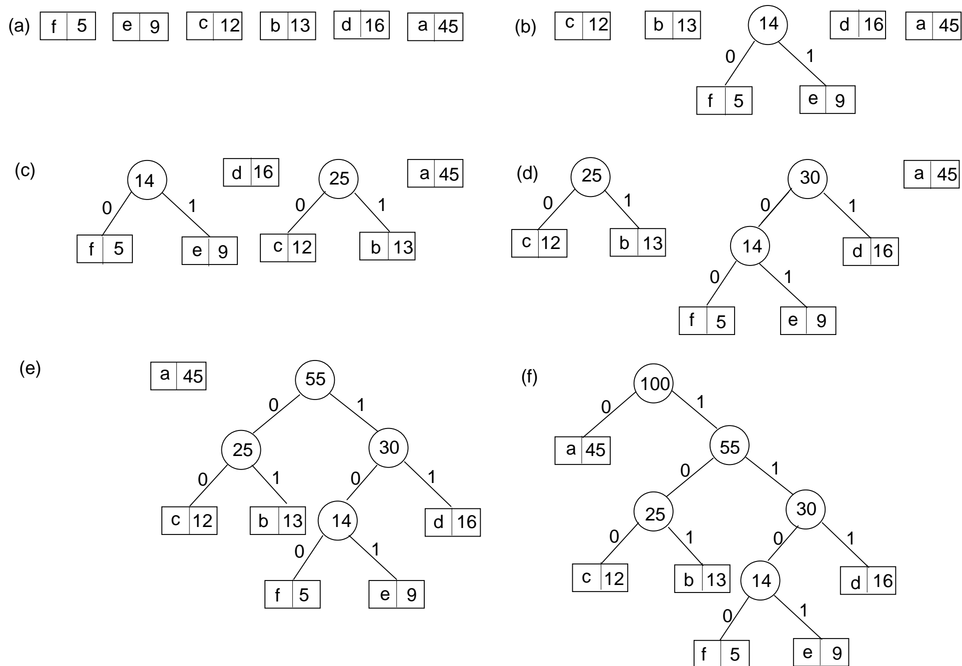
Tehát az $Apa(i)$ függvényérték előjelével kódoljuk, hogy i bal, avagy jobb fia apjának.

```
public class HuffmanKod{

   private static class KulcsPar implements Comparable<KulcsPar> {
      public float kulcs;
      public int adat;
      public int compareTo(KulcsPar z){
         return kulcs < z.kulcs ? -1: kulcs > z.kulcs ? 1: 0;
      }
   }

public static String[] Huffman(float[] F){
   int n=F.length;
   int[] Apa=new int[2*n];
   PriSor<KulcsPar> S = new PriSorT<KulcsPar>(n);
   for (int i=0; i<n; i++){
      KulcsPar p=new KulcsPar();
      p.kulcs=F[i];
      p.adat=i;
      S.SorBa(p);
   }

   KulcsPar x ,y, z;
   int gyoker=n-1;
   for (int i=1; i<n; i++){
      x=S.SorBol();
      y=S.SorBol();
      gyoker++;
      Apa[x.adat]=-gyoker;
      Apa[y.adat]=gyoker;
      z=new KulcsPar();
      z.kulcs=x.kulcs+y.kulcs;
      z.adat=gyoker;
      S.SorBa(z);
   }

   String[] Kodok=new String[n];
   for (int i=0; i<n; i++){
      String kod="";
      int j=i;
      while (j!=gyoker){
         if (Apa[j]<0)
            kod='0'+kod;
         else
            kod='1'+kod;
         j=Math.abs(Apa[j]);
      }
      Kodok[i]=kod;
   }
   return Kodok;
 }
}
```

A példánkban szereplő adatokra a Huffman algoritmus a 5. ábrán látható módon működik. Mivel hat kódolandó karakter van, a sor mérete kezdetben $n = 6$ és 5 összevonási lépés szükséges a fa felépítéséhez. A végén kapott fa megfelel az optimális prefix-kódnak. Minden karakter kódja a gyökértől a megfelelő levélig vezető úton lévő élek címkéinek sorozata.

Az algoritmusban a 2. sor inicializálja a Q prioritási sort a C-beli karakterekkel. A 3-8. sorokban adott ciklus ismétlődően kiválasztja a Q sorból az x és y két legkisebb gyakoriságú csúcsot és beteszi a sorba azt a z új csúcsot, amely x és y összevonását ábrázolja. A z új csúcs gyakorisága x és y gyakoriságának összege lesz, amit a 7. sorban számítunk ki. A z csúcs bal gyereke x, jobb gyereke pedig az y csúcs lesz. (Itt a sorrend nem lényeges, bármely csúcs bal és jobb gyereke felcserélhető, különböző, de azonos költségű fát eredményezve.) $n-1$ számú összevonás végrehajtása után a sorban egy csúcs marad (a kódfa gyökere), az algoritmus a 9. sor végrehajtásával ezt adja eredményül.

A Huffman algoritmus időigényének elemzésénél feltételezzük, hogy a felhasznált prioritási sor absztrakt adattípust úgy valósítjuk meg, hogy a SORBOL és SORBA műveletek futási ideje $O(\lg n)$. A 3-8. sorokban adott ciklus pontosan $(n-1)$ -szer hajtódik végre, és mivel a prioritási sor minden művelete $O(\lg n)$ időt igényel, a ciklus teljes futási ideje $O(n \lg n)$. Tehát a HUFFMAN algoritmus futási ideje $O(n \lg n)$ minden n karaktert tartalmazó C halmazra.



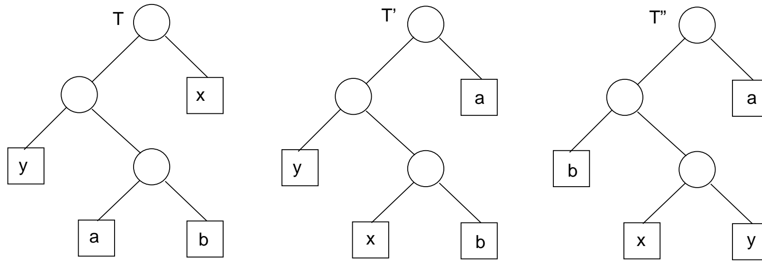
5. ábra. A Huffman algoritmus lépései a 2. ábrán szereplő gyakoriságokra. Minden részábra a sor aktuális tartalmát gyakoriság szerint növekvően. A leveleket téglalapok jelölik. A belső csúcsokat kör jelöli, amelyekben a csúcs két gyereke gyakoriságának összege van. Minden karakter kódszava az a jelsorozat, amelyet úgy kapunk, hogy a gyökértől a karaktert tartalmazó levélig vezető úton az élek címkéit egymás után írjuk. (**a**) A kezdeti állapot $n = 6$ csúccsal. (**b**) - (**e**) A közbülső állapotok. (**f**) A fa az eljárás végén.

A Huffmann algoritmus helyessége

A HUFFMAN mohó algoritmus helyességének igazolásához megmutatjuk, hogy az optimális prefix-kód meghatározása teljesíti a mohó-választási és az optimális részproblémák tulajdonságokat. A következő lemma azt bizonyítja, hogy a mohó-választási tulajdonság teljesül.

8.2. lemma. *Legyen C tetszőleges karakter halmaz, és legyen $f[c]$ a $c \in C$ karakter gyakorisága. Legyen x és y a két legkisebb gyakoriságú karakter C -ben. Ekkor létezik olyan optimális prefix-kód, amely esetén az x-hez és y-hoz tartozó kódszó hossza megegyezik, és a két kódszó csak az utolsó bitben különbözik.*

Bizonyítás. A bizonyítás alapötlete az, hogy vegyünk egy optimális prefix-kódot ábrázoló T fát és módosítsuk úgy, hogy a fában x és y a két legmélyebben lévő testvércsúcs legyen. Ha ezt meg tudjuk tenni, akkor a hozzájuk tartozó kódszavak valóban azonos hosszúságúak lesznek és csak az utolsó bitben különböznek.



6. ábra. A 2. lemma bizonyításának kulcslépése. Az optimális prefix-kód T fájában b és c a két legmélyebb testvércsúcs. Az x és y az a két levél csúcs, amelyet a Huffman algoritmus elsőnek von össze. Ezek bárhol lehetnek a fában. A b és x csúcsok felcserélésével kapjuk a T' fát. Ezután a c és y csúcsokat felcserélve adódik a T'' fa. Mivel egyik lépés hatására sem növekszik a fa költsége, a kapott T'' fa is optimális lesz.

Legyen a és b a T fában a két legmélyebb testvércsúcs. Az általánosság megszorítása nélkül feltehetjük, hogy $f[a] \leq f[b]$ és $f[x] \leq f[y]$. Mivel $f[x] \leq f[y]$ a két legkisebb gyakoriság, valamint $f[a] \leq f[b]$ tetszőleges gyakoriságok, így azt kapjuk, hogy $f[x] \leq f[a]$ és $f[y] \leq f[b]$. A 6. ábrán látható módon felcseréljük a T fában a és x helyét, ezzel kapjuk a T' fát, majd ebből a fából, felcserélve a b és y csúcsok helyét, kapjuk a T'' fát. A (3.) egyenlet szerint a T és a T' fák költségének különbsége

$$\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) - f[a]d_{T'}(a) \\
&= f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) - f[a]d_T(x) \\
&= (f[a] - f[x])(d_T(a) - d_T(x)) \\
&\geq 0.
\end{aligned}$$

Az egyenlőtlenség azért teljesül, mert $f[a] - f[x]$ és $d_T(a) - d_T(x)$ nem-negatív. Pontosabban, $f[a] - f[x]$ nem-negatív, mert x egy legkisebb gyakoriságú karakter, és $d_T(a) - d_T(x)$ azért nem-negatív, mert a maximális mélységű a T fában. Hasonlóan bizonyítható, hogy b és y felcserélése esetén sem növekszik a költség, így $B(T') - B(T'')$ nem-negatív. Tehát $B(T'') \leq B(T)$, és mivel T optimális így $B(T) \leq B(T'')$, tehát $B(T'') = B(T)$. Tehát T'' olyan optimális fa, amelyben x és y maximális mélységű testvércsúcsok, amiből a lemma állítása következik. ■

A 2. lemmából következik, hogy az optimális fa felépítése, az általánosság megszorítása nélkül, kezdhető a mohó választással, azaz a két legkisebb gyakoriságú karakter összevonásával. Miért tekinthető ez mohó választásnak? Azért, mert tekinthetjük a két összevont elem gyakoriságának összegét egy összevonás költségeként. A HUFFMAN algoritmus az összes lehetséges lépések közül mindig azt választja, amelyik a legkisebb mértékben járul hozzá a költséghez.

A következő lemma azt mutatja, hogy az optimális prefix-kód konstrukciója teljesíti az optimális részproblémák tulajdonságot.

8.3. lemma. *Legyen C tetszőleges ábécé, és minden $c \in C$ karakter gyakorisága $f[c]$. Legyen x és y a két legkisebb gyakoriságú karakter C-ben. Tekintsük azt a C' ábécét, amelyet C-ből úgy kapunk, hogy eltávolítjuk az x és y karaktert, majd hozzáadunk egy új z karaktert, tehát $C' = C - \{x,y\} \cup \{z\}$. Az f gyakoriságok C'-re megegyeznek a C-beli gyakoriságokkal, kivéve z esetét, amelyre $f[z] = f[x] + f[y]$. Legyen T' olyan fa, amely optimális prefix-kódját ábrázolja a C' ábécének. Ekkor az a T fa, amelyet úgy kapunk, hogy a z levélcsúcshoz hozzákapcsoljuk gyerek csúcsként x-et és y-t, olyan fa lesz, amely a C ábécé optimális perix-kódját ábrázolja.*

Bizonyítás. Először megmutatjuk, hogy a T fa $B(T)$ költsége kifejezhető a T' fa $B(T')$ költségével a 5. egyenlet alapján. Minden $c \in C - \{x,y\}$ esetén $d_T(c) = d_{T'}(c)$, így $f[c]d_T(c) = f[c]d_{T'}(c)$. Mivel $d_T(x) = d_T(y) = d_{T'}(z) + 1$, így azt kapjuk, hogy

$$\begin{aligned}
f[x]d_T(x) + f[y]d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) \\
&= f[z]d_{T'}(z) + (f[x] + f[y]),
\end{aligned}$$

amiből az következik, hogy

$$B(T) = B(T') + f[x] + f[y].$$

Indirekt módon bizonyítunk. Tegyük fel, hogy T nem optimális prefix-kódfa a C ábécére. Ekkor létezik olyan T'' kódfa C-re, hogy $B(T'') < B(T)$. Az általánosság megszorítása nélkül (a 2. lemma alapján) feltehetjük, hogy x és y testvérek. Legyen T''' az a fa,

amelyet T''-ből úgy kapunk, hogy eltávolítjuk az x és y csúcsokat, és ezek közös z szülőjének gyakorisága az $f[z] = f[x] + f[y]$ érték lesz. Ekkor

$$\begin{aligned} B(T''') &= B(T'') - f[x] - f[y] \\ &< B(T) - f[x] - f[y] \\ &= B(T'), \end{aligned}$$

ami ellentmond annak, hogy T' a C' ábécé optimális prefix-kódját ábrázolja. Tehát T szükségképpen a C ábécé optimális prefix-kódját ábrázolja. ■

8.4. tétel. *A* HUFFMAN *eljárás optimális prefix-kódot állít elő.*

Bizonyítás. Az állítás közvetlenül következik a 2. és a 3. lemmákból. ■