sapientia
tankönyvek

Kátai Zoltán

Algoritmustervezési stratégiák

Scientia Kiadó

Kátai Zoltán

Algoritmustervezési stratégiák

Kátai Zoltán

Algoritmustervezési stratégiák

Scientia Kiadó
Kolozsvár ▪ 2020

sapientia tankönyvek informatika

Felelős kiadó:
Kása Zoltán

Lektor:
Păţcaş Csaba-György (Kolozsvár)

Descrierea CIP a Bibliotecii Naţionale a României
KÁTAI, ZOLTÁN
Algoritmustervezési stratégiák / Kátai Zoltán. - Cluj-Napoca : Scientia, 2020
Conţine bibliografie
ISBN 978-606-975-037-7
004

TARTALOM

CONTENTS

CUPRINS

1. BEVEZETÉS

Comenius, akit a modern oktatás megteremtőjének tartanak, a következő kijelentést tette a tanítási módszereket illetően: „Tanítani szinte nem is jelent mást, mint megmutatni, miben különböznek egymástól a dolgok a különböző céljukat, megjelenési formájukat és eredetüket illetően… Ezért aki jól megkülönbözteti egymástól a dolgokat, az jól is tanít" (Sadler 2013). Jelen könyv elsősorban erre a didaktikai alapelvre épül. Egy olyan tanítási, illetve tanulási módszert ajánl, amely segít a tanulóknak úgymond felülnézetből látni a megvizsgált öt programozási módszert (algoritmustervezési stratégiát):

– mohó keresés (greedy),
– visszalépéses keresés (backtracking),
– oszd-meg-és-uralkodj (divide-and-conquer, divide-et-impera),
– elágazás és korlátozás (branch-and-bound),
– dinamikus programozás (dynamic programming).

Tehát nem csak az a célunk, hogy bemutassuk e módszereket, hanem hogy olyan nézőpontba juttassuk az olvasót, amelyből feltárulnak előtte a technikák közötti elvi, alapvető, sőt árnyalatbeli különbségek, illetve hasonlóságok. A comeniusi alapelvvel összhangban ez nélkülözhetetlen, ha uralni szeretnénk a programozás e területét. A következőkben úgy utalunk erre a megközelítési módra, mint *felülnézetmódszer*.

Algoritmus

Az „algoritmus" kifejezés a bagdadi arab tudós, al-Hvárizmi (780–845) nevének eltorzított, rosszul latinra fordított változatából ered.

Az első, számítógépre kigondolt algoritmust Ada Lovelace írta 1843-ban Charles Babbage analitikai gépére a Bernoulli-számok kiszámítására, tehát ő tekinthető az első programozónak.

1.1. A felülnézetmódszer általános leírása

Mit jelent „felülről látni" valamit? Képzeljük el a következő helyzeteket: a rendőrségen, egy bűneset kapcsán, a különböző forrásokból érkező bizonyítékokat feltűzik egy táblára. Miért? A polgármesteri hivatal városrendezésért felelős szakosztálya elkészíti a város egy makettjét és körbeállják. Miért? Azért, hogy

átfogó képet kapjanak az „egész”-ről, valamint hogy jobban érzékelhetőek legyenek az egyes „részek” közötti hasonlóságok, különbségek, illetve kapcsolatok.

A két esetben különböző módon alakították ki az illetékesek a felülnézetet. A rendőröknek szükségük volt egy olyan „platformra” (a tábla), amelyen elhelyezve a bizonyítékokat, „egymás mellett” láthatták őket. A műépítészek kicsinyítést és absztrakciót használtak a makett elkészítésekor. Az absztrakció azért fontos, mert elvonatkoztat attól, ami a tanulmányozás szempontjából lényegtelen.

A két módszer ötvözéséből látható, hogy egy felülnézet kialakításához szükséges lehet egy úgynevezett „absztrakt platform”, amelyen az elemzett entitások úgy helyezhetők egymás mellé, hogy szembetűnővé váljanak a vizsgálat szempontjából lényeges tulajdonságok és kapcsolatok.

Azt, hogy mit fogunk felülnézet alatt érteni ebben a könyvben, a következőképpen lehetne összefoglalni:

1. „Egymás mellett” látjuk a vizsgálat tárgyát képező entitásokat.
2. Csak az látszik, ami a vizsgálat szempontjából lényeges.
3. Nyilvánvalóak a hasonlóságok és a különbségek, szembetűnők a kapcsolatok.

A módszer célja olyan helyzetbe juttatni a tanulót, hogy ilyen rálátása legyen a tanulmányozás alatt álló anyagra. Természetesen különböző felülnézetek alakíthatók ki attól függően, hogy miként valósítjuk meg az említett lépéseket. Ez kívánatos is, hiszen minden újabb felülnézet egy másik szemszöget jelent. A módszer egyik erősségének számít az is, hogy segít a diákoknak a vizsgált entitások egymáshoz viszonyított „értékét” is felmérni. Például az algoritmustervezési stratégiák tanulmányozása esetén nyilvánvalóvá válnak az egyes technikák erős, illetve gyenge pontjai, valamint az, hogy adott helyzetben melyiknek az alkalmazása a legcélszerűbb és miért.

1.2. Felülnézetek az informatikaoktatásban

Hogyan lehet felülnézeteket kialakítani informatikaórán? A legtöbb tanár megteszi ezt – még ha nem is így nevezi –, amikor a rendezéseket tanítja. A fejezet végén veszünk egy konkrét számsorozatot, amelyen elmímeljük vagy elmímeltetjük a tanulókkal az összes megtanított rendezési algoritmust, és felhívjuk a figyelmet a hasonlóságokra, valamint a különbségekre. Amikor így járunk el, felülnézetet alakítunk ki a diákokban, hiszen:

– „egymás mellé” helyezzük a rendezési algoritmusokat azáltal, hogy ugyanazon a számsorozaton mímeljük el őket;
– felhívjuk a diákok figyelmét arra, hogy egy adott felülnézetből mi lényeges és mi nem; például az összehasonlításon alapuló rendezések esetén arra

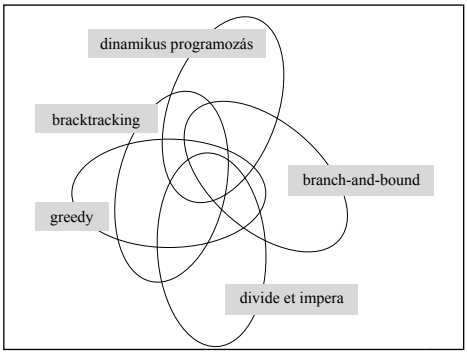
összpontosíthatnak a diákok egy adott felülnézetből, hogy milyen stratégiák szerint történnek az elemek hasonlítgatásai;
– segítünk a tanulóknak meglátni a hasonlóságokat és a különbségeket, az algoritmusok erősségeit és gyenge pontjait.

Kitűnő számítógépes vizualizációk léteznek, amelyek megkönnyíthetik a felülnézet kialakítását a rendezési algoritmusokat illetően. Például a Sapientia Erdélyi Magyar Tudományegyetem Marosvásárhelyi Karán működő AlgoRythmics kutatócsoport Erdélyben honos etnikumok néptáncaival illusztrál számos rendezési algoritmust (AlgoRythmics 2020).

1.3. Algoritmustervezés felülnézetből

Hogyan alakíthatunk ki felülnézeteket az algoritmustervezési stratégiák tanításakor? A kihívás abban áll, hogy amíg a rendezési algoritmusok ugyanazt a feladatot oldják meg, addig minden egyes algoritmustervezési stratégiának megvan – többé-kevésbé – a saját felségterülete. Az 1.1. ábra ezt szemlélteti. Az egyes ellipszisek azon feladatok halmazát ábrázolják, amelyek megoldhatók az illető stratégiával, függetlenül attól, hogy optimális megoldást nyújtanak-e vagy sem, hatékony-e az algoritmus vagy nem.

Az, hogy az ellipszisek metszik egymást, azt szemlélteti, hogy léteznek olyan feladatok, amelyek több technikával is megoldhatók, sőt egyesek az összessel. Ebből arra következtethetünk, hogy léteznie kell egy „sík”-nak, amelyen az egyes technikák feladathalmazai alapvető hasonlóságokat mutatnak. Melyik ez a „sík”? Ahhoz, hogy megtaláljuk a választ erre a kérdésre, be kell vezetnünk a fastruktúra fogalmát.

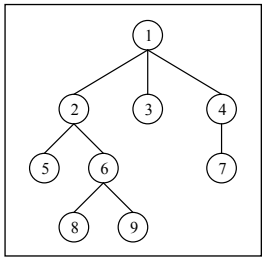


1.1. ábra. *Az algoritmustervezési stratégiák „felségterületei”*[1]

1 Az ábra csak az elvet szemlélteti, az ellipszisek méretének, valamint a metszési felületek nagyságának nincs külön jelentése.

1.4. Fastruktúrák

Egy fastruktúrának (más szóval gyökeres fának[2]) van egy kitüntetett csomópontja, amely a fa nulladik szintjén helyezkedik el, és amit gyökércsomópontnak nevezünk. Az első szinten találhatók a gyökér fiúcsomópontjai, a másodikon ezeknek a fiúcsomópontjai és így tovább. Azokat a csomópontokat, amelyekből nem ágazik le egyetlen fiúcsomópont sem, a fa leveleinek nevezzük. Tehát egy fastruktúra csomópontokból épül fel, és ezek között apa-fiú típusú kapcsolatok vannak. Minden csomópont egyetlen apacsomóponthoz kapcsolódik, amely a közvetlen felette lévő szinten található. Ez alól egyedül a gyökércsomópont kivétel. Ezzel szemben egy csomóponthoz több fiúcsomópont is kapcsolódhat, amelyek a közvetlen alatta lévő szinten helyezkednek el. A fiak között, általában, aszerint teszünk különbséget, hogy – balról jobbra számolva – hányadik fia az apjának (1.2. ábra).



1.2. ábra. *Négyszintes fastruktúra (1: gyökér; 3, 5, 7, 8, 9: levelek); Az élek implicit fentről lefele irányítottak, hisz apa-fiú kapcsolatokat ábrázolnak*

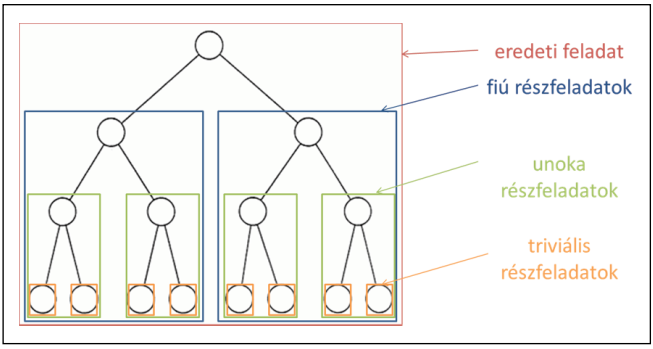
Megfigyelhető, hogy bármely csomópont a leszármazottjaival együtt ugyancsak fa (gyökeres fa). Ez a következő rekurzív definíciót teszi lehetővé: fastruktúrának nevezzük a csomópontoknak egy olyan halmazát és szerkezetét, amelyben létezik egy kitüntetett gyökércsomópont, és a többi csomópont olyan diszjunkt halmazokba van szétosztva, amelyek maguk is fák, és e részfák gyökerei, mint fiak, a fa gyökeréhez kapcsolódnak. Ha egy fastruktúrában mindenik csomópontnak legfennebb két fia van, akkor bináris fáról beszélünk. Ez esetben a fiúcsomópontokat úgy azonosítjuk, mint bal, illetve jobb fiú.

2 A fastruktúra, illetve fa, bináris fa, részfa megnevezések alatt a továbbiakban is gyökeres fákat értünk.

1.5. Egy „absztrakt platform”

Amint látni fogjuk a későbbiekben, az összes bemutatásra kerülő módszert általában olyan feladatok esetében használjuk, amelyek fastruktúrával modellezhetők (lásd *A technikák mint útkereső stratégiák* betétet). E fastruktúra fogja betölteni azon absztrakt platform szerepét, amelyen a technikák egymás mellé helyezhetők, és amely a felülnézet kialakításához szükséges.

Például a visszalépéses keresés és oszd-meg-és-uralkodj technikák esetében nagyon kézenfekvő a rekurzív implementáció (lásd az *Iteratív vs. Rekurzív* betétet). Ez már önmagában is fastruktúrát sugall, hisz a rekurzív gondolatmenet a következő: Hogyan vezethető vissza a feladat hasonló, egyszerűbb részfeladatokra, majd ezek további hasonló, még egyszerűbb részfeladatokra, egészen addig, míg triviális részfeladatokhoz nem jutunk? A teljes fa (amelyet a gyökércsomópont képvisel) nyilván magát a feladatot ábrázolja, a fiúrészfák (amelyeket az első szinti fiúcsomópontok képviselnek) azokat a részfeladatokat, amelyekre a feladat első lépésben lebontható, és így tovább. Végül a fa levelei fogják ábrázolni a lebontásból adódó triviális részfeladatokat (lásd az 1.3. ábrát).



1.3. ábra. *Fastruktúra szerkezetű feladat*

A mohó, dinamikus programozás és branch-and-bound technikát gyakran olyan feladatok esetében alkalmazzuk, amelyek döntéssorozatként foghatók fel. Ez a megközelítés megint csak egy fastruktúrához vezet, amelyben a gyökér a feladat kezdeti állapotát jelképezi, az első szint csomópontjai azokat az állapotokat, amelyekbe a feladat az első döntés nyomán kerülhet, a második szinten lévők azokat, amelyek a második döntésből adódhatnak stb. Egy csomópontnak annyi fia lesz, ahány lehetőség közül történik a választás az illető döntés alkalmával.

Eddig arra összpontosítottunk, hogy mi a közös az egyes technikákban, de a felülnézet azt is jelenti, hogy nyilvánvalóak a köztük lévő különbségek is. Amint

látni fogjuk, az egyik ilyen különbség az, ahogyan bejárják a feladatnak megfeleltethető fát. Tekintettel azon olvasóinkra, akiknek még nincsenek gráfelméleti ismereteik, az alábbiakban bemutatjuk a gyökeres fák azon bejárási módjait, amelyekre a későbbi fejezetekben gyakran hivatkozunk majd.

1.6. A fastruktúrák bejárása

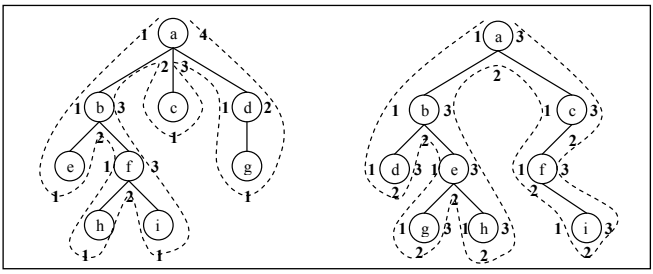
Mit jelent bejárni egy gyökeres fát? Alapvetően azt, hogy a gyökértől indulva (az élek mentén haladva), egy jól meghatározott sorrendben meglátogatjuk a csomópontjait úgy, hogy mindegyiket csak egyszer dolgozzuk fel. Két alapvető bejárás a mélységi és a szélességi.

1.6.1. Mélységi bejárás (DF – Depth First)

A mélységi bejárás a következő rekurzív algoritmust követi: a fa mélységi bejárását lebontjuk a gyökér fiúrészfáinak a mélységi bejárására. A mellékelt ábrára (1.4. ábra) vonatkoztatva ez azt jelenti, hogy indulunk a gyökérből, majd pedig „balról jobbra” haladva sorra bejárjuk a gyökér minden egyes fiúrészfájának összes csomópontját. A rekurzióból adódóan az egyes fiúrészfák bejárása hasonlóképpen megy végbe: indulunk az illető részfa gyökércsomópontjából, majd „balról jobbra” sorrendben bejárjuk ennek a fiúrészfáit. Ugyancsak a rekurzió következménye, hogy miután bejártuk egy csomópont összes fiúrészfáját, visszalépünk az apacsomópontjához, és folytatjuk a bejárást e csomópont következő fiúrészfájával (amennyiben ez létezik). A fentiekkel összhangban úgy foghatjuk fel a mélységi bejárást, mintha körbejárnánk a fa koronáját, szorosan követve annak vonalát.

Az 1.4. ábra azt is jól érzékelteti, hogy a bejárás során az egyes csomópontokat többször is érintjük. Egészen pontosan, ha egy csomópontnak f fia van, akkor $f + 1$ alkalommal találkozunk vele a fa mélységi bejárása során. Attól függően, hogy melyik találkozáskor „látogatjuk meg”, vagyis mikor dolgozzuk fel a csomópontban tárolt információt, megkülönböztetünk preorder (első érintéskor látogatunk), illetve postorder (utolsó érintéskor látogatunk) mélységi bejárásokat. Bináris fák esetén beszélhetünk inorder bejárásról is, amikor a két fiúrészfa bejárása közötti érintéskor látogatjuk meg a csomópontokat. Az 1.4.a. ábra egy általános fa, a 1.4.b. egy bináris fa bejárásait szemlélteti.

Megfigyelhető, hogy preorder bejárás esetén a bejárás előre-szakaszain foglalkozunk a csomópontokkal, postorder bejáráskor viszont a visszautakon. Egy másik különbség, hogy preorder esetben először meglátogatjuk a csomópontot, és azután járjuk be ennek fiúrészfáit, postorder esetben pedig fordítva, először bejárjuk a csomópont fiúrészfáit, és csak azután látogatjuk meg magát a csomópontot.



1.4. ábra. *(a) Általános gyökeres fa mélységi bejárásai (preorder sorrend: a, b, e, f, h, i, c, d, g; postorder sorrend: e, h, i, f, b, c, g, d, a); (b) Bináris fa mélységi bejárásai (preorder sorrend: a, b, d, e, g, h, c, f, i; inorder sorrend: d, b, g, e, h, a, f, i, c; postorder sorrend: d, g, h, e, b, i, f, c, a); (a számok azt jelzik, hogy hányadik érintése az illető csomópontnak)*

Az alábbiakban megadjuk a bemutatott mélységi bejárásokat implementáló rekurzív eljárásokat [mindegyik az aktuális (cs)omóponton dolgozik]:

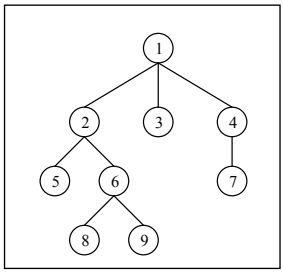
```
preorder(cs)
   meglátogat(cs)
   minden f_i fiára cs-nek végezd
      preorder(f_i)
   vége minden
vége preorder

postorder(cs)
   minden f_i fiára cs-nek végezd
      postorder(f_i)
   vége minden
   meglátogat(cs)
vége postorder

inorder(cs)
   inorder(f_bal)
   meglátogat(cs)
   inorder(f_jobb)
vége inorder
```

Mindhárom eljárást természetesen a gyökércsomópontra hívjuk meg:

```
preorder(gyökér)
postorder(gyökér)
inorder(gyökér)
```

1.6.2. Szélességi bejárás (BF – Breadth First)

A szélességi bejárás a következő gondolatmenetet követi: meglátogatjuk a gyökércsomópontot, ezután ennek a fiúcsomópontjait („balról jobbra” sorrendben), majd ezeknek a fiúcsomópontjait, … Ahogy az 1.5. ábra is szemlélteti, szintről szintre haladunk a fában.



1.5. ábra. *Általános gyökeres fa szélességi bejárása (szélességi bejárás szerinti sorrend: 1, 2, 3, 4, 5, 6, 7, 8, 9)*

A szélességi bejárás implementálásához egy várakozási sor szerkezetet (Q) használunk (ellentétben a mélységi bejárással, amely – összhangban rekurzív voltával – verem szerkezetre épül: LIFO – Last In First Out). A várakozási sor struktúrára az jellemző, hogy betevéskor az elemek a sor végére kerülnek, a kivétel pedig a sor elejéről történik. Ebből adódik, hogy ugyanabban a sorrendben történik az elemek eltávolítása a sorból, mint amilyen sorrendben betettük őket (FIFO – First In First Out).

```
szélességi_bejárás(gyökér)
   Q = ∅
   betesz(Q,gyökér)
   amíg (nem üres(Q)) végezd
      cs = kivesz(Q)
      meglátogat(cs)
      minden f_i fiára cs-nek végezd
         betesz(Q,f_i)
      vége minden
   vége amíg
vége szélességi_bejárás
```

A bejárások úgy is tekinthetők, mint módszerek a feladat szerkezetét ábrázoló fastruktúra felderítéséhez/generálásához. A kiindulópont általában egy irá-

nyított gráf[3] (reprezentációs gráf), amelyből valamely bejárás nyomán nyerjük a keresési térnek is nevezett gyökeres fát (lásd *A technikák mint útkereső stratégiák* című betétet). A bejárás úgy is felfogható, mint a reprezentációs gráf felderítése. A bejárt rész, a már felderített rész (kezdetben csak a gyökér), a többi pedig a még felderítésre váró terület. A bemutatott bejárási módszerek abban különböznek, hogy milyen stratégia szerint ágaztatják tovább a már feltárt részfa koronája peremét képező csomópontokat. A DF stratégia mindig a gyökértől legtávolabbi, a BF pedig a legközelebbi csomóponttól lép elsőként tovább (a fenti példákban az ugyanolyan távolságú pontokat a címkéjük alapján rangsoroltuk). Mindez azt feltételezi, hogy folyamatosan nyilvántartjuk legalább a növekvő korona peremén lévő csomópontokat. Ezt teszi a DFS (depth-first search) algoritmus a veremben, a BFS (breadth-first search) pedig a Q sorban. Amikor egy pont (be/ki)kerül a verem(be/ből)/sor(ba/ból), akkor úgymond (fel/le)kerül a korona peremé(re/ről).

A technikák mint útkereső stratégiák

Mindenik megvizsgálásra kerülő technika kapcsán olyan feladatokkal foglalkozunk, amelyek *problématere* egy úgynevezett *reprezentációs gráf* adott csúcsából (startcsúcs) induló irányított utak halmazaként fogható fel, amelyek között olyan utat keresünk, amely adott csúcsok (célcsúcsok) egyikébe vezet. Gyakran az optimális út megtalálása a célunk. Általános módszer egy feladat útkeresési feladattá való átfogalmazására az *állapottér-reprezentáció*. Ez négy dolog pontos megadását jelenti (ami egy állapotgráfot eredményez): a feladat állapotterét (csúcsok halmaza), a műveleteit (irányított élek), a kezdőállapotait (startcsúcsok) és egy kezdőállapothoz tartozó célállapotait (célcsúcsok) (Gregorics 2014). A feladat: a startcsúcsból indulva találjunk célcsúcsba vezető utat/utakat, esetleg optimális utat.

A különböző programozási technikák különböző keresési stratégiákra vetítődnek le. Ezek lényeges tulajdonsága, hogy végeznek-e *duplikátumfigyelést*, azaz egy állapot elérésekor vizsgálják-e, hogy azt már korábban felfedezték-e (Gregorics 2014). Ez az ellenőrzés időigényes lehet, vagy akár lehetetlen is, amennyiben nem kerültek eltárolásra a korábban már felfedezett állapotok. Duplikátumfigyelés hiányában viszont torzult formájában érzékelik a keresések az állapotgráfot. Az állapotgráf irányított fává (lásd a fastruktúra definícióját; 1.6. alpont) egyenesedik ki, amelyben a startcsúcs lesz a gyökér, az ágak pedig a startcsúcsból kiinduló utakat ábrázolják. A fában egy állapotot annyi darab csúcs képvisel, ahányféleképpen el lehet jutni az illető állapotba a kezdőállapotból. Ha körök is voltak az állapotgráfban, akkor a fa

3 Egy irányított gráf csomópontok és az ezeket összekötő irányított élek halmazaiként fogható fel. Egy gráf tartalmazhat köröket. A körmentes összefüggő gráfokat nevezzük fáknak. A fastruktúrát, amely egy gyökeres irányított fának tekinthető, az 1.4. alfejezetben definiáltuk.

végtelen hosszú ágakat fog tartalmazni. Ezt a fát, amelyet a „keresés az állapotgráfból lát”, *keresési térnek* nevezik, amelyben a startcsúcsból kivezető utak alapvetően ugyanazt a problémateret jelölik ki, mint az állapotgráf. Ha a keresési tér identikus állapotokat képviselő csúcsait egymásra csúsztatjuk, akkor visszanyerjük az állapotgráfot.

Egy másik közös, kapcsolódó fogalom a *globális munkaterület* (Gregorics 2014), *amely* annak az információnak a tárolására szolgál, amelyet a technikák a keresés során megszereznek, és megőrzésre fontosnak ítélnek. Ez a reprezentációs gráfnak a startcsúcsból elérhető része, amelyet a keresés felfedezett és megjegyzett. Ez lehet csak egyetlen csúcs, esetleg annak környezete, lehet egy út, amelyik a startcsúcsból az aktuálisan vizsgált csúcsba vezet, de lehet egy ennél tágabb részgráf is. Kezdetben a globális munkaterület a startcsúcsot tárolja. A megoldások megtalálása kéz a kézben jár azzal, hogy célcsúcs jelenik meg a globális munkaterületen. Annál céltudatosabb a keresés, minél kevesebbet kell hogy feltárjon a keresési térből ahhoz, hogy célcsúcsba érjen.

A függelékben bemutatjuk három híres probléma (N-királynő, 3. fejezet; Hanoi tornyai, 4. fejezet; Tili-toli, 7. fejezet) állapottér-reprezentációját.

Iteratív vs. rekurzív

Minden jelentősebb feladat megoldása részfeladatok ismételt elvégzését feltételezi. Ha ciklusutasítás révén valósítjuk meg az ismétlést, akkor iteratív implementációról beszélünk. Egy másik lehetőség a rekurzív megvalósítás. A rekurzív eljárások/függvények sajátossága, hogy meghívják önmagukat. Olyan, mintha klónoznának magukból további példányokat. Úgy is mondhatnánk, hogy a rekurzivitás mechanizmusa révén az illető eljárásból/függvényből példányok sora jön létre, amelyek között szétosztódik a megoldandó feladat. Mindegyik példány bevállal egy adott részt a kapott feladatból, a maradék részfeladatot pedig egy következő példányra ruházza át. Mivel az átruházás egy klónra történik, ezért a kapott feladatnak és az átruházott részfeladatnak hasonlónak kell lennie. A rekurzív implementálás kulcskérdése tehát az, hogy miként vezethető vissza a kapott feladat megoldása hasonló, egyre egyszerűbb részfeladatok megoldására. E redukálási folyamat révén végül annyira leegyszerűsödik a feladat, hogy a sorvégi példány egészében be tudja vállalni azt. Összegezve:

– Minden eljárás/függvény-példány hasonló feladatot kap, hogy megoldjon.
– A kapott feladat „méretétől” függ, hogy a kurrens példány rekurzívan fog-e viselkedni, vagy bevállalja a teljes megoldást.
    • Rekurzívan viselkedni azt jelenti, hogy önmaga meghívása által a kapott feladat oroszlánrészét (hasonló, egyszerűbb részfeladatot) átruházza egy következő példányra (a fennmaradt részt bevállalja).
– Egyszerűsített, általános forgatókönyv a kurrens példányra megfogalmazva:

```
rekuzív_eljárás (<kapott feladat paraméterlistája>)
   ha <triviálisan egyszerű> akkor
      [teljes feladat megoldása]
   különben
      rekuzív_eljárás (<átruházott részfeladat paraméterlistája>)
      [saját rész megoldása]
   vége ha
vége rekuzív_eljárás
```

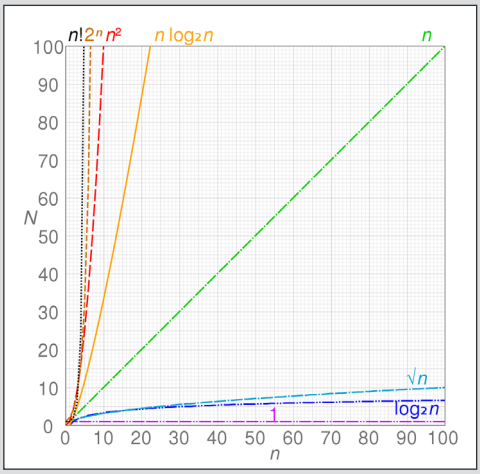
A rekurzív hívások láncolata generálta redukálási folyamatnak konvergálnia kell („el kell találnia”) a trivialitási (megállási) feltételéhez. Ezzel elkerülhető a végtelen rekurzió.

Algoritmusok bonyolultsága (komplexitása)

Alkalmazva a bemutatásra kerülő algoritmustervezési stratégiákat, különböző hatékonyságú algoritmusokhoz jutunk. Egy algoritmus hatékonysága egyik fokmérője az időbonyolultsága. Egyszerűen fogalmazva ez azt jelenti, hogy a bemenet mérete függvényében meghatározzuk az elvégzésre kerülő elemi műveletek számát (ami kéz a kézben jár az algoritmus időigényével). Hogy mit tekintünk a bemenet méretének, az feladatfüggő. Hasonlóképpen az is, hogy mit választunk elemi műveletnek. Például egy összehasonlításon alapuló rendezési algoritmus esetében ajánlatos bemenetméretnek a rendezendő sorozat elemei számát (jelöljük n-nel), elemi műveletnek pedig két elem összehasonlítását választani. Bizonyos gráfalgoritmus esetében célszerű a csúcs- és élszámot bemenetméretnek tekinteni.

Mivel az elvégzésre kerülő elemi műveletek száma bemenetfüggő, ezért bevezették a legjobb, legrosszabb és átlagos eset fogalmát. A növekvő sorrendbe rendezésre nézve a legjobb eset nyilván az, ha egy már növekvő sorozatot kell hogy rendezzen az algoritmus. Hány összehasonlításból tud rájönni az algoritmus, hogy már rendezett a sorozat? A legrosszabb eset az, amikor a bemeneti sorozat csökkenő. Átlagos esetnek számít, ha egy véletlen sorozatot kell rendezni. Például a buborékrendezés elnevezésű algoritmus (lásd a 2. fejezetet) (n-1) összehasonlításból tudja kideríteni, hogy a sorozat már rendezett. Ezzel szemben egy csökkenő sorozat rendezése $n(n\text{-}1)/2$ összehasonlításba (és cserébe) kerül neki. Általában a legrosszabb esetbeli viselkedést tekintjük egy algoritmus hatékonysága fokmérőjének. Mivel a buborékrendezés esetében, legrosszabb esetben az elvégzendő elemi műveletek száma négyzetes függvénye a bemenet méretének, ezért azt mondjuk, hogy időbonyolultsága $\theta(n^2)$ (a futási idő növekedési rendje négyzetes; csak a növekedés domináns fokmérőjének számító tagot tekintjük: $\frac{1}{2}\,\boldsymbol{n^2} - \frac{1}{2}\,n$). Ha csak azt szeretnénk kifejezni, hogy az elvégzendő elemi műveletek száma *legfennebb* négyzetes függvény szerint függ a bementet méretétől, akkor az $O(n^2)$ jelölést használjuk. Látni fogjuk, hogy

vannak ennél hatékonyabb rendezési algoritmusok is, amelyek időbonyolultsága $\theta(n\log n)$ (a *log* itt 2-es alapú logaritmust jelöl).



1.6. ábra. *Néhány növekedési rend függvényhatékonyság szerint sorba állítva (n: a bement mérete; N: elemi műveletek száma)*

Forrás: *https://en.wikipedia.org/wiki/Time_complexity*

Hasonló módon beszélhetünk egy algoritmus tárhelybonyolultságáról is. Mekkora az algoritmus tárhelyigénye a bemenet mérete függvényében? Mivel a buborékrendezés egy n elemű sorozatot egy n elemű tömb felhasználásával rendez, ezért tárhelybonyolultsága $\theta(n)$.

1.7. A könyv felépítése

E bevezetőt követően, a 2. fejezet egy „felülnézet-kóstolót” nyújt a bemutatásra kerülő algoritmustervezési stratégiákból, azáltal, hogy ugyanazon feladaton szemlélteti őket. A következő öt fejezet lehetővé teszi a módszerekben való elmélyülést (3. Backtracking; 4. Oszd meg és uralkodj; 5. Mohó módszer; 6. Branch-and-bound; 7. Dinamikus programozás). Igyekeztünk mindenik technika esetében egy-egy történelmi példával indítani, recepteket sugallni a használatukhoz, illetve több megoldott feladaton is szemléltetni, hogy miként alkalmaz-

hatók különböző helyzetekben. Gyakran utalunk olyan további példákra, amelyek az *Algoritmusok felülnézetből* egyetemi jegyzetben találhatók (Kátai 2007).

A felülnézet kialakítása érdekében újra és újra olyan feladatok kerülnek terítékre, amelyek több módszerrel is megoldhatók. Ilyenkor explicite hangsúlyozzuk a stratégiák közti hasonlóságokat, különbségeket és kapcsolatokat. Az utolsó fejezet egyfajta „felülnézet visszapillantásként" is felfogható, amikor is a híres utazóügynök-problémán mutatjuk be, váll váll mellett, az öt stratégiát.

1.8. A könyvben használt pszeudokód nyelv leírása

Az algoritmusok leírására az alábbi pszeudokód nyelvet használjuk:

MŰVELETEK (OPERÁTOROK)
  Aritmetikai operátorok:
    +, -, * , / , % (egész osztási maradék)
    Megjegyzés: Ha mindkét operandus egész szám, a / operátor egész osztást, különben valós osztást jelent.
  Összehasonlítási operátorok:
    <, ≤ , > , ≥ , == (egyenlőségvizsgálat) ,≠ (különbözőségvizsgálat)
  Logikai operátorok:
    és, **vagy** , **nem**

MEGJEGYZÉSEK (KOMMENTEK)
  Ha egy algoritmus valamely sorát dupla backslash jel (//) előzi meg, akkor megjegyzésnek tekintendő.

MŰVELETEK
  Értékadás:
    <változó> = <kifejezés>
  Elágazás:
    ha <feltétel> **akkor**
      <műveletek1>
    különben
      <műveletek2>
    vége ha
  Elöltesztelő ciklus:
    amíg <feltétel> **végezd**
      <műveletek>
    vége amíg

Hátultesztelő ciklus:
végezd
<műveletek>
amíg <feltétel>
Megjegyzés: Mindkét **amíg** ciklusból akkor lépünk ki, ha a feltétel hamissá vált.
Ismert lépésszámú ciklus:
minden <változó> = <kezdőérték>,<végsőérték>,<lépés> **végezd**
<műveletek>
vége minden
Megjegyzés: Ha a lépésszám hiányzik, implicit 1-nek tekintjük.

Beolvasási művelet:
be: <változó lista>
Kiírási művelet:
ki: <kifejezés lista>

KONSTANSOK (PÉLDÁK)
13, −524 (egész)
−12.027, 0.22 (valós)
'A', 'c', '1', '!' (karakterek)
"informatika", "1802", "Neumann Janos" (karakterlánc)
IGAZ, HAMIS (logikai)

ADATSZERKEZETEK
Egydimenziós tömb (vektor) (példák):
a[] – egydimenziós tömb
a[1..n] – egydimenziós tömb, amelynek elemei 1-től n-ig vannak indexelve
a[i] – hivatkozás egy egydimenziós tömb i-edik elemére
Kétdimenziós tömb (példák):
b[][] – kétdimenziós tömb
b[1..n][1..m] – kétdimenziós tömb, amelynek sorai 1-től n-ig, oszlopai pedig 1-től m-ig vannak indexelve
b[i][j] – hivatkozás egy kétdimenziós tömb i-edik sorának, j-edik oszlopbeli elemére
Bejegyzés (rekord, struktúra) (példák):
$r.m$ – hivatkozás az r bejegyzés típusú változó m mezőjére
a[i].x – az a bejegyzés típusú tömb i-edik elemének az x mezője

ELJÁRÁS
<eljárás neve> (<formális paraméter lista>)
<műveletek>
vége <eljárás neve>

Megjegyzés: Egy eljárásnak lehet több visszatérési pontja is (tartalmazhat üres return utasítást).

FÜGGVÉNY
  <függvény neve> (<formális paraméter lista>)
    <műveletek>
    return <eredmény>
  vége <függvény neve>

Megjegyzés: Egy függvénynek lehet több visszatérési pontja is.

PARAMÉTERÁTADÁS

A formális paraméterek listájában jelezni fogjuk, mely paraméterek egyszerű típusúak és melyek tömbök. Az érték szerinti és cím szerinti paraméterátadás között úgy teszünk különbséget, hogy az utóbbi esetében a formális paramétereket félkövér karakterekkel írjuk.

Példa:

```
eljárás(x[], y[][], a, b, c[])
  ...
vége eljárás
```

1. megjegyzés: *a* és *b* egyszerű típusú változók, *x* és *c* egydimenziós tömbök, y pedig kétdimenziós tömb. Az *x*, *y* és *b* paraméterek érték szerint, *a* és *c* cím szerint kerül átadásra.

2. megjegyzés: Amikor tömböket adunk át, ha az algoritmus szempontjából nem lényeges, melyik fajta paraméterátadást használjuk, akkor az implementálásnál – memóriatakarékossági megfontolásból – célszerű a cím szerinti paraméterátadást használni.

2. ÁLTALÁNOS KÉP AZ ALGORITMUSTERVEZÉSI STRATÉGIÁKRÓL

Mielőtt rátérnénk az öt stratégia madártávlatból való bemutatására, hadd lássuk, mit jelent a stratégia hiánya.

2.1. A nyers erő megközelítés (avagy stratégia híján)

A nyers erő (brute force) módszere általában a legkézenfekvőbb algoritmust jelenti, amelyet a feladat megfogalmazása vagy a hivatkozott fogalmak definíciója elsőre sugall (Levitin 2008). Ezek a megoldások általában kevésbé hatékonyak, hiszen, különösebb stratégia hiányában, nagy erőforrás (idő, tárhely) igényűek lehetnek a számítógépre nézve. Például ha érdekeltek vagyunk az x^n érték kiszámításában, akkor a nyers erő módszere szerinti algoritmus, az $x^n = x \cdot x \cdot \ldots \cdot x$ definíció értelmében, az alábbi:

```
hatvány(x,n)
   p = 1
   minden i = 1,n végezd
      p = p * x
   vége minden
   return p
vége hatvány
```

A fenti algoritmus időbonyolultsága nyilvánvalóan $\theta(n)$. Látni fogjuk a 4. fejezetben, hogy létezik hatékonyabb megoldás is erre a feladatra, amennyiben van stratégiánk.

2.1.1. Kiválasztó rendezés

Tegyük fel, hogy egy számsorozatot (amelyet az $a[1..n]$ tömb tárol) szeretnénk növekvő sorrendbe rendezni. A legkézenfekvőbb módszer, hogy az $a[1..n]$ tömbszakasz legkisebb elemét előrehozzuk az $a[1]$ pozícióba (csere által), majd hasonlóképpen járunk el az $a[2..n]$, $a[3..n]$, …, $a[n-1..n]$ tömbszakaszokkal is. (A min_index(a,i,n) függvényhívás visszatéríti az $a[i..n]$ tömbszakasz legkisebb elemének indexét.)

```
kiválasztásos_rendezés(a[],n)
   minden i = 1,n−1 végezd
      j = min_index(a,i,n)
      csere(a[i],a[j])
   vége minden
vége kiválasztásos_rendezés
```

Mivel a min_index függvény $(n-1)$-szer kerül végrehajtásra, és az i. hívása $(n-i)$ összehasonlítást feltételez, ezért az elvégzett elemi műveletek száma négyzetes függvénye n-nek. A kiválasztásos rendezés sajátossága, hogy legjobb esetben (növekvő számsorozatra futtatjuk) is megmarad az $\theta(n^2)$ bonyolultsága (bár egyetlen cserére sem kerül sor, a szükséges összehasonlítások száma ugyanannyi marad).

2.1.2. Buborékrendezés

Gyakori, hogy egy algoritmus nyers változata tovább finomítható minimális erőráfordítással. Erre példa a buborékrendezés. Az is egy kézenfekvő megközelítése a rendezési feladatnak, hogy az $a[1..n]$ tömbszakasz legnagyobb elemét kibuborékoltatjuk az $a[n]$ pozícióba (a szomszédos elemek összehasonlítása és esetleges cseréje által; lásd lentebb), majd hasonló módon teszünk az $a[1..n-1]$, $a[1..n-2]$, …, $a[1..2]$ tömbszakaszokkal is. (A végérebuborékoltat(a,i) eljáráshívás, az $a[1..i]$ tömbszakasz legnagyobb elemét buborékoltatja ki az $a[i]$ pozícióba.)

```
buborékrendezés(a[],n)
   minden i = n,2,−1 végezd
      végérebuborékoltat(a,i)
   vége minden
vége buborékrendezés

végérebuborékoltat(a[],i)
   minden j = 1,i − 1 végezd
      ha a[j]>a[j+1] akkor
         csere(a[j],a[j+1])
      vége ha
   vége minden
vége végérebuborékoltat
```

Mivel a végérebuborékoltat függvény $(n-1)$-szer kerül végrehajtásra, és az i. hívása $(i-1)$ összehasonlítást feltételez, ezért az elvégzett elemi műveletek száma négyzetes függvénye n-nek (időbonyolultsága $\theta(n^2)$).

Megfigyelhető, hogy a fenti algoritmus azt előlegezi meg, hogy minden végérebuborékoltatással csak az illető szakasz legnagyobb eleme kerül a helyére. Hogyan finomítható ez a változat? A végérebuborékoltat eljárás kurrens meghívása térítse vissza az utolsó csere helyét (index_utolsó_csere), hogy a következő hívás már csak az *a*[1.. index_utolsó_csere] tömbszakaszt kelljen végigjárnia (az utolsó csere mögötti elemek már mind a helyükön vannak). Ha egyetlen csere sem történt, akkor a visszatérített érték legyen 0.

```
finomított_buborék_rendezés(a[],n)
   vég = n
   amíg vég > 0 végezd
      vég = végérebuborékoltat(a,vég)
   vége minden
vége finomított_buborék_rendezés

végérebuborékoltat(a[],vég)
   index_utolsó_csere = 0
   minden j = 1,vég−1 végezd
      ha a[j]>a[j+1] akkor
         csere(a[j],a[j+1])
         index_utolsó_csere = j
      vége ha
   vége minden
   return index_utolsó_csere
vége végérebuborékoltat
```

Nem nehéz átlátni, hogy a finomított változat időbonyolultsága rendezett inputra $\theta(n)$. Amint látni fogjuk, a rendezési feladatokra is léteznek $\theta(n^2)$ időbonyolultságnál hatékonyabb algoritmusok, amennyiben van stratégiánk.

2.2. A stratégiák bemutatkozása

A bevezető fejezetben bemutatott felülnézetmódszer alkalmazásának első lépése a vizsgálat tárgyát képező entitások – jelen esetben a programozási stratégiák – egymás mellé helyezése. A legegyszerűbben ezt úgy tudjuk megtenni, ha ugyanazt a feladatot oldjuk meg mind az öt módszerrel. Például sokatmondó felülnézet alakítható ki, ha feladatnak az alábbi optimalizálási problémát választjuk.

Háromszög: Egy n soros négyzetes mátrix főátlóján és főátló alatti háromszögében természetes számok találhatók. Feltételezzük, hogy a mátrix egy a nevű kétdi-

menziós tömbben van tárolva. Határozzuk meg azt a „legrövidebb” utat, amely az $a[1][1]$ elemtől indul és az n-dik sorig vezet, figyelembe véve a következőket:

– egy úton az $a[i][j]$ elemet az $a[i+1][j]$ elem (függőlegesen le) vagy az $a[i+1][j+1]$ elem (átlósan jobbra le) követheti, ahol $1 \leq i < n$ és $1 \leq j < n$.

– egy út „hossza” alatt az út mentén található elemek összegét értjük.

A 2.1. ábrán látható mátrixban a legrövidebb utat, amely a csúcsból az alapra vezet, besatíroztuk. Ennek az útnak a hossza 32:

7				
9	5			
1	99	4		
21	7	33	17	
2	15	8	3	1

2.1. ábra. *Példamátrix a háromszög feladathoz*

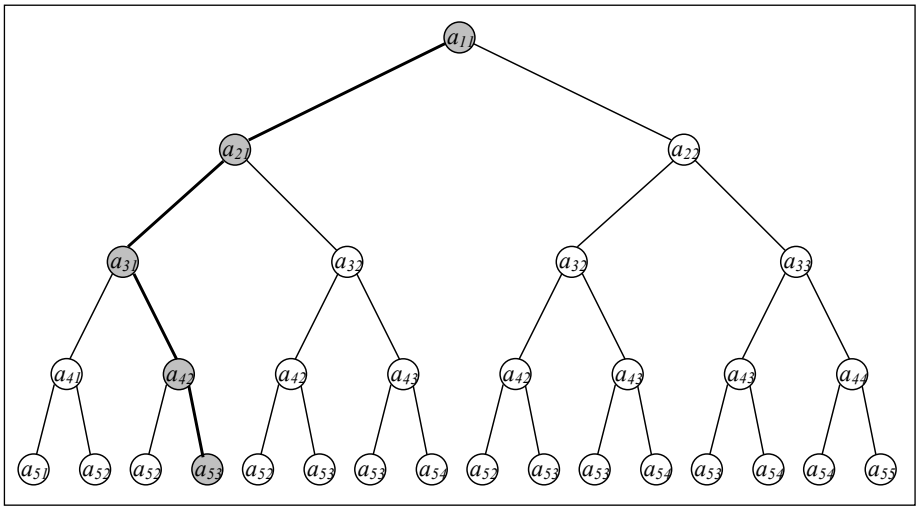
A felülnézet kialakításának második lépése az absztrakció. Az előző fejezetben kifejtettük, hogy ez nem jelent egyebet, mint azonosítani a feladat fastruktúráját (lásd továbbá az 1. fejezet *A technikák mint útkereső stratégiák* című betétjét).

Elemezve a feladatot észrevehetjük, hogy egy olyan optimalizálási problémáról van szó, amelyben az optimális megoldáshoz $n-1$ döntés nyomán juthatunk el, és mindenik döntésnél 2 választásunk van (melyik irányba lépjünk tovább, függőlegesen le vagy átlósan jobbra). Minden döntéssel a feladat egy hasonló, de egyszerűbb feladatra redukálódik. Tehát az absztrakt platform szerepét a 2.2. ábrán látható bináris fa tölti be.

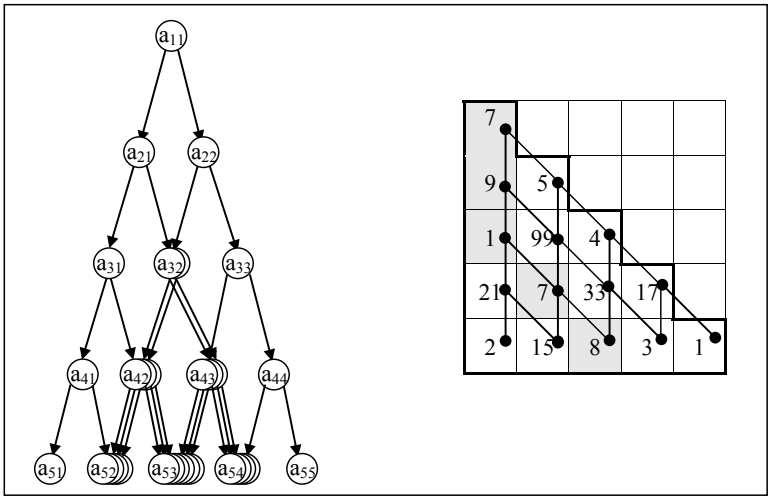
Az optimális megoldás meghatározása az optimális döntéssorozat megtalálását jelenti. Úgy is mondhatnánk, hogy meg kell találnunk a feladat keresési terét képező fastruktúra 2^{n-1} darab, gyökértől levélhez vezető útja közül a „legjobbat”. Más szóval, meg kell keresnünk a fa „legjobb levelét”, azt, amelyikhez a „legjobb út” vezet. (Minden gyökér–levél út *megoldásnak* tekinthető, a „legjobb” pedig az *optimális megoldásnak*.)

A keresési tér egy alaposabb vizsgálata további észrevételekhez vezethet el:

1. A fa csomópontjainak száma, $1+2+2^2+\ldots+2^{n-1}=2^n-1$. Ez azt jelenti, hogy bármely algoritmus, amely generálja/bejárja a teljes fát ahhoz, hogy megtalálja az optimális utat, exponenciális bonyolultságú lesz.
2. Amíg a fa a teljes feladatot képviseli, addig a részfái azokat a hasonló, de egyszerűbb részfeladatokat (a levelek a triviális részfeladatokat), amelyre ez lebontható. Konkrétan: az a_{ij} gyökerű részfa az $a[i][j]$ elemtől az alapra vezető „legrövidebb út” meghatározásának feladatát ábrázolja.



2.2. ábra. *A példamátrix mögött megoldástérként meghúzódó bináris fa (a feladat keresési tere)*

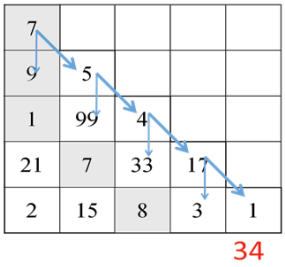


2.3. ábra. *A feladat állapotterét képviselő összevont döntési fa*

3. A 2.2. ábra azt is kiemeli, hogy különböző döntéssorozatok azonos részfeladatokhoz vezethetnek, ami azt jelenti, hogy a fának vannak azonos részfái. Ha egymásra csúsztatjuk ezeket, akkor visszakapjuk a feladat állapotterét ábrázoló irányított gráfot (2.3. ábra). Nem nehéz átlátni, hogy a különböző részfeladatok száma azonos a mátrix elemeinek számával, azaz $n(n+1)/2$. Tehát az algoritmus, amelynek sikerül elkerülni az azonos részfeladatok többszöri megoldását, négyzetes bonyolultságú lesz.

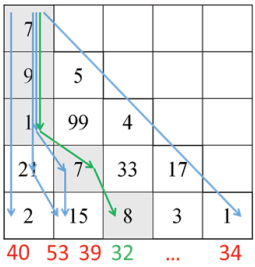
A következőkben meghívjuk az öt technikát egy képzeletbeli *talkshow*-ra, hadd mutatkozzanak be ők maguk, elmagyarázva, miként közelítenék meg a bemutatott feladatot.

Greedy: Jelszavam, hogy *„élj a mának!”*. Ha minden napból sikerül a legtöbbet kihoznom, akkor remélhetőleg a lehető legtartalmasabb életem lesz. Jelen esetben ez azt jelenti, hogy indulva a csúcsból, minden lépésben a kisebbik elem irányába lépek tovább. Elvégre a legkisebb összegben vagyunk érdekeltek. [A példamátrix esetében a mohó-út 34 hosszú lesz, és ez a következő: (1,1),(2,2),(3,3),(4,4),(5,5).]



2.4. ábra. *A mohó döntéssorozat nem eredményez optimális megoldást*

Backtracking: Greedy „élj a mának!” filozófiája nem eredményez optimális megoldást, ha a per-pillanat legígéretesebb választás elvág jövőbeni „nagy lehetőségektől”, vagy elkerülhetetlenné tesz későbbi buktatókat (a példa-mátrixban, az 5-ös érték mohó elválasztásával beleszaladt a 99, 33, 17 értékek képezte „gátvonalba”). Ezért az én elvem – biztos, ami biztos alapon – az, hogy *„legbizonyosabban úgy találod el a verebet, ha ágyúval lősz a fára”*. Más szóval, sorra generálom az összes megoldásutat, azaz az összes csúcsból alapra vezető utat, és kiválasztom közülük a „legjobbat”. Elsőnek az (1,1),(2,1),(3,1),(4,1),(5,1) utat állítom elő, majd az (1,1),(2,1),(3,1),(4,1),(5,2) utat, és így tovább. Utolsóként generálom az (1,1),(2,2),(3,3),(4,4),(5,5) utat (lásd az 1.6.1. alfejezetben bemutatott mélységi bejárást). Nincs, ahogy kicsússzon az ujjaim közt az optimális megoldás.



2.5. ábra. *A backtracking stratégia generálta megoldásutak, kiemelve az optimális (hossza 32)*

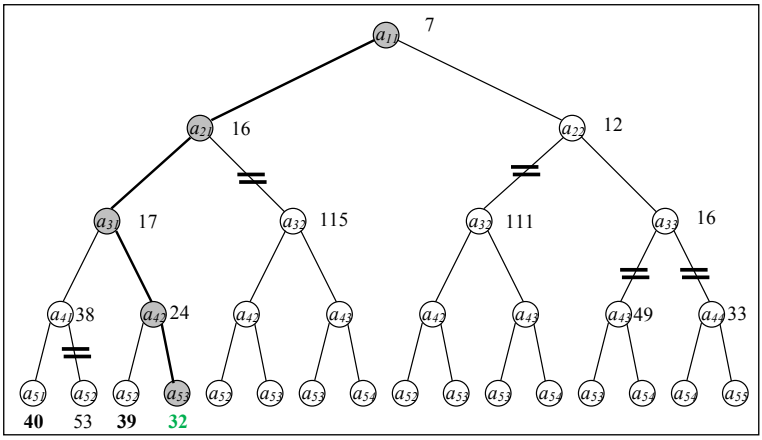
Branch-and-bound: Lenne egy javaslatom, hogy miként tudna Bracktracking gyorsítani algoritmusán. Ha a kurrens út (gyökértől a kurrens csomópontig) hossza *már több*, mint az addig talált legjobb megoldásúté, akkor értelmetlen az illető irányba folytatni az útgenerálást. A 2.6. ábrán bejelöltük, hogy mely részfák metszhetők le a generálandó fáról (a csomópontok mellett a csúcsból odavezető út hossza lett feltüntetve; 53>40, 115>32, 111>32, 49>32, 33>32).

Ebben a gondolatmentben kívánatos minél előbb egy minél jobb megoldást találni, ha nem is garantáltan az optimálisat. Ha az épülő fa koronája peremének mindig a legkisebb részösszegű csomópontjától ágazunk tovább („best first"), akkor – talán – még jobban meg tudjuk metszeni a fát (2.7. ábra). A korona peremén lévő csomópontok (részösszeg szerint csökkenő sorrendben), lépésről lépésre, a következők:

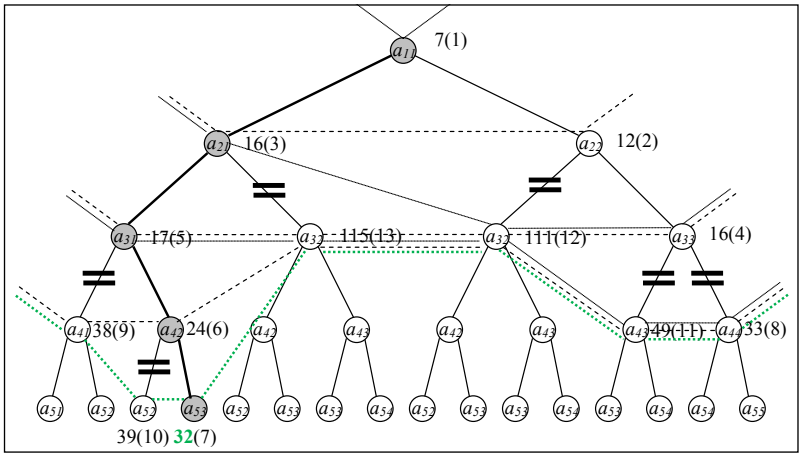
1. {}
2. {(1,1,**7**)}
3. {(2,2,**12**), (2,1,**16**)}
4. {(2,1,**16**), (3,3,**16**), (3,2,**111**)}
5. {(3,3,**16**), (3,1,**17**), (3,2,**111**), (3,2,**115**)}
6. {(3,1,**17**), (4,4,**33**), (4,3,**49**), (3,2,**111**), (3,2,**115**)}
7. {(4,2,**24**), (4,4,**33**), (4,1,**38**), (4,3,**49**), (3,2,**111**), (3,2,**115**)}
8. {(5,3,**32**), (4,4,**33**), (4,1,**38**), (5,2,**39**), (4,3,**49**), (3,2,**111**), (3,2,**115**)}
9. {(4,4,**33**), (4,1,**38**), (5,2,**39**), (4,3,**49**), (3,2,**111**), (3,2,**115**)}
10. {(4,1,**38**), (5,2,**39**), (4,3,**49**), (3,2,**111**), (3,2,**115**)}
11. {(5,2,**39**), (4,3,**49**), (3,2,**111**), (3,2,**115**)}
12. {(4,3,**49**), (3,2,**111**), (3,2,**115**)}
13. {(3,2,**111**), (3,2,**115**)}
14. {(3,2,**115**)}
15. {}

Minden lépésben a sorelsőt helyettesítettük a fiaival (feltéve, ha a megfelelő részösszeg nem már nagyobb, mint az addig talált legjobb megoldás értéke). Ezzel a módszerrel első megoldásként – jó eséllyel – egy optimális közeli értéket

találunk (ha nem éppen az optimumot), bár nem olyan korán, mint a Backtracking (amely a legbaloldaliabb ágon leszaladva, a lehető legkorábban talál egy valamilyen megoldást).



2.6. ábra. *Branch-and-bound ötlettel javított backtracking algoritmus metszette fastruktúra. A csomópontok mellett feltüntettük a megfelelő részösszegeket (a csúcsból odavezető út hosszát)*



2.7. ábra. *„Best first” ötletre épülő branch-and-bound algoritmus metszette fastruktúra. Zárójelben feltüntettük, hogy milyen sorrendben hajtanak ágakat (vagy hajtanának ágakat, ha nem bizonyulnának száraz iránynak) a növekvő fa koronáját képező csomópontok (pontozott/szaggatott vonalak jelzik, ahogy lépésről lépésre bővül a korona pereme)*

Divide-et-impera: Az én megközelítésemet már a római császárok is használták: *„oszd meg, és uralkodj!”*. A kurrens részfeladatnak (kezdetben az eredeti feladatnak) mint apafeladatnak a megoldását visszavezetem a fiúrészfeladatai megoldásaira (egy-egy rekurzív hívás által; „oszd meg” fázis), majd e fiúmegoldásokból felépítem az apafeladat megoldását; „uralkodj” fázis). Ha a kurrens apafeladatnak az (i, j) pozícióból alapra vezető legjobb út problémáját tekintjük, akkor ennek fiúrészfeladatai az $(i+1,j)$ és $(i+1,j+1)$ pozíciókból alapra vezető legjobb utak problémái. A stratégiám alapjául szolgáló rekurzív képlet nyilvánvalóan a következő: $legjobbút(i,j) = a[i][j] + min\{legjobbút(i+1,j), legjobbút(i+1,j+1)\}$, ahol $1 \leq i < n$.

Dinamikus programozás: A divide-et-impera megközelítéssel két gondom is van. Először is csak a legjobb út hosszát szolgáltatja, és nem magát az utat is. Még súlyosabb gond, hogy mivel nem tart nyilvántartást a már megoldott részfeladatokról, azonos részfeladatokat többször is megold. Például a „(3,2) részfeladatot” kétszer oldja meg: egyszer a „(2,1) részfeladat” jobb-fiúrészfeladataként, majd a „(2,2) részfeladat” bal-fiúrészfeladataként is. Én egy másik tömbben (például $c[1..n][1..n]$) eltárolnám minden részfeladat első példányának megoldását (a megfelelő legjobb út hosszát), és ha újra találkoznék ugyanazzal a részfeladattal, akkor csak elővenném a megoldását.

A branch-and-bound megközelítés is javítható az alapötletemmel: ha ugyanazon cella többszörösen kerülne a növekvő fa koronájára, csak attól a példánytól ágazzunk tovább, amelyhez tartozó részösszeg a legkisebb (a többit nyugodtan tekinthetjük száraz iránynak). Persze ez megint csak azt feltételezi, hogy nyilvántartást vezetünk. Jelen feladat esetében az elsőként koronára kerülő példány részösszege lesz a minimális.

A divide-et-imperát javító ötletem iteratív megközelítésben így foglalható össze: indulva a triviálisan egyszerű részfeladatok $[(n,j)$ alakú részfeladatok, $j=1,n]$ nyilvánvaló megoldásainak szintjéről, a már rendelkezésre álló (eltárolt) fiúrészmegoldásokból felépítjük (a rekurzív képlet alapján) az aparészfeladatok megoldásait (utolsóként az eredeti feladatét) (2.8. ábra).

$c[n][j] = a[n][j], j=1..n$

$c[i][j] = a[i][j] + \min\{c[i+1][j], c[i+1][j+1]\}, i=n-1, n-2, \ldots, 1; j=1..i$

A c tömbből azonnal adódik a legjobb út is: a csúcsból alapra vezető mohó vonal.

Ugyanez branch-and-bound irányból (a gyökértől a levelek irányába) is megfogalmazható (2.9. ábra):

$c[1][1] = a[1][1]$

$c[i][1] = a[i][1] + c[i-1][1], i=2, 3, \ldots, n$

$c[i][i] = a[i][i] + c[i-1][i-1], i=2, 3, \ldots, n$

$c[i][j] = a[i][j] + \min\{c[i-1][j], c[i-1][j-1]\}, i=2, 3, \ldots, n; j=2..i-1$

32				
25	27			
16	114	22		
23	15	36	18	
2	15	8	3	1

2.8. ábra. *Levelek–gyökér irányú dinamikus programozással feltöltött optimális részmegoldások-tömb*

7				
16	12			
17	111	16		
38	24	49	33	
40	39	32	36	34

2.9. ábra. *Gyökér–levelek irányú dinamikus programozással feltöltött optimális részmegoldások-tömb*

Moderátor összegzése: A Greedy javasolta algoritmus a leggyorsabb (lineáris idő bonyolultságú), de nem garantál optimális megoldást. A Backtracking és Divide-et-impera biztosítják ugyan a legjobb megoldást, de algoritmusaik exponenciális bonyolultságúak. „Szerencsétlen esetekben" még a „Branch-and-bound javításokkal" is marad az exponenciális bonyolultság. Ennél a feladatnál a pálmát a dinamikus programozás viszi el, hiszen képes polinomiális időben (négyzetes időbonyolultság) optimális megoldással szolgálni.

A bemutatkozott technikák más-más stratégiák szerint igyekeznek a keresési teret leszűkíteni, azaz a keresési fát megmetszeni, mintha mindeniknek meglenne a maga ollója. E feladat kapcsán úgy tűnhet, mintha a backtracking módszernek nem lenne ollója (jelen esetben ez számított a nyers erő megközelítésnek), de ez nem így van. Például ha a „legjobb" *szigorúan növekvő* gyökér–levél útban lettünk volna érdekeltek, akkor a backtracking is csak e szabálynak megfelelő utakat generált volna. Más szóval lemetszette volna azokat a részfákat, amelyek bejárása/generálása nyomán nem növekvően folytatódott volna a kurrens út.

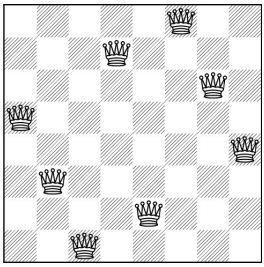
A fenti bemutató természetesen csak azt a célt szolgálta, hogy fogalmat alkothassunk a módszerekről, melyeknek részletes bemutatása a következő fejezettel kezdődik.

3. BACKTRACKING

Backtracking a sakktáblán. A nyolckirálynő-probléma először 1848-ban tűnt fel egy híres sakkújságban (*Schachzeitung*, szeptemberi szám) egy sakkbarát (Schachfreund), Max Bezzel feladványaként. 1850-ben Franz Nauck tollából újra megjelent a probléma az *Illustrirte Zeitung*ban (Leipzig), ahol Gauss is rátalált. Egy csillagász barátjával folytatott levelezéséből kiderül, hogy a matematika fejedelme azonnal elkezdett foglalkozni a feladvánnyal, de nem találva fejedelmi fontosságúnak, valószínűleg csak félgőzzel. Gauss először 76-ra becsülte az összes megoldások számát, majd ezt 72-re javította. Nauck volt, aki először közölte a 92-t, mint az összes megoldások helyes száma (ő 60-ról javított 92-re) (Campbell 1977). 1972-ben Dijkstra egy részletes tanulmányt közölt a problémát megoldó backtracking algoritmusról.

Bástyák/királynők: Legyen egy $n \times n$ méretű sakktábla: (1) helyezzünk el rajta, az összes lehetséges módon, n bástyát úgy, hogy ne üssék egymást; (2) helyezzünk el rajta, az összes lehetséges módon, n királynőt úgy, hogy ne üssék egymást.

Mielőtt megoldanánk a bástya/királynő feladatot, foglalkozzunk egy másik problémával.



3.1. ábra. *Helyes királynő-elhelyezés 8×8-as sakktáblán*

3.2. ábra. *4-pozíciós kerékpárzár*

Kerékpárzár: Emlékezzünk a klasszikus 4-pozíciós kerékpárzárra (lásd a 3.2. ábrát). Mindegyik pozícióba valamelyik számjegy választható ki: 0, 1, …, 9. A feladatunk az, hogy generáljuk az összes lehetséges 4 számjegyű kódvektort: (0,0,0,0), (0,0,0,1), (0,0,0,2), …, (9,9,9,9).

Úgy is fogalmazhatunk, hogy érdekel az összes (v_1, v_2, v_3, v_4) alakú vektor, ahol $v_i \in \{0, 1, \ldots, 9\}$, $i = 1..4$. Milyen algoritmust követhetünk?

– 1. pozíción: minden számjegyet egyszer állítunk be. (Az 1. szintre egyszer generáljuk a 0, 1, …, 9 számjegysort.)

– 2. pozíción: minden 1. pozíciós számjegy mellé társítjuk, rendre, a teljes számjegysort. (A 2. szintre 10-szer generáljuk a 0, 1, …, 9 számjegysort.)

– 3. pozíción: minden 1..2 pozíciós számjegypár mellé társítjuk, rendre, a teljes számjegysort. (A 3. szintre 10^2-szer generáljuk a 0, 1, …, 9 számjegysort.)

– 4. pozíción: minden 1..3 pozíciós számjegyhármas mellé társítjuk, rendre, a teljes számjegysort. (A 4. szintre 10^3-szor generáljuk a 0, 1, …, 9 számjegysort.)

Ez az algoritmus megvalósítható 4 egymásba ágyazott minden ciklussal (a kódvektorokat az *x*[1..4] tömbben generáljuk).

```
minden x[1] = 0,9 végezd
   minden x[2] = 0,9 végezd
      minden x[3] = 0,9 végezd
         minden x[4] = 0,9 végezd
            ki: x[1..4]    //10^4-szer hajtódik végre
         vége minden
      vége minden
   vége minden
vége minden
```

Általánosítsuk a *Kerékpár-zár* feladatot: generáljuk az összes *n* számjegyű kódvektort.

Ez azt jelenti, hogy *n* darab minden ciklust kell egymásba ágyaznunk, amelyre a megoldás a *bactracking* (BT) stratégia!

```
BTd(x[],n,k)
   minden x[k] = 0,9 végezd
      ha k < n akkor
         BTd(x,n,k+1)
      különben
         kiír(x,n)
      vége ha
   vége minden
vége BTd
```

A módszer e rekurzív implementációja a következőképpen foglalható össze:

– A BTd(x,n,k) rekurzív eljárás feladata, hogy generálja, az $x[k..n]$ tömbszakaszon az összes $(v_k, v_{k+1}, \ldots, v_n)$ kódszakaszt. Egy eljáráspéldány a kapott feladatból csak annyit „vállal személyesen be", hogy generálja a k. szintre (az $x[k]$ cellában) a 0, 1, …, 9 számjegyeket.
  • Azok a példányok, amelyek $k<n$ értékre lettek meghívva, a $(v_{k+1}, \ldots, v_n)$ kódszakaszok generálását $(k+1)$. szinti példányokra ruházzák át. (Minden $x[k]$ érték mellé generáltatják, egy-egy BTd(x,n,k+1) rekurzív hívás révén, az $x[(k+1)..n]$ tömbszakaszon, az összes $(v_{k+1}, \ldots, v_n)$ kódszakaszt).
  • A $k=n$ értékre meghívott példányok feladata lesz a generált kódvektorok kiíratása, a kiír(x,n) eljárás révén.
– A BTd(x,n,1) példány (amelyet a főprogram/függvény hív meg) kapja feladatul, hogy generálja az összes n hosszú kódvektort. Ezen 1. szinti példány 10-szer hív majd 2. szinti példányokat, ezek pedig összes 10^2-szer 3. szintieket, és így tovább. Végül a 10^{n-1} darab n. szinti példány mindenike kiír tíz teljes kódot. Összesen 10^n kód jelenik meg a képernyőn.
– Ténylegesen megvalósul a minden ciklusok egymásba ágyazása, hiszen az 1. szinten egyszer, a 2. szinten 10-szer, …, és végül az n. szinten pedig 10^{n-1}-szer hajtódik végre a minden x[k] = 0,9 végezd ciklus.

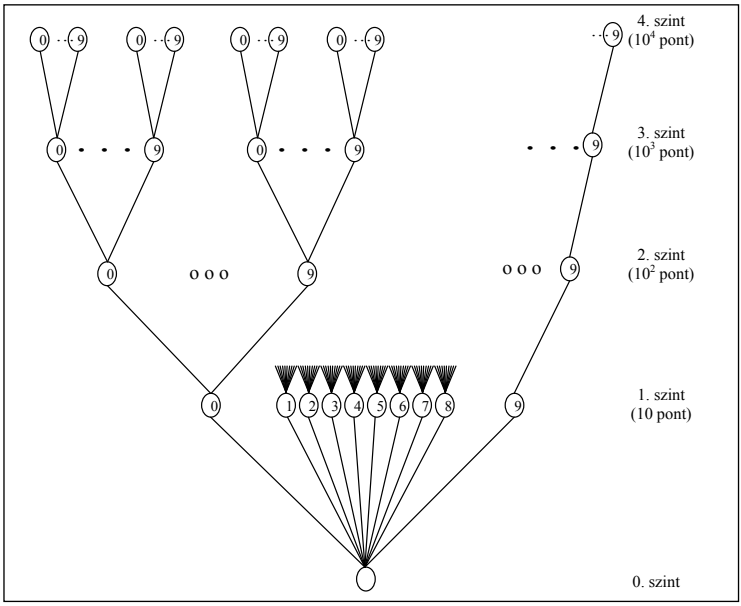
Megjegyzések:

– A módszert azért nevezik backtrackingnek, mert amennyiben befejeződött a kurrens szinti értéksor generálása, *visszalép*, hogy folytassa az időszakosan felfüggesztett, előző szintit.
– Vegyük észre, hogy a generált kódvektorok a {0, 1, …, 9} halmaz n-szeres Descartes-szorzatának elemeit adják meg. (Ezért neveztük az eljárást BT**d**-nek.)
– A generált kódvektorokat n-szintes fastruktúraként is felfoghatjuk. A 0. szinti virtuális gyökérnek 10 első szinti fia van, ezek mindenikének tíz-tíz második szinti (összesen 10^2), és így tovább. Az n. szinten 10^n levele lesz a fának. Ha mindenik testvér csomópontsorhoz a 0, 1, …, 9 számjegyeket rendeljük, akkor a fa 10^n gyökér–levél ága/útja egy-egy n hosszú kódvektort képvisel majd (3.3. ábra).

Módosítsunk a *Kerékpár-zár* feladaton! Olyan n hosszú kódvektorok érdekelnek, amelyek elemei az {1, 2, …, n} halmazból vehetnek értékeket. Kössük ki azt is, hogy a kódok nem tartalmazhatnak identikus elemeket. Tömören fogalmazva az {1, 2, …, n} halmaz permutációi érdekelnek: (1, 2, …, n), …, (n, $n-1$, …, 1).

Hogyan módosítsuk a BTd eljárást, hogy permutációgeneráló (BTp) legyen belőle?

– Nyilvánvaló, hogy ez esetben a minden ciklus, az $x[k]$ tömbelemben, az 1, 2, …, n értékeket kell hogy generálja. Ha csak ennyit változtatnánk,



3.3. ábra. *A {0,…,9}×{0…,9}×{0,…,9}×{0,…,9} Descartes-szorzat elemei fastruktúraként megjelenítve (a kerékpárzár-feladat megoldásai)*

akkor az {1, 2, …, n} halmaz n-szeres Descartes-szorzatának elemeit kapnánk eredménynek.

– Hogyan zárhatjuk ki az identikus kódelemeket? Az $x[k]$ tömbértéket csak akkor tekintjük úgy, mint amely ígéretesen bővíti a $x[1..(k-1)]$ kódprefixet, ha különbözik e szakasz mindegyik elemétől. Ennek vizsgálatát bízzuk az ígéretes(x,k) függvényre. Az ígéretes megnevezés azt sugallja, hogy amennyiben az $x[k]$ érték összefér az $x[1..(k-1)]$ tömbszakasz elemeivel (a generálandó megoldás-kódvektorokkal szemben támasztott belső tulajdonság értelmében; jelen esetben, hogy elemeik páronként különbözzenek), akkor, remélhetőleg, a bővített x[1..k] tömbszakasz egy megoldás-kódvektorprefixet tárol.
– Sajátos esetekben (ilyen a permutációs feladat is), ha az $x[k]$ érték összefér az $x[1..(k-1)]$ szakasszal, akkor a bővített $x[1..k]$ szakasz garantáltan megoldás-kódvektorprefixet tárol. Ilyenkor találóbb lehet a megfelelő függvénynév-azonosító használata.

```
ígéretes(x[],k)
  minden i = 1,k−1 végezd
    ha x[i] == x[k] akkor
      return HAMIS
```

```
        vége ha
    vége minden
    return IGAZ
vége ígéretes

BTp(x[],n,k)
    minden x[k] = 1,n végezd
        ha ígéretes(x,k) akkor
            ha k < n akkor
                BTp(x,n,k+1)
            különben
                kiír(x,n)
            vége ha
        vége ha
    vége minden
vége BTp
```

Megjegyzés:

– Vegyük észre, hogy máris megoldottuk a bástyafeladatot. Minden permutáció egy-egy helyes bástyaelhelyezést kódol. A kiír(x,n) eljárásnak az (1,x[1]), (2,x[2]), …, (n,x[n]) (sor,oszlop) koordinátákon kell a bástyákat elhelyeznie.

Mi a teendőnk, amennyiben a helyes királynő-elhelyezések érdekelnek? Csupán az ígéretes függvényt kell módosítanunk, hogy csak olyan permutációk kerüljenek generálásra, amelyek helyes királynő-elhelyezést kódolnak. Tekintsük továbbra is úgy, hogy a tömbindexek sakktáblasorokat, a tömbértékek pedig sakktáblaoszlopokat jelentenek. Mikor ígéretes egy x[k] érték? Ha a (k,x[k]) koordinátájú pozíciót nem ütik az (i,x[i]) pozíciókra (i=1..(k−1)) már elhelyezett királynők. Ennek plusz feltétele (a bástyafeltételhez képest), hogy a (k,x[k]) és (i,x[i]) koordinátájú pozíciók ne legyenek egyazon átlón: azaz a (k−i) sortávolság ne legyen azonos a $|x[k]-x[i]|$ oszloptávolsággal.

Íme az n-királynő feladat megoldását implementáló backtracking algoritmus (BTkirálynő):

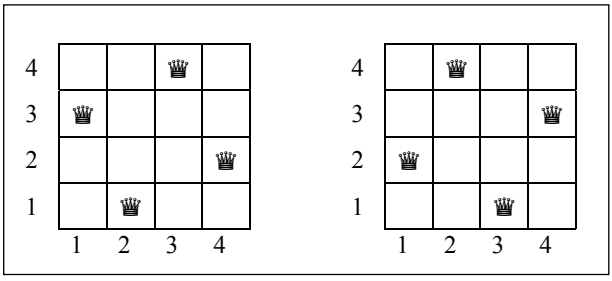
```
ígéretes_királynő(x[],k)
    minden i = 1,k−1 végezd
        ha (x[i] == x[k]) vagy ((k−i) == |x[k]−x[i]|) akkor
            return HAMIS
        vége ha
    vége minden
    return IGAZ
vége ígéretes_királynő
```

```
kiír_királynő(x[],n)
   minden i = 1,n végezd
      ki: i,x[i]
   vége minden
vége kiír_királynő

BTkirálynő(x[],n,k)
   minden x[k] = 1,n végezd
      ha ígéretes_királynő(x,k) akkor
         ha k < n akkor
            BTkirálynő(x,n,k+1)
         különben
            kiír_királynő(x,n)
         vége ha
      vége ha
   vége minden
vége BTkirálynő
```

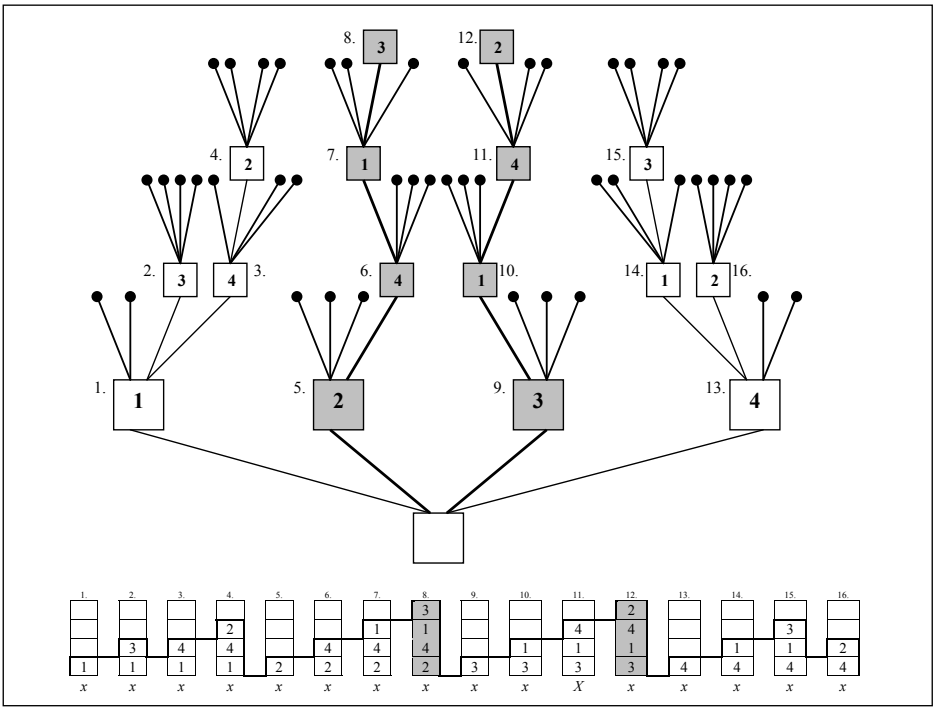
Megjegyzések:

– Az $\{1, 2, \ldots, n\}$ halmaz n-szeres Descartes-szorzatának elemeit ábrázoló fastruktúra (3.3. ábra) úgy is felfogható, mint a megoldások keresési tere. Keressük azokat a gyökér–levél ágakat/utakat, amelyek helyes bástya/királynő elhelyezést kódoló vektoroknak felelnek meg. Ebből a megvilágításból az ígéretes függvény úgy tekinthető, mint amely révén leszűkítjük (megmetsszük) a keresési teret. E függvény kezében van a BT-olló (lásd a 3.2. alfejezetet).

– Az alábbi ábrák (3.4. és 3.5. ábrák) az $n=4$ esetet teszik szemléletesebbé:



3.4. ábra. *Helyes királynő-elhelyezések 4×4-es sakktáblán*



3.5. ábra. *A 4-királynő feladathoz kapcsolódó keresési tér mint fastruktúra. A ponttal jelölt csomópontokban az ígéretes függvény ollóként működik. A négyzettel jelölt csomópontok kapcsán megadtuk az x tömb megfelelő állapotát (a preorder sorrend szerinti sorszámozás teremt egy-egy kapcsolatot a két ábrázolás között)*

3.1. Általános backtrackingmodell

Az előbbi feladatokban a kódvektorok mindegyik eleme az {1, 2, …, n} halmazból származott. Ezért bármely k=1..n-re az $x[k]$ cellában az {1, 2, …, n} halmaz elemeit kellett generálni. A generálást meg tudtuk valósítani egy klasszikus minden ciklussal. Általános esetben a generálandó kódvektorok k. elemei egy A_k={a_{k1}, a_{k2}, …}halmazból származhatnak. Mivel az a_{k1}, a_{k2}, … értéksorozatot generálnunk kell az $x[k]$ cellában, ezért, nyilván, valamely szabály szerint kell hogy kövessék az értékek egymást.

A fentebb megtárgyalt feladatok egy másik sajátossága az volt, hogy a megoldáskódok azonos hosszúságúak voltak. Ebből kifolyólag a generált kódszakasz

hosszából ($k=n$) egy az egyben adódott, hogy megoldáskódhoz jutottunk-e. Általános esetben használhatunk egy külön megoldásfüggvényt ennek ellenőrzésére.

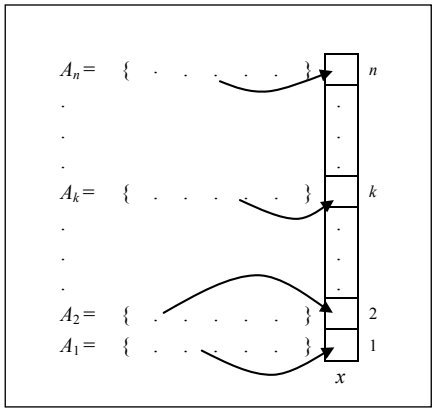
```
BT(x[],n,k)
  minden x[k] = a_k1, a_k2, ... végezd
    ha ígéretes(x,n,k) akkor
      ha megoldás(x,n,k) akkor
        kiír(x,n,k)
      különben
        BT(x,n,k+1)
      vége ha
    vége ha
  vége minden
vége BT
```

Megjegyzések:

– A 3.6. ábra jól modellezi az általánosabb esetet. Az $A_1 \times A_2 \times \ldots \times A_n$ Descartes-szorzat elemei közül keressük azokat, amelyek megoldásokat kódolnak. Ezeket általában jellemez egy belső tulajdonság, amely például a bástya feladat esetében az volt, hogy elemei legyen páronként különbözőek.

– Az ígéretes függvény általános esetben is azt ellenőrzi, hogy az $x[k]$ érték ígéretesen bővíti-e, összefér-e az $x[1..(k-1)]$ tömbszakasz tárolta kódszakasszal, a megoldásvektorokkal szemben támasztott belső tulajdonság értelmében.

– A kiír eljárás általános esetben is a kurrens megoldás-kódvektort írja ki (esetleg, a kód alapján, a megoldást szemléletesebben is megjelenítheti).



3.6. ábra. *Általános modell backtracking feladatokhoz*

3.2. Backtracking a fán

Az alábbiakban a keresési teret ábrázoló fán követjük nyomon a backtracking módszer stratégiáját (további részletek végett lásd az *Algoritmusok felülnézetből* jegyzet 2.1. alfejezetében; Kátai 2007). A gyökértől a levelekhez vezető utak az $A_1 \times A_2 \times \cdots \times A_n$ Descartes-szorzat elemeit ábrázolják (feltételezzük, hogy a halmazok $n_1, n_2, \ldots, n_n$ elemet tartalmaznak). Az első szintre az $x[1]$ tömbelembe n_1-féleképpen választhatunk elemet az A_1 halmazból, tehát a fa gyökeréből n_1 első szinti fiúcsomópont ágazik le, amelyekhez az A_1 halmaz elemeit rendeljük. Minden első szinti választáshoz a második szintre, az $x[2]$-be n_2-féleképpen választhatunk elemet az A_2 halmazból. A fán ez abban tükröződik, hogy minden első szinti csomópontnak n_2 fia lesz a második szinten, amelyekhez az A_2 halmaz elemeit rendeljük. Ez összesen $n_1 \times n_2$ második szinti csomópontot eredményez. Végül az n-edik szinten a fának $n_1 \times n_2 \times \cdots \times n_n$ csomópontja (levele) lesz. Általános szabályként megjegyezhető, hogy ha a csomópont a fa k-adik szintjén található, és az apacsomópontjának ez az i-edik fia, akkor az A_k halmaz i-edik elemét rendeljük hozzá. Mivel a fa bármely csomópontjához a gyökérből pontosan egy út vezet, ezért elmondható, hogy minden csomópont képviseli ezt az utat. A levelek mindegyike a Descartes-szorzat valamelyik elemét képviseli – azt, amelyiket a gyökérből az illető levélhez vezető út ábrázol.

Hogyan jelennek meg a fában a feladat megoldásvektorai? Ha a megoldások a Descartes-szorzat adott tulajdonságú elemei, akkor ezeket azok az utak ábrázolják, amelyek a gyökértől egy-egy levélhez vezetnek. Nevezzük ezeket megoldásutaknak vagy megoldáságaknak, a hozzájuk tartozó leveleket pedig megoldásleveleknek. Egyes feladatokban a megoldásokat nem feltétlenül gyökér–levél utak ábrázolják, hanem a gyökértől adott tulajdonságú csomópontokhoz vezető utak. Általános esetben tehát megoldáscsomópontokról beszélünk. Ezek után nem nehéz átlátni, miért nevezzük a feladathoz rendelt fát a megoldások terének.

Mindezeket figyelembe véve egy backtracking-feladat az alábbi módon is megfogalmazható: keressük meg a feladathoz rendelhető fa megoldáscsomópontjait, illetve építsük fel azokat a megoldásutakat, amelyek a gyökérből ezekhez vezetnek, és amelyekhez, természetesen, éppen a megoldásvektorok vannak rendelve. Mindezt az x tömbben fogjuk megvalósítani, amelyet – amint látni fogjuk – úgy használunk, mint egy vermet.[4]

Szemléltetésül lássuk, hogyan működik a 4-királynő feladatot megoldó backtracking algoritmus az imént bemutatott fán. A 3.5. ábrán, helyhiány miatt, nem rajzoltuk le a teljes fát (a negyedik szinten 256 levele van). Csak azt a részfát rajzoltuk le, amely potenciálisan tartalmazza a megoldásleveleket a fa élő ré-

4 A verem egy lineáris adatszerkezet, amelynek ugyanazon végén (a „tetején") történik mind a betevés, mind a kivevés. Ebből kifolyólag az utoljára betett elem lesz az elsőnek kivett. (**L**ast **I**n **F**irst **O**ut)

szén). A csomópontjait aszerint sorszámozzuk, hogy milyen sorrendben látogatjuk meg őket. A jobb áttekinthetőség kedvéért bejelöltük az „*élő* csomópontok” „*száraz* irányba” vivő fiait is. A fa alatt megadtuk a csomópontok meglátogatásának pillanatában a verem (az *x* tömb) tartalmát. A vastagított vonalat követve láthatjuk, mikor mely elemek kerülnek be, illetve ki a veremből.

Indulunk a gyökérből. Sorra megvizsgáljuk a gyökér utáni első szint csomópontjait, a gyökér fiait, és fellépünk ahhoz, amelyről elsőként találjuk azt, hogy potenciálisan megoldáscsomóponthoz vezet. Itt hasonlóképpen járunk el: megvizsgáljuk a fiait (a belőle ágazó második szint csomópontjait), és ahogy olyat találunk, amely ígéretes irányba vezet, odalépünk. Így járunk el egészen addig, míg olyan csomóponthoz nem jutunk, amelynek nyilvánvalóan egyetlen fia sem vezet megoldáshoz. Ilyenkor visszalépünk az előző szinti apacsomóponthoz, ahol folytatjuk a keresést, további ígéretes fiak után kutatva. Ha találunk egy újabb csomópontot, amely potenciálisan megoldáscsomóponthoz vezet, akkor újból fellépünk a következő szintre. Ha egy csomópontból már minden irányt ellenőriztünk – az ígéreteseket be is járva –, akkor innen is visszalépünk. Az algoritmus akkor ér véget, amikor a gyökérből ágazó összes első szinti csomóponttal foglalkoztunk már. Valahányszor megoldáscsomópontot találunk, a gyökérből hozzá vezető út egy megoldásvektornak felel meg.

A fa ily módon történő bejárását mélységi bejárásnak[5] nevezzük (lásd az 1.6.1. alfejezetet). Vegyük észre, hogy nem jártuk be a teljes fát, csak azt a részfát, amely potenciálisan tartalmazza a megoldásokat. Ezt neveztük a fa *élő részének*, a többi *száraz ág* a feladatra nézve. Nyilván minél jobban sikerül leszűkíteni a bejárt részfát, annál hatékonyabb az algoritmusunk. Ez attól függ, hogy mennyire hatékonyan tudjuk meghatározni, hogy egy bizonyos irány ígéretes-e. A 4-királynő feladata esetén sikerült csupán 16 csomópont meglátogatása által megtalálni a megoldásokat, holott a teljes fának összesen $4+16+64+256 = 340$ csomópontja van (nem számoltuk a gyökeret, lévén csupán virtuális).

De mit is jelent az, hogy egy irány ígéretes, vagy hogy az illető fiúcsomópont potenciálisan megoldáscsomóponthoz vezet? Először is kövessük nyomon, miként alakul át a keresés alatt az aktuális út (a gyökérből az aktuális csomóponthoz vezető út) megoldásúttá. Amikor felfele lépünk a fában, új csomópont kerül az aktuális út végére. Visszalépéskor egy bizonyos csomópont eltűnik a végéről. Nos, egy csomópont akkor vezet potenciálisan megoldáscsomópont felé, ha ígéretesen bővíti a gyökérből az aktuális csomóponthoz vezető utat. Hogyan értendő ez? Emlékezzünk, hogy egy út attól megoldásút, hogy a csomópontjaihoz rendelt elemek megoldásvektort alkotnak. Más szóval, az illető úthoz rendelt elemek rendelkeznek a megoldásvektorokat azonosító belső tulajdonsággal. A backtracking módszer kulcsötlete az, hogy amennyiben a szóban forgó fiúcsomópont-

5 A fákat általában a gyökerükkel fent és a leveleikkel lent szokás ábrázolni. Mivel mi megfordítottuk, ezért esetünkben találóbb lenne a „fa magassági bejárása” kifejezés.

hoz rendelt elem máris nem fér össze – a megoldásvektorokat azonosító belső tulajdonság szempontjából – a már az aktuális úton lévő csomópontokhoz rendelt elemekkel, akkor értelmetlen az illető csomópontra építve keresni az újabb megoldásokat. Tehát egy megvizsgált fiúcsomópont akkor ígéretes, ha – az imént bemutatott értelemben – nem kerül „konfliktusba" az apacsomópontjához vezető aktuális út már ígéretesnek talált csomópontjaival. Például az n-királynő feladata esetén a fa csomópontjaihoz rendelt elemek a sakktábla négyzeteit képviselik. Ha a szóban forgó csomópont által képviselt négyzetet támadják az aktuális úton lévő csomópontok négyzetein már elhelyezett királynők, akkor nyilván „száraz irányról" van szó.

Az a feltétel, amelynek alapján eldönthető, hogy az illető csomóponttal ígéretesen bővíthető-e az aktuális út, a feladat megoldásait azonosító belső tulajdonságok alapján határozható meg. Amint megfigyelhettük, ennek a feltételnek kulcsszerepe van minden backtracking algoritmus esetében, hiszen alapvetően ez határozza meg, mekkora lesz a bejárt részfa.

És most lássuk, miként valósult meg a fán bemutatott algoritmus az x tömbben? Nem építettük fel a fát a számítógép memóriájában, csak szimuláltuk a bejárását az x tömb segítségével. A 3.5. ábra és az algoritmus tanulmányozása a következő fontos észrevételekhez vezet:

– Az x tömböt úgy kell használnunk, mint egy vermet. Figyeljük meg, hogy a fa bejárásának egy adott pillanatában a tömb az aktuális úton található csomópontok képviselte elemeket tartalmazza. Amikor felfele lépünk a fában, a verem tetejére új elem kerül. Visszalépéskor eltűnik a verem tetején lévő elem.
– A fa bejárásakor minden érintett csomópontban alapvetően ugyanúgy kellett eljárnunk: sorra megvizsgáltuk a fiait, és valahányszor ígéretest találtunk, felléptünk hozzá, ahol hasonlóképpen jártunk el. Természetesen minden csomópontban ellenőriznünk kell, hogy nem képvisel-e éppen megoldást.
– A fának a fenti algoritmus szerinti bejárását úgy is felfoghatjuk, hogy egy csomóponthoz tartozó részfa bejárását visszavezetjük ígéretes fiúrészfáinak a bejárására (egy részfa akkor ígéretes, ha potenciálisan tartalmaz megoldáscsomópontot).

Mindhárom észrevétel azt sugallja, hogy az algoritmust célszerű lehet rekurzívan megvalósítani, ahogy tettük is. Ez egy olyan eljárás megírását feltételezi, amely mindig az aktuális szinten dolgozik (legyen ez a k-adik): sorra generálja az előző szinti apa ($x[k-1]$) fiait, és valahányszor ígéretest talál, meghívja magát az illető fiúcsomópont (mint újdonsült apa) következő szinti ($k+1$) fiaira. A visszalépés automatikusan történik a rekurzió mechanizmusa által, amikor a kurrens apa összes fiának a vizsgálata befejeződött. Ha az a részfa, amely potenciálisan tartalmazza a megoldásokat, véges, akkor nem kell tartanunk a végtelen rekurzív hívásoktól: mivel a leveleknek nincsenek ígéretes fiaik, ezekre az algoritmus nem hívja meg önmagát.

Visszalépéses keresés

A backtracking algoritmusokat szokás visszalépéses keresésnek is nevezni. Mi úgy mutattuk be ezt a stratégiát, mint amelyet valamely feladat *összes* megoldása generálására használunk. Másfelől, keresésként felfogni egy feladatot azt jelenti, hogy érdekeltek vagyunk a feladat reprezentációs gráfjának startcsúcsból *valamely* célcsúcsba vezető útjában.

A visszalépéses keresés a startcsúcsból elindulva egyetlen úton hatol be a gráf belsejébe. A mélységi bejárás stratégiáját követve egy-egy utat olyan mélyen tár fel, amennyire csak lehetséges. Ezért is szokták depth-first keresésként (DFS) is emlegetni. A startcsúcsból a keresés aktuális csomópontjába vezető utat teljes hosszában nyilvántartja (ez magába foglalja az útról leágazó, még ki nem próbált élek nyilvántartását is). Akkor lép vissza, (1) ha a keresés zsákutcába jutott, (2) ha az aktuális csúcsból kivezető összes útról kiderült, hogy nem vezet célba, (3) ha körre futott, illetve (4) ha az aktuális út hossza egy előre megadott mélységi korláton túl nőne. A visszalépéses keresés akkor ér véget, ha talált egy célcsúcsba vezető utat, vagy ha már minden lehetséges utat végignézett. (Gregorics 2014)

3.3. Recept backtracking feladatok megoldásához

A backtracking feladatok egyik sajátossága, hogy a feladat *összes* megoldásában érdekeltek vagyunk. Ha optimalizálási feladatot akarunk megoldani backtrackinggel, akkor a módszer az, hogy generáljuk az összes potenciális megoldást, és optimumot keresünk ezek között (a kiír eljárást lecseréljük egy min/max keresőre; az optimális megoldást utólag írjuk ki).

Az alábbi ötlépéses receptet javasoljuk backtracking feladatok megoldásához:

1. Hogyan kódolhatók a feladat megoldásai vektorokként? Előnyt jelentenek az olyan kódolások, amelyek azonos hosszú megoldáskódokat eredményeznek.

– Nyomra vezethet, ha egy konkrét példán kísérletezünk. Például ha a 4×4-es sakktáblára feltesszük a két helyes királynő konfigurációt (lásd a 3.4. ábrát), akkor nem nehéz átlátni, hogy ezek a (2,4,1,3) és (3,1,2,4) vektorokkal kódolhatók.
– Fontos azonosítani, hogy milyen belső tulajdonság jellemzi a megoldáskódvektorokat!

2. Igyekszünk felállítani a 3.6. ábrán bemutatott modellt (az *x* tömb magassága kéz a kézben jár a megoldásvektorok hosszával).

– Azonosítjuk az A_k (k=1,2,…) halmazokat, ahonnan a megoldásvektorok elemei származnak. Már a kódoláskor figyelnünk kell arra, hogy az a_{k1}, a_{k2}, … értéksorozatok generálhatók legyenek (általában nem léteznek külön eltároltan). Fontos megjegyezni, hogy az A_k halmazok általában azonosak.

Ebből adódik, hogy minden szinten alapvetően ugyanazt a forgatókönyvet kell követni, amiért is olyan elegánsan egyszerű tud lenni a BT eljárás (főleg a rekurzív implementációja). Például a 4-királynő feladat esetén a modell a 3.7. ábra szerint alakul.

– A modellből, általában, egy az egyben adódik a BT eljárás.

3. Megírjuk az ígéretes *függvényt, amely az algoritmus kulcselemének tekintendő!*

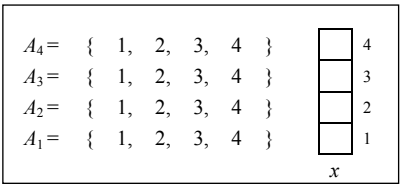
– Az ígéretességi feltétel a megoldás-kódvektorokra jellemző belső tulajdonságból következtethető ki.

4. Megírjuk a megoldás *függvényt!*

– Az ígéretességi feltételen túl még milyen feltételnek kell teljesülnie ahhoz, hogy a generált vektor megoldáskódnak bizonyuljon?

5. Megírjuk a kiír *eljárást!*

– A kiír eljárás a kurrens megoldásvektor dekódolását is tartalmazhatja. Például az n-királynő feladatnál megjeleníthetjük, sakktáblán, a megfelelő királynő-felállítást.



3.7. ábra. *A 4-királynő feladat modellezése*

3.8. ábra. *A 4-királynő feladat backtracking megoldásának balettkoreográfiája (AlgoRythmics 2020)*

3.4. Megoldott feladatok

A fenti recept használatának begyakorlásához az olvasó figyelmébe ajánljuk az *Algoritmusok felülnézetből* jegyzet (Kátai 2007) 2.3. alfejezetében bemutatott megoldott feladatokat.

Halmazműveletek: Generáljuk az $\{1, 2, \ldots, n\}$ halmaz
– p-szeres Descartes-szorzatának elemeit;
– összes permutációit;
– p-edrendű variációit;
– p-edrendű kombinációit;
– összes részhalmazát;
– összes partícióját.

Osztályterem: Adott n tanuló neve a *tanuló*[1..n] tömbben, akiket kétszemélyes padokba szeretnénk leültetni. Pontosan annyi pad van, amennyi szükséges (ha a tanulók száma páratlan, egy ülőhely üres marad). Írjuk ki az összes lehetőséget, ahogy a tanulók leültethetők az $(n+1)/2$ padba!

Urna: Adott egy urna, amelyben n piros és m fekete golyó található. Generáljuk az összes lehetőséget, ahogyan p ($p \leq n+m$) golyó kivehető az urnából!

Pénzérmék: Adottak az $a_1, a_2, \ldots, a_n$ értékű pénzérmék, mindenik típusból bármennyi. Írjuk ki az összes lehetséges módot, ahogyan egy s összeg kifizethető ezen pénzérmék segítségével.

Szuperprímek: Generáljuk az összes n számjegyű szuperprímet. Egy természetes számot szuperprímnek nevezünk, ha prím és a számjegyei jobbról balra sorrendben történő egyenkénti levágásával nyert számok (tekintsük ezeket a szám prefixeinek) is mind prímek. Például 239 szuper prím, mert 239, 23 és 2 mind prímek.

1/2 halmazok: Határozzuk meg az összes p elemű halmazt, amely rendelkezik az alábbi tulajdonságokkal:
– minden eleme n ($n \leq 32$) számjegyű;
– az elemek csak 1-es és 2-es számjegyeket tartalmaznak;
– bármely két eleme pontosan m ($m < n$) pozíción tartalmaz azonos számjegyeket;
– a számok egyetlen pozícióban sem tartalmaznak azonos számjegyeket.

Békák: Legyen egy s[1 . . $2n+1$] karakterlánc, amelyben az első n elem 0, az $(n+1)$-edik szóköz, a többi pedig 1-es. Generáljuk az összes lehetőséget, ahogyan

a nullások (fehér békák) helyet cserélhetnek az egyesekkel (fekete békák), a következő feltételek mellett:

– a fehér békák csak balról jobbra, a feketék pedig csak jobbról balra szökdöshetnek;
– egy béka akkor haladhat előre, ha előtte szóköz van, vagy ha az előtte lévő béka előtt szóköz van, ez utóbbi esetben átugorhatja az előtte lévő békát.

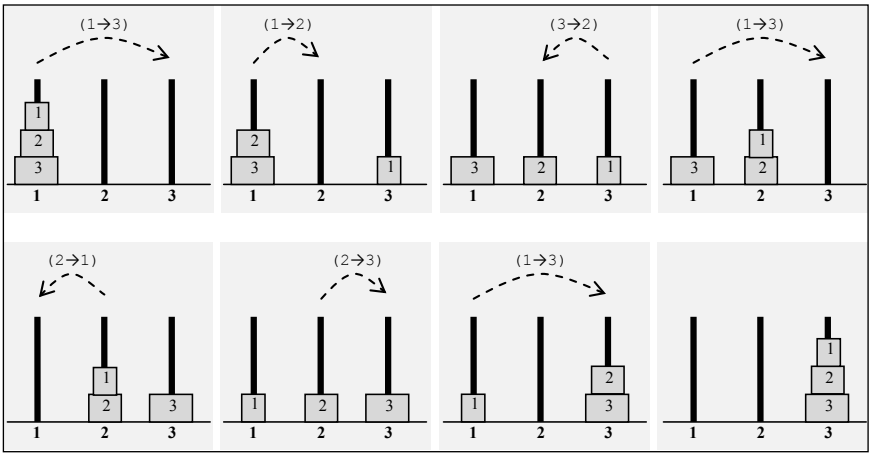
Labirintus: Adott egy labirintus az $a[1..n][1..m]$ bináris tömbben (1-essel kódoljuk a falat, 0-val a folyosót). Adott még egy személy pozíciója a labirintusban (az x koordináta a sorindex, az y az oszlopindex). Generáljuk az összes hurokmentes utat, amelyen a személy kijuthat a labirintusból.

Fénykép: Adott egy fekete-fehér kép, amelyet az $a[1..n][1..m]$ bináris mátrixban tároltunk (a 0 a fehér tartományokat, az 1 a fekete foltokat /tárgyakat/ ábrázolja). Állapítsuk meg a képen található tárgyak számát!

4. OSZD-MEG-ÉS-URALKODJ

Oszd-meg-és-uralkodj Hanoiban. A *Hanoi tornyai* matematikai játékot 1883-ban Édouard Lucas francia matematikus javasolta. Egy legendából inspirálódott, amely szerint a világ megteremtésekor egy 3 rúdból és 64 korongból álló feladványt kezdtek el „játszani” Brahma szerzetesei. A játék szabályai szerint az első rúdról a harmadikra kell átrakni a korongokat (felhasználva a másodikat is) úgy, hogy minden lépésben egy korongot lehet áttenni, és nagyobb korong nem tehető kisebb korongra. A legenda szerint, amikor a szerzetesek végeznek majd a korongok átjuttatásával, a kolostor összeomlik, és a világunk megszűnik létezni (Hanoi tornyai 2020).

A 4.1. ábra három korongra tartalmazza a megoldást (egy lépés definiálható egyszerűen úgy, hogy mely rúdról mely rúdra mozdítunk korongot, hiszen nyilvánvaló, hogy a forrásrúd tornyának legfelső korongját tesszük át a célrúd tornyának tetejére). (A feladat állapottér-reprezentációja végett lásd a Függeléket.)



4.1. ábra. *A Hanoi tornyai feladvány megoldása 3 korongra*

Az oszd-meg-és-uralkodj (divide-et-impera) stratégia felderítése végett maradjunk továbbra is Brahma szerzeteseinél:

– Tekintsük úgy, hogy a kolostor legidősebb szerzetese vállalta be a 64 (n) korong áthelyezésének felelősségét az 1. rúdról (forrásrúd) a 3. rúdra (célrúd), a 2. rúd (segítőrúd) segítségével;

- E legidősebb szerzetes felszólítja (akár egy uralkodó) legidősebb tanítványát, hogy ő helyezze át a felső 63 ($n-1$) korong alkotta tornyot, először a forrásrúdról (1. rúd) a segítőrúdra (2. rúd), a célrúd (3. rúd) segítségével;
- Ezután a legidősebb szerzetes átteszi a 64. korongot (legalsót) a forrásrúdról (1. rúd) a célrúdra (3. rúd);
- Majd újra int legidősebb tanítványának, hogy méltóztasson a korábban segítőrúdra (2. rúd) helyezett 63 ($n-1$) magasságú tornyot innen áthelyezni a célrúdra (3. rúd), a felszabadult forrásrudat (1. rúd) használva segítségként.

– A legidősebb tanítvány (valahányszor munkára fogják) nyilván hasonlóképpen uraskodik, mint mestere: szólítja a korban következő tanítványt (az ő közvetlen tanítványát), és leosztja neki a 62 ($n-2$) magas torony…

– A legifjabb tanítvány feladata lesz majd az 1 magasságú torony, azaz a legkisebb korong mozgatása.

Vegyük észre a következőket:

– Az 1. szerzetes, a legidősebb, 1-szer teszi a kezét az n. korongra; a 2. szerzetes 2-szer teszi a kezét az ($n-1$). korongra; a 3. szerzetes 4-szer teszi a kezét az ($n-2$). korongra;… az n. szerzetes, a legifjabb, 2^{n-1}-szer teszi a kezét az 1. korongra.

- mindez 2^n-1 korongelmozdítást jelent, ami $n=64$ esetén: 18 446 744 073 709 551 615 lépés;
- ha 1 lépés/másodperccel számolunk, akkor ez kb. 590 000 000 000 évet jelent.

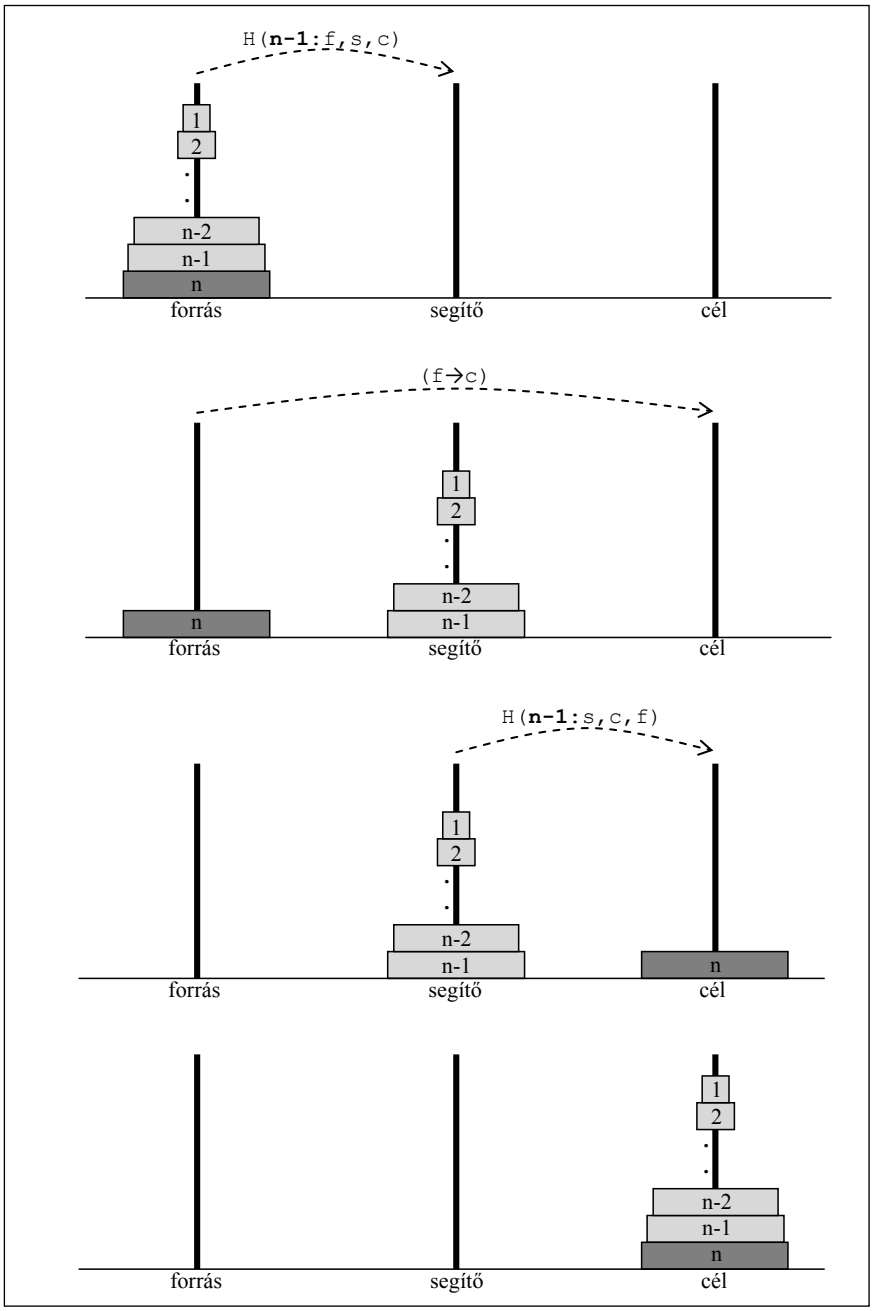
– „Oszd meg”: bármely $i>1$ magas torony egyszeri áthelyezésének problémája vissza lett vezetve az ($i-1$) magas torony *kétszeri* áthelyezésének problémájára;

- bármely szerzetesnek kiosztott feladat általános alakja: valamennyi korong áthelyezése adott forrásrúdról adott célrúdra, a fennmaradt rúd segítségével;
- az i. szerzetes 2^{i-1}-szer kell hogy megbirkózzon az i magasságú toronnyal (1,2,…i korongok).

– „és uralkodj” ($i>1$): a kurrens szerzetes kurrens feladatának (i torony: forrásrúdról, célrúdra, segítőrúddal) elvégzése azt feltételezi, hogy ütemezi a bevállalt egyetlen korong (a neki leosztott i magas torony legalsója) elmozdítását és a keze alá dolgozó közvetlen tanítványnak továbbosztott munkákat (lásd a 4.2. ábrát is):

- közvetlen tanítványa ($i-1$ torony: forrásrúdról, segítőrúdra, célrúddal);
- kurrens szerzetes (i. korong: forrásrúdról, célrúdra);
- közvetlen tanítványa ($i-1$ torony: segítőrúdról, célrúdra, forrásrúddal).

– Az $n=3$ esetre a $2^3-1=7$ lépés „oszd-meg-és-uralkodj zárójelezéssel”: ((1→3), 1→2, (2→3)), 1→3, ((2→1), 2→3, (1→3)).

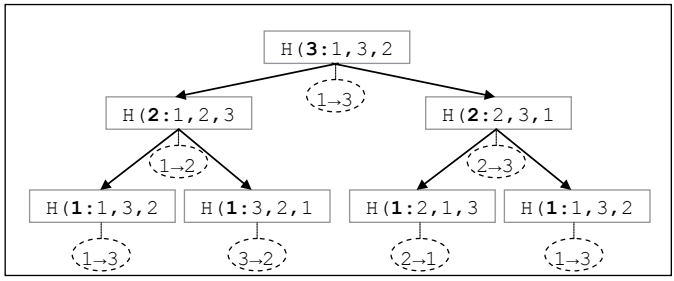


4.2. ábra. *Oszd-meg-és-uralkodj stratégia a Hanoi tornyai feladatra: H(***n***:f,c,s)*

4.1. Az oszd-meg-és-uralkodj módszer stratégiája

Milyen feladatok oldhatók meg az oszd-meg-és-uralkodj módszer segítségével? Azok, amelyek visszavezethetők, más szóval lebonthatók *két vagy több* hasonló, egyszerűbb (kisebb méretű) részfeladatra. E részfeladatok hasonlóak lévén, maguk is visszavezethetők további hasonló, még egyszerűbb részfeladatokra. Addig járunk így el, míg triviálisan egyszerű részfeladatokhoz jutunk, amelyek tovább nem bonthatók.

Egy fát látunk magunk előtt, amelynek gyökerében az eredeti feladat található. A fa első szintjén helyezkednek el azok a részfeladatok, amelyekre első lépésből bontható le a feladat. Mely részfeladatok kerülnek a második szintre? Nyilván azok, amelyek közvetlenül adódnak az első szinti részfeladatok lebontásából. A fa leveleibe a lebontásból adódó triviális részfeladatok kerülnek. Tekintettel, hogy bármely csomópont képviseli az illető gyökerű részfát, ezért úgy is gondolkodhatunk, hogy a teljes fa ábrázolja az eredeti feladatot, a fiúrészfák pedig a részfeladatokat. Mivel e fastruktúra minden csomópontjában hasonló feladatok találhatók, a részfeladatok általános alakjáról beszélhetünk. Az eredeti feladat úgy tekinthető, mint az általános feladat határesete, abban az értelemben, hogy méretben a legnagyobb. A triviális részfeladatok a másik határesetként foghatók fel, hiszen méretben a lehető legkisebbek (tovább nem darabolhatók). A 4.3. ábra a „Hanoi tornyai fát” mutatja be $n=3$ esetre.



4.3. ábra. *A Hanoi tornyai feladat részfeladatokra bontása n=3 esetben*

Az oszd-meg-és-uralkodj algoritmus rekurzívan közelíti meg a feladatot. Ebből adódik eleganciája. Az algoritmust implementáló rekurzív eljárást (vagy függvényt) nyilván az általános feladatra kell megírnunk és az eredetire meghívnunk. Az eljárásnak (függvénynek) különbséget kell tennie triviális és nem triviális részfeladat között. Más szóval két forgatókönyvet tartalmaz:

a) Triviális esetben fel kell vállalnia az illető részfeladat teljes megoldását. Ez fog a rekurzió megállási feltételeként szolgálni.

b) Ha nem triviális a feladat, megoldását, rekurzív hívások által, vissza kell vezetnie a közvetlen részfeladatai megoldására. Ez három lépésben valósítható meg:

1. meghatározzuk a közvetlen részfeladatokat, amelyekre az általános feladat lebontható („oszd meg…”);

2. rekurzív hívások által megoldatjuk e közvetlen részfeladatokat („… és uralkodj”);

3. a közvetlen részfeladatok megoldásaiból felépítjük az általános feladat megoldását.

Íme egy oszd-meg-és-uralkodj algoritmus lehetséges váza:

```
oszd_meg_és_uralkodj(<általános_feladat_paraméterei>)
   ha <triviális> akkor
      [oldd meg]
   különben
      [határozd meg a fiúrészfeladatokat]
      [rekurzív hívások által oldd meg ezeket]
      [építsd fel a megoldást]
   vége ha
vége oszd_meg_és_uralkodj
```

E stratégia, a Hanoi tornyai feladatra vonatkoztatva:

```
Hanoi(i,forrás,cél,segítő)                    //kurrens apafeladat
   ha i == 1 akkor                            //triviális eset
      ki: '(', forrás, '→', cél, ')'
   különben
      Hanoi(i−1,forrás,segítő,cél)            //fiúrészfeladat
      ki: '(', forrás, '→', cél, ')'
      Hanoi(i−1,segítő,cél,forrás)            //fiúrészfeladat
   vége ha
vége Hanoi
```

A fenti eljárást nyilván a Hanoi(n,1,3,2) alakban hívjuk meg.

Hogyan kerül bejárásra egy oszd-meg-és-uralkodj algoritmus révén a feladat lebontásából származó fastruktúra? A rekurzióból adódóan nyilván mélységében: a kurrens csomópont képviselte részfeladat megoldása ennek első fiúrészfeladata megoldásával kezdődik, amely a maga részéről az ő első fiúrészfeladatáéval, és így tovább. Bármely nem triviális részfeladat megoldása feltételezi a fiúrészfeladatok előzetes megoldást.

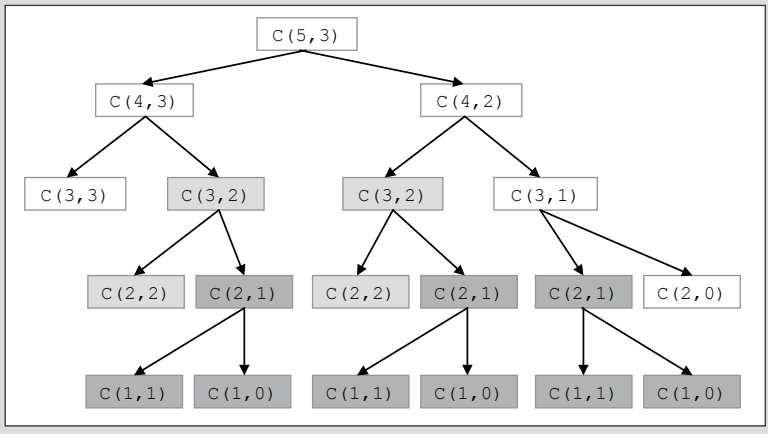
Bár a Hanoi tornyai problémát nagyon eltalálja az oszd-meg-és-uralkodj megközelítés, ez nem jelenti azt, hogy a többi módszer nem tudna hozzászólni. Egy sajátos felülnézet végett lásd a (Kátai–Kovács 2014) tanulmányt.

„Nem tudja a jobb kéz, mit csinál a bal”

Mi történik, ha a feladat lebontásakor a fa különböző ágain azonos részfeladatok jelennek meg? Példaként tegyük fel, hogy n elem k-adrendű kombinációinak számát szeretnénk kiszámítani oszd-meg-és-uralkodj algoritmussal a $C(n,k) = C(n-1,k) + C(n-1, k-1)$ képlet alapján [triviális esetek $C(i,i) = C(i,0) = 1$]. Első lépésben a $C(n,k)$ kiszámítása a $C(n-1,k)$ és a $C(n-1, k-1)$ kiszámítására vezetődik vissza. A második lépésben megjelenő részfeladatok pedig a következők lesznek: $C(n-2,k)$ és $C(n-2,k-1)$, valamint $C(n-2,k-1)$ és $C(n-2,k-2)$. Íme máris megjelent két azonos részfeladat. Mivel az oszd-meg-és-uralkodj stratégia a részfeladatokat egymástól függetlenül oldja meg („nem tudja a jobb kéz, mit csinál a bal”), az azonosak újra és újra megoldásra kerülnek, ahányszor csak találkozunk velük a fa bejárása közben (lásd 4.4. ábra).

Ilyen feladatok megoldására előnytelen oszd-meg-és-uralkodj algoritmust írni. Például megtörténhet, hogy a fa összes csomópontjának száma exponenciálisan függ a bemenet méretétől, bár a különböző részfeladatokat képviselők száma csak polinom értéke a bemenetnek. A 7. fejezetben megismerkedünk majd a dinamikus programozás módszerével, amelyre jellemző a már megoldott részfeladatok nyilvántartása és ezek eredményeinek tárolása (az azonos részfeladatok ismételt megoldása elkerülése céljából).

Érdekes, hogy a Hanoi tornyai feladat esetében is jelennek meg identikus részfeladatok a különböző lebontási ágak mentén [$n=3$ esetén a H(**1**:1,3,2) részfeladat; általános esetben a csatolt fastruktúrának már a második szintjén identikus csomópontok/részfák jelennek meg, a H(**n−2**:1,3,2) részfeladat képviseletében], de ezek ismételt végrehajtása szükségszerű.



4.4. ábra. *A C(5,3) feladat részfeladatokra bontásának fája. Szürkével jelöltük az identikus részfákat, amelyek ismételt végrehajtását nem kerüli el az oszd-meg-és-uralkodj stratégia*

4.2. Recept oszd-meg-és-uralkodj stratégiával megoldható feladatokhoz

Ha tisztázzuk az alábbi kérdéseket, akkor ezekből az algoritmus természetszerűen fog adódni:

1. *Hogyan vezethető vissza a feladat hasonló, egyszerűbb részfeladatokra?*
2. *Mi a feladat általános alakja? Melyek a paraméterei?*
   (Ezek lesznek az eljárás/függvény formális paraméterei.)
3. *Milyen paraméterértékekre kapjuk az eredeti feladatot?*
   (Ezekre hívjuk meg kezdetben az eljárást/függvényt.)
4. *Mi a triviálitás feltétele? Hogyan oldhatók meg a triviális részfeladatok?*
5. *Nem triviális esetben melyek az általános feladat közvetlen részfeladatainak a paraméterei?*
   (Ezek lesznek a rekurzív hívások aktuális paraméterei.)
6. Hogyan építhető fel a fiúrészfeladatok megoldásaiból az általános feladat megoldása?

A fenti sablont természetesen nem lehet egy az egyben minden feladatra ráhúzni. Az elv megértésében segítenek a most következő megoldott feladatok.

4.3. Megoldott feladatok

Legközelebbi pontpár keresése (Cormen 2003). Adott egy n ($n \geq 2$) elemű $Q=\{p_1(x_1,y_1), p_2(x_2,y_2), \ldots, p_n(x_n,y_n)\}$ síkbeli ponthalmaz, és érdekeltek vagyunk a legközelebbi pontpárban, illetve az elemei közti távolságban. Például légi vagy tengeri forgalomirányításnál célszerű lehet nyomon követni a két legközelebbi jármű helyzetét, a lehetséges ütközések előrejelzése érdekében. A nyers erő módszere azt jelentené, hogy egy $\theta(n^2)$ algoritmus révén párosítunk minden pontot minden ponttal, és kiválasztjuk a legközelebbi párost.

```
mintávolság = ∞
minden i = 1,n−1 végezd
   minden j = i+1,n végezd
      ha távolság(p_i,p_j) ≤ mintávolság akkor
         mintávolság = távolság(p_i,p_j)
      vége ha
   vége minden
vége minden
ki: mintávolság
```

Az „oszd meg és uralkodj” stratégia révén viszont elérhető az $\theta(n\log n)$ bonyolultság. Alkalmazzuk az előbbiekben javasolt 6 lépéses receptet:

1. Hogyan vezethető vissza a feladat hasonló, egyszerűbb részfeladatokra?

A Q ponthalmazt egy függőleges egyenessel kettéosztjuk egy bal oldali P és egy jobb oldali $Q\backslash P$ „egyenlő” elemszámú halmazra, és a Q-beli legközelebbi pontpárkeresés problémáját visszavezetjük az ezekben történő legközelebbi pontpár keresésére.

2. Mi a feladat általános alakja? Melyek a paraméterei?

Az általános alak egy kurrens P ponthalmaz legközelebbi pontpártávolságának meghatározása. A paraméterek legyenek a $k=|P|$ érték, valamint az $X[1..k]$ és $Y[1..k]$ tömbök, amelyek x, illetve y koordináták szerint rendezve (monoton növekvő sorrendben) tartalmazzák a P halmaz pontjait.

3. Milyen paraméterértékekre kapjuk az eredeti feladatot?

Az eredeti feladatra vonatkozó függvényhívás paraméterekként az n értéket, valamint az $X[1..n]$ és $Y[1..n]$ rendezett tömböket kapja meg (a rendezések előzőleg kell megtörténjenek $\theta(n\log n)$ időben).

4. Mi a triviálitás feltétele? Hogyan oldhatók meg a triviális részfeladatok?

A $k=2$ vagy $k=3$ méretű részfeladatokat fogjuk triviálisan egyszerűnek tekinteni. A $k=3$ esetben minimumot számolunk a három lehetséges pontpár-távolság között.

5. Nem triviális esetben melyek az általános feladat közvetlen részfeladatainak a paraméterei?

Jelölje P_{bal} és P_{jobb} a kurrens P halmaz kettéosztásából adódó halmazokat. Az $\theta(n\log n)$ bonyolultság biztosításához fontos, hogy a P_{bal} és P_{jobb} halmazoknak megfelelő (X_{bal},Y_{bal}) és (X_{jobb},Y_{jobb}) rendezett tömböket lineáris időben állítsuk elő a paraméterként kapott, P halmazra vonatkozó $X[1..k]$ és $Y[1..k]$ rendezett tömbökből. Mivel a kettéosztás függőleges egyenessel történik, ezért P_{bal}-ba az egyenes pozíciójánál kisebb (vagy egyenlő) x koordinátájú pontok ($k/2$ darab), P_{jobb}-ba pedig a nagyobb (vagy egyenlő) koordinátájúak (k-$k/2$ darab) kerülnek. Ennek megfelelően: $X_{bal}[1..(k/2)] = X[1..(k/2)]$ és $X_{jobb}[1..(k\text{-}k/2)] = X[(k/2+1)..k]$. Az $Y[1..k]$ rendezett tömb szétválogatása $Y_{bal}[1..(k/2)]$ és $Y_{jobb}[1..(k\text{-}k/2)]$ rendezett tömbökre történhet az alábbi eljárással (a merge-sor összefésülési technikájának a fordítottja):

```
a = b = 0
minden i = 1,k végezd
   ha Y[i] ∈ P_bal akkor
      a = a+1
      Y_bal[a] = Y[i]
   különben
      b = b+1
      Y_jobb[b] = Y[i]
   vége ha
vége minden
```

6. Hogyan építhető fel a fiúrészfeladatok megoldásaiból az általános feladat megoldása?

Ez a lépés is fontos, hogy lineáris időben történjen, ahhoz, hogy a végső algoritmus elérje az $\theta(n\log n)$ bonyolultságot. Ha P legközelebbi pontpárjának mindkét pontja vagy P_{bal}-ban vagy P_{jobb}-ban található, akkor a P-re vonatkozó visszatérési érték (jelölje d) közvetlenül felépíthető a P_{bal}-ra vagy P_{jobb}-ra vonatkozó rekurzív hívások visszatérítette értékekből (jelölje d_{bal} és d_{jobb}): $d = \min(d_{bal}, d_{jobb})$. De mi van akkor, ha P legközelebbi pontpárjának egyik pontja P_{bal}-ba, a másik pedig P_{jobb}-ba esik?

Tekintsünk a kettéosztó egyenestől balra is és jobbra is egy-egy d széles függőleges sávot. Ha van P-nek d-nél közelibb pontpárja az egyenes két oldalán, akkor ezek garantáltan ebben a sávban találtatnak. Töröljük Y-ból azokat a pontokat, amelyek e sávon kívül esnek. E szűkített Y (legyen Y') minden pontpárjának vizsgálata sem jöhet szóba, mert négyzetes időt feltételezne. Kell hát még egy ötlet.

Függőleges irányba is kijelenthető, hogy ha két pont közelebb van egymáshoz, mint a d érték, akkor ezek y koordináta szerinti távolsága sem lehet d-nél nagyobb. Tekintsünk ezért a $2d$ széles függőleges sávon belül egy d magas vízszintes sávot is. Belátható, hogy egy ilyen $2d \times d$ méretű téglalapba legfennebb 8 pont eshet. Mivel P_{bal}-ban a pontok távolsága nagyobb vagy egyenlő, mint d, ezért a téglalap bal négyzetébe legtöbb 4 P_{bal}-beli pont eshet, amennyiben ezek a 4 sarokban helyezkednek el. Ugyanez kijelenthető P_{jobb}-ra vonatkozóan is. A 8-as maximumot akkor érjük el, ha a kettéosztó egyenes és a téglalap metszéspontjaiba két-két azonos koordinátájú pontpár esik (ilyenek létezését nem zártuk ki), és a szétosztásnál ezek egyik tagja P_{bal}-ba, a másik pedig P_{jobb}-ba kerül. Ezen észrevétel hozadéka abban áll, hogy Y' elemeinek a pásztázásakor minden pontot csak a következő 7-tel kell párosítani, ami azt jelenti, hogy a belső ciklus konstansszor fut le, és így a művelet lineáris időben valósul meg. Amennyiben d-nél közelibb pontpárt találnánk, frissítjük d-t az ennek megfelelő távolsággal.

Az oszd-meg-és-uralkodj elnevezést (angolul „divide and conquer”) néha olyan algoritmusokra is alkalmazzák, ahol minden lépésben csak *egy* hasonló egyszerűbb részfeladattá redukálódik a kurrens feladat. Ilyen esetben találóbb lehet a „kicsinyítsd és urald” elnevezés (angolul „decrease and conquer”). Ilyen algoritmus a bináris vagy logaritmikus keresés, amikor is minden lépésben a keresés a rendezett sorozatnak vagy, az alsó vagy a felső felében folytatódik (lásd továbbá az *Algoritmusok felülnézetből* jegyzetben a 11.1.1. alfejezetet, Kátai 2007). Egy másik példa a gyorshatványozási algoritmus.

Gyorshatványozás: A 2.1. alfejezetben jeleztük, hogy létezik hatékonyabb, mint $\theta(n)$ algoritmus a hatványozásra, amennyiben van stratégiánk. Az alábbi oszd-meg-és-uralkodj szellemű algoritmus $\theta(\log n)$ bonyolultságú. Mivel minden

lépésben vagy az egyik, vagy a másik rekurzív hívásra kerül sor, ezért másfelől „kicsinyítsd és urald” algoritmusról van szó. Továbbá, mivel minden lépésben felére csökken az elvégzendő szorzások száma, ezért csak (log$n$) rekurzív hívásra kerül sor. Ez azt jelenti, hogy (log$n$)-szer kerül sor egy négyzetre emelésre, vagy egy négyzetre emelésre és egy plusz szorzásra (páratlan n-ek esetén).

Kicsinyítsd és urald

Akkor beszélünk „kicsinyítsd és urald” stratégiáról, ha minden lépésben a feladat (általában rekurzív hívások révén) egyetlen hasonló, egyszerűbb feladattá redukálódik. A kicsinyítés történhet egy adott értékkel (például $a^n = a^{n-1} \cdot a$), egy adott faktorral (például $a^n = a^{n/2} \cdot a^{n/2}$) vagy változó értékkel. Az utolsó esetre példa az alábbi rekurzív algoritmus, amely a legnagyobb közös osztót határozza meg:

```
lnko(m,n)
   ha m == n akkor
      return n
   különben
      return lnko(n,m%n)
   vége ha
vége lnko
```

Alakítsd át, hogy uralhasd

Hasonló gondolatmenetet alkalmaznak az „alakítsd át, hogy uralhasd” megközelítések. Ennek egyik változata az „egyszerűsítsd, hogy uralhasd”. Például gyakran hatékonyabb algoritmust eredményez, ha előre rendezzük a bemenetet. Tegyük fel, hogy az a feladat, hogy határozzuk meg, melyik érték fordul elő legtöbbször egy adott sorozatban. $\theta(n\log n)$ idő bonyolultságú megoldást eredményez, ha először rendezzük a számsort egy hatékony algoritmussal, majd megkeressük a leghosszabb konstans részsorozatát.

Egy másik változat az „ábrázold másként, hogy uralhasd” stratégia. Erre példa a Gauss-módszer lineáris egyenletrendszerek megoldásához.

Időnként a „vezesd vissza, hogy uralhasd” megközelítést célszerű alkalmazni. Ez azt jelenti, hogy igyekszünk a feladatot egy olyan feladatra visszavezetni, amelyre már ismerünk hatékony algoritmust. Példa lehet erre az, ahogy a legkisebb közös többszörös kiszámítását visszavezetjük a legnagyobb közös osztó kiszámítására:

lkkt(m,n) = (m·n) / lnko(m,n)

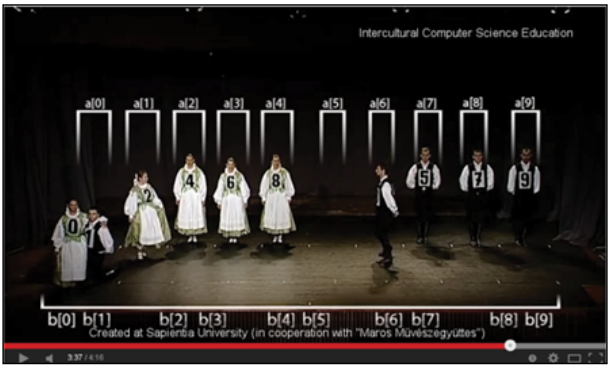
Az *Algoritmusok felülnézetből* című jegyzet (Kátai 2007) további nyolc oszd-meg-és-uralkodj algoritmust mutat be megoldott feladatokon:

Maximumkeresés: Adott egy számsorozat, amelyet az $a[1..n]$ tömbben tároltunk. Határozzuk meg a legnagyobb elem indexét!

Bináris keresés: Adott egy szigorúan növekvő számsorozat, amelyet az $a[1..n]$ tömbben tároltunk, valamint egy x szám. Keressük meg a számot a számsorozatban, és ha megtalálható, térítsük vissza a sorszámát, különben nullát!

4.5. ábra. *Bináris keresés flamenco tánccal (AlgoRythmics 2020)*

Mergesort: Rendezzük növekvő sorrendbe az $a[1..n]$ tömbben tárolt számsorozatot, a növekvő számsorozatok összefésülésére vonatkozó algoritmusra alapozva.



4.6. ábra. *Mergesort szász néptánccal (AlgoRythmics 2020)*

Quicksort: Rendezzük növekvő sorrendbe az $a[1..n]$ tömbben tárolt számsorozatot, a számsorozatok szétválogatására vonatkozó algoritmusra alapozva.

4.7. ábra. *Quicksort küküllőmenti legényessel (AlgoRythmics 2020)*

Koch-fraktál: Rajzoljuk ki a képernyőre az n-edik szintű, D szélességű Koch-fraktált.

Lemezdarabolás: Adott egy n méter hosszú és m méter széles téglalap alakú lemezdarab. A lemez bal alsó sarka a (0, 0), a jobb felső pedig az (n, m) koordinátájú pontban van. A lemezen p pontszerű lyuk található egész koordinátájú pozíciókban, amelyeket az $Ly[1..p]$ tömb tárol. Az i-edik lyuk koordinátáit az $Ly[i].x$ és $Ly[i].y$ mezők tárolják ($i = 1, n$). A koordinátatengelyekkel párhuzamos teljes vágásokat (minden vágással a lemez két részre esik) alkalmazva, vágjuk ki a legnagyobb lyukmentes területet a lemezből.

Négyzet és kör: Adott egy négyzet a bal alsó sarkának koordinátái és az oldalhossza által (a négyzet oldalai párhuzamosak a koordinátatengelyekkel), valamint egy kör a középpontja koordinátái és a sugara által. Határozzuk meg a metszési felületük területét háromtizedes pontossággal.

Mátrixszorzás: Adott két $n \times n$ méretű (n kettőnek a hatványa) mátrix, A és B. Szorozzuk össze őket hatékonyabban, mint a klasszikus módszer, amelyekkel mátrixokat szorzunk.

5. MOHÓ MÓDSZER

„Mohó kódok”. 1951-ben, az MIT-n (Massachusetts Institute of Technology) David Huffman és osztálytársai választhattak a vizsga és egy tudományos dolgozat írása között, „leghatékonyabb bináris kód” témakörben. Huffman már azon volt, hogy nekilát tanulni a vizsgára (merthogy hiába próbálkozott bármelyik kód optimalitásának bizonyításával), amikor hirtelen az az ötlete támadt, hogy használjon gyakoriságon alapuló bináris fát a kódoláshoz, amelyről könnyűszerrel bizonyítani is tudta, hogy a lehető leghatékonyabb módszer. Ez joggal lephette meg professzorát, Robert M. Fanot, aki Claude Shannonnal egy hasonló, de kevésbé hatékony kódot fejlesztett ki.

Huffman mohó megközelítést használt: ha szeretnénk, hogy egy szöveg *karakterenkénti* kódja (minden karaktert helyettesítünk a kódjával) a lehető legrövidebb legyen, akkor a gyakoribb karaktereknek rövidebb, a ritkábbaknak pedig hosszabb kódot célszerű választani.

Példaként tekintsük azt a szöveget, hogy „SAPIENTIA–INFORMATIKA”. A szöveg karaktereinek gyakorisági táblázatát az 5.1. ábra mutatja be.

A	I	N	T	E	F	K	M	O	P	R	S	–
4	4	2	2	1	1	1	1	1	1	1	1	1

5.1. ábra. *A „SAPIENTIA–INFORMATIKA” jelmondat karaktereinek gyakorisági táblázata*

Huffman egy bináris fát épített a gyakorisági tábla alapján. A szöveg karakterei a fa leveleibe kerülnek, és a megfelelő kódok a gyökér–levél utak bináris ábrázolásai lesznek (balra: 0, jobbra: 1). A mohó elvvel összhangban azt szeretnénk, hogy a gyakoribb karaktereket képviselő levelek magasabbra, a ritkábbakat ábrázolók pedig mélyebbre kerüljenek a fában. Huffman a fát lentről felfele (a levelektől a gyökér felé) irányba építette fel. Kezdetben minden karakter csomópontja külön álló egypontú fának tekintendő. E gyökércsomópontok mindegyike tárolja az illető karakter gyakoriságát (lásd az 5.2. ábrát). Minden lépésben összevonjuk a *két legkisebb* gyakoriságértékű gyökérrel rendelkező részfát (vastagított körök jelzik az ábrákon) oly módon, hogy létrehozunk részükre egy közös apacsomópontot. Az új apacsomópont (az egyesült részfák közös gyökere) gyakoriságértéke a fiúrészfák gyökerei gyakoriságértékeinek összege lesz.

A megadott példa esetében a felépült bináris fa gyökér–levél útjai képviselte kódok:

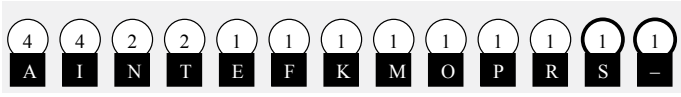
S: 1110; **A**: 000; **P**: 1000; **I**: 001; **E**: 0101; **N**: 011; **T**: 110; **F**: 01000; **O**: 1011; **R**: 1001; **M**: 1010; **K**: 01001; **–**: 1111.

Látható, hogy a gyakoribb karakterek (például az 'A' és az 'I') rövidebb kódokat kaptak, a ritkábbak pedig hosszabbakat. A „SAPIENTIA–INFORMATIKA" szöveg Huffman-kódja 74 bit hosszú lesz:

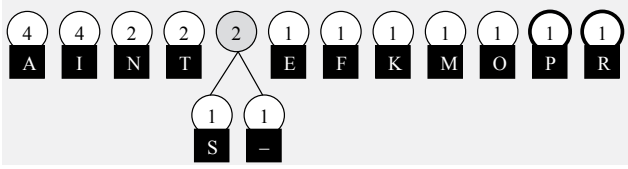
1110000**1000**001**0101**011**110**001**000**1111**001**011**01000**1011**001**1010**000**1 10**001**01001**000**

Ha minden karakternek azonos hosszú kódot választanánk, akkor a 13 karakter-kód kialakítása legalább 4 bit hosszú kódokat feltételezne, ami 84 hosszra rúgna fel a 21 karakter hosszú szöveg elkódolásakor.

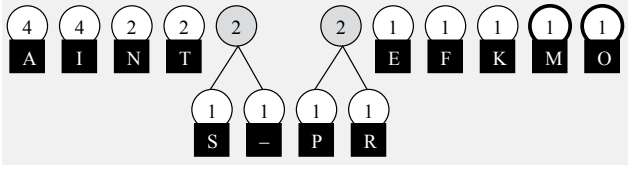
De miként tudjuk dekódolni a szöveget, ha változó hosszú kódokat használtunk? Meddig tart az első karakter kódja? Vegyük észre, hogy egyetlen kód sem prefixe a másiknak. Ez a tény összecseng azzal, hogy az *összes* kódolandó karakter *levél*-pontja a Huffman-fának (különböző gyökér–levél utak értelemszerűen nem lehetnek egymásnak prefixei). A Huffman-fán alapuló dekódolási algoritmus kurrens lépése abból áll, hogy indul a gyökérből, és megkeresi a fa azon levelét, amelyhez a szöveg-kód soron következő bitjei mint valami útjelző táblák (0=bal / 1=jobb) elvezetnek. Például az első karakterhez, az 'S' betűhöz úgy jutunk, hogy 1 (jobb) → 1 (jobb) → 1 (jobb) → 0(bal). Tehát a szöveg-kód 1110 szakasza kódolja az első betűt, azaz az 'S'-t.



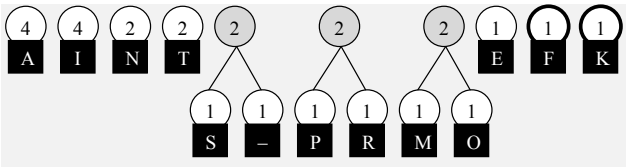
5.2.a. ábra. *Huffman-fa megépítése: 1. lépés*



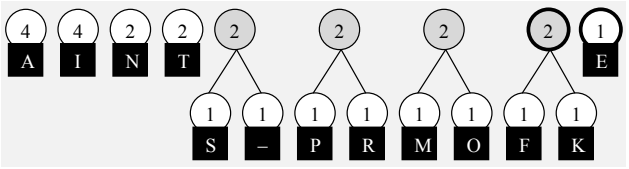
5.2.b. ábra. *Huffman-fa megépítése: 2. lépés*



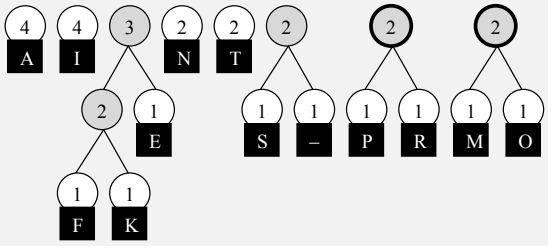
5.2.c. ábra. *Huffman fa megépítése: 3. lépés*



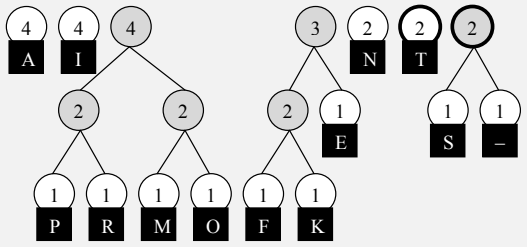
5.2.d. ábra. *Huffman-fa megépítése: 4. lépés*



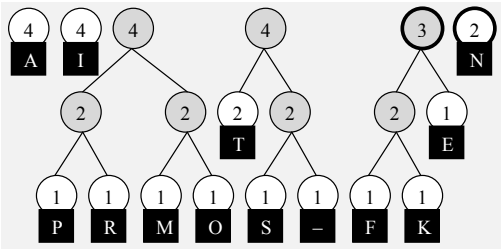
5.2.e. ábra. *Huffman-fa megépítése: 5. lépés*



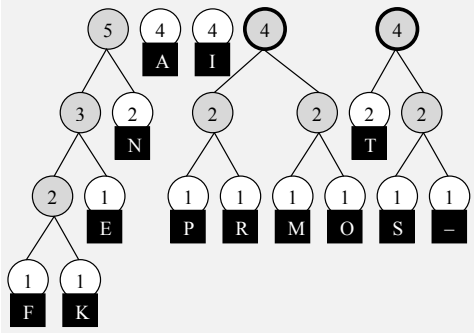
5.2.f. ábra. *Huffman-fa megépítése: 6. lépés*



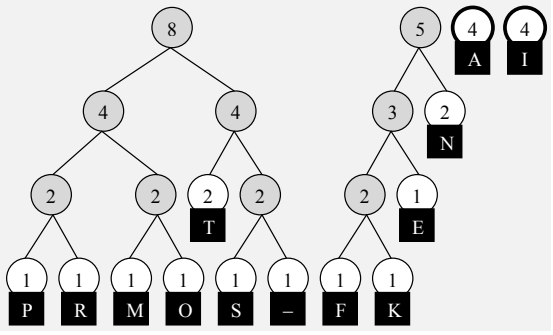
5.2.g. ábra. *Huffman-fa megépítése: 7. lépés*



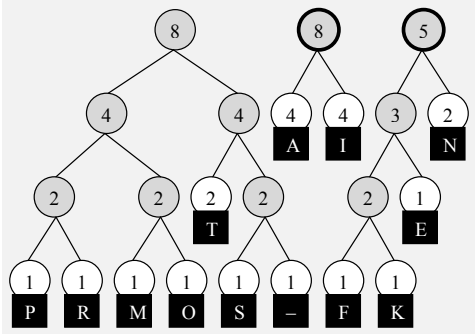
5.2.h. ábra. *Huffman-fa megépítése: 8. lépés*



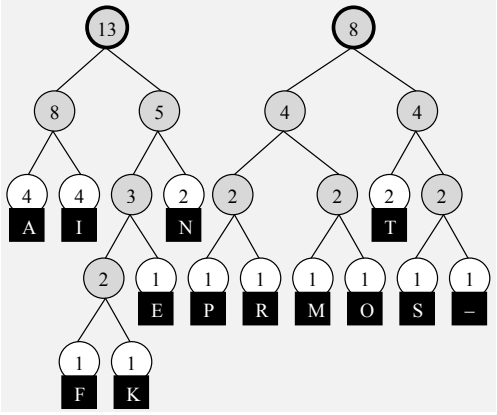
5.2.i. ábra. *Huffman-fa megépítése: 9. lépés*



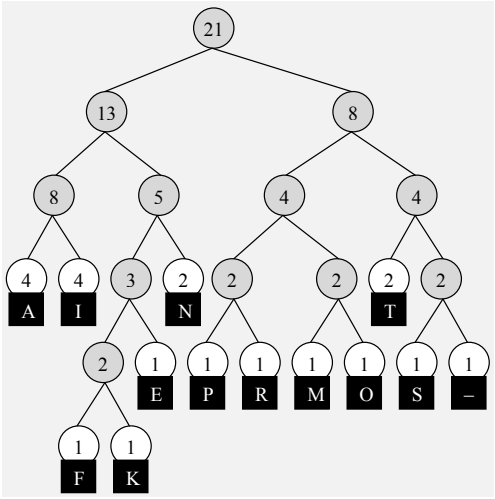
5.2.j. ábra. *Huffman-fa megépítése: 10. lépés*



5.2.k. ábra. *Huffman-fa megépítése: 11. lépés*



5.2.l. ábra. *Huffman-fa megépítése: 12. lépés*



5.2.m. ábra. *Huffman-fa megépítése: 13. lépés*

Általánosan is kijelenthető, hogy a mohó módszert optimalizálási feladatok megoldására használjuk. Íme néhány további példa mohó feladatra (*Algoritmusok felülnézetből* jegyzet; Kátai 2007):

1. Felvonó: Egy egyszemélyes felvonó előtt n személy áll, akikről ismert, hogy hányadik emeletre szeretnének feljutni (e_1, e_2, …, e_n). Milyen sorrendben kellene hogy használják a felvonót, ha azt szeretnék, hogy a várakozási idejük összege *minimális* legyen? Mennyi lesz ez az összidő, ha tudjuk, hogy a felvonó időegységekként egy emeletmagasságot tesz meg, és a kiszállás és beszállás „szempillantás alatt” történik?

2. Műsorok: Adott n tévéműsor, amelyeknek ismert a kezdési és befejezési időpontjuk: (b_1, e_1), (b_2, e_2), . . . , (b_n, e_n). Egy család, amelynek egy tévékészüléke van, úgy dönt, hogy a $[B,E]$ időintervallumban ($b_i \geq B$, $e_i \leq E$, $i = 1,n$) tévézni fog. Mely műsorokat választják (lehetnek átfedődő műsorok is, amelyeket természetesen más-más kanálison közvetítenek), ha azt szeretnék, hogy a *legtöbb* műsort lássák (a kiválasztott műsorokat teljes egészében megnézik)? *Legkevesebb* hány tévékészülékre lenne szükségük (és legalább hány tagú kellene hogy legyen a család) ahhoz, hogy minden műsort megnézhessen legalább egy családtag?

3. Telefonhálózat[6]: N számú város között telefonhálózatot szeretnének kiépíteni. Egy $d[1..m]$ egydimenziós tömbben adott, hogy mely várospárok között építhető ki direkt telefonvonal, valamint hogy ezek a kapcsolatok egyenként mennyibe kerülnének. Az i-edik közvetlen vonal végpontjait, valamint megépí-

6 Ma már nyilván nem így történik a telefonkapcsolatok kiépítése.

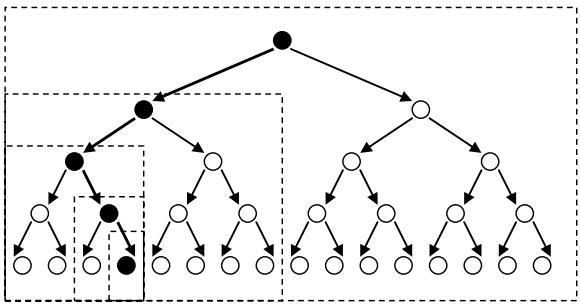
tési költségét a $d[i].x$, $d[i].y$ és $d[i].k$ mezők tárolják. Mely városok között építsék ki a közvetlen telefonvonalakat ahhoz, hogy összekapcsoljanak minden várost (legalább közvetve), és az összköltség *minimális* legyenek?

4. Madarak: Adott $n+1$ fa, amelyek 1-től $(n+1)$-ig vannak megszámozva. Az első n fa mindenikén elhelyezkedett egy-egy madár. Ezeket is megszámozzuk 1-től n-ig. Az i-edik fára az i-edik madár szállt ($i = 1, n$). A madarak elkezdenek áthelyezkedni. Minden lépésben valamelyik madár átrepül az éppen üres fára (egyszerre csak egyetlen madár van a levegőben). Ismerve, hogy egy idő után melyik madár éppen melyik fára szállt, „repítsük vissza" a madarakat eredeti helyükre *minimális* számú repüléssel.

5. Legrövidebb utak: Adott egy $n \times n$ méretű d mátrix, amely egy n várost összekötő úthálózatot ábrázol. A $d[i][j]$ elem az i és j városok közti közvetlen út hosszát tárolja (ha két város közt nincs direkt út, a megfelelő mátrixelemek értéke ∞). Határozzuk meg az első várostól az összes többihez vezető *legrövidebb* utakat és ezeknek a hosszát.

A feladatok szövegében dőlt betűkkel írtuk azokat a szavakat, amelyek az optimalizálás gondolatát hordozzák.

A mohó feladatokban általában arról van szó, hogy egy döntéssorozattal kell meghatározni az optimális megoldást. Egyes esetekben egyszerűen az optimális sorrendet kell megtalálni, máskor egy halmaznak valamilyen szempontból vett optimális részhalmazát, stb. Fontos megjegyezni azonban, hogy a második esetben is általában az vezet el az optimális részhalmazhoz, ha optimális sorrendben vizsgáljuk meg a halmaz elemeit.



5.3. ábra. *Mohó döntéssorozat a feladat struktúráját szemléltető fán*

Újra egy fa elevenedik meg előttünk, éspedig egy döntési fa, amely a feladathoz rendelhető (lásd az 5.3. ábrát). Minden döntéssel a feladat kisebb és kisebb méretű hasonló feladatokká redukálódik, amelyek az eredeti feladat egymásba ágyazott részfeladatainak tekinthetők (ezt szemléltettük a szaggatott vonalú keretekkel). Az eredeti fa az első döntéssel leszűkül valamelyik fiúrészfájára (azzal,

hogy választunk, mintha lemetszenénk a fáról a többi fiúrészfát), majd a következő döntéssel ennek valamelyik fiúrészfájára, és így tovább, míg már csak egy levél marad belőle.

Az optimális döntéssorozatot, amely elvezet az optimális megoldáshoz, a fa valamelyik gyökér–levél útja képviseli. Nevezzük el ezt az utat optimális útnak, az illető levelet pedig optimális levélnek. Ezek után úgy is megfogalmazható egy mohó feladat, hogy keressük meg a feladathoz rendelhető döntési fa optimális gyökér–levél útját.

5.1. A mohó módszer stratégiája

A mohó algoritmusok filozófiája nagyon egyszerű, és azokra az emberekre emlékeztet, akik a mának élnek. Neve jelentésével összhangban, mindig az adott lépésben optimálisnak látszó döntést hozza (mi a legígéretesebb választás tekintettel az optimalizálási követelményekre), nem számolva az esetleges hosszú távú következményekkel. Úgy gondolkodik, hogy a lokális optimum majd globális optimumhoz vezet. Amennyiben ez nem igaz az illető feladatra, vagy nem talál megoldást, vagy nem találja meg az optimálisat (legfennebb véletlenül, bizonyos sajátos esetekben).

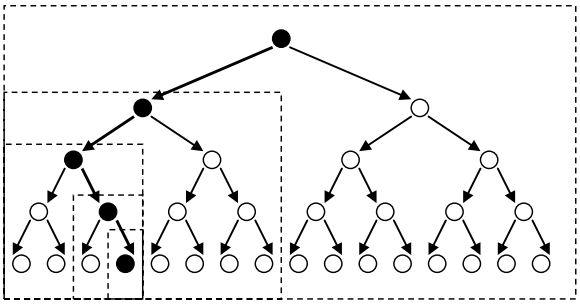
Például ha a feladat az, hogy fizessünk ki egy S összeget az a_1, a_2, …, a_n értékű pénzérmékkel úgy, hogy minimális számú érmét használunk, akkor egy mohó megközelítés az lehetne, hogy az érmékkel értékeik szerinti csökkenő sorrendben próbálkozunk. A következő inputokra az előbbiekben jelzett módokon fogna mellé a mohó módszer: $S=11$, $a[1..3]=\{2,3,5\}$; $S=9$, $a[1..3]=\{1,3,5\}$. A 7. fejezetben dinamikus programozásos megoldást adunk e feladatra.

Megjegyzés: Mivel a mohó módszer alapgondolata, miszerint a lokális optimum globális optimumhoz vezet, általánosságban nem igaz, ezért a teljes megoldáshoz az is hozzátartozik, hogy bizonyítjuk, hogy a szóban forgó feladatra viszont igaz. Ennek ellenére – ha beérjük kevesebbel is, mint az optimális megoldás – célszerű lehet minden olyan esetben alkalmazni a mohó megközelítést (még ha nem is bizonyítható a nyert algoritmus helyessége), amikor minden más stratégia túl bonyolult vagy túl nagy időigényű algoritmust eredményezne.

Lássuk, melyek lennének a mohó választások a fenti példákban:

1. Az elsőként felvonózó személy utazási ideje (pontosabban ennek kétszerese, mert a felvonó vissza kell hogy jöjjön a földszintre) bekerül az összes többi lakó várakozási idejébe, azaz $(n-1)$-szeresen kerül bele az összvárakozási időbe. Általánosan, az i-edikként utazó lakó feljutási ideje $(n-i)$-szeresen kerül bele az összvárakozási időbe. Nyilvánvaló hát, hogy minden lépésben a következő *legalacsonyabban lakó* személy célszerű, hogy használja a felvonót, hogy a kisebb utazási idők kerüljenek bele sokszorosabban az összvárakozási időbe.

2. Elsőre úgy gondolhatnánk, hogy a legrövidebb műsort célszerű elsőként programra tűzni. Erre ellenpélda az, amikor $E=0$, $B=24$, $n=3$, $b_1=0$, $e_1=12$, $b_2=12$, $e_2=24$, $b_3=11$, $e_3=13$ (lásd az 5.4. ábrát). Ha a legrövidebb műsort (a 3-ast, amely 2 óra hosszúságú) választjuk, akkor ezzel a másik kettőt, mint ezt átfedőket, alapból kizárjuk. Ezzel szemben választhattuk volna a két 12 órás műsort.



5.4. ábra. *Szemléltető példa arra az esetre, amikor a legrövidebb műsor (3-as) nem része az optimális megoldásnak (1-es és 2-es műsorok)*

A helyes mohó választást a *leghamarabb befejeződő* műsor jelenti, merthogy így a lehető leghosszabb *folytonos* időszakasz marad fenn további választásokra. Ha a soron következő legígéretesebb műsor átfedődik a már kiválasztottakkal, egyszerűen kihagyjuk.

3. Mindig a *következő legolcsóbb* szakaszon építjük ki a telefonvonalat, ügyelve arra, hogy ne építsünk közvetlen vonalat ott, ahol már létrejött közvetett kapcsolat. Egy másik lehetőség, hogy kezdve bármelyik várossal, minden lépésben a *legolcsóbban csatolható* várost kapcsoljuk a már összekötöttekhez.

4. Arra törekszünk, hogy mindenik madár a lehető *legkevesebb repüléssel* kerüljön a helyére. Tehát mindig az a madár fog repülni, amelyiknek a fája éppen üres. Ily módon az illető madár egy repüléssel a helyére kerül. Ha egy adott pillanatban az $(n+1)$-edik fa válik üressé, és nincs még minden madár a helyén, akkor ezek közül valamelyik elrepül az $(n+1)$-edik fára. Ez a madár végül két repülésből kerül a fájára.

5. Mindig a már elért városokhoz *legközelebb eső* város lesz a következő, amelyikhez meghatározzuk a legrövidebb utat. Ez a sorrend biztosítja majd, hogy minden városhoz a legrövidebb úton jussunk el.

Számos mohó feladat esetében az alábbi helyzet ismerhető fel.

Létezik egy úgynevezett „jelöltek halmaza” (az elemei arra pályáznak, hogy az optimális megoldás részét képezzék), és egy bizonyos „célfüggvény”, amely a jelölthalmaz részhalmazainak halmazán értelmezett. A feladat abból áll, hogy

a jelölthalmaz adott tulajdonságú részhalmazai közül meg kell határozni azt a részhalmazt, amelyik optimalizálja a célfüggvényt. Íme a mohó algoritmus erre a helyzetre:

```
greedy(J)
  S = ∅
  amíg nem megoldás(S) és J ≠ ∅ végezd
      x = legígéretesebb(J)                    (1)
      J = J \ {x}                              (2)
    ha bővíthető(S,x) akkor
      S = S ∪ {x}                              (3)
    vége ha
  vége amíg
  ha megoldás(S) akkor
    kiír(S)
  különben
    ki: „Nincs megoldás”
  vége ha
vége greedy
```

Ahol:
– J: jelölthalmaz.
– S: ebben a halmazban építjük fel a megoldást.
– legígéretesebb(J): a mohó döntés alapján kiválasztja a jelölthalmaznak az adott pillanatban legígéretesebbnek tűnő elemét. Mivel ennek a függvénynek a szerepe a célfüggvény optimalizálása, ezért ez utóbbiból következtethető ki. A legígéretesebb függvény a lokális optimumot biztosítja, a célfüggvény pedig a globálist.
– bővíthető(S,x): Ellenőrzi, hogy az *x* elemmel bővíthető-e az *S* halmaz, azzal a kilátással, hogy végül a megoldáshalmaz a kívánt tulajdonságú legyen. Tehát onnan vezethető le, hogy a jelölthalmaz mely tulajdonságú részhalmazai között keressük az optimálist. Ha a feladat azt kéri, hogy az eredeti halmazra határozzuk meg a célfüggvény optimumát (lásd az 1. példafeladatot), akkor a bővíthető függvény elveszíti szerepét.
– megoldás(S): ellenőrzi, hogy felépült-e a megoldás.
– kiír(S): kiírja a megoldást.

A mohó stratégia kulcsgondolata az eljárás (1), (2) és (3) soraiban testesül meg:
1. Mindig az adott pillanatban legígéretesebbnek látszó jelöltet választjuk (a döntési fa aktuális csomópontjának legígéretesebb fiát).
2. A kiválasztott jelöltet „örökre” eltávolítjuk a jelölthalmazból.
3. Ha a megoldáshalmaz bővíthető az illető elemmel, akkor végérvényesen beletesszük, ha nem, akkor végképp lemondunk róla.

Tehát egy mohó algoritmusban nincs visszalépés (back-track). Amint mondani szokás, mindent feltesz egy lapra. Hogyan állapítható meg, hogy egy feladat megoldható-e a mohó stratégiával? Nincs általános módszer, azonban van két alapelv, amelyek ha érvényesek az illető feladatra, akkor bizonyíthatóan alkalmazható rá a mohó stratégia. Szerencsére az esetek nagy többségében ez járható út.

5.2. A mohó stratégia háttere

5.2.1. A mohó választás alapelve

Emlékezzünk, hogy a mohó stratégia szerint mindig az adott lépésben legjobbnak tűnő választást hajtjuk végre, bármelyik legyen is az, majd ezt követően megoldjuk azokat a részfeladatokat, amelyekre választásunk nyomán a feladat leszűkült. Ezzel összhangban az aktuális választás függhet (figyelembe tudja venni) az előző döntésektől, de nem függhet a későbbiektől, hiszen mindez még a jövő titka. Úgy is fogalmaztunk, hogy a greedy algoritmusok fentről lefele haladva, egymást követő mohó döntések által a feladatot mind kisebb és kisebb méretű feladattá redukálják. Ez azt jelenti, hogy a döntési fának egyetlen gyökér–levél útját generálják, bízva abban, hogy pontosan ez lesz az optimális.

Ezek után a mohó választás alapelve a következőképpen jelenthető ki:

1. a feladat *optimális* megoldása mohó választással *kezdődi*k (vagy módosítható úgy, hogy mohó választással kezdődjön),
2. és e választás nyomán a feladat *hasonló* feladattá redukálódik.

Természetesen, ha a mohó választás nyomán nyert feladat hasonló, akkor ennek az optimális megoldása is mohó választással fog kezdődni (vagy módosítható, hogy azzal kezdődjön), és így tovább… A feladat optimális megoldásának ez a tulajdonsága szükséges feltétel annak bizonyításához, hogy a globális optimum felépíthető lokális optimumok (mohó döntések) sorozataként. Mi hiányzik még a teljes bizonyításhoz?

Legyen $D_1, D_2, \ldots, D_n$ a feladat optimális megoldása. Pontosabban, az az optimális döntéssorozat, amely a globális optimumot biztosítja. Azt szeretnénk bebizonyítani, hogy ezen döntések mindenike egyenként mohó választás, azaz lokális optimum. Mit biztosít ebből a mohó választás alapelve? Azt, hogy D_1 biztosan mohó választás. Mire van még szükség ahhoz, hogy a mohó választás alapelvéből következzen a D_2 döntés mohó volta is? Ehhez a $D_2, D_3, \ldots, D_n$ döntéssorozatnak is optimálisnak kell lennie, annak a részfeladatnak az optimális megoldásának, amellyé a feladat az első mohó döntés nyomán redukálódott. Általánosan: a D_i ($i = 2,n$) döntések mohó voltának bizonyításához további szükséges feltétel, hogy a $D_i, D_{i+1}, \ldots, D_n$ alakú részdöntéssorozatok ugyancsak optimálisak legyenek. Pontosan ezt mondja ki a következő alapelv.

5.2.2. Az optimalitás alapelve

A feladat optimális megoldása a részfeladatok (amelyekre a feladat a mohó döntések nyomán redukálódik) optimális megoldásaiból épül fel. Ez azt jelenti, hogy ha $D_1, D_2, \ldots, D_n$ egy optimális döntéssorozat, akkor ebből következik, hogy a $D_i, D_{i+1}, \ldots, D_n$ alakú részdöntéssorozatok ugyancsak optimálisak az illető részfeladatokra nézve.

Végkövetkeztetésként leszögezhetjük, hogy ha egy feladat optimális megoldására érvényes a mohó választás, valamint az optimalitás alapelve, akkor megoldható greedy módszerrel.

5.2.3. A mohó stratégia helyességének bizonyítása

Az alábbiakban be fogjuk mutatni a műsorok feladat kapcsán a mohó algoritmus helyességének bizonyítását.

Legyen $K = \{1, 2, \ldots, n\}$ a műsorok (jelöltek) halmaza, és $S \subseteq K$ egy optimális részhalmaz (a legszámosabb, amely nem tartalmaz átfedődő műsorokat). Továbbá legyen p a legígéretesebb műsor, amely leghamarabb fejeződik be.

Először is bebizonyítjuk, hogy van olyan optimális megoldás, amely a mohó döntéssel kezdődik, azaz hogy $p \in S$, vagy módosítható S úgy, hogy p eleme legyen. Tegyük fel, hogy $p \notin S$, és legyen q az S-nek azon eleme, amelynek a legkisebb a befejezési ideje. Ha felcseréljük q-t p-vel (ezt megtehetjük, mert a p műsor hamarabb befejeződik, mint a q műsor), egy másik optimális megoldást kapunk. Tehát létezik olyan optimális részhalmaz, amely tartalmazza a legígéretesebb műsort.

Vegyük észre, hogy az első mohó döntés nyomán a feladat a következő részfeladattá redukálódott: *Legyen $K_1 \subset K$ az a halmaz, amelyet úgy kapunk, hogy eltávolítjuk K-ból a p műsort és a vele átfedődő műsorokat. Határozzuk meg K_1-nek egy legszámosabb részhalmazát, amely nem tartalmaz átfedődő műsorokat.* Ez valóban az eredeti feladattal azonos, de kisebb méretű.

Be fogjuk bizonyítani, hogy az $S-\{p\}$ részhalmaz optimális megoldása a fenti részfeladatnak. Mivel S nem tartalmaz átfedődő műsorokat, $S-\{p\} \subset K_1$. Tegyük fel, hogy létezik $B \subset K_1$ úgy, hogy B számosabb, mint $S-\{p\}$, és nem tartalmaz átfedődő műsorokat. Ez esetben viszont $B \cup \{p\}$ jobb megoldása lenne az eredeti feladatnak, mint S. Ez viszont ellentmond a feltételnek, miszerint S optimális megoldása a feladatnak. Tehát a feladatra érvényes az optimalitás alapelve is.

Mindezen elméleti fejtegetések után biztosan érdekli az olvasót, miként is közelíthet meg egy mohó feladatot? Sajnos (vagy éppen ez a szép benne) nincs recept ezt illetően. Talán ez a módszer az, amelyik, bár a legegyszerűbb, mégis a legtöbb leleményességet követeli. Különösen ahhoz kell leleményesség, hogy helyesen azonosítsuk a mohó választást, amely a feladatot hasonló feladattá redukálja. Mindenképpen ez az első lépés. Az vezethet nyomra ebben, hogy mit is

kell optimalizálni. Ezért mondtuk, hogy a legígéretesebb függvény a célfüggvényből következtethető ki. Ha az előbbiekben bemutatott algoritmusváz alkalmazható a feladatra, akkor második lépésben a bővíthető függvényt kellene megírni. A többi függvény általában magától értetődő.

A fenti algoritmusvázzal azonban nem az volt a célunk, hogy sablont adjunk az olvasónak, inkább azt ajánljuk, hogy sajátítsa el a mohó gondolkodás szellemét, és sajátosan alkalmazza az egyes feladatokra. Ebben segíthet a fentebb bemutatott hat feladat megoldása, ahogyan ezeket az *Algoritmusok felülnézetből* jegyzet kifejti (Kátai 2007).

5.2.4. A mohó algoritmusok heurisztikája

Ahogy arra már utaltunk az 5.1. alfejezetben, bizonyos mohó algoritmusok nem mindig nyújtják az optimális megoldást, sőt egyes bemenetekre úgy találhatják, hogy nincs megoldás, holott a valóságban létezik. Az ilyen algoritmusokat heurisztikus megoldásoknak nevezzük. Az alábbiakban két olyan feladatot mutatunk be, amelyek esetében csak heurisztikus mohó algoritmusok ismertek. Mivel e feladatok esetében a „tökéletes megoldást” nyújtó algoritmusok időigénye nagy bemenetekre túl nagy lehet, úgy dönthetünk, hogy beérjük a mohó algoritmusok nyújtotta elég jó megoldásokkal.

Térképszínezés: Adott egy térkép, amely n országot tartalmaz, és ismert, melyik ország melyikkel szomszédos. Színezzük ki a térképet úgy, hogy ne kapjon két szomszédos ország azonos színt, és minimális legyen a használt színek száma.

Egy mohó megközelítés a következő lehetne: Veszem az első színt, és kifestem vele a lehető legtöbb országot. A következő színekkel hasonlóképpen járok el. Bár bizonyítható, hogy bármely térkép kifesthető legfennebb négy színnel, az imént felvázolt algoritmus nem mindig fogja megtalálni a minimális színű megoldást.

Utazó kereskedő: Ismert egy ország úthálózata, azaz hogy mely városok között vannak direkt utak, és mekkora ezek hossza. Határozzuk meg (egy utazó kereskedő részére) a legrövidebb olyan körutat, amely minden várost érint egyszer és csakis egyszer (kivéve az indulási várost).

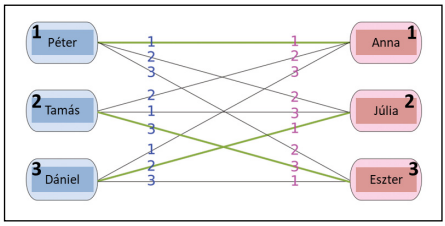
Egy lehetséges mohó gondolatmenet, hogy az utazó kereskedő mindig az aktuális városból közvetlen elérhető legközelebbi – még nem érintett – várost válassza. Nem nehéz találni olyan példákat, amikor ez az algoritmus nem adná meg a legjobb megoldást, vagy egyszerűen nem találna megoldást (holott létezik). Visszatérünk erre a feladatra a 8. fejezetben.

A fenti feladatokat megoldó heurisztikus algoritmusok implementálását az olvasóra hagyjuk.

5.3. Egy díjazott mohó algoritmus

Az alábbiakban az úgynevezett stabilházasság-problémát mutatjuk be, amelyre 1962-ben David Gale és Lloyd Shapley közöltek egy mohó algoritmust (Gale–Shapley 1962). Azóta az algoritmust számos tudományterületen alkalmazták sikeresen. A történet rendkívülisége abban áll, hogy 2012-ben a szerzők megkapták algoritmusukért a közgazdasági Nobel-emlékdíjat.

A stabil házasság problémája. Legyen n férfi és n nő, illetve ezek preferencialistái a másik nem tagjait illetően (lásd az 5.5. ábrát). Párosítsuk össze a nőket és a férfiakat n házasság keretében úgy, hogy a „házasságok" stabilak legyenek (nincs olyan férfi és nő, akik egymás preferencialistáján előrébb szerepelnek, mint azok, akivel ténylegesen összepárosítottuk őket).

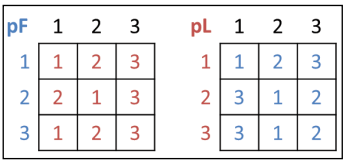


5.5. ábra. *Példa n=3 esetre. Minden élre ráírtuk, hogy az illető kapcsolatot milyen szinten preferálja a megfelelő fiú, illetve lány (például Tamás a Júlia, Anna és Eszter sorrendben preferálja a lányokat, Júlia pedig Dániel, Péter és Tamás sorrendben a fiúkat). A megvastagított nyilak ábrázolta kapcsolatok stabil házasságokat jelentenek.*

Megoldás: Bizonyított, hogy a következő mohó stratégia stabil házasságokhoz vezet:

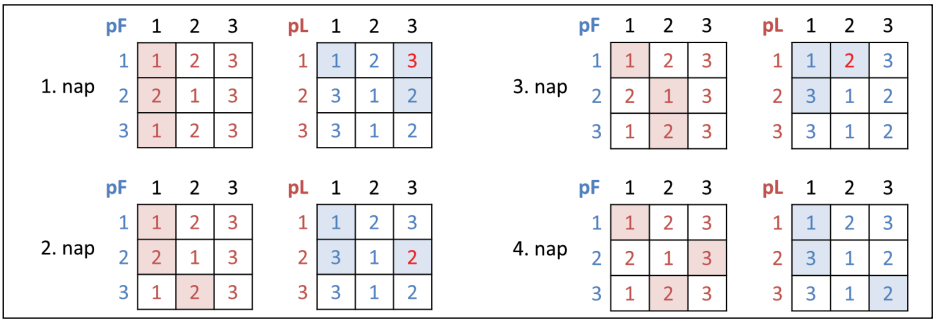
- 1. nap: Minden férfi elmegy a legpreferáltabb lány udvarába szerencsét próbálni. Ha egy lány udvarába többen érkeznének, akkor a legpreferáltabb marad (az udvarban), a többi pedig ki van „kosarazva" (hazaküldve).
- 2. nap: Minden „kikosarazott" férfi elmegy a következő legpreferáltabb lány udvarába. Ahogy az előző nap, a legpreferáltabb marad, a többi pedig ki van „kosarazva". Ha valamely lány udvarába preferáltabb férfi érkezik, mint aki az előző napról ott maradt, akkor ez utóbbit, a tegnapit, hazaküldi.
- Stb. Mindezt addig folytatjuk, míg mindenkinek megkerül a párja.

Lássuk, hogy működik az algoritmus a megadott példán. A *pF* és *pL* tömbök a fiúk, illetve a lányok preferencialistáit tárolják (5.6. ábra).



5.6. ábra. *Inputadatok a szemléltető példához (a sorindexek fiúkat/lányokat, az oszlopindexek sorrendet, az értékek pedig lányokat/fiúkat jelentenek; kék színnel ábrázoltuk a fiúkat és rózsaszínnel a lányokat)*

Az 5.7. ábra azt szemlélteti, hogy miként vezet megoldáshoz az algoritmus 4 menetből.

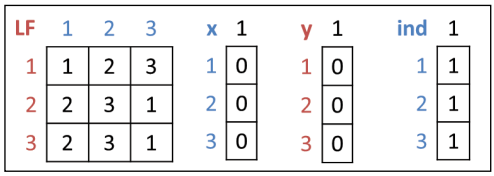


5.7. ábra. *A pF tömbben azt satíroztuk be, hogy melyik lánynál tart az illető fiú, a pL-ben pedig azt, hogy mely fiúk érkeztek az illető lányhoz. A piros tömbelem kikosarazott fiút ábrázol*

A bemeneti tömbökből előállítjuk a $LF[1..n][1..n]$ tömböt, amelynek a $LF[i][j]$ cellája azt tárolja, hogy az i lány mennyire preferálja a j fiút. Az alábbi algoritmus használja még az $ind[1..n]$, $x[1..n]$ és $y[1..n]$ segédtömböket. Az $ind[i]$ elem azt tárolja, hogy hányadik próbálkozásnál tart az i fiú, az $x[i]$ azt, hogy melyik lány udvarában van az i fiú (ha $x[i]=0$, akkor az i fiú szabad), az $y[i]$ pedig azt, hogy melyik fiú van az i lány udvarában (ha $y[i]=0$, akkor az i lány szabad). Az 5.8. ábra a segédtömbök kezdeti értékeit tartalmazza.

```
végezd
   ok = IGAZ
   minden i = 1,n végezd
      ha x[i] == 0 és ind[i] ≤ n akkor          // i fiú szabad, és még van lány
         ok = HAMIS
         kovL = pF[i][ind[i]]                   // i fiú következő esélyese
```

```
        ha y[kovL] == 0 akkor                          // ha szabad a kiválasztott
          y[kovL] = i;                                 // potenciálisan az egymáséi
          x[i] = kovL;
        különben                                       // nem szabad a lány
          ha LF[kovL][i] < LF[kovL][y[kovL]] akkor     // i preferáltabb
            x[y[kovL]] = 0                             // az aktuális fiút hazaküldik
            ind[y[kovL]] = ind[y[kovL]] + 1
            y[kovL] = i                                // potenciálisan az i fiúé a lány
            x[i] = kovL
          különben                                     // nem preferáltabb
            ind[i] = ind[i] + 1                        // i majd a következő lánynál próbálkozik
          vége ha
        vége ha
      vége ha
    vége minden
amíg nem ok                                            // addig folytatjuk, amíg egyetlen fiú x-értéke se lesz 0
```



5.8. ábra. *A segédtömbök kezdőértékei (LF: például a 2. fiú másodikként preferálja az 1. lányt, harmadikként a 2. lányt és elsőként a 3. lányt; x: kezdetben minden fiú otthon van; y: kezdetben minden lány udvara üres; ind: mindenik fiú az első próbálkozásnál tart)*

6. BRANCH-AND-BOUND

Mielőtt a branch-and-bound (elágazás és korlátozás) módszerre rátérnénk, egy felülnézet erejéig tekintsünk vissza az eddig bemutatott három technikára.

6.1. Backtracking vagy oszd-meg-és-uralkodj

A 3. és 4. fejezetekben felfigyelhettünk arra, hogy mind a backtracking, mind az oszd-meg-és-uralkodj algoritmusok mélységükben járják be a feladatok szerkezetét ábrázoló fát. Ez a fő ok, amiért a rekurzív implementálás annyira kézenfekvő mindkét esetben. Mi akkor az alapvető különbség a két módszer között?

A backtracking algoritmusokban a megoldások felépítése a gyökértől a levelek irányába történik. Így az ígéretes részfa leveleiben hirdetnek megoldást, pontosabban a megoldáslevelekben. Ezért is rajzoltuk a fát a megszokottól eltérően, a gyökerével alul, hiszen az a természetes, hogy felfele építkezzünk. Úgy is mondhatnánk, hogy a mélységi bejárás előre szakaszain épít, a visszamutatókon pedig bont. Emlékezzünk most arra, miként jártuk körbe a fát a mélységi bejárás bemutatásakor az 1.6.1. alfejezetben. Többször is érintettük a csomópontokat, és attól függően, hogy az első vagy utolsó érintéskor látogattuk meg ezeket, beszéltünk preorder, illetve postorder mélységi bejárásról. Mivel a visszalépéses keresés a feladathoz rendelt fa csomópontjait első érintésükkor használja a megoldások építésében (ekkor kerülnek be a verembe, és majd utolsó érintésükkor lesznek onnan eltávolítva), ezért fogalmazhatunk úgy, hogy preorder mélységi bejárást alkalmaz.

Ezzel szemben az oszd-meg-és-uralkodj algoritmusok a gyökértől a levelek irányába (fentről lefele) lebontják a feladatot egyre egyszerűbb részfeladatokra, majd a visszaúton (lentről felfele) a részfeladatok megoldásaiból felépítik az eredeti feladat megoldását. Egy részfeladat csak azután kerül megoldásra, miután a fiúrészfeladatai már meg lettek oldva. Milyen sorrendben lesznek hát megoldva a fa csomópontjai által képviselt részfeladatok? Posztorder mélységi bejárás szerinti sorrendben.

Vegyük észre azt a különbséget is, hogy a backtracking a megoldáslevelekbe leérkezve (felérkezve), az oszd-meg-és-uralkodj pedig a gyökérbe visszaérkezve hirdet megoldást. Ez megmagyarázhatja azt, hogy a backtracking feladatoknak általában miért van több megoldásuk is, az oszd-meg-és-uralkodj típusúaknak viszont csak egy (egy fának sok levele van, de csak egyetlen gyökere).

Az előbbi észrevételnek van egy másik következménye is, főleg optimalizálási feladatok esetében (bár egyik technikát sem elsősorban ilyen feladatok meg-

oldására „találták ki”). Ilyenkor a feladat mögött egy döntési fa húzódik meg, és az optimális megoldást az optimális gyökér–levél út képviseli (lásd a 2.2. alfejezetet). Az optimális levélből nézve egyértelmű, melyik az optimális gyökér–levél út, viszont a gyökérből nézve egyáltalán nem. Ez a magyarázata annak, hogy optimalizálási feladatok esetén a backtracking alkalmas mind az optimális megoldás (optimális gyökér–levél út), mind az ehhez tartozó optimumérték kiíratására, az „oszd meg és uralkodj” módszernek viszont csak az utóbbi kézenfekvő. Persze felmerülhet bennünk az a gondolat, hogy nem lehetne-e minden csomópontban eltárolni ennek optimális fiát, hiszen ezen információ alapján később játszi könnyedséggel előállítható lenne az optimális gyökér–levél út is. Emlékezzünk viszont, hogy a fa exponenciális méretű. Ezért az oszd-meg-és-uralkodj stratégia – hogy elkerülje a fa megépítését – az egyes részfeladatok megoldásait csak addig tárolja el, amíg az apafeladat megoldását fel nem építi belőlük.

6.2. Backtracking és greedy

Ahhoz, hogy két dolog jól kiegészíthesse egymást, szükséges, hogy eléggé hasonlítsanak egymásra, de kellő mértékben különbözzenek is. Ebben a megközelítésben elmondható, hogy mind a backtracking (3. fejezet), mind a mohó stratégiák (5. fejezet) mélységükben viszonyulnak a feladatok szerkezetét ábrázoló fákhoz. Mindkét módszer gyökér–levél irányba építkezik: a backtracking megoldásutakat, a greedy pedig optimális levélhez vezető utat. Optimalizálási problémák esetében az alapvető különbség köztük az, hogy amíg a backtracking a teljes fa vagy ennek egy jelentős részfája, mélységi bejárása révén potenciális megoldások között válogatva keres, addig a mohó módszer egyetlen gyökér–levél úton szalad le. A 6.5. alfejezetben egy példát látunk majd arra, hogy e hasonlóságoknak és különbségeknek betudhatóan miként ötvözhető a két módszer. Egészen pontosan arról lesz szó, hogy miként növelhető a backtracking és mohó stratégiák eredményessége, branch-and-bound szellemben való kombinálásuk révén.

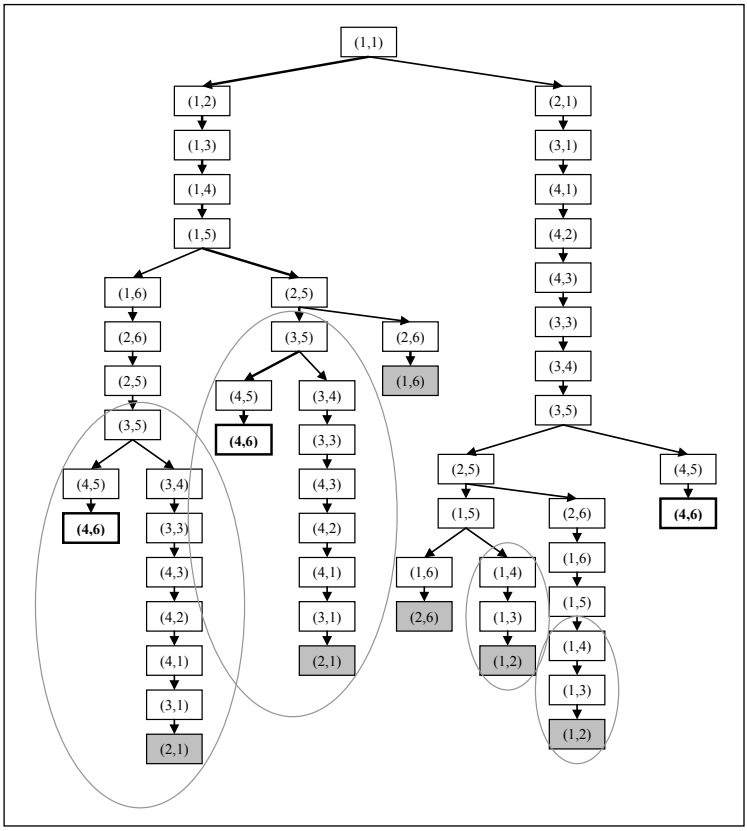
6.3. Egerek a labirintusban

Azért, hogy úgymond egymás mellett lássuk az eddig bemutatott három technikát, és felvezessük az e fejezet témáját képező branch-and-bound módszert, a következőkben egy olyan feladatot vizsgálunk meg, amely mind a négy stratégiával megközelíthető.

Egér–sajt probléma: Egy labirintusban, amelyet az $a[1..n][1..m]$ bináris mátrix ábrázol (a 0 érték utat jelöl, az 1-es falat), ismert egy egér (xe, ye) és egy darab sajt (xs, ys) pozíciója. Határozzuk meg az egértől a sajthoz vezető legrövidebb utat (a labirintusban négy irányban lehet közlekedni). Példa:

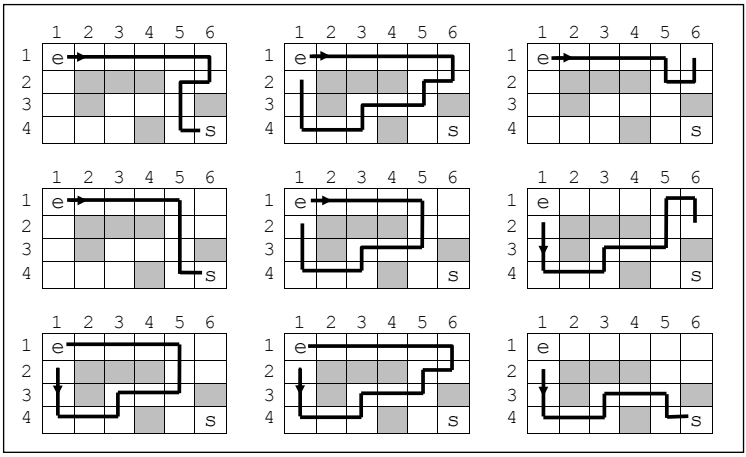
	1	2	3	4	5	6
1	e					
2		1	1	1		
3		1				1
4				1		s

6.1. ábra. *Példa 4×6 méretű labirintusra, ahol az egér az (1, 1), a sajt pedig a (4, 6) pozíciókban található*



6.2. ábra. *Az egér–sajt feladat hátteréül szolgáló fastruktúra (a példára vonatkoztatva)*

A 6.2. ábrán ábrázoltuk az (1, 1) pozícióból induló hurokmentes „egérutak” fáját, amely alapvetően egy döntési fa. A besatírozott levelekhez „zsákutcák” vezetnek. A megoldásleveleket (amelyeknek mindegyike a sajt pozícióját képviseli) megerősített kerettel rajzoltuk. A legmagasabban lévő, (4, 6) koordinátaértékű levélhez vezető gyökér–levél út ábrázolja az optimális egér–sajt utat. Bejelöltük az identikus részfákat is. A 6.3. ábra bemutatja a gyökér–levél utaknak megfelelő egér–sajt utakat (amennyiben bármely cellából fel/jobbra/le/balra sorrendben próbál továbbhaladni az egér).



6.3. ábra. *A példa-labirintus hurokmentes egérútjai: „zsákutcák” és sajthoz vezető utak*

Mi lehetne egy *mohó megközelítés* e feladatra? Egy mohó egér minden lépésben a legígéretesebb vagy a legígéretesebbnek tűnő irányt választaná. Más szóval, az ennek megfelelő gyökér–levél úton szaladna le a 6.2. fában. Mi alapján választhatna a szomszédos szabad cellák között? Például a szomszéd cellák és a cél cella (a sajt cellája) közti Manhattan távolságok (minimum hány vízszintes/függőleges lépésre van az illető pozíció a célpozíciótól) alapján. A megadott példára a 6.3. ábra szerinti első megoldást találja meg, amely nyilván nem az optimális. Más labirintusok esetén egy mohó egér akár zsákutcába is szaladhat. Mivel e feladat esetében nem vezet megoldáshoz a mohó stratégia, ezért nem is implementáljuk.

Ezzel szemben mind a backtracking, mind az oszd-meg-és-uralkodj módszer alkalmas e feladat megoldására, bár egyik sem vezet hatékony algoritmushoz.

– *Backtracking stratégia:* Generálja az összes hurokmentes utat, amely az egértől a sajthoz vezet, és kiválasztja közülük a legrövidebbet.

– *Oszd-meg-és-uralkodj stratégia:* Az egértől a sajthoz vezető legrövidebb út *hosszának* meghatározása visszavezethető az egérrel szomszédos szabad pozíciókból induló – sajthoz vezető – legrövidebb utak hosszának megtalálására. Miután ezek rendelkezésre állnak (jelentés érkezik felőlük), a legrövidebbhez hozzáadva egyet, meg is van a keresett út hossza.

Figyeljük meg, hogy a backtracking több potenciális *megoldásutat* is generál (az összes egér–sajt utat), az oszd-meg-és-uralkodj viszont csak az optimális út *hosszát* építi fel.

Megjegyzések a backtracking_egér eljáráshoz:
– út[]: a kurrens utat tárolja;
– útmin[]: a legrövidebb utat tárolja;
– kmin: a legrövidebb út hosszát tárolja;
– x, y: a kurrens pozíció;
– k: hányadik állomás a kurrens pozíció az aktuális úton;
– ALGORITMUS:
  – regisztráljuk az (x,y) pozíciót mint a kurrens út k-adik állomását;
  – ellenőrizzük, hogy nem állunk-e éppen a sajton;
    • ha igen, akkor, amennyiben jobb megoldást találtunk, mint az eddigi legjobb, regisztráljuk a kmin és az útmin változókban;
  – ha nem vagyunk még a sajtnál, akkor
    • mielőtt továbblépnénk, befalazzuk lábunk alatt a labirintust (ezzel elkerüljük, hogy ezen az ágon újra visszajussunk ugyanebbe a pozícióba, és a kialakult hurokban az egeret a végtelenségig körbejárassuk; olyan ez, mintha nyomot hagynánk);
    • a szomszédos pozíciókra vonatkozó rekurzív hívások által sorra elmegyünk minden szabad irányba;
    • mielőtt visszalépnénk az (x, y) pozícióból, kifalazzuk lábunk alatt a labirintust (ezzel biztosítjuk, hogy más ágakon továbbra is elérhető legyen e pozíció; úgy is fogalmazhatnánk, hogy nyomtalanul lépünk vissza);
– figyeljük meg, hogy akkor érkeztünk „zsákutcába”, ha az illető pozícióból egyetlen irányba sem tudunk továbblépni;
– az eljárás meghívása: backtracking_egér(a,ut,utmin,kmin,xs,ys,xe,ye,0).

```
backtracking_egér(a[][],ut[],utmin[],kmin,xs,ys,x,y,k)
  út[k].x = x
  út[k].y = y
  ha x == xs és y == ys akkor                    // a sajthoz érkeztünk
    ha k < kmin akkor                            // rövidebb utat találtunk
      útmin[1..k] = út[1..k]
      kmin = k
    vége ha
```

```
  különben
    a[x][y] = 1                                    // hurkok elkerülése végett
    ha a[x−1][y] == 0 akkor
      backtracking_egér(a,ut,utmin,kmin,xs,ys,x−1,y,k+1)
    vége ha
    ha a[x][y+1] == 0 akkor
      backtracking_egér(a,ut,utmin,kmin,xs,ys,x,y+1,k+1)
    vége ha
    ha a[x+1][y] == 0 akkor
      backtracking_egér(a,ut,utmin,kmin,xs,ys,x+1,y,k+1)
    vége ha
    ha a[x][y−1] == 0 akkor
      backtracking_egér(a,ut,utmin,kmin,xs,ys,x,y−1,k+1)
    vége ha
    a[x][y] = 0                                    // visszaállítjuk a szabad utat
  vége ha
vége backtracking_egér
```

Megjegyzések az oszd_meg_és_uralkodj_egér függvényhez:
– a függvény a legrövidebb út *hosszát* téríti vissza;
– a ∞ hossz jelentése: nincs út;
– x, y: a kurrens pozíció;
– a h1, h2, h3, h4 lokális változók a kurrens feladat fiúrészfeladatainak optimumértékeit tárolják, időszakosan;
– a minimum függvény visszatéríti a paraméterértékek minimumát;
– a függvény meghívása: oszd_meg_és_uralkodj_egér(a,xs,ys,xe,ye).

```
oszd_meg_és_uralkodj_egér(a[][],xs,ys,x,y)
  ha x == xs és y == ys akkor                      // a sajthoz érkeztünk
    return 0
  vége ha
  h1 = ∞
  h2 = ∞
  h3 = ∞
  h4 = ∞
  a[x][y] = 1                                      // hurkok elkerülése végett
  ha a[x−1][y] == 0 akkor
    h1 = oszd_meg_és_uralkodj_egér(a,xs,ys,x−1,y)
  vége ha
  ha a[x][y+1] == 0 akkor
    h2 = oszd_meg_és_uralkodj_egér(a,xs,ys,x,y+1)
  vége ha
```

```
    ha a[x+1][y] == 0 akkor
      h3 = oszd_meg_és_uralkodj_egér(a,xs,ys,x+1,y)
    vége ha
    ha a[x][y−1] == 0 akkor
      h4 = oszd_meg_és_uralkodj_egér(a,xs,ys,x,y−1)
    vége ha
    a[x][y] = 0                                  // visszaállítjuk a szabad utat
    return minimum(h1,h2,h3,h4) +1
vége oszd_meg_és_uralkodj_egér
```

Mindkét rekurzív algoritmusban az eljárás/függvényhívások úgy tekinthetők, mintha újabb egérklón születne a megfelelő cellába (visszalépéskor felszámolódik az illető klón). A kurrens backtracking-egér a szomszédos szabad cellákba született klónjai révén halad tovább a labirintusban. A befalazás ebből a megközelítésből azt jelenti, hogy „egértől foglalt". A kurrens oszd-meg-és-uralkodj-egér a jelentését (visszatérített értékét) a szomszédos szabad cellákba született klónjaitól kapott jelentésekből állítja össze. Kivételesen (triviális részfeladatok) a zsákutcák végeibe született egerek végtelent jelentenek, a sajt cellájába születettek pedig nullát, tele szájjal. A mélységi bejárás alatt a backtracking-egerek azt tartják nyilván (preorder momentumokban), hogy milyen távolra jutottak a kezdeti pozíciótól, az oszd-meg-és-uralkodj típusúak pedig arról tesznek jelentést (postorder momentumokban), hogy milyen távol van tőlük a sajt. Egy (i, j) cellába annyiszor születik egér, ahányszor szerepel mint csomópont a 6.2. fában.

Vegyük észre, hogy adott (i, j) gyökerű részfák nem föltétlenül azonosak. Ez azzal magyarázható, hogy az (i, j) pozícióhoz vezető kurrens úton az „egértől foglalt" cellák időszakosan átkonfigurálják a labirintust. Ebből adódóan nem mindenik oszd-meg-és-uralkodj-egérnek, amelyik adott (i, j) cellába született, lesz ugyanaz a jelentése. Persze ha minimumot számolnánk minden (i, j) cellában az ideszületett klónok jelentései között (például egy $b[1..n][1..n]$ segédtömbben), akkor ezzel az oszd-meg-és-uralkodj algoritmus meghatározná minden cellától a sajthoz vezető legrövidebb út hosszát. E b tömbből könnyűszerrel kiolvasható lenne, mohó módon, a legjobb egér–sajt útvonal is. Maradjon ennek implementálása az olvasóra (lásd a 6.4. ábrát).

	1	2	3	4	5	6
1	**8**	**7**,∞,∞	**6**,∞,∞	**5**,∞,∞	**4**,∞,∞	**5**,∞,∞,∞
2	∞,∞,**9**				**3**,3,∞	**4**,∞,∞,∞
3	∞,∞,**8**		∞,∞,**4**	∞,∞,**3**	**2**,2,2	
4	∞,∞,**7**	∞,∞,**6**	∞,∞,**5**		**1**,1,1	**0**,0,0

6.4. ábra. *Minden cellában annyi szám szerepel, ahány egérklón születik oda az algoritmusok alatt. Az értékek az oszd-meg-és-uralkodj-klónok visszatérített értékeit jelentik. A megvastagított számok, a minimumok lesznek*

*a b tömb végső elemei*A két algoritmus időbonyolultsági ekvivalenciája abban tükröződik, hogy mindkettő nyomán ugyanannyi egér születik (a fa csomópontjaival, illetve a rekurzív hívásokkal megegyező számú), sőt ugyanabban a sorrendben. Az alábbiakban látni fogjuk, hogy miként lehet javítani e megoldásokon branch-and-bound szellemben.

6.4. A branch-and-bound módszer stratégiája

Az egér–sajt feladathoz a következő reprezentációs gráf[7] társítható (lásd az első fejezet *A technikák mint útkereső stratégiák* című betétjét): a labirintus szabad cellái a csomópontok, két csomópont között pedig akkor van oda-vissza él, ha a megfelelő cellák szomszédosak (minden él hossza 1). Érdekeltek vagyunk a startcsúcs (egér pozíciója) és a célcsúcs (sajt pozíciója) közötti legrövidebb útban.

A gráfkereső algoritmusok a startcsúcsból kiindulva fokozatosan tárják fel a reprezentációs gráfot úgy, hogy startcsúcsból kiinduló utakat derítenek fel. A keresés nyomán a gráf (például: 6.1. ábra) egy irányított fává egyenesedik ki (keresési tér; például: 6.2. ábra), amelyben a startcsúcs lesz a gyökér, az ágak pedig a startcsúcsból kiinduló utakat ábrázolják (a gráf egy csomópontja annyiszor jelenik meg a fában, ahány úton érhető el a startcsúcsból).

Egy általánosabb megfogalmazás a következő lehet: Határozzuk meg a feladat keresési terét ábrázoló fában az optimális startcsúcs–célcsúcs utat. Lévén szó exponenciális ütemben terebélyesedő fáról, a szóban forgó utat szeretnénk úgy megtalálni, hogy a fának, ha lehet, csak egy töredékét kelljen effektíve generáljuk/bejárjuk. Ebben segíthet a branch-and-bound stratégia.

A branch-and-bound módszert Land és Doig (1960) szerzők javasolták *optimalizálási feladatok* megoldására 1960-ban. A „Branch and bound” elnevezést Little és társai (1963) használták először, miközben a módszert a híres utazóügynök-problémára alkalmazták. Az alábbiakban részletezzük, hogy mit értünk a következőkben a „branching” és „bounding” műveletek alatt (megjegyezzük, hogy nincs egységesen elfogadott definíció a branch-and-bound módszerre):

– „Branching”: A keresés a teljes fában kezdődik: generáljuk a gyökerét. Ha nem éppen a gyökér képviseli az optimális megoldást, akkor a megoldáslevélnek a gyökér valamelyik fiúrészfájában kell lennie: generáljuk ezek gyökereit (azaz a gyökér fiúcsomópontjait). Úgy is fogalmazhatnánk, hogy a generált fa koronája peremén kezdetben a gyökércsomópont van, majd ennek fiúcsomópontjai, és így tovább. A módszer nevében a „branch” szó arra utal, hogy minden lépésben a generált fa *koronája peremén* lévő *valamelyik* csomópont elágazik (branching) fiúcsomópontjaira. Az illető apa-

7 Az *a* mátrix úgy tekinthető, mint a gráf implicit ábrázolása.

csomópont lekerül a korona pereméről, a fiai pedig felkerülnek (e töröl/beszúr-szerű műveletek implementálása feltételezi az aktuálisan peremen lévő csomópontok explicit/implicit nyilvántartását). Szokás a korona peremén levő csomópontokat nyílt, a generált fa többi csomópontját (a peremről lekerülteket) pedig zárt pontnak nevezni. Az elágazás úgy is felfogható, mintha a keresés redukálódna a szóban forgó csomópont fiúrészfáira (amennyiben az illető apacsomópont nem bizonyult megoldáslevélnek).

– „Bounding": Az előző pont önmagában még nem sok újat hoz. A mélységi és szélességi keresések (lásd az 1.6. alfejezetet) is tekinthetők úgy, hogy minden lépésben a már bejárt részfa koronája peremről egyik csomópont elágazik fiúcsomópontjaira. A módszer erőssége a „bound" fázisban van. Ez annyit jelent, hogy ha egy fiúcsomópontról eldönthető, hogy a hozzá tartozó fiúrészfa garantáltan nem tartalmaz megoldáslevelet, akkor lemetsszük mint száraz ágat (nem kerül fel a fa koronájára; elmarad az illető részfa generálása/bejárása).

Azt mondtuk, hogy a fa koronája peremén lévő nyílt csomópontok *valamelyikét* cseréljük le a fiúcsomópontjaira. Ezt a választást megtehetjük BF (mindig a gyökérhez legközelebbit választjuk), DF (mindig a gyökértől legtávolabbit választjuk) vagy „best-first" szellemben.

6.4.1. DF- és BF-branch-and-bound stratégiák

Nevezzük DF-branch-and-bound stratégiának azt, amikor a feltárt utak közül mindig a leghosszabbal próbálunk továbblépni. A BF-branch-and-bound stratégia esetében pedig éppen fordítva: minden lépésben a legrövidebb utat részesítjük előnyben. E sorrendeket úgy biztosítjuk, hogy az elsőnél vermet (LIFO adatszerkezet), a másiknál pedig sort (FIFO adatszerkezet) használunk a nyílt csúcsok tárolásához. Hogyan építhető be e stratégiákba a „bounding" művelet? Például úgy, hogy amennyiben valamely úton újra megjelenne egy korábban már feltárt csomópont, az illető irányba csak akkor folytatjuk a keresést, ha előnyösebb, mint az előző. Más szóval, az illető fiúcsomópont csak akkor kerül be a nyílt csúcsok halmazába, ha ígéretesebben bővíti a fát, mint a korábbi példány.

Mit jelenthet ez az egér–sajt feladat esetében? Mivel a legrövidebb útban vagyunk érdekeltek, és a legrövidebb út részútjai is legrövidebb utak, ezért nem lépünk tovább olyan cella irányába, ahova korábban találtunk már rövidebb utat. Ez persze azt feltételezi, hogy minden cellára tartsunk nyilván egy „kurrens optimumot" (6.5. és 6.6. ábrák).

A DF-branch-and-bound stratégia esetében ezt az optimalizálást úgy implementálhatjuk, hogy amikor a verem tetején lévő cellát lecseréljük a szomszédjaira, akkor ezt nem tesszük meg olyan szomszédokra, amelyekre a kurrens út hossza hosszabb, mint egy korábban beállított kurrens optimum. Ez többet

jelent a körök elkerülésénél, amit a fentebb tárgyalt backtracking_egér eljárásba is beépítettünk (kör alatt az aktuális útra való visszakanyarodást értjük). A DF-branch-and-bound algoritmus akkor sem lép tovább olyan cella irányába, ahova korábban már talált rövidebb utat, amennyiben ez a keresés egy másik ágán történt (lásd a *Visszalépéses keresés versus mélységi gráfkeresés* betétet). A 6.5. ábrán a tömbelemek azt tárolják, hogy miként finomodnak a cellákhoz társított kurrens optimumok, a DF-branch-and-bound stratégia nyomán, végső optimumokká. Például a (2,5) cella az (1,1), (1,2), (1,3), (1,4), (1,5), (1,6), (2,6), (2,5) ágon kerül be először a nyílt csúcsok halmazába, és a megfelelő optimumérték 7 lesz. Amikor az algoritmus az (1,1), (1,2), (1,3), (1,4), (1,5), (2,5) ágon újra a (2,5) cellához érkezik, frissül az eltárol optimum (7-ről 5-re), és a (2,5) cella újra bekerül a verembe. Ezzel ellentétben a (2,6) cella, a (1,1), (1,2), (1,3), (1,4), (1,5), (2,5) ág folytatásaként nem kerül újra be a nyílt halmazba. Amennyivel kevesebb értéket tartalmaznak a 6.5. ábra cellái a 6.2. ábra csomópontjainak számánál, annyival eredményez hatékonyabb algoritmust e feladatra, e branch-and-bound megközelítés, mint a backtracking.

	1	2	3	4	5	6
1	**0**	**1**	**2**	**3**	**4**	**5**,
2	15,13,**1**				7,**5**	**6**,
3	14,12,**2**		10,8,**6**	9,7,**7**	8,**6**	
4	13,11,**3**	12,10,**4**	11,9,**5**		9,**7**	10,**8**

6.5. ábra. *Optimumok tömbje a DF-branch-and-bound stratégia nyomán (a cellák azt tárolják, hogy miként finomodott a megfelelő kurrens optimum)*

Mellesleg már az is egy branch-and-bound javítás lenne a backtracking algoritmuson, ha egyszerűen csak azt figyelnénk, hogy a kurrens út hossza nem hosszabb-e, mint az addig talált legrövidebb útnak a hossza. Ez a példafeladat esetében abban tükröződne, hogy miután megtaláltuk az első, 10 hosszú egér–sajt utat, többet ennél mélyebbre nem megyünk a keresési fában. Miután frissült a minimum 8-ra, azután már ennél mélyebbre nem megyünk. Ezen ötlet implementálása csak annyit jelent, hogy a backtracking_egér eljárás elejére beiktatjuk a következő feltételt:

```
ha k ≥ kmin akkor
   return
vége ha
```

E feladatra a pálmát a BF-branch-and-bound stratégia viszi el, amely minden cellával csak egyszer foglalkozik, és a célcsúcsot a legrövidebb úton éri el először (lásd a 6.6. ábrát). Sőt, *minden* cellát a legrövidebb úton ér el először. Ez annak a

következménye, hogy az algoritmus minden lépésben, a BF bejárás értelmében, a gyökérhez legközelebbi nyílt csúcsból lép tovább. Minden cella (amely elérhető az egérnek) számára csak egyszer generálódik csomópont a keresési fában. Ha a kurrens nyílt csomópont (a sorelső) valamely szomszédjaként újra jelentkezne egy korábban már elért cella, akkor ez garantáltan lemetszhető a fáról. E stratégia szellemében, amint látni fogjuk a 7. fejezetben, dinamikus programozásként is felfogható, hiszen optimumokból épít optimumokat.

	1	2	3	4	5	6
1	**0**	**1**	**2**	**3**	**4**	**5**
2	**1**				**5**	**6**
3	**2**		**6**	**7**	**6**	
4	**3**	**4**	**5**		**7**	**8**

6.6. ábra. *Optimumok tömbje a BF-branch-and-bound stratégia nyomán (minden lépésben a legkisebb értékű cella szabad szomszédjai eggyel nagyobb értéket kapnak)*

Az alábbi kódrészlet értelmében, a $b[1..n][1..m]$ tömbben, a labirintus szabad celláit kezdetben 0 értékek, a befalazott pozíciókat pedig -1 értékek képviselik. A BF algoritmusok igényelte sort a q egydimenziós tömb tárolja. Az eleje és vége változók a sor elejét és végét jelzik. A sor bejegyzéseket tartalmaz, amelyek mezői a megfelelő cella koordinátáit (x és y) tárolják. Az algoritmus csak a legrövidebb út hosszát írja ki, de a feltöltött b tömbből kiolvasható (vissza irányban) maga a legrövidebb út is.

```
eleje = 0                          //kezdetben csak az egér pozíciója van a q sorban
vége = 0
q[0].x = xe
q[0].y = ye

amíg (eleje ≤ vége) és             //a sor nem üres és a sorelső nem a sajt pozíciója
     nem(q[eleje].x == xs és q[eleje].y == ys) végezd
  x = q[eleje].x
  y = q[eleje].y
  ha x > 1 és b[x−1][y] == 0 akkor                    //felfele üres cella van
    b[x−1][y] = b[x][y]+1                  //1-gyel távolabbra van a sorelsőnél
    vége = vége+1                                    //betesszük a sor végére
    q[vége].x = x−1
    q[vége].y = y
  vége ha
  ha y < m és b[x][y+1] == 0 akkor                    //jobbra üres cella van
```

```
        b[x][y+1] = b[x][y]+1                    //1-gyel távolabbra van a sorelsőnél
        vége = vége+1                            //betesszük a sor végére
        q[vége].x = x
        q[vége].y = y+1
    vége ha
    ha x < n és b[x+1][y] == 0 akkor             //lefele üres cella van
        b[x+1][y] = b[x][y]+1                    //1-gyel távolabbra van a sorelsőnél
        vége = vége+1                            //betesszük a sor végére
        q[vége].x = x+1
        q[vége].y = y
    vége ha
    ha y > 1 és b[x][y−1] == 0 akkor             //balra üres cella van
        b[x][y−1] = b[x][y]+1                    //1-gyel távolabbra van a sorelsőnél
        vége = vége+1                            //betesszük a sor végére
        q[vége].x = x
        q[vége].y = y−1
    vége ha
    eleje = eleje+1
vége amíg

ha q[eleje].x == xs és q[eleje].y == ys akkor
    ki: b[q[eleje].x][q[eleje].y]                //a legrövidebb út hossza
különben
    ki: "Nem elérhető a sajt pozíciója"
vége ha
```

Visszalépéses keresés versus mélységi gráfkeresés

Amit mi DF-branch-and-bound és BF-branch-and-bound stratégiáknak neveztünk, azt Gregorics (2014) mélységi és szélességi *gráfkeresésnek* nevezi.

Vegyük észre, hogy szigorú értelemben a mélyégi gráfkeresés nem ugyanaz, mint a visszalépéses keresés (lásd a 3. fejezetet), bár nyilván sok a hasonlóság van köztük. Például mindkettő mélységi bejárásra épül, és irányított fákon ugyanúgy működnek. Egy lényeges különbség azonban az, hogy a gráfkereső algoritmusok a reprezentációs gráfnak a már felfedezett részét teljes egészében eltárolják, a visszalépéses keresés pedig csak egyetlen, a startcsúcsból kivezető aktuális utat tart nyilván. Ebből kifolyólag a visszalépéses keresés, ahhoz, hogy a kurrens ágról átjusson egy másik ág valamely csomópontjához, vissza kell lépjen a két ág legközelebbi közös pontjáig. Ezzel szemben a mélységi gráfkeresés akár egy lépésből is tud váltani valamely korábban feltárt nyílt vagy zárt csúcshoz.

Fentebb a backtracking_egér eljárást úgy mutattuk be, mint amely visszalépéses keresést alkalmaz (egyes szerzők sík-backtrackingnek nevezik). A valóságban viszont közelebb áll a mélységi gráfkereséshez. Minimális változtatással átírható oly módon, hogy a 6.5. ábra szerinti optimalizált algoritmust valósítsa meg (az *a* tömböt a 6.5. ábra szellemében használjuk). A 6.5. ábrán bemutatott tömbre úgy is tekinthetünk, mint a reprezentációs gráf felfedezett részének implicit ábrázolására. Azok az elemek képviselnek feltárt csomópontokat, amelyek már kaptak értéket.

Egy „tiszta visszalépéses keresési algoritmus" erre a feladatra az lenne, ha a 3. fejezetben bemutatott módszer *x* tömbjében tárolnánk a kurrens utat mint koordinátapársorozatot. Ez az ábrázolás is lehetővé tenné a körök elkerülését az ígéretes függvény révén. Másfelől az *x* tömb „elfelejti", hogy mi történt azokon az ágakon, ahonnan az algoritmus már visszalépett (nem így viszont egy olyan kétdimenziós tömb, amelyet a 6.5. ábra szerint használunk).

6.4.2. Best-first-branch-and-bound stratégia

A branch-and-bound módszer e változatát egy kirakójáték segítségével szemléltetjük, amely közel 150 éves múltra tekint vissza, és alapul szolgált az 1990-es Nemzetközi Informatika Olimpián kitűzött egyik feladathoz.

Tili-toli: Adott egy 4×4 méretű mátrix, amely 0-tól 15-ig tartalmazza a természetes számokat (lásd lennebb). A nulla pozíciója üres helynek számít. Egy lépés alatt azt értjük, hogy valamelyik nullával szomszédos (fel, le, jobbra, balra irányban) számot az üres helyre húzzuk (a valóságban az illető szám helyet cserél a nullával). Keverjük össze a mátrix elemeit, véletlenszerű lépéseket téve.

Adva lévén egy összekevert állapot, állítsuk vissza a kezdeti állapotot *minimális* számú lépésből.

Példa:

Kezdeti állapot				Összekevert állapot			
1	2	3	4	8	2	5	11
5	6	7	8	1	6	9	10
9	10	11	12	3	15	0	7
13	14	15	0	12	14	13	4

Megjegyzések:

– A feladat állapotait a 16 szám konfigurációja határozza meg, nevezetesen az, hogy éppen hol találhatók a (4x4)-es mátrixban.

– Egy lépést úgy is felfoghatunk, mintha a nulla lépne, helyet cserélve az elmozdított számmal.
– A feladathoz rendelhető döntési fában a gyökér a kezdeti állapotot (az összekevert konfigurációt) ábrázolja, az első szinti csomópontok azokat a konfigurációkat, amelyekhez egy lépésből juthatunk, a második szintiek, amelyek két lépésből adódhatnak, és így tovább (lásd a Függeléket, amely megjeleníti a 8-as kirakójáték keresési terének egy részletét).
– A sarkokból két irányba, az oldallapokról három irányba, egyébként négy irányba „léphet a nulla". Ezzel összhangban, és figyelembe véve, hogy nem lépünk vissza oda, ahonnan ideléptünk, a fa csomópontjai fiainak száma 1, 2 vagy 3 lesz (kivéve a gyökeret, amelynek 2, 3 vagy 4 fiúkonfigurációja lehet, hiszen esetében nem volt előző lépés).
– Ugyanaz a konfiguráció nyilván többször is megjelenhet a fában. Bár minden kirakott konfigurációt képviselő csomópont *megoldáslevélnek* tekinthető, mi az *optimális megoldáslevélben* vagyunk érdekeltek (a hozzá vezető gyökér–levél úttal együtt), amely a feladat *optimálisan megoldott* állapotát képviseli. A *minimális* lépésszámú megoldást, természetesen, a fa *legmagasabban* elhelyezkedő megoldáslevele ábrázolja.

Mivel e feladat kapcsán is a legmagasabban elhelyezkedő megoldáslevélben vagyunk érdekeltek, ezért ez esetben is egy BFS alapú megközelítés (szintről szintre halad a fában) tűnik kézenfekvőnek. Ennél jóval hatékonyabb algoritmushoz jutunk azonban, ha „best-first" szellemben mindig azt a csomópontot választjuk, amelyik részfája a legnagyobb eséllyel tartalmazza az optimális levelet.

Társítsunk a keresési fa csomópontjaihoz két értéket, egy g és egy h értéket, és legyen $f = g+h$. A g érték jelölje a csomópont szintjét (a gyökértől hozzá vezető út hossza), a h érték pedig legyen a csomópont–megoldáslevél távolság egy *becsült* értéke (heurisztikus[8] függvény). A h érték lehet, például, (1) azon nullától különböző számok száma, amelyek nincsenek a célállapot szerinti végső helyükön (Hamming-távolság), vagy (2) a számok kurrens pozíciója és a célpozíciója közti Manhattan-távolságok (minimum hány vízszintes/függőleges lépésre van az illető szám a célpozíciójától) összege. (A kezdeti állapotot képviselő gyökérpont g értéke nyilván 0, az első típusú h értéke 12, a második típusú pedig: $1+0+4+3+3+0+2+4+3+3+3+4+2+0+2 = 34$; célállapotokra mindkét h érték 0).

Az alábbi algoritmus best-first-branch-and-bound megközelítésnek számít a Tili-toli feladathoz: (1) minden lépés „branching" szakaszában a legkisebb f értékű nyílt csomópontot válasszuk; (2) amennyiben e csomópont elágaztatásából adódó

8 A feladattal kapcsolatos olyan információ, amit nem a reprezentációban rögzítünk, hanem közvetlenül az algoritmusba építünk be. A heurisztika a vezérlési stratégiának a feladatra történő ráhangolását végzi (Gregorics 2014).

valamely fiúcsomópont már szerepel a korábban felderített zárt vagy nyílt csomópontok halmazában, akkor lemondunk róla (nem kerül be a nyílt csomópontok halmazába; „bounding" szakasz). E stratégia úgy is felfogható, mintha céltudatosabbá tennénk a BF-branch-and-bound algoritmust. Szemléletesen: ahelyett, hogy minden levél irányába ugyanazzal a sebességgel haladnánk, az ígéretesebbnek tűnő irányokat előnybe helyezzük. De a per pillanat kevésbé ígéretes irányokról sem mondunk le. Ott vannak a sor végén, arra az eshetőségre, ha netalántán melléfogtunk (ne feledjük, hogy az ígéretesség meghatározása csak megbecsülésen alapult).

Mivel a h függvény egy *nemnegatív* értékkel, *alulról* becsüli („optimista": közelebb látja a célt) a célhoz vezető hátralevő optimális út költségét (h mindkét változatára igaz ez), ezért bizonyítható, hogy ez az algoritmus[9] garantáltan az optimális megoldást találja meg elsőnek (Gregorics 2014). Érdekes megfigyelni, hogy az egér–sajt feladatra adott BF-branch-and-bound és DF-branch-and-bound algoritmusok sajátos esetei a tili-toli feladatra adott best-first-branch-and-bound stratégiának. A megfelelő f értékek a következők: $f = g+0$, illetve $f = -g+0$. A 6.3.1. alfejezetben bemutatott mohó egér esetében pedig a megfelelő f érték $f = 0+h$ alakú.

Az algoritmus implementálási részletei végett lásd az *Algoritmusok felülnézetből* című jegyzet 10.2. alfejezetét.

6.5. Backtracking–greedy házasítás branch-and-bound módra

Egy másik módszer a „bound" fázis implementálására, hogy minden csomópontra meghatározunk egy korlátértéket (egy korlátfüggvényt definiálunk a csomópontok halmazán), hogy *legfeljebb* „mennyire jó megoldásleveleket" tartalmazhat az illető csomópont gyökerű részfa. Ha egy csomópontra e korlátérték sem jobb, mint az addig talált legjobb *megoldás* értéke, akkor a szóban forgó részfa garantáltan nem tartalmazza az optimális levelet (száraz ág). Persze mindez azt feltételezi, hogy a generálás/bejárás alatt nyilvántartsunk egy úgynevezett kurrens optimumot, az adott pontig talált legjobb *megoldáslevél* értékét.

Az alábbiakban a híres hátizsákproblémán szemléltetjük, hogy miként kombinálhatók a visszalépéses keresés és mohó stratégiák e szellemben.

Hátizsákprobléma: Egy üzletben n tárgy (áru) található, amelyeknek ismert az áruk és a súlyuk (tömegük). Az árakat a t bejegyzés típusú tömb elemei a mezőiben,

9 A tili-toli feladatra adott megoldásunkat a szakirodalom A* algoritmusnak nevezi: az f függvény $g+h$ alakú, és h-ra igaz, hogy egy *nem-negatív* értékkel, *alulról* becsüli a célhoz vezető hátralevő optimális út költségét. Több szerző is sajátos branch-and-bound stratégiának tekinti az A* algoritmust (Nau et al. 1984).

a súlyokat pedig az elemek g mezőiben tároljuk, ahol $t[i].g$ ($i = 1,n$) természetes számok. Állapítsuk meg, hogy mely tárgyakat fogja magával vinni egy tolvaj ahhoz, hogy a lehető legnagyobb nyereséggel távozzon (a hátizsákja legtöbb G súlyt bír meg).

A feladat szövege két változatban is ismert:

a) a tárgyak elvághatók (folytonos változat),

b) a tárgyak nem vághatók el (diszkrét változat).

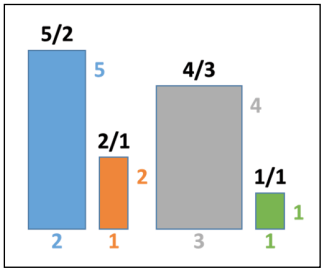
Példa:

Bemenet (6.7. ábra): $n = 4$, $G = 5$, $t[1..4].g = \{2, 1, 3, 1\}$, $t[1..4].a = \{5, 2, 4, 1\}$

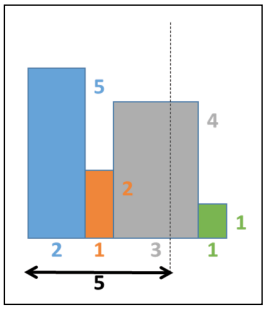
Kimenet:

a) (1, 1, 2/3, 0) – jelentése: az első és második tárgy egészében, a harmadiknak pedig 2/3 része kerül a hátizsákba (a negyedik áru az üzletben marad). Ez 29/3 = 9,66 egység nyereséget jelent (6.8. ábra).

b) (1, 0, 1, 0) – jelentése: az első és harmadik tárgy kerül a hátizsákba (a második és negyedik áru az üzletben marad). Ez 9 egység nyereséget jelent (6.9. ábra).



6.7. ábra. *A tárgyak vízszintes irányú mérete a súlyukkal arányos, a függőleges irányú pedig az árukkal; mindenik tárgy fölé az egységnyi értékét írtuk*

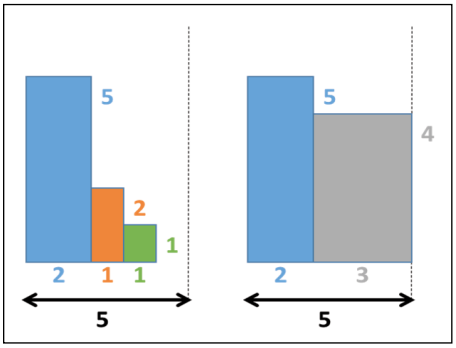


6.8. ábra. *A mohó megoldást szemlélteti a példafeladat folytonos változatára*

Megoldás: A feladat a) változata megoldható mohó stratégiával (6.8. ábra). A tárgyakat érték (ár/súly) szerint csökkenő sorrendben próbáljuk betenni a hátizsákba (a példabemenet éppen ebben a sorrendben tartalmazza a tárgyakat: 5/2 >

$2/1 > 4/3 > 1/1$; ha nem így lenne, akkor a tárgyak megfelelő rendezésével tudjuk biztosítani a mohó sorrendet). Az első áruból, amelyik már nem fér egészében a hátizsákba, levágunk annyit, hogy azzal teljesen megteljen. Bizonyítható, hogy ez a stratégia mindig az optimális megoldáshoz vezet.

Ha a feladat b) változatát próbáljuk mohó algoritmussal megoldani, a fenti megközelítés nem mindig vezet optimális megoldáshoz (6.9. ábra). A fenti példa esetében is az (1, 1, 0, 1) kódú megoldást találnánk, holott az optimálisnak a kódja, amint láttuk, az (1, 0, 1, 0).



6.9. ábra. *(a) A mohó megoldást szemlélteti a példafeladat diszkrét változatára; (b) A példafeladat diszkrét változatának optimális megoldását szemlélteti*

Hogyan közelítené meg ezt a feladatot a backtracking stratégia? Mivel mind az n tárgy esetén két lehetőség közül választhatunk: vagy beletesszük a tárgyat a hátizsákba, vagy nem, a feladat keresési tere egy $n+1$ szintes bináris fa lesz. Ezt a fát mutatja be a 6.10. ábra, a példánkra felrajzolva. A fa gyökér–levél útjai a tárgyak halmazának részhalmazait ábrázolják. A már megszokott szóhasználat szerint nevezzük optimális gyökér–levél útnak azt, amelyik a feladat optimális megoldását képviseli, és optimális levélnek azt, amelyikhez ez az út vezet. A nyers erő módszere az lenne, hogy generáljuk a tárgyak halmazának összes részhalmazát, kiválasztva közülük először azokat, amelyek belefférnek a hátizsákba, majd pedig azt, amelyik a legtöbb nyereséggel jár (maximumkeresést végzünk a potenciális megoldások között). Mivel az n elemű halmaz részhalmazainak száma 2^n (mindenik részhalmazhoz rendelhető egy n elemű bináris kód), ez a megoldás 2^n bonyolultságú algoritmust jelent. Mindez megvalósítható a teljes fa mélységi bejárásával, ami felfogható egy olyan backtracking algoritmusként, amely csak akkor lép vissza, ha már nincs további bepakolható tárgy. Hogyan lehetne javítani ezen az algoritmuson? Csak olyan részhalmazokat generálunk, amelyek belefférnek a hátizsákba, azaz nem folytatjuk az építkezést olyan *bal* irányokba, amelyek túlterhelt hátizsákot eredményeznének. A „backtracking ollót”, amely ebből a szempontból metszi meg a fát, vastagított szimpla vonalkával jelöltük.

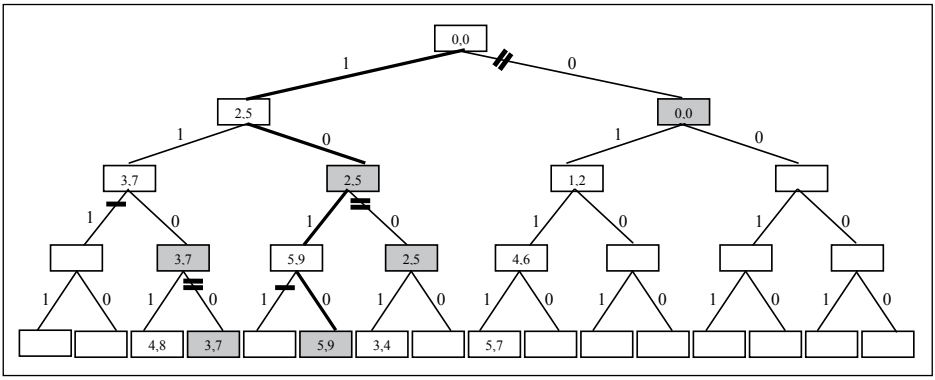
Mindezek után is úgy találhatjuk, hogy míg a mohó stratégia nem volt kielégítő, a backtracking túl időigényes. Egy lehetséges (és jobb) megoldáshoz vezet a két módszer kombinálása.

Hogyan lehetne még hatékonyabban optimalizálni a fenti backtracking algoritmust a mohó stratégia segítségével? Foglalkozzon a backtracking is mohó sorrendben (rendezzük a tárgyakat az értékük szerint csökkenő sorrendbe) a tárgyakkal, és először mindig azt a lehetőséget próbáljuk ki, hogy a tárgyat beletesszük (nyilván csak akkor, ha még belefér) a hátizsákba. Ez azt jelenti, hogy a bináris fában a bal ágakat kódoljuk 1-essel (beletesszük), és a jobb ágakat 0-val (nem tesszük bele). Ily módon a backtracking algoritmus elsőnek éppen a mohó megoldást találja meg, amely ha nem is garantáltan optimális, de elég jó megoldásnak számít. Egy viszonylag jó megoldás korai megtalálása javít az alábbiakban bemutatott optimalizálás hatékonyságán.

Tartsuk nyilván, hogy minden csomópont képviselte állapotban mekkora összsúly van már a hátizsákban (ezt az értéket írtuk az egyes csomópontokba, a hozzájuk tartozó nyereséggel; akt_g és akt_ny az alábbiakban bemutatásra kerülő algoritmusban). Legyen továbbá egy globális változó (max_ny), amely a kurrens optimumértéket, az addig megtalált legjobb megoldás értékét tárolja. Ha ezt kezdetben nullára is állítjuk, az első megoldás megtalálásakor frissül a mohó megoldás értékére (lásd az előbbi bekezdést). Ahhoz, hogy konstans időben tudjuk ellenőrizni, hogy adott pillanatban a hátralevő tárgyak nem férnek-e mind bele a hátizsákba, célszerű előre feltölteni az $a[1..n]$ és $b[1..n]$ tömböket úgy, hogy $a[i] = {}_i\Sigma^n\, t[j].g$, $b[i] = {}_i\Sigma^n\, t[j].a$, (ahol $j=i..n$). Ha $a[1] < G$, akkor a feladat triviális, hiszen ez azt jelenti, hogy az összes tárgy belefér a hátizsákba, és a maximális nyereség $b[1]$. Ellenkező esetben a feladat úgy is átfogalmazható, hogy mely tárgyakat hagyja a tolvaj az üzletben, szem előtt tartva, hogy a maximális nyereséggel szeretne távozni. Mivel valamely tárgy elvetésének esetét a megfelelő *jobb* fiúrészfa képviseli, ezért a következőkben vázolt optimalizálás ezekre fókuszál.

Milyen esetekben kerülhető el a kurrens részfa *jobb* fiúrészfájának bejárása? Tegyük fel, hogy a teljes fa gyökerétől a kurrens részfa gyökeréhez vezető út az $1..k$ tárgyakra vonatkozó, már meghozott döntéseket képviseli. A kurrens részfa *jobb* fia nyilván azt az esetet ábrázolja, hogy az $(k+1)$-edik tárgy nem kerül bele a hátizsákba. Amennyiben a hátralevő tárgyak $(k+2 \ldots n)$ mind belefférnek a hátizsákba, akkor úgymond „gondolkodás nélkül” (konstans időben eldönthető: akt_g + a[k+2] ≤ G) az összest beletesszük (és frissítjük a kurrens optimumot, amennyiben indokolt). Mivel a szóban forgó jobb fiúrészfa e legbaloldalibb levele garantált legjobb (az illető részfára nézve), ezért ennek bejárása, jobb levelek reményében, értelmetlenné vált (6.10. ábra).

Ha a fenti logikával nem kerülhető el az illető *jobb* fiúrészfa bejárása, akkor esetleges terméketlenségét (nem tartalmazza az optimális levelet) az alábbi módon próbálhatjuk meg kideríteni. Vizsgálat céljából folytassuk a bejárást egy olyan mohó algoritmussal, amely elvághatja a tárgyakat. Ezt valósítja meg



6.10. ábra. *A feladat keresési terét ábrázoló bináris fa (megvastagítottuk az optimális gyökér–levél utat; szimpla olló: az illető bal fiú képviselte tárgy már nem fér bele a hátizsákba; a szürke jobb fiúkból hívódik meg, vizsgálat céljából, egy folytonos greedy; a dupla ollóval levágott jobb fiúrészfák garantáltan nem tartalmazzák a megoldáslevelet)*

az alábbi algoritmus supremum függvénye, amely a szóban forgó jobb fiúrészfa egyetlen gyökér–levél útján „szalad le”, tehát lineáris bonyolultságú. Az így kapott nyereség nagyobb (vagy egyenlő) lesz, mint a teljes fa gyökeréből az illető részfa bármelyik leveléhez vezető út nyeresége. A kapcsolódó optimalizálás alapötlete a következő: ha ez a supremum érték sem nagyobb, mint a kurrens optimum, akkor az illető jobb fiúrészfa biztosan nem tartalmazza az optimális levelet, tehát értelmetlen lenne bejárni (így teljesen lemetszhető a fáról).

A 6.10. ábrán besatíroztuk azokat a csomópontokat, amelyekhez tartozó részfákra, a példánk esetében, szóba jön a supremum függvény meghívása. Vastagított dupla vonalkával jelöltük a *jobb* fiúrészfákra vonatkozó optimalizálás „ollóját”. A „dupla ollóval” lemetszett részfákban bejelöltük a vizsgálati mohó utat (ez megegyezhet az illető részfa legbaloldalibb levelébe vezető úttal). Az optimális gyökér–levél utat, amelynek kódja 1010, a vastagított vonal jelzi. Üresen hagytuk azokat a csomópontokat, amelyeknek a bejárását sikerült elkerülni.

A hátizsák rekurzív backtracking eljárás k-adik szinti meghívása az akt_g és akt_ny paraméterekben megkapja, hogy az aktuális állapotban mekkora súly van már a hátizsákban és mennyi nyereséggel. Ezt a két értéket fogja érték szerint átadni a supremum függvénynek is, amely – amint már említettük – vizsgálat céljából mohó módon (folytonos változatban) folytatja a $(k+2), \ldots, n$ tárgyak bepakolását [mivel jobb fiúról van szó, a $(k+1)$-edik tárgy nem kerül a hátizsákba]. A max_ny és opt_x[] cím szerint átadott paraméterekben tárolódik a maximális nyereség és az optimális megoldás kódja.

```
supremum(t[],n,G,akt_g,akt_ny,k)
  minden i=k,n végezd
    ha akt_g + t[i].g ≤ G akkor
      akt_g = akt_g + t[i].g
      akt_ny = akt_ny + t[i].ár
    különben
      akt_ny = akt_ny + (G−akt_g) * t[i].ár / t[i].g
      return akt_ny
    vége ha
  vége minden
  return akt_ny
vége supremum

hátizsák(x[],t[],n,G,akt_g,akt_ny,k,max_ny,opt_x[])
  ha k == n akkor
    ha akt_ny > max_ny akkor                    // frissítjük a kurrens optimumot
      max_ny = akt_ny
      opt_x[1..n] = x[1..n]
    vége ha
  különben
    // bal fiú
    ha akt_g + t[k+1].g ≤ G akkor          // a (k+1). tárgy még belefér a hátizsákba
      x[k+1] = 1
      hátizsák(x,t,n,G,akt_g+t[k+1].g,akt_ny+t[k+1].ár,k+1,max_ny,opt_x)
    vége ha
    // jobb fiú
    ha akt_g + a[k+2] ≤ G akkor       //a maradék (k+2)..n tárgyak mind belefférnek
      ha akt_ny + b[k+2] > max_ny akkor          // frissítjük a kurrens optimumot
        max_ny = akt_ny + b[k+2]
        opt_x[1..n] = x[1..k]+[0]+[1..1]                // a + jel összefűzést jelent
      vége ha
    különben
      sup_ny = supremum(t,n,G,akt_g,akt_ny,k+2)
      ha sup_ny > max_ny akkor       // a jobb fiúrészfa esélyes optimális levélre
        x[k+1] = 0
        hátizsák(x,t,n,G,akt_g,akt_ny,k+1,max_ny,opt_x)
      vége ha
      // hiányzik a különben-ág: a jobb fiúrészfa esélytelen optimális levélre
    vége ha
  vége ha
vége hátizsák
```

Az alábbiakban közöljük azt az algoritmusrészletet, amely tartalmazza a hátizsákeljárás meghívását és az optimális megoldás kiíratását:

```
…
max_ny = 0
hátizsák(x,t,n,G,0,0,0,max_ny,opt_x)
ki: max_ny, opt_x[1..n]
…
```

Vegyük észre, hogy a dupla olló branch-and-bound szellemben metszi meg a fát. A felvezető fejezetben már adtunk egy kis ízelítőt a branch-and-bound olló használatáról. Branch-and-bound ollóval olyan részfákat metszünk le, amelyekről kideríthető, hogy garantáltan nem tartalmaznak jobb megoldásleveleket, mint az addig ismert legjobb megoldás (ha nem jobbak az addigi legjobbnál, akkor nyilván esélytelenek az optimális levél státusra). A supremum függvény egy felső korlátot térít vissza a kurrens csomópont jobb fiúrészfája tartalmazta esetleges megoldáslevelekre vonatkozóan. Ha e felső korlát sem nagyobb az addigi legjobb megoldás értékénél (melyet a max_ny változó képvisel), akkor a szóban forgó részfa lemetszhető a keresési fáról (bejárása átugorható; a sup_ny>max_ny feltételű ha utasításnak hiányzik a különben ága).

Backtracking versus branch-and-bound olló

A backtracking algoritmusok a feladat összes megoldásában érdekeltek, a branch-and-bound stratégia viszont csak az optimálisban. Egy backtracking algoritmus akkor lép vissza (metszi a fát), ha bármely megoldásra nézve száraz irányt érzékel, a branch-and-bound algoritmusok viszont olyan irányokba nem ágaztatják tovább a keresési fát, amerre kizárható, hogy találnának optimális megoldáslevelet.

A fentebb tárgyalt algoritmust úgy is jellemezhetnénk, hogy Backtracking–greedy házasítás branch-and-bound módra.

Hatékony megoldás adható a Hátizsák problémára dinamikus programozás alkalmazásával is, mely technikát a következő fejezet mutatja be. A hátizsákprobléma dinamikus programozásos megoldása végett lásd az *Algoritmusok felülnézetből* jegyzet (Kátai 2007) 9.1. megoldott feladatát.

Az *Algoritmusok felülnézetből* című jegyzet további példákat tartalmaz branch-and-bound algoritmusokra is (lásd a 6.2. és 10.2. megoldott feladatokat).

Alma/körte: Legyen $n+1$ szoba, amelyek vagonszerűen követik egymást. Az első n szoba mindenikében egy bizonyos mennyiségű alma és körte található. Egy elég nagy hátizsákkal rendelkező személynek a következőképpen kell végigmennie a szobákon, szobáról szobára haladva: megérkezve valamely i-edik szobába ($i = 1, \ldots, n$), először kiüresíti a hátizsákját (a megfelelő gyümölcscsomóba), majd bepakolja a szobában található összes almát vagy összes körtét, és rakományát átviszi a következő szobába, az $(i+1)$-edikbe, ahol természetesen hasonlóan jár el (az első szobába üres hátizsákkal lép be). Állapítsuk meg, hogy az egyes szobákban mely típusú gyümölcsöket kell bepakolnia a személynek, ha azt szeretné, hogy minimális kalóriaveszteséggel érkezzen meg az $(n+1)$-edik szobába (két egymás utáni szoba között a szállított gyümölcsök számával egyenlő számú kalóriaegységet veszít a személy; a hátizsák kipakolása és megrakása nem jár kalóriaveszteséggel).

Misszionárius-kannibál: Egy folyó egyik partján van k kannibál, m misszionárius és egy kétszemélyes csónak. Hogyan tudnak átjutni a kannibálok és a misszionáriusok a túlsó partra anélkül, hogy mészárlás történne? Erre akkor kerülne sor, ha valamelyik parton többségben maradnának a kannibálok (és van misszionárius is az illető parton).

7. DINAMIKUS PROGRAMOZÁS

1801-ben karácsonyra Thomas Jefferson, az Amerikai Egyesült Államok akkori elnöke levelet kapott egyik matematikus barátjától, Robert Pattersontól, aki egy általa tökéletesnek nevezett titkosítási rendszerről számolt be. Jefferson nyilván nem tudta feltörni a „tökéletes rendszert", és az ezt követő 200 évben mások sem, viszont a közelmúltban egy Lawren Smithline nevű matematikusnak, számítógépes programok segítségével sikerült. Mi a közös ebben a történetben, a felhőkarcolók liftjeinek ütemezési problémáiban, a vonalkód-generálásban, a nagy halháborúban, a sakk-végjátékokban stb.? E területek mindenikén alkalmazták már a dinamikus programozást optimalizálási problémák megoldására.

A dinamikus programozást mint optimalizálási módszert Richard Bellman javasolta a múlt század közepén, és azóta számos tudományterületen nyert jelentős alkalmazást. A dinamikus programozásos feladatokat többféleképpen is osztályozhatjuk: diszkrét/folytonos, determinisztikus/sztochasztikus, véges/végtelen horizontú stb. Ebben a fejezetben a diszkrét, determinisztikus, véges horizontú problémák kerülnek megvizsgálásra.

7.1. Milyen feladatok oldhatók meg dinamikus programozással?

Amint már többször is utaltunk rá, gyakori jelenség, hogy egy feladat megoldása feltételezi ennek hasonló, egyszerűbb részfeladataira való lebontását. A cél általában az, hogy a részfeladatok megoldásaiból építsük fel az eredeti feladat megoldását (vagy megoldásait). Feladatokat bontunk le, és megoldásokból építkezünk. A bontás és építés ellentétes irányú műveletek. A bontást a triviális részfeladatok (megoldásuk a feladat bemeneti adataiból triviálisan adódik) szintjéig történik, az építkezés pedig erről a szintről indul.

Gyakori eset, hogy a feladat többféleképpen is lebontható részfeladataira. A különböző lebontások szerkezetei meghatározzák a rájuk épülő megoldások felépítését. Bár a hangsúly a megoldások felépítésén van, az építkezési irányokat a bontási vonalak határozzák meg. Egy apafeladat megoldásai azon fiúrészfeladatok megoldásaiból építhetők fel, amelyek az illető apafeladat lebontásából közvetlenül adódnak. Úgy is fogalmazhatnánk, hogy ahhoz, hogy le tudjuk programozni a megoldásépítés folyamatát, át kell hogy lássuk a feladat szerkezetét (ez általában azt feltételezi, hogy legalább gondolatban lebontjuk a feladatot részfeladataira).

A dinamikus programozással megoldható feladatok egyik jellemzője, hogy a lebontásukból származó *különböző* részfeladatok száma gyakran a bemenet mére-

ténak polinom függvénye. Ez általában abból adódik, hogy a lebontásból származó exponenciálisan sok részfeladat közül számottevően sok azonos.

Amennyiben optimalizálási problémáról van szó, akkor egy másik követelmény az, hogy a feladatra igaz legyen az „optimalizálás alapelve", miszerint „az optimális megoldás optimális részmegoldásokból épül fel". Ez garanciát jelent arra vonatkozóan, hogy az optimális megoldás felépíthető a részfeladatok optimális megoldásaiból. Azért annyira lényeges ez, mert ily módon elegendő minden részfeladatnak *csak az optimális megoldását (az ezt képviselő optimumértéket) tárolni*, ami csupán polinomsok értéket jelent. A tárolás (rendszerint egy-, két- vagy többdimenziós tömbben) stratégiailag is fontos, mert ezzel elkerülhető a részfeladatok többszöri megoldása (amennyiben a megoldási folyamat alatt többször is találkoznánk ugyanazzal a részfeladattal).

Természetesen az optimális megoldás felépítése azt is feltételezi, hogy az optimális részmegoldásokból *optimális módon építkezzünk* (akkor lesz optimális épületünk, ha optimális anyagokat optimális módon építünk össze). Az optimális építkezés módját matematikailag egy rekurzív képlettel szokás megadni, amelyen belül az optimalizálás egy minimum vagy maximum függvényben fogalmazódik meg.

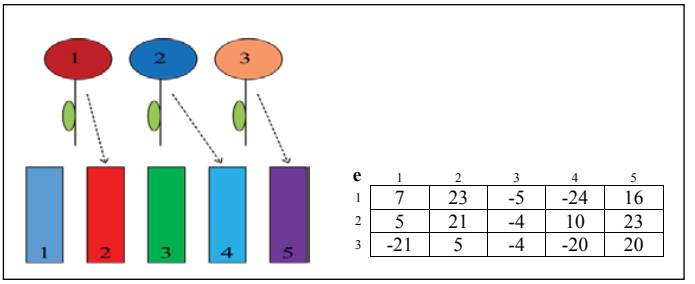
A fentiekkel összhangban a dinamikus programozás lentről felfele (egyszerűtől a bonyolult felé) építkezést jelent: kiindulva a triviálisan egyszerű részfeladatok nyilvánvaló optimális megoldásaiból, felépítjük, lépésről lépésre, az egyre bonyolultabb részfeladatok optimális megoldásait, végül az eredeti feladatét. Ez általában annyit jelent, hogy az optimumértékeket tároló tömb triviális részfeladatokat képviselő, implicite kitöltött celláitól elindulva egyre több „szomszédos cellát" töltünk ki (a rekurzív képlet alapján), míg végül ki tudjuk tölteni az eredeti feladatot képviselő cellát is. Ha minden egyes cellában nemcsak az optimumértéket tároljuk, hanem kódoljuk az optimális döntést is, amely ezt szolgáltatta, akkor az optimumértékek tömbjéből egy az egyben visszaolvasható lesz az optimális döntéssorozat is, amely az optimális megoldást eredményezte.

Akkor is részfeladatokként egy értékkel van dolgunk, ha olyan feladatot oldunk meg, amelyben a megoldások *száma* érdekel. Tehát egy másik feladatcsalád, amely dinamikus programozással megoldható: a megszámlálási feladatok. Itt is feltétel, hogy az eredeti feladat polinom-sok hasonló, egyszerűbb részfeladatra legyen lebontható. Ez esetben, mivelhogy nem optimalizálásról van szó, a rekurzív képlet nem fog minimum vagy maximum függvényt tartalmazni.

7.2. Recept dinamikus programozásos feladatmegoldáshoz

Az előbbi gondolatsor egy 5 lépéses dinamikus programozásos feladatmegoldási módszert sugall. Szemléltetésül tekintsük az alábbi feladatot:

Virágüzlet: Egy virágüzlet kirakatában van m váza (1, 2, …, m sorrendben), és ezekbe úgy kell elhelyezni az 1, 2, …, n virágokat (ebben a sorrendben; $n \leq m$), hogy az esztétikai összhatás maximális legyen (7.1. ábra). (Az $e[1..n][1..m]$ tömb $e[i][j]$ cellája azt tárolja, hogy az i virág a j vázában milyen esztétikai hatást kelt; az üresen maradt vázák esztétikai hatása nulla.) (Nemzetközi Informatika Olimpia, Törökország, 1999)



e	1	2	3	4	5
1	7	23	-5	-24	16
2	5	21	-4	10	23
3	-21	5	-4	-20	20

7.1. ábra. *Példa 3 virágra és 5 vázára. Az optimális megoldás: 1. váza üres; 2. vázában 1. virág; 3. váza üres; 4. vázában 2. virág; 5. vázában 3. virág. A maximális esztétikai összhatás: 53*

Íme a javasolt recept:

1. Meghatározzuk a részfeladatok általános alakját. Ha egy adott példára, gondolatban, lefuttatjuk a részfeladatokra bontás folyamatát, akkor ez segíthet érzékelni, hogy mi az általános alak (egy paraméteres alak, amely általánosan jellemzi a lebontásból adódó összes részfeladatot). Elgondolkodhatunk azon is, hogy mely paraméterértékekre kapunk triviálisan egyszerű részfeladatokat, illetve mely értékek eredményezik az eredeti feladatot. Milyen irányú paraméterérték-változás jelenti a lentről felfele építkezést?

Általános alak: az 1..i virágok optimális elhelyezése az 1..j vázákba ($0 \leq i \leq n$, $i \leq j \leq m-n+i$). Az i. virág nem kerülhet az i. vázánál előbbre (hogy legyen váza az 1..($i-1$) virágnak), illetve az ($m-(n-i)$)-edik vázánál hátrább (hogy maradjon váza a fennmaradt $n-i$ virágnak is).

Optimumérték: az optimális elhelyezés keltette esztétikai összhatás értéke.

Optimális megoldás: az optimális elhelyezés módja.

Triviális részfeladatok:

$i=0$ (nulla virág elhelyezése bármennyi vázába); méretében triviális

$i=j$ (ugyanannyi a virág, mint a váza); minőségében triviális, mert egyértelmű, hogy az i. virág az i. vázába kell hogy kerüljön.

Eredeti feladat: $i=n$, $j=m$.

Lentről felfele irány: i és j növekednek.

2. Hol tároljuk a részfeladatok optimális megoldásait képviselő optimumértékeket? Általában, ahány független paramétert tartalmaz az általános alak, annyi dimenziós tömbre lesz szükségünk. Mely cellák fogják tárolni a triviális részfeladatok optimumértékeit, és melyik az eredeti feladatét?

Optimumértékek tömbje: $c[0..n][0..m]$ 2-dimenziós tömb satírozott területe ($i=1..n$, $j=i..m-n+$i). (7.2. ábra)

Triviális részfeladatokat képviselő cellák: $c[0][j]$, $j=0..m-n$; $c[i][i]$, $i=1..n$.

Eredeti feladatot képviselő cella: $c[n][m]$.

3. Meghatározunk egy általános rekurzív képletet, amely matematikailag leírja az optimális építkezés módját: az optimumok tömbje valamely „apacellája", *mely közvetlen* „fiúcellák" értékeitől, *milyen módon* függ(het)? Segíthet átlátni a képletet, ha érzékeljük egy általános apafeladat megoldása feltételezte lentről felfele döntéssorozat utolsó döntését, azt, amely a lebontásából adódó közvetlen fiúrészfeladatokra támaszkodik.

„Utolsó döntés" az „(i,j) feladatot" illetően: (1) az i. virág a j. vázába kerül, vagy (2) a j. váza üresen marad. Az első változat esetében a fiúrészfeladat: ($i-1$,$j-1$) ($1..i-1$ virágok optimális elhelyezése az $1..j-1$ vázákba). A második esetben az (i,$j-1$) fiúrészfeladhoz jutunk: $1..i$ virágok optimális elhelyezése az $1..j-1$ vázákba.

A képlet optimalizálási ága: $c[i][j] = \max\{c[i-1][j-1] + e[i][j]; c[i][j-1] + 0\}$, $i=1..n$, $j=i+1..m-n+i$.

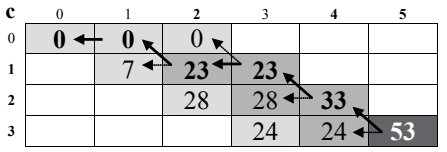
A képlet triviális ágai: $c[0][j] = 0$, $j=0..m-n$; $c[i][i] = c[i-1][i-1] + e[i][i]$, $i=1..$n.

4. Megírjuk az iteratív algoritmust, amely a rekurzív képlet alapján („lentről felfele" irányba) feltölti az optimumértékek tömbjét.

```
minden j = 0,m − n végezd                    // méretükben triviális részfeladatok
   c[0][j] = 0
vége minden
minden i = 1,n végezd                        // minőségükben triviális részfeladatok
   c[i][i] = c[i−1][i−1] + e[i][i]
vége minden
minden i = 1,n végezd                        // nem triviális részfeladatok
   minden j = i+1,m − n+i végezd
      ha c[i−1][j−1] + e[i][j] > c[i][j−1] akkor
         c[i][j] = c[i−1][i−1] + e[i][j]
      különben
         c[i][j] = c[i][i−1]
      vége ha
   vége minden
vége minden
ki: c[n][m]
```

5. Az optimumtömbből kiolvassuk („fentről lefele” irányba) az optimális döntéssorozatot (amely az optimális megoldást eredményezi) (7.2. ábra). Ha egy rekurzív függvényt írunk az optimális döntéssorozat visszaolvasásához, és a rekurzió vissza-útján írjuk ki a döntéseket, akkor ezek előre sorrendben jelennek meg a képernyőn.

```
Optimális_megoldás(e[][],c[][],i,j)
  ha i > 0 és j > i akkor                          // nem triviális részfeladatok
    ha c[i][j] == c[i−1][i−1] + e[i][j] akkor
      Optimális_megoldás(e,c,i−1,j−1)
      ki: i, „-edik virág, ”, j, „-edik vázába”
    különben
      Optimális_megoldás(e,c,i,j−1)
      ki: j, „-edik váza üres”
    vége ha
  különben
    ha i == 0 és j > 0 akkor                       // méretükben triviális részfeladatok
      Optimális_megoldás(e,c,i,j−1)
      ki: j, „-edik váza üres”
    vége ha
    ha i > 0 és i == j akkor                       // minőségükben triviális részfeladatok
      Optimális_megoldás(e,c,i−1,j−1)
      ki: i, „-edik virág, ”, j, „-edik vázába”
    vége ha
  vége ha
vége Optimális_megoldás
```



7.2. ábra. *A feltöltött c tömb, amelyből visszaolvasható (mohó módon) az optimális döntéssorozat (világosszürke: „triviális szegély”; sötétszürke: célcella)*

Ötlépéses receptünk, tömören:

1. Meghatározzuk a részfeladatok általános, paraméteres alakját. (Lévén szó „különböző méretű” hasonló részfeladatok egy családjáról, nyilván beszélhetünk ezek általános alakjáról.)

2. Eldöntjük, hogy hol tároljuk a részfeladatok optimális megoldásait képviselő optimumértékeket. (Az optimalizálandó célfüggvény optimumértékeit a részfeladatokra vonatkoztatva.)
3. Meghatározunk egy rekurzív képletet, amely matematikailag leírja az optimális építkezés módját. (Mi a módja, hogy a már rendelkezésre álló „fiú-optimumokból" „apaoptimumokat" építsünk?)
4. Implementáljuk az iteratív algoritmust, amely a rekurzív képlet alapján („lentről felfele irányba") feltölti az optimumértékek tömbjét. (A „triviális szegélyen" lévő celláktól „optimumhidat" építünk az „ellenkező oldalon" található célcellához.)
5. Az optimumtömbből kiolvassuk („fentről lefele irányba") az optimális döntéssorozatot (amely az optimális megoldást eredményezi). (Meghatározzuk az „optimumhíd" célcellába vezető „optimális útját".)

7.3. Dinamikus programozásos stratégiák osztályozása

A dinamikus programozásos feladatok igen sokfélék lehetnek. Azért, hogy egy viszonylag átfogó képet kapjunk a módszerről, célszerű lehet követni az alábbi osztályozást: 1. monadikus (monadic) / poliadikus (polyadic); 2. soros(serial) / nem-soros(non-serial).

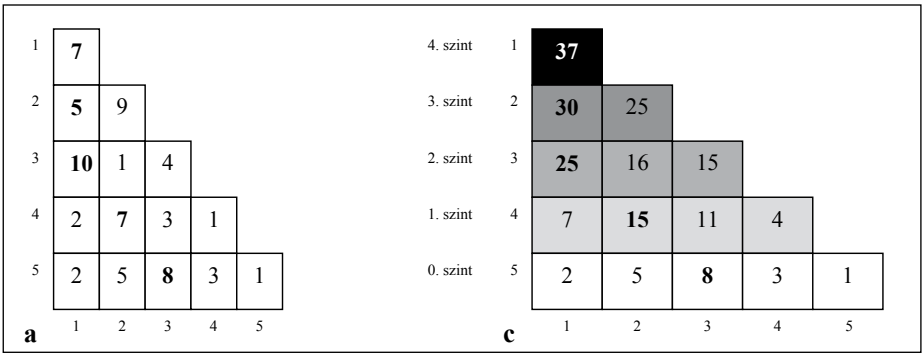
Ahogy az eddigiekből is kiderült, a dinamikus programozásos építkezés szintről szintre halad, lentről felfele. A 0. szinten találhatók a triviális részfeladatok, amelyek optimális megoldásai implicite adódnak az input adatokból. Első szinten azok a részfeladatok oldhatók meg, amelyek optimális megoldásai közvetlenül adódnak a „triviális optimumokból". Általánosan: A k. szinti részfeladatok optimális megoldásai kizárólag $0..(k-1)$ szinti optimumoktól függnek. A legfelső szinten található, nyilván, az eredeti feladat.

Ha a k. szinti feladatok megoldásai *kizárólag* $(k-1)$. szintiek megoldásaitól függnek (vagy függhetnek), akkor a feladatot *sorosnak* nevezzük, különben *nemsorosnak*. Ha bármely részfeladat optimális megoldásába *egyetlen* alsóbb szinti részfeladat optimuma épül be, akkor *monadikus* feladatról beszélünk, különben *poliadikusról*. Az alábbiakban felsorolunk mindenik kategóriából egy-egy példát.

Monadikus–soros (Háromszög-optimalizálás): Az $a[1..n][1..n]$ tömb főátló menti és főátló alatti cellái természetes számokat tárolnak. Határozzuk meg a csúcsból ($a[1][1]$ cella) alapra (n. sor) vezető „legjobb utat" (amely mentén a legnagyobb az összeg), ha a megengedett továbblépési irányok: le vagy átlósan jobbra (7.3. ábra). (Nemzetközi Informatika Olimpia, Svédország, 1994; e feladaton keresztül nyújtottunk előzetest az öt módszer stratégiájáról a 2.2. alfejezetben.)

1. Általános alak: (i,j) pozícióból $(1 \le i < n,\ 1 \le j \le i)$ induló alapra vezető legjobb út meghatározása.
   Triviális részfeladatok: (n,j) pozíciókból induló legjobb utak meghatározása.
   Eredeti feladat: az (1,1) pozícióból induló legjobb út meghatározása.
2. Optimumértékek tömbje: $c[1..n][1..n]$. (A $c[i][j]$ cella az (i,j) pozícióból alapra vezető legjobb út optimumértékét tárolja.)
3. Rekurzív képlet: mivel egy (i,j) pozícióból, egy lépésből, az $(i+1,j)$ vagy $(i+1,j+1)$ pozíciókba juthatunk, ezért a $c[i][j]$ apaoptimum, a $c[i+1][j]$ és $c[i+1][j+1]$ közvetlen fiúoptimumok építhető fel. Hogyan?
   $c[n][j] = a[n][j]$, $1 \le j \le n$;
   $c[i][j] = \max\{a[i][j] + c[i+1][j],\ a[i][j] + c[i+1][j+1]\}$, $1 \le i < n$, $1 \le j \le i$.
4. Iteratív algoritmus a c tömb feltöltésére:

```
minden j = 1,n végezd                                  // triviális optimumok
   c[n][j] = a[n][j]
vége minden
minden i = n−1,1,−1 végezd                             // nem triviális optimumok
   minden j = 1,i végezd
      ha c[i+1][j] > c[i+1][j+1] akkor                 // lefele az optimális
         c[i][j] = a[i][j] + c[i+1][j]
      különben                                         // átlósan jobbra az optimális
         c[i][j] = a[i][j] + c[i+1][j+1]
      vége ha
   vége minden
vége minden
ki: c[1][1]
```



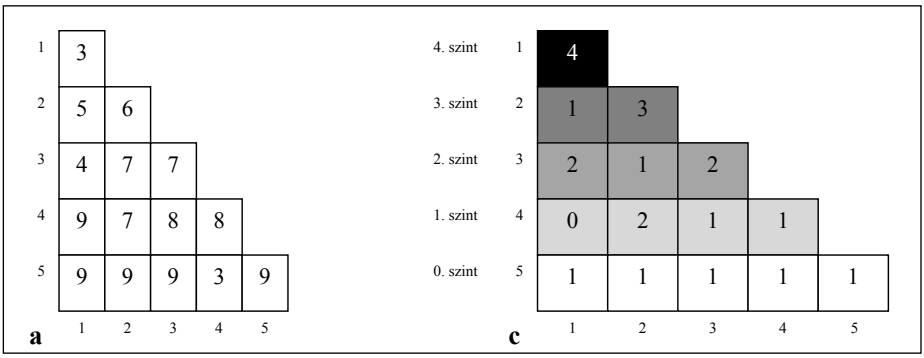
7.3. ábra. *Példatömbök a háromszög feladat optimalizálásos változatához (a[1..5][1..5], c[1..5][1..5]); megvastagítottuk az optimális út menti számokat. A feladat azért soros, mert a k. szinti optimumok csak (k-1). szintiektől függnek*

5. Mi eredményezte a $c[1][1]$ optimumot (lásd a 7.3. ábrát)? A $c[2][1]$. És a $c[2][1]$-et? A $c[3][1]$… Tehát a c tömbbeli csúcsból alapra vezető mohó út (minden lépésben a nagyobb elem irányába lépünk tovább) az eredeti mátrix optimális útját adja meg. Az implementálás maradjon az olvasóra. (Vegyük észre, hogy az a tömbbeli mohó út nem jelentett optimálist.)

Háromszög-számlálás: A háromszögfeladat egy számlálási változata, hogy hány csúcsból alapra vezető szigorúan növekvő út létezik, ha a megengedett irányok: le vagy átlósan jobbra (7.4. ábra).

1. Általános alak: (i,j) pozícióból ($1 \le i < n$, $1 \le j \le i$) induló, alapra vezető, szigorúan növekvő utak száma.
2. A részfeladatok megoldásai számát tároló tömb: $c[1..n][1..n]$. (A $c[i][j]$ cella az (i,j) pozícióból alapra vezető szigorúan növekvő utak számát tárolja.)
3. Rekurzív képlet: ha $a[i][j]$ kisebb $a[i+1][j]$-nél is és $a[i+1][j+1]$-nél is, akkor az $a[i][j]$ elem beékelődhet mind az $(i+1,j)$, mind az $(i+1,j+1)$ pozícióból induló növekvő részsorozatok elé (a képlet ezen ága poliadikus jelleget ölt).
   $c[n][j] = 1$, $1 \le j \le$ n;
   $c[i][j] = 0$, ha $a[i][j] \ge a[i+1][j]$ és $a[i][j] \ge a[i+1][j+1]$;
   $= c[i+1][j]$, ha $a[i][j] < a[i+1][j]$ és $a[i][j] \ge a[i+1][j+1]$;
   $= c[i+1][j+1]$, ha $a[i][j] \ge a$[i+1][j] és $a[i][j] < a[i+1][j+1]$;
   $= c[i+1][j] + c[i+1][j+1]$, ha $a[i][j] < a[i+1][j]$ és $a[i][j] < a[i+1][j+1]$.
4. Iteratív algoritmus a c tömb feltöltésére (kezdetben minden eleme 0):

```
minden j = 1,n végezd                          // triviális méretű részfeladatok
  c[n][j] = 1
vége minden
minden i = n−1,1,−1 végezd                     // nem triviális részfeladatok
  minden j = 1,i végezd
    ha c[i+1][j] > c[i][j] akkor               // lefele növekvő
      c[i][j] += c[i+1][j]
    vége ha
    ha c[i+1][j+1] > c[i][j] akkor             // átlósan-jobbra növekvő
      c[i][j] += c[i+1][j+1]
    vége ha
  vége minden
vége minden
ki: c[1][1]
```



7.4. ábra. *Példatömbök a háromszög feladat számlálási változatához (a[1..5][1..5], c[1..5][1..5]). A feladat azért soros, mert a k. szinti eredmények csak (k-1). szintiektől függnek*

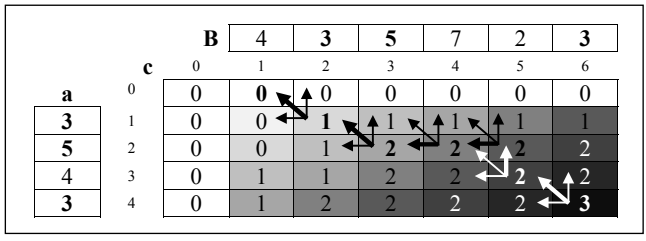
Monadikus–nem-soros (Leghosszabb közös részsorozat): Határozzuk meg az a[1..n] és b[1..m] tömbök leghosszabb közös részsorozatát (7.5. ábra).

1. Általános alak: az $a[1..i]$ és $b[1..j]$ tömbszakaszok leghosszabb közös részsorozatának meghatározása.
   Triviális részfeladatok: $i=0$ vagy $j=0$ (valamelyik részsorozat üres).
   Eredeti feladat: $i=n$, $j=m$.
2. Optimumértékek tömbje: $c[0..n][0..m]$. (A $c[i][j]$ cella az $a[1..i]$ és $b[1..j]$ tömbszakaszok leghosszabb közös részsorozatának hosszát tárolja.)
3. Rekurzív képlet: ha $a[i]=b[j]$, akkor e közös érték bekerül a leghosszabb közös részsorozatba, és ily módon, a $c[i][j]$ apaoptimum a $c[i-1][j-1]$ fiúoptimumra épül; ha $a[i]\neq b[j]$, akkor a $c[i][j-1]$ és $c[i-1][j]$ fiúoptimumok valamelyike szolgáltatja a $c[i][j]$ apa-optimumot.
   $c[i][0] = 0$; $c[0][j] = 0$, ahol $0\leq i\leq n$, $0\leq j\leq m$.
   $c[i][j] = c[i-1][j-1] + 1$, ha $1\leq i\leq n$, $1\leq j\leq m$, $a[i]=b[j]$.
   $c[i][j] = \max\{c[i][j-1], c[i-1][j]\}$, ha $1\leq i\leq n$, $1\leq j\leq m$, $a[i]\neq b[j]$.
4. Iteratív algoritmus a c tömb feltöltésére (kezdetben minden eleme 0):

```
minden i = 1,n végezd                          // triviális optimumok
   c[i][0] = 0
vége minden
minden j = 1,m végezd                          // triviális optimumok
   c[0][j] = 0
vége minden
minden i = 1,n végezd                          // nem triviális optimumok
   minden j = 1,m végezd
```

```
        ha a[i] == b[j] akkor
          c[i][j] = c[i−1][j−1] + 1
        különben
          ha c[i−1][j] > c[i][j−1] akkor
            c[i][j] = c[i−1][j]
          különben
            c[i][j] = c[i][j−1]
          vége ha
        vége ha
      vége minden
    vége minden
    ki: c[n][m]
```

5. Mi eredményezte a $c[4][6]$ optimumot (lásd a 7.5. ábrát)? A $c[3][5]$. És a $c[3][5]$-öt? A $c[2][5]$… Maradjon az olvasóra az optimális döntéssorozat rekonstruálása a feltöltött c tömb alapján. (Ha rekurzív eljárást írunk, és a rekurzió visszaútján íratjuk ki az optimális döntéseket, akkor előre sorrendben jelenik meg a leghosszabb közös részsorozat a képernyőn.)



7.5. ábra. *Leghosszabb közös részsorozat: 3,5,3. A 0. szint háttere fehér. Az „egyszínű” átlókon található cellák ugyanazon szinten levő részfeladatokat képviselnek. A feladat azért „nem-soros”, mert a k. szinti optimális megoldások függhetnek mind a (k-1), mind a (k-2). szinti optimumoktól (lásd a berajzolt nyilakat; megvastagítottuk az optimális döntéseket képviselőket)*

Poliadikus–soros (Floyd algoritmusa): Legyen n város. Az $a[1..n][1..n]$ tömb $a[i][j]$ cellája azt tárolja, hogy mekkora az i és j városok közti direkt útvonal költsége (nem létező útvonalon a költség ∞; a negatív költségértékek nyereségként tekintendők). Határozzuk meg minden várospár közt a legelőnyösebb utak költségét (7.6. ábra).

1. Általános alak: Az (i,j) várospár között azon legelőnyösebb út meghatározása, amely legfennebb az $1..k$ köztes állomásokon halad át. A szint-

ről szintre való egyszerűtől bonyolult felé haladás a k növekedésével (k=0,1,..,n) jár kéz a kézben.

Triviális részfeladatok: k=0 (direkt utak; nincsenek köztes állomások).

Eredeti feladat: k=n (bármely város lehet köztes állomás).

2. Optimumértékek tömbje: c[1..n][1..n] [a kurrens k-ra tárolja az optimumokat; a k. szinti optimumok felülírhatók a (k+1). szintiekkel, lásd a 7.7. ábrát].
3. Rekurzív képlet: az (i,j) várospár közötti, legfennebb az 1..k köztes állomásokon áthaladó, legelőnyösebb út (c_k[i][j] apaoptimum) tartalmazhatja (a c_{k-1}[i][k] és c_{k-1}[k][j] fiúoptimumokra épül), vagy nem (a c_{k-1}[i][j] fiúoptimumra épül), a k. állomást.

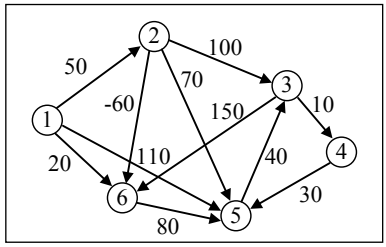
   c_0[i][j] = a[i][j], ahol 1≤i≤n, 1≤j≤n.

   c_k[i][j] = min {c_{k-1}[i][j], c_{k-1}[i][k] + c_{k-1}[k][j]}, ahol 1≤i≤n, 1≤j≤n.

   (Azért soros, mert a k. szinti optimumok csak (k−1). szintiektől függnek. Azért poliadikus, mert megtörténhet, hogy valamely optimumérték *két* másik összegéből adódik.)
4. Iteratív algoritmus az optimumértékek felépítésére a c tömbben:

```
minden i = 1,n végezd                    // a c tömb kezdetben az a mátrixot tárolja
   minden j = 1,n végezd                 // ezek triviális optimumokat jelentenek
      c[i][j] = a[i][j]
   vége minden
vége minden
minden k = 1,n végezd                    // k. szinti optimumok, a (k−1). szintiekből
   minden i = 1,n végezd
      minden j = 1,n végezd
         ha i ≠ k és j ≠ k akkor         // nem lehet köztes állomás a k
            c[i][j] = min(c[i][j], c[i][k]+c[k][j])
            // ck[i][j] = min(ck−1[i][j], ck−1[i][k]+ck−1[k][j])
         vége ha
      vége minden
   vége minden
vége minden
minden i = 1,n végezd
   minden j = 1,n végezd
      ki: c[i][j]
   vége minden
vége minden
```

5. Ha érdekelnek a legelőnyösebb utak is, nem csak a költségük, akkor célszerű nyilvántartani az optimális utak utolsó előtti állomásait is. Használhatunk erre egy ue[1..n][1..n] tömböt. Valahányszor egy c[i][j] cella értéke felülíródik a c[i][k]+c[k][j] összeggel, a megfelelő ue[i][j] cella frissül az ue[k][j] értékkel.



7.6. ábra. *Példa 6 város közti irányított úthálózatra. Az élekhez rendelt értékek a megfelelő irányított szakaszok költségét jelentik. A negatív költségértékek nyereségként tekintendők. Bármely pontpár között a vastagított élek menti a legelőnyösebb út*

$\mathbf{C_0}$

	1	2	3	4	5	6
1	0	50	∞	∞	110	20
2	∞	0	100	∞	70	-60
3	∞	∞	0	10	∞	150
4	∞	∞	∞	0	30	∞
5	∞	∞	40	∞	0	∞
6	∞	∞	∞	∞	80	0

→ $\mathbf{C_1}$

	1	2	3	4	5	6
1	0	50	∞	∞	110	20
2	∞	0	100	∞	70	-60
3	∞	∞	0	10	∞	150
4	∞	∞	∞	0	30	∞
5	∞	∞	40	∞	0	∞
6	∞	∞	∞	∞	80	0

$\mathbf{C_1}$

	1	2	3	4	5	6
1	0	50	∞	∞	110	20
2	∞	0	100	∞	70	-60
3	∞	∞	0	10	∞	150
4	∞	∞	∞	0	30	∞
5	∞	∞	40	∞	0	∞
6	∞	∞	∞	∞	80	0

→ $\mathbf{C_2}$

	1	2	3	4	5	6
1	0	50	**150**	∞	110	**-10**
2	∞	0	100	∞	70	-60
3	∞	∞	0	10	∞	150
4	∞	∞	∞	0	30	∞
5	∞	∞	40	∞	0	∞
6	∞	∞	∞	∞	80	0

$\mathbf{C_2}$

	1	2	3	4	5	6
1	0	50	150	∞	110	-10
2	∞	0	100	∞	70	-60
3	∞	∞	0	10	∞	150
4	∞	∞	∞	0	30	∞
5	∞	∞	40	∞	0	∞
6	∞	∞	∞	∞	80	0

→ $\mathbf{C_3}$

	1	2	3	4	5	6
1	0	50	150	**160**	110	-10
2	∞	0	100	**110**	70	-60
3	∞	∞	0	10	∞	150
4	∞	∞	∞	0	30	∞
5	∞	∞	40	**50**	0	**190**
6	∞	∞	∞	∞	80	0

$\mathbf{C_3}$

	1	2	3	4	5	6
1	0	50	150	160	110	-10
2	∞	0	100	110	70	-60
3	∞	∞	0	10	∞	150
4	∞	∞	∞	0	30	∞
5	∞	∞	40	50	0	190
6	∞	∞	∞	∞	80	0

→

$\mathbf{C_4}$

	1	2	3	4	5	6
1	0	50	150	160	110	-10
2	∞	0	100	110	70	-60
3	∞	∞	0	10	**40**	150
4	∞	∞	∞	0	30	∞
5	∞	∞	40	50	0	190
6	∞	∞	∞	∞	80	0

$\mathbf{C_4}$

	1	2	3	4	5	6
1	0	50	150	160	110	-10
2	∞	0	100	110	70	-60
3	∞	∞	0	10	40	150
4	∞	∞	∞	0	30	∞
5	∞	∞	40	50	0	190
6	∞	∞	∞	∞	80	0

→

$\mathbf{C_5}$

	1	2	3	4	5	6
1	0	50	150	160	110	-10
2	∞	0	100	110	70	-60
3	∞	∞	0	10	40	150
4	∞	∞	**70**	0	30	**220**
5	∞	∞	40	50	0	190
6	∞	∞	**120**	**130**	80	0

$\mathbf{C_5}$

	1	2	3	4	5	6
1	0	50	150	160	110	-10
2	∞	0	100	110	70	-60
3	∞	∞	0	10	40	150
4	∞	∞	70	0	30	220
5	∞	∞	40	50	0	190
6	∞	∞	120	130	80	0

→

$\mathbf{C_6}$

	1	2	3	4	5	6
1	0	50	**110**	**120**	**70**	-10
2	∞	0	**60**	**70**	**20**	-60
3	∞	∞	0	10	40	150
4	∞	∞	70	0	30	220
5	∞	∞	40	50	0	190
6	∞	∞	120	130	80	0

7.7. ábra. *A c tömb n=6 lépésben való átalakulása a direkt utak mátrixából az optimális utak mátrixává. A c_k mátrix a k. szinti optimumokat tárolja. Minden lépésben szürke háttérrel jeleztük, hogy mely (k−1). optimumok felhasználásával mely k. szinti optimumok kerülnek meghatározásra. E szürke területek átfedődés mentessége teszi lehetővé, hogy a c_{k-1} mátrixot felülírjuk a c_k mátrixszal. Minden lépésben megvastagítottuk a frissülő értékeket*

Poliadikus–nem-soros (tükörszó): Az $s[1..n]$ karakterláncot bontsunk minimális számú tükörszóra (7.8. ábra).

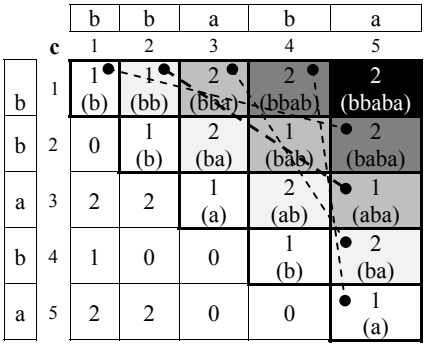
1. Általános alak: Egy $s[i..j]$ karakterláncszakasz minimális tükörszóra bontása.
   Méretében triviális részfeladatok: $i=j$ (egy elemű szakaszok).
   Minőségükben triviális részfeladatok: az $s[i..j]$ szakasz tükörszó.
   Eredeti feladat: i=1, j=n.
2. Optimumértékek tömbje: $c[1..n][1..n]$ (főátló és főátló feletti háromszög).

3. Rekurzív képlet: az $s[i..j]$ nem szimmetrikus szakasz, egy vágással, $j-i$ féleképpen bontható $s[i..k]$, $s[k+1..j]$ alakú párosra, ahol $k=i,j-1$.
   $c[i][i] = 1$, $1 \le i \le n$;
   $c[i][j] = 1$, ha $s[i..j]$ tükörszó;
   $c[i][j] = \min \{ c[i][k] + c[k+1][j]$, minden $k=i,j-1\}$, ha $s[i..j]$ nem tükörszó.
4. Iteratív algoritmus a c tömb feltöltésére (a főátló alatti háromszöget felhasználjuk az optimális vágások helyeinek eltárolására):

```
minden i = 1,n végezd                          // méretükben is triviális optimumok
  c[i][i] = 1
vége minden
minden i = n−1,1 végezd                                  // lentről felfele
  minden j = i+1,n végezd                                // balról jobbra
    ha tükörszó(s,i,j) akkor                  // minőségükben triviális optimumok
      c[i][j] = 1
      c[j][i] = 0                                        // nem igényel vágást
    különben                                             // nem triviális optimumok
      c[i][j] = n                    // „nagy” kezdőérték a minimumszámoláshoz
      minden k = i,j−1 végezd
        ha c[i][k] + c[k+1][j] < c[i][j] akkor
          c[i][j] = c[i][k] + c[k+1][j]
          c[j][i] = k                                    // eltárolom a vágás helyét
        vége ha
      vége minden
    vége ha
  vége minden
vége minden
ki: c[n][n]
```

5. Mi eredményezte a $c[1][5]$ optimumot (lásd a 7.8. ábrát)? A $\{c[1][c[5][1]], c[c[5][1]+1][5]\}$ páros, azaz a $\{c[1][2], c[3][5]\}$ páros. Hát e páros elemeit, egyenként, mik eredményezték? … Íme a rekurzív eljárás, amely kiírja a minimális tükörszóra bontást:

```
Optimális_megoldás(s[],c[][],i,j)
  ki: '('
  ha i == j vagy c[j][i] == 0 akkor                  // s[i..j] tükörszó
    ki: s[i..j]
  különben
    Optimális_megoldás(s,c,i,c[j][i])
    Optimális_megoldás(s,c,c[j][i]+1,j)
  vége ha
  ki: ')'
vége Optimális_megoldás
```



7.8. ábra. *A c tömb feltöltése a s[1..5]=”bbaba” példára. Megadtuk az egyes cellák képviselte rész-karakterláncokat is. Bejelöltük az s[1..5] szakaszt képviselő c[1][5] cella kapcsán a 4 lehetséges vágásból (k=1,2,3,4) adódó fiúcellapárokat. A főátló alatti háromszög tárolja az optimális k értékeket. Az „egyszínű” átlókon található cellák ugyanazon szinten levő részfeladatokat képviselnek (adott hosszúságú karakterláncokra vonatkoznak). A feladat azért „nem-soros”, mert a k. szinti nem triviális optimumok nem csak (k-1). szinti optimumoktól függnek. Azért poliadikus, mert a nem triviális optimumértékek két másik összegéből adódnak.*

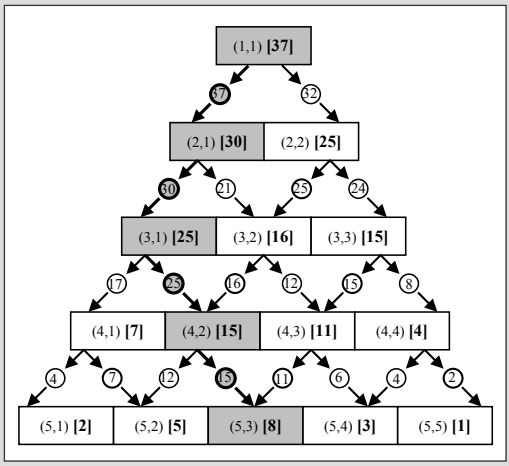
***d*-gráfok**

A monadikus feladatok esetében minden lépésben (döntéssel) a kurrens feladat egy hasonló, egyszerűbb részfeladattá *redukálódik*. A poliadikus feladatok esetében viszont a kurrens feladat minden lépésben (döntéssel) *szétesik* két vagy több hasonló, egyszerűbb részfeladatra. E jelenség egységes modellezése érdekében Kátai (2010) bevezette a *d*-gráfok fogalmát (sajátos ÉS/VAGY gráfok). A *d*-gráfoknak vannak *p* és *d* típusú pontjai. A *p*-pontok képviselik a részfeladatokat. Az eredeti feladatot egy *p*-forrás, a triviális részfeladatokat pedig *p*-nyelők ábrázolják. A *d*-pontok egymásnak közvetlenül alárendelt *p*-pontok közé ékelődnek be a következőképen: (1) minden *p*-pontnak (kivéve a nyelőket) annyi *d*-fia van, ahány lehetőség közül választhatunk az illető részfeladatra vonatkozó döntés kapcsán; (2) egy *d*-pont *p*-fiai azok a *p*-pontok lesznek, amelyek az illető választásból adódó részfeladatokat képviselik. Vegyük észre, hogy a *d*-pontoknak pontosan egy *p*-apjuk van, a *p*-fiaik száma viszont lehet egy (monadikus esetben) vagy több (poliadikus esetben) is.

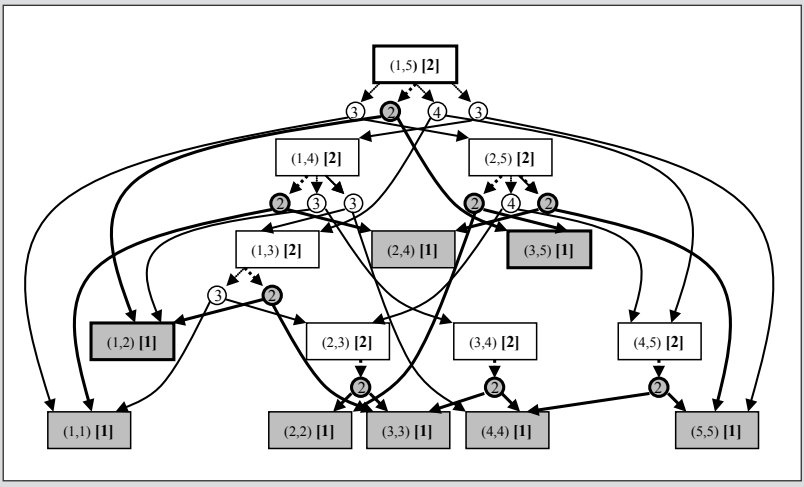
Szemléltetésül tekintsük a háromszög feladathoz rendelhető *d*-gráfot (7.9. ábra: a *p*-pontokat téglalapok, a *d*-pontokat pedig körök ábrázolják). Lévén szó monadikus feladatról, minden *d*-pontnak egyetlen *p*-fia van. A poliadikus eset illusztrálásához a 7.10. ábrán megtekinthető a tükörszó feladathoz rendelhető *d*-gráf. A *p*-pontoknak

annyi d-fiuk van, ahány helyen vágható el a megfelelő karakterláncszakasz. Mivel minden vágásból két részkarakterlánc adódik, ezért minden d-pontnak két p-fia van.

További részletek végett lásd a Kátai 2010 tanulmányt.



7.9. ábra. *A háromszög feladat* d*-gráfja*



7.10. ábra. *A tükörszó feladat* d*-gráfja*

7.4. Két sajátos példa

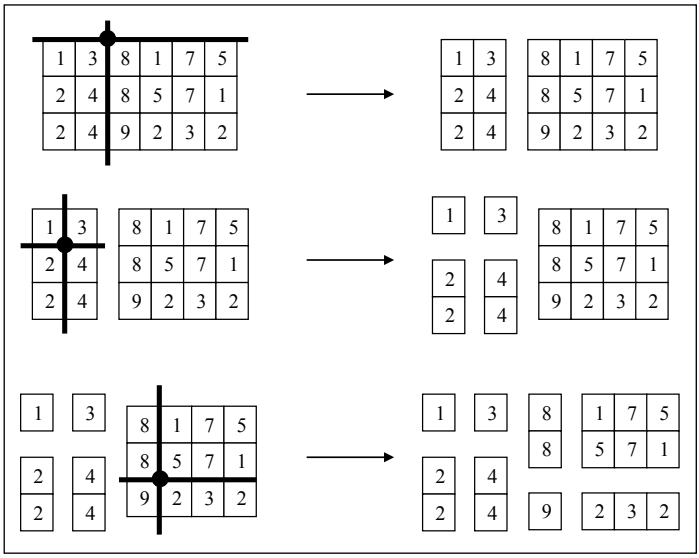
Az eddig bemutatott példákban az optimumértékek tárolására egy- vagy kétdimenziós tömböket használtunk. Bár a Floyd-algoritmus kapcsán a részfeladatokat három független paraméterrel jellemeztük – a soros jelleg és a rekurzív képlet sajátosságaiból adódóan –, elégséges volt egy kétdimenziós tömb használata ez esetben is. Az alábbi feladatot a Sapientia-ECN verseny 2011-es kiadásán tűzték ki, és a szerző megoldása négydimenziós tömböt használt a részfeladatok optimumainak tárolására.

Mozaik: Legyen egy $n \times m$ méretű téglalap alakú mozaikkép, amelyet az $a[1..n][1..m]$ tömb elemei kódolnak ($1 \le n,m \le 32$). Az (i,j) pozíciójú tömbelem az (i,j) pozíciójú képelem színkódját jelenti (0..255). A mozaikképet egy golyósorozat éri, amely téglalap alakú darabokra töri (lásd a 7.11. ábrát). Minimum hány, középpontjaikra nézve szimmetrikus színösszetételű téglalapra (középpontra szimmetrikus pozíciójú képelem-párok színe azonos) darabolható a kép?

Megjegyzés: Ha a golyó a kurrens téglalap szélét találja el (első lövés a példában), akkor is tekinthető úgy, hogy a téglalap négy darabra esik szét, amelyből kettő nulla területű (egyik oldalhosszuk nulla). Elég csak a felső és a bal oldali széleket tekinteni, mert a szemközti oldalakat ért találatok azonos darabolásokat eredményeznének. A sarok-találatok kizárandók, mert nem eredményeznek darabolást.

1. Általános alak: (i,j,li,lj)-feladat, azaz az (i,j) sarkú, li és lj oldalhosszú téglalap optimális feldarabolása.
   Méretükben is triviális részfeladatok: az $(i,j,1,1)$ alakú részfeladatok (amelyek 1×1-es méretű résztéglalapokra vonatkoznak); azok is méretükben triviális részfeladatokként kezelendők, amelyek megfelelő téglalapok egyik oldalhossza 0.
   Minőségükben triviális részfeladatok: azok a nem triviális méretűek, amelyek önmagukban szimmetrikus színösszetételű résztéglalapokra vonatkoznak.
   Eredeti feladat: az $(1,1,n,m)$-feladat (az (1,1) sarkú $n \times m$ oldalhosszú téglalapra vonatkozó).
2. Optimumértékek tömbje: $c[1..n][1..m][0..n][0..m]$ (A $c[i][j][li][lj]$ cella azt tárolja, hogy minimum hány lövésből bontható fel az (i,j) sarkú, li és lj oldalhosszú téglalap szimmetrikus résztéglalapokra).
   Méretükben triviális részfeladatok optimumai: a $c[i][j][1][1]$, $c[i][j][0][lj]$, $c[i][j][li][0]$ alakú cellák tárolják.
   Eredeti feladat optimumértéke: a $c[1][1][n][m]$ cella tárolja.
3. Rekurzív képlet: az (i,j,li,lj) téglalapot $li \cdot lj - 1$ helyen érheti érvényes lövés. A 7.12. ábra azt ábrázolja, hogy egy (pi,pj) pozíciói találatból ($pi=0,li-1$; $pj=0,lj-1$) mely résztéglalapok adódnak (a $pi=pj=0$ bal felső sarki találat kizárandó; lásd fentebb).

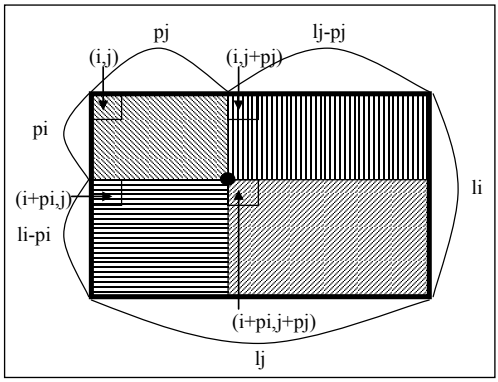
$c[i][j][0][lj] = 0$, $1 \le i \le n$, $1 \le j \le m$, $0 \le lj \le m-j+1$
$c[i][j][li][0] = 0$, $1 \le i \le n$, $1 \le j \le m$, $0 \le li \le n-i+1$
$c[i][j][1][1] = 0$, $1 \le i \le n$, $1 \le j \le m$
Továbbá, bármely $1 \le i \le n$, $1 \le j \le m$, $1 \le li \le n-i+1$, $1 \le lj \le m-j+1$ esetén
ha a megfelelő téglalap szimmetrikus, akkor
$c[i][j][li][lj] = 0$
különben, minden $pi=0,li-1$; $pj=0,lj-1$ esetén,
$c[i][j][li][lj] = 1 + \min\{c[i][j][pi][pj] + c[i+pi][j][li-pi][pj] + c[i][j+pj][pi][lj-pj] + c[i+pi][j+pj][li-pi][lj-pj]\}$



7.11. ábra. *Egy példa mátrix minimális számú szimmetrikus elemre való szétdarabolása 3 lépésben*

4. Iteratív algoritmus a c és b tömbök feltöltésére. A b bejegyzés típusú tömb az optimális döntéseket tárolja (a -1 érték azt jelzi, hogy a megfelelő darab szimmetrikus). Ha gondoskodunk róla, hogy a c tömb elemei 0 kezdőértékeket kapjanak, akkor a méretükben triviális részfeladatok optimumai implicit meglesznek.

```
// minden téglalapra méretük szerint növekvő sorrendben (li, lj, i, j ciklusok)
// a kurrens téglalapra (pi és pj ciklusok)
   // bevesszük a felső szélt, de kizárjuk az alsót (pi = 0,li−1)
   // bevesszük a bal szélt, de kizárjuk a jobbat (pj = 0,lj−1)
   // kizárjuk a bal felső sarkot (pi > 0 vagy pj > 0)
```



7.12. ábra. *Az (i,j,li,lj) téglalap egy lehetséges felbontása*

```
minden li = 1,n végezd
   minden lj = 1,m végezd
      minden i = 1,n−li+1 végezd
         minden j = 1,m−lj+1 végezd
            ha szimmetrikus(a,i,j,li,lj) akkor                    // triviális eset
               c[i][j][li][lj] = 0;
               b[i][j][li][lj].pi = −1
               b[i][j][li][lj].pj = −1
            különben          // kurrens téglalap lehetséges feldarabolásai egy lövéssel
               min = n * m
               minden pi = 0,li−1 végezd
                  minden pj = 0,lj−1 végezd
                     ha pi > 0 vagy pj > 0 akkor
                        x = c[i][j][pi][pj] + c[i+pi][j][li−pi][pj] +
                            c[i][j+pj][pi][lj−pj] + c[i+pi][j+pj][li−pi][lj−pj]
                        ha x < min akkor
                           min = x
                           mpi = pi
                           mpj = pj
                        vége ha
                     vége ha
                  vége minden
               vége minden
               c[i][j][li][lj] = min + 1
               b[i][j][li][lj].pi = mpi
               b[i][j][li][lj].pj = mpj
            vége ha
```

```
        vége minden
      vége minden
    vége minden
  vége minden
  ki: c[1][1][n][m]
```

5. Az optimális felbontás kiíratása maradjon az olvasóra.

Egy másik sajátossága az előbbiekben tárgyalt algoritmusoknak, hogy az optimalizálás vagy minimum, vagy maximum számítást foglalt magába. A következő példában egy időben szerepel mindkét optimalizálási függvény.

Kétszemélyes játék: Legyen egy n elemű (n páros) természetes számokat tartalmazó sorozat ($a[1..n]$). A játékosok (piros és kék) felváltva választanak egy-egy elemet a számsor valamelyik végéről. Mennyi a maximális összeg, amit a kezdő garantáltan összeszedhet? (A 7.13. ábra egy példa-számsorozatot mutat be.)

Megjegyzés: Az, hogy garantált összeg, azt jelenti, hogy a második játékos is mindig jól választ, persze a kezdő által diktált keretek között.

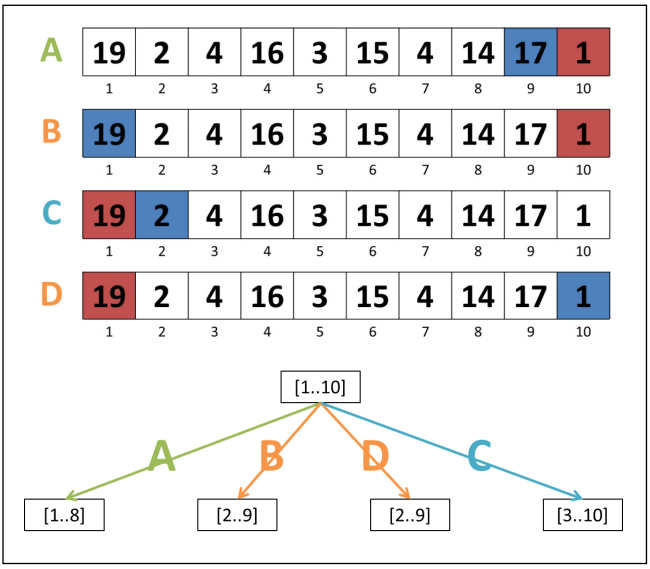
19	2	4	16	3	15	4	14	17	1
1	2	3	4	5	6	7	8	9	10

7.13. ábra. *Példaszámsorozat a kétszemélyes játék nevű feladathoz*

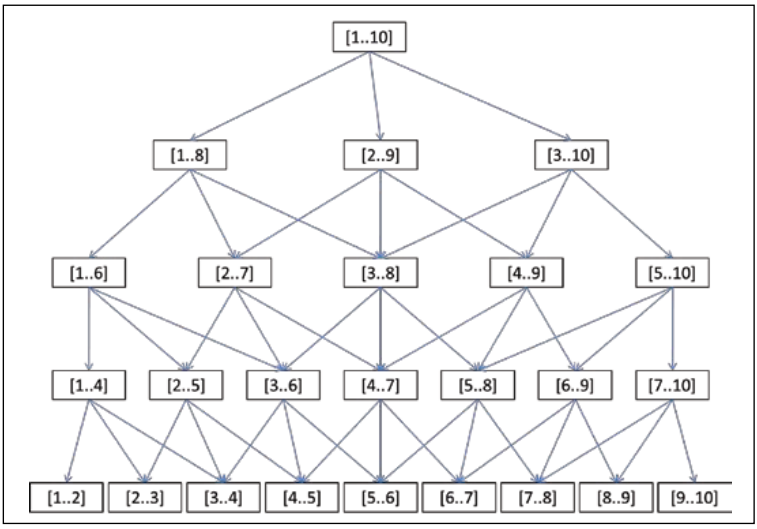
A 7.14. ábra négy lehetséges forgatókönyvet (A, B, C és D) mutat be az első fordulóhoz, illetve azt, hogy mely részfeladatokra redukálódhat a feladat az első forduló nyomán. Máris érzékelhető, hogy a B és D variánsok ugyanahhoz a részfeladathoz vezetnek. A 7.15. ábra a teljes „döntési fát” eleveníti meg (a példa-számsorozatra vonatkoztatva), miután egymásra csúsztattuk az identikus részfeladatokat. Minden fordulóval kettővel rövidül a számsorozat. A fa egy adott szintjén a csomópontok a megfelelő páros hosszúságú tömbszakaszokra vonatkozó játékokat képviselik.

Az egyszerűtől a bonyolult felé haladás azt jelenti, hogy lentről felfele, szintről szintre haladunk a fában, és megoldjuk a feladatot minden 2-hosszú, 4-hosszú stb. szakaszra, és végül a teljes számsorozatra. A részfeladatok általános alakja nyilván az, hogy mennyi a maximális összeg, amelyet az első játékos garantáltan össze tud gyűjteni az $a[i..j]$ szakaszon ($j-i+1$ páros), ha ő következik (7.16. ábra).

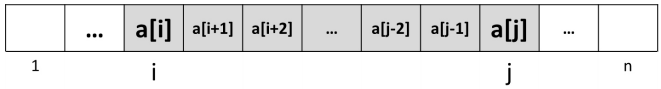
Mi a dinamikus programozás szerinti gondolkodás az első játékos (piros) szemszögéből a kurrens fordulóhoz?



7.14. ábra. *(a) A négy lehetséges forgatókönyv az első fordulóhoz (A: mindkét játékos jobbról választ; B: piros jobbról, kék balról választ; C: mindkét játékos balról választ; D: piros balról, kék jobbról választ). (b) Az első fordulói lehetséges választásokból adódó részfeladatok*



7.15. ábra. *Az összevont döntési fa*



7.16. ábra. *Az „általános részfeladat”*

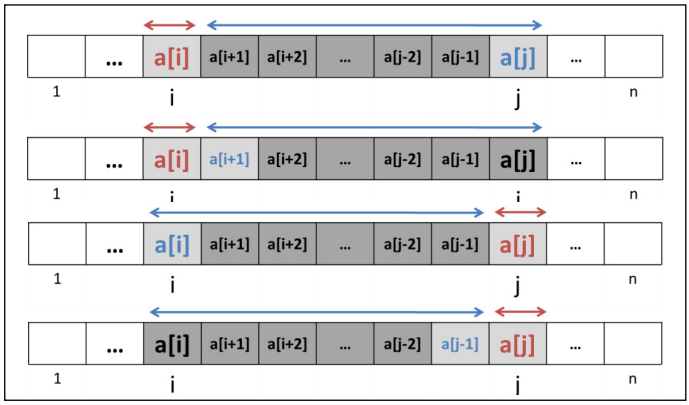
– Két út áll előttem ($a[i]$ vagy $a[j]$), melyiket válasszam? Nyilván a számomra előnyösebbet. Ezt tükrözi a képlet maximum függvénye.
– Ha a piros $a[i]$-t választja, akkor két út áll a kék előtt ($a[i+1]$ vagy $a[j]$), melyiket válassza? Nyilván azt, amelyik olyan szakaszra vezeti vissza a feladatot, ahonnan a piros kevesebbet tud kinyerni.

Ha a piros $a[j]$-t választja, akkor másik két út áll a kék előtt ($a[i]$ vagy $a[j-1]$), melyiket válassza? Ez esetben is nyilván azt, amelyik olyan szakaszra vezeti vissza a feladatot, ahonnan a piros kevesebbet tud kinyerni.

A kék játékos pirosra nézve előnytelen választásait tükrözik a képlet minimum függvényei.

A 7.17. ábra e forgatókönyvet illusztrálja. Az ebből adódó rekurzív képlet (amennyiben a $c[1..n][1..n]$ tömb tárolja az optimumértékeket), az első játékos szemszögéből megfogalmazva, a következő (a $c[i][j]$ tömbelem azt tárolja, hogy mennyi a maximális összeg, amit a piros játékos garantáltan meg tud szerezni az $a[i..j]$ szakaszon):

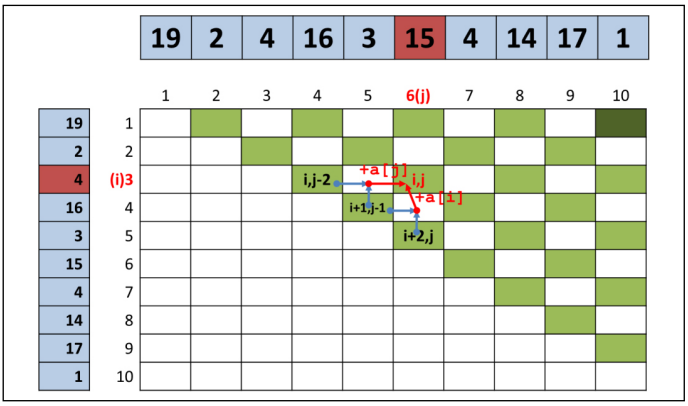
$$c[i][j] = \max\{a[i] + \min\{c[i+1][j-1], c[i+2][j]\}, a[j] + \min\{c[i+1][j-1], c[i][j-2]\}\}$$



7.17. ábra. *Az a[i..j] szakaszra vonatkozó kurrens forduló*

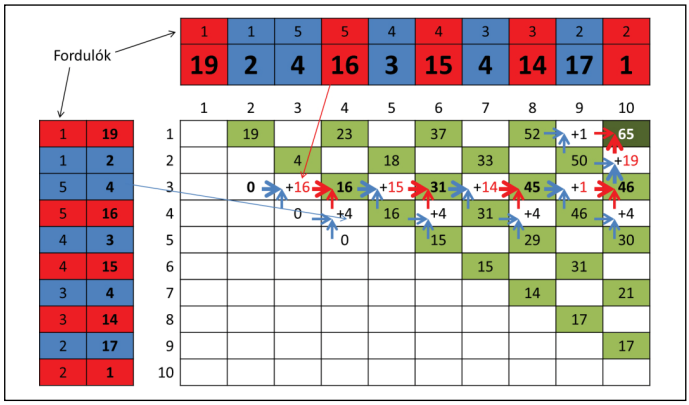
A 7.18. ábra azt mutatja be, hogy miként tölthető fel a c tömb, átlóról átlóra, a fenti képlet alapján. A megfelelő algoritmus pedig a következő:

```
// a 0-hosszú szakaszokat a főátló alatti átló elemei képviselik (értékük 0)
minden k = 1,n−1,2 végezd                                   //átlóról, átlóra
   i = 1
   j = i+k                                  // a k. átló, a k+1 hosszú szakaszokat képviseli
   amíg j ≤ n végezd                                        // a kurrens átló mentén
      c[i][j] = max(a[i] + min(c[i+1][j−1],c[i+2][j]),
      a[j] + min(c[i+1][j−1],c[i][j−2]))
      i = i+1
      j = j+1
   vége amíg
vége minden
ki: c[1][n]
```

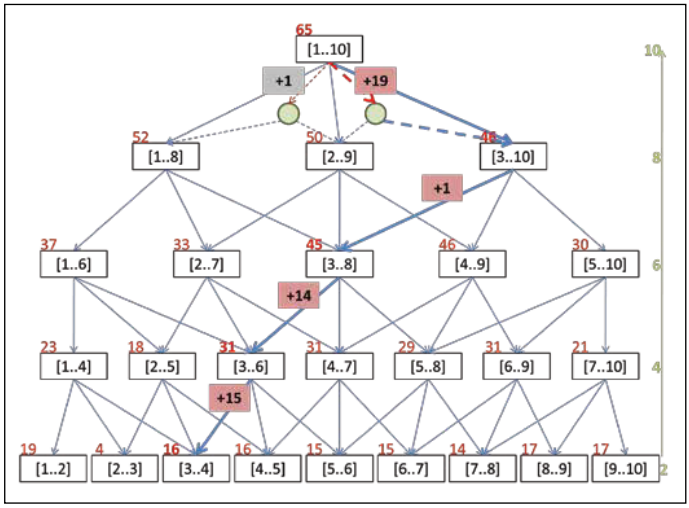


7.18. ábra. *A c tömb feltöltése, átlóról átlóra (csak a páros hosszú szakaszoknak megfelelő átlókban vagyunk érdekeltek). Bejelöltük, hogy egy kurrens (i,j) cellához mint apacellához képest hol helyezkednek el a közvetlen fiúcellái. Az eredeti feladatot képviselő cella a c[1][n]. A 2-hosszú szakaszoknak megfelelő részfeladatokat képviselő cellák a főátlót követő átlón helyezkednek el*

A feltöltött *c* tömbből, vissza irányban, kiolvasható az optimális döntéssorozat. Ezt szemléltetik a 7.19. és a 7.20. ábrák. Az egyik a tömbön, a másik pedig az összevont döntési fán rekonstruálja az „optimális játékot”. A *c* tömb alapján akár a kétszemélyes játék is implementálható, ahol a piros játékos a számítógép, a kék pedig a felhasználó. A gép a *c* tömb alapján lép, a felhasználó pedig a billentyűzetről írhatja be a választását. Ennek megvalósítása maradjon az olvasóra.



7.19. ábra. *A példa-számsorozatra feltöltött optimumtömb és az optimális döntéssorozat (az optimális választásokat képviselő nyilakat megvastagítottuk). Az első fordulóban mindketten balról választanak, majd újra és újra, mindketten jobbról*



7.20. ábra. *Az „optimális játék” a döntési fán. Az optimálisdöntés-sorozatot képviselő gyökér–levél utat kiemeltük. Ha a kék játékos másik ágra terelné a játékot, akkor a piros még nagyobb összeget is összegyűjthet, mint a garantált optimum. Mindenik csomópont felett megadtuk, hogy mekkora a garantált maximum, amit a piros ki tud nyerni a megfelelő számsorszakaszon (A gyökér esetében, a beékelt körök révén, explicite ábrázoltuk, hogy a szintek között kétlépéses „max-min döntésekre” kerül sor)*

7.5. Egyszerűtől a bonyolult felé: variációk egy témára

Amint már említettük, a dinamikus programozás az egyszerűtől a bonyolult felé, lentről felfele irányba oldja meg a feladat lebontásából adódó részfeladatokat. A megoldást *felépítjük*, lépésről lépésre, szintről szintre. Ez az „egyszerűtől bonyolult felé" motívum azonban nagyon sokszínűen jelentkezhet a dinamikus programozásos feladatok kapcsán. Ezért szerepel ennek az alfejezetnek a címében a „variációk egy témára" kifejezés.

7.5.1. Értékről értékre

Az egyszerűtől a bonyolult felé jelentheti azt, hogy „értékről értékre". Szolgáljon szemléltetésül az alábbi feladat.

Pénzérmék: Adott n-féle pénzérme, melyek értékét a $w[1\ldots n]$ tömb tárolja, valamint egy S összeg. Fizessük ki az S összeget minimális pénzérme felhasználásával (bármelyikből bármennyi felhasználható).

Példa:

INPUT: $S=11$, $n=3$, $w[1..3] = \{2,3,5\}$

OUPTUP: $11 = 3+3+5$

Megoldás: Bár az S összeg kifizetésében vagyunk érdekeltek, megoldjuk a feladatot az összes $i = 0, 1, \ldots, S$ összeg-értékre (értékről értékre haladunk) (7.21. ábra).

Összeg	Min-érmeszám	Felhasznált érmék
0	**0**	-
1	**∞**	-
2	**1**	**2**
3	**1**	**3**
4	**2**	**2, 2**
5	**1**	**5**
6	**2**	**3, 3**
7	**2**	**2, 5**
8	**2**	**3, 5**
9	**3**	**2, 2, 5**
10	**2**	**5, 5**
11	**3**	**3, 3, 5**

7.21. ábra. *A pénzérmék feladat optimális megoldásai (minimális érmeszám és a megfelelő érmék) 0, 1, …, 11 összegekre*

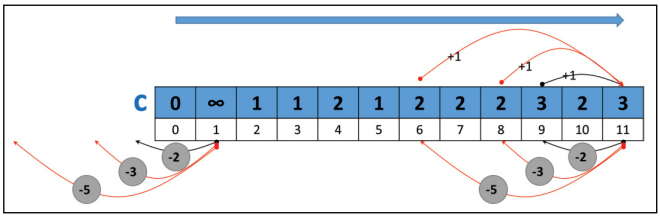
Az érmék optimális számát a $c[0..S]$ tömbben tároljuk. Adott i-re az érmék minimális számát a $c[i]$ cella tárolja. Kurrens i-re a $c[i]$-t, a már rendelkezésre álló $c[j]$ értékekből ($j<i$) számítjuk ki.

Amint fennebb láttuk, a lentről felfele építkezés egy rekurzív képlet révén történik, amely optimumról optimumra vezet. Az $i=0$ esetre az érmék minimális száma nyilván 0 ($c[0] = 0$). A képletre rávezethet a következő kérdés: a kurrens i összeg ($i=1..S$) mely j összegekből ($j<i$) építhető fel *egy* pénzérme hozzáadásával? Mivel ez az egy pénzérme n-féle lehet, íme a képlet (lásd a 7.22–23. ábrákat):

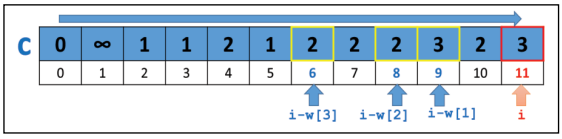
$c[i] = \min\{c[i-w[k]]+1$; minden $k=1..n$-re, ahol $w[k] <= i\}$

A képlet implementálása az alábbi algoritmushoz vezet:

```
c[0] = 0
minden i = 1,S végezd
   c[i] = ∞
   minden k = 1,n végezd
      ha i−w[k] ≥ 0 és c[i−w[k]] < c[i] akkor
         c[i] = c[i−w[k]]
      vége ha
   vége minden
   c[i] = c[i]+1
vége minden
ki: c[S]
```



7.22. ábra. *Adott összegnek ($i=1$ és $i=11$-re) megfelelő apafeladat közvetlen fiúrészfeladatai*

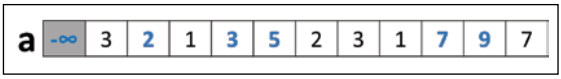


7.23. ábra. *Kurrens i-re ($i=11$), az apa- (piros kerettel) és fiúrészfeladatokat (sárga kerettel) képviselő cellák*

7.5.2. Szuffixről szuffixre

Az egyszerűtől a bonyolult felé időnként azt jelentheti, hogy a feladatot például egy számsorozat egyre hosszabb szuffix-részsorozataira oldjuk meg. Íme egy példa.

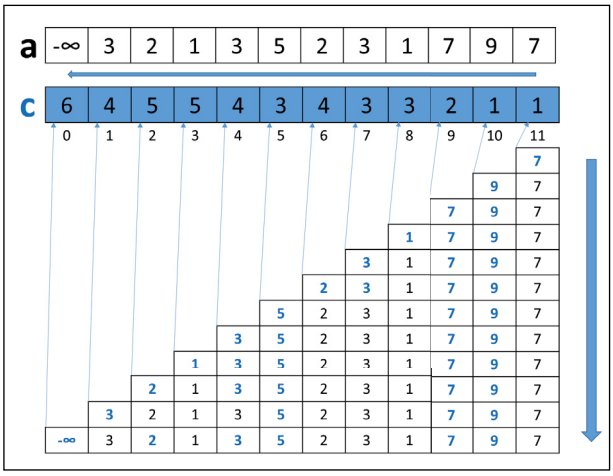
Leghosszabb növekvő részsorozat: Adott az $a[1..n]$ számsorozat. Határozzuk meg a leghosszabb szigorúan növekvő részsorozatát (a 7.24. ábra egy példát mutat be).



7.24. ábra. *Az a[0] cellába eltároltunk egy fiktív -∞ értéket, hogy a leghosszabb növekvő részsorozat (kékkel jelöltük) garantáltan ezzel kezdődjön.*

Megoldás: Sorra meghatározzuk az $a[n]$, $a[n-1]$, …, $a[1]$, és végül az $a[0]$ elemekkel kezdődő leghosszabb növekvő részsorozatokat (szuffixről szuffixre haladunk a tömbben: $a[n..n]$, $a[n-1..n]$, …, $a[1..n]$, $a[0..n]$). Az optimumértékeket a $c[0..n]$ tömbben tároljuk, azaz $c[i]$ tárolja az $a[i]$ elemmel kezdődő leghosszabb részsorozat hosszát ($i=0..n$). A „lentről felfele” ez esetben azt jelenti, hogy $c[n]$, $c[n-1]$, …, $c[1]$, $c[0]$ sorrendben számoljuk ki az optimumokat (7.25. ábra).

A $c[n]$ cella értéke nyilván implicite 1. A c[i] elemnek mint kurrens „apacellának” melyek a közvetlen „fiúcellái”? Mivel szigorúan növekvő, nem föltétlenül egymás utáni elemekből álló részsorozatokban vagyunk érdekeltek, ezért nyilván



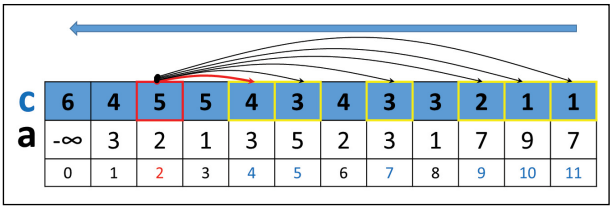
7.25. ábra. *Szuffixről szuffixre haladunk. Kékkel jelöltük a c tömbben tárolt optimumértékeknek megfelelő leghosszabb növekvő részsorozatokat*

azon c[j] cellák (j=i+1..n), amelyekre a[j] > a[i] (7.26. ábra). Íme a képlet minden i-re ($n-1$)-től 0-ig:

$c[i] = 1 + \max\{ c[j]$, ahol $j=i+1..n$ és $a[j] > a[i] \}$

A képlet implementálása az alábbi algoritmushoz vezet:

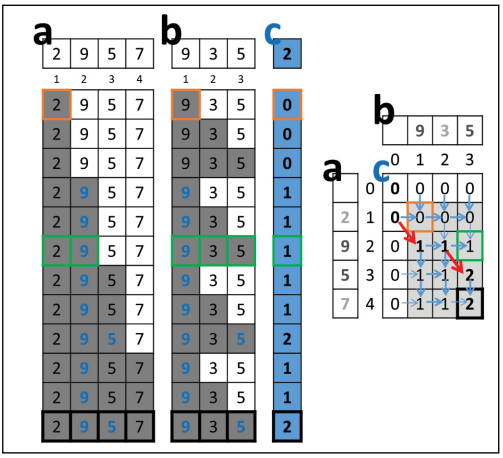
```
c[n] = 1
minden i = n−1,0 végezd
   c[i] = 1
   minden j = i+1,n végezd
      ha a[i] < a[j] és c[j]+1 > c[i] akkor
         c[i] = c[j]+1
      vége ha
   vége minden
vége minden
ki: c[0]−1
```



7.26. ábra. *Kurrens i-re (i=2), az apa- (piros kerettel) és fiúrészfeladatokat (sárga kerettel) képviselő cellák*

7.5.3. Prefixpárról prefixpárra

Ebbe a kategóriába tartozik például a *leghosszabb közös részsorozat* nevű feladat, amelyet fentebb már tárgyaltunk (7.3. alfejezet, második példa). Itt csak azt hangsúlyozzuk, hogy milyen értelemben halad a dinamikus programozásos algoritmus prefixpárról prefixpárra. A 7.27. ábra azt emeli ki, hogy a c tömb soronkénti bejárása milyen értelemben jelenti a feladat egyre hosszabb prefixpárokra való megoldását.



7.27. ábra. *Prefixpárról prefixpárra (vesd össze a 7.5. ábrával)*

7.5.4. Szakaszról szakaszra

Az alábbi megoldott feladat azt szemlélteti, hogy miként jelentheti az egyszerűtől a bonyolult felé haladás azt, hogy a feladatot egy sorozat egyre hosszabb szakaszaira oldjuk meg.

Leghosszabb palindrom részsorozat: Határozzuk meg az $a[1..n]$ számsorozat/karakterlánc leghosszabb palindrom részsorozatának hosszát (esetleg ezek számát is).

Példa

INPUT: $a[1..5] = \{2, 1, 4, 2, 2\}$

OUTPUT: 3

Öt darab 3-hosszú megoldás van: (2,1,2); (2,1,2); (2,4,2); (2,4,2); (2,2,2)

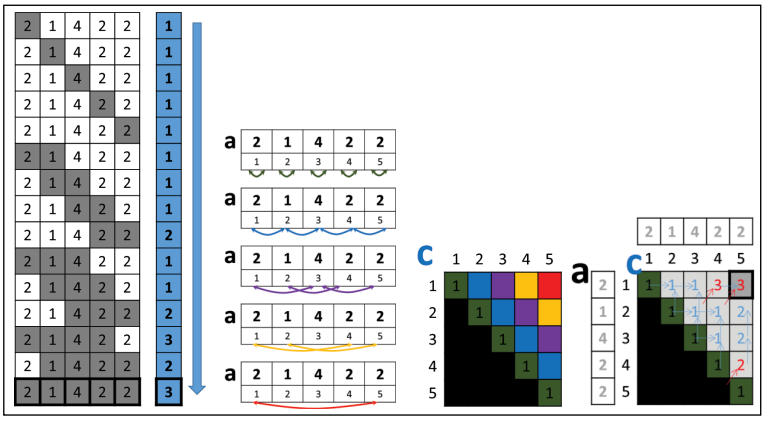
Megoldás: Sorra meghatározzuk az 1, 2, …, n hosszú szakaszok tartalmazta leghosszabb palindrom hosszát (n darab 1-hosszú, $(n-1)$ darab 2-hosszú, …1 darab n-hosszú szakasz van; lásd 7.29. ábrát). Tárolja a $c[i][j]$ tömbelem az $a[i..j]$ szakasz leghosszabb palindromjának hosszát ($i \le j$).

Az 1-hosszú szakaszoknak megfelelő $c[i][i]$ cellák értékei nyilván 1-gyel lesznek egyenlők minden $i=1..n$ esetén. Melyek a $c[i][j]$ kurrens apacella közvetlen fiúcellái $j > i$ esetben? Ha a megfelelő $a[i]$ és $a[j]$ elemek egyenlők, akkor belekerülnek a leghosszabb palindromba, és a megfelelő fiúcella a $c[i+1][j-1]$. Ha $a[i] \neq a[j]$, akkor valamelyik kerülhet bele a leghosszabb palindromba, és a megfelelő fiúcellák a $c[i][j-1]$ és a $c[i+1][j]$. A mindezekből adódó képlet rekurzív ágai a következők:

$c[i][j] = 2 + c[i+1][j-1]$, ha $a[i] == a[j]$
$c[i][j] = \max\{ c[i][j-1], c[i+1][j] \}$, ha $a[i] \neq a[j]$

A szakaszról szakaszra haladás legtermészetesebb implementálása, ha átlóról átlóra haladunk (h ciklusváltozó) a c tömbben (7.28. ábra).

```
minden h = 1,n végezd
   minden i = 1,n–h+1 végezd
      j = i+h−1;
      ha i == j akkor
         c[i][i] = 1
      különben
         ha a[i] == a[j] akkor
            c[i][j] = c[i+1][j−1] + 2
         különben
            ha c[i][j−1] > c[i+1][j] akkor
               c[i][j] = c[i][j−1]
            különben
               c[i][j] = c[i+1][j]
            vége ha
         vége ha
      vége ha
   vége minden
vége minden
```

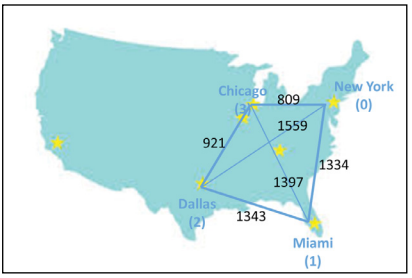


7.28. ábra. *Szakaszról szakaszra*

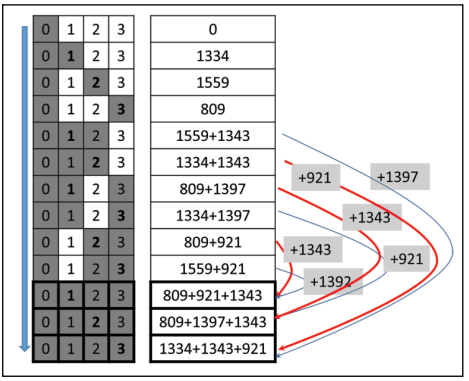
7.5.5. Részhalmazról részhalmazra

Talán a legösszetettebb eset, amikor részhalmazról részhalmazra haladva kell hogy megoldjuk a részfeladatokat. Szemléltessük ezt a híres utazóügynök-problémával, amelyet részletesen tárgyalunk majd a 8. fejezetben.

Utazóügynök: Legyen n város (0,1,…,n−1). Az a[0..n−1][0..n−1] mátrix valamely $a[i][j]$ eleme azt tárolja, hogy mekkora az i és j városok között a közvetlen távolság. Határozzuk meg a legrövidebb Hamilton-kör hosszát (érinti az *összes* várost egyszer és csakis egyszer)! (A 7.29. ábra egy példát mutat be.)



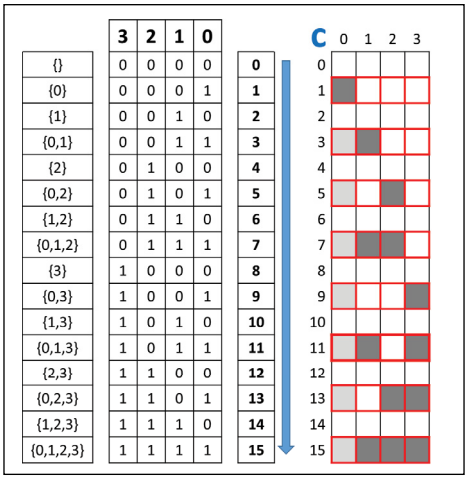
7.29. ábra. *Példa négy városra (megvastagítottuk a legrövidebb Hamilton-kört)*



7.30. ábra. *Részhalmazról részhalmazra (besatíroztuk a megfelelő elemeket; bal oldali oszlop). Mindenik részhalmaz annyiszor szerepel a sorban, ahány 0-tól különböző eleme van. A jobb oldali oszlop a legrövidebb úthosszakat tartalmazza (ahogy lépésről lépésre felépülnek). A bal oldali oszlopban megvastagítottuk azt az elemet, amelyhez vezető legrövidebb úthosszra (az összes többi érintésével) vonatkozik a jobb oldali optimum. A 4-elemű halmazokat képviselő apacellákra megjelöltük a megfelelő fiúcellákat (piros nyilak ábrázolják az optimális fiakat)*

Megoldás: Jelölje minden $i = 1..n-1$ esetre $h[i]$ a minimális út hosszát 0-ból i-be, a *teljes városhalmaz érintésével*. A keresett optimumot nyilván a következő képlet adja meg: $\min\{h[i] + a[i][0]$, ahol $i = 1..n-1\}$. Az egyszerűtől a bonyolult felé haladás ez esetben azt jelenti, hogy meghatározzuk az összes részhalmazra, amely tartalmazza a 0-t, hogy mennyi a minimális út hossza a részhalmaz egyes elemeihez, a *részhalmaz összes elemének érintésével* (lásd a 7.30. ábrát).

Implementációs szempontból az jelenthet kihívást, hogy miként tároljunk a részhalmazokra vonatkozó optimumokat. Elsőre furcsán hangozhat, hogy egy tömb adott „részhalmazadik" eleme. Hogyan lehet egy tömb indexe halmaz? Például úgy, hogy a tömbindexekhez, egészen pontosan ezek bináris alakjához részhalmazokat társítunk (a $0..2^n-1$ számsor n biten történő bináris ábrázolása egy-egy kapcsolatba hozható bármely n elemű halmaz 2^n-ediken részhalmazával; lásd a 7.31. ábrát). További részletek végett az olvasó figyelmébe ajánljuk a 8. alfejezetet.

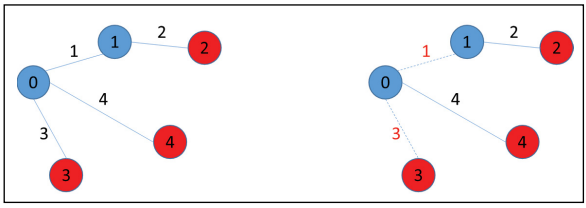


7.31. ábra. *A $c[0..2^n-1][0..n-1]$ tömböt használjuk az optimumok tárolására (balra részleteztük, hogy a $0..2^4-1$ indexsornak mely részhalmazsor felel meg, és miért). A c tömb csak azon sorait használjuk, amelyek olyan részhalmazokat képviselnek, amelyek tartalmazzák a 0 elemet (pirossal bekereteztük). Minden részhalmaz kapcsán annyi optimumot kell hogy tároljunk, ahány nullától különböző eleme van (ezért kétdimenziós a c tömb). Erre a kurrens sor azon celláit használjuk, amelyek oszlopindexe éppen a megfelelő részhalmazelem (sötétszürkével jeleztük). A c tömb felhasznált celláinak száma megegyezik a 7.31. ábrán megjelenített tömb sorainak számával*

7.5.6. Részfáról részfára

Utolsó példaként szolgáljon az az eset, amikor az egyszerűtől a bonyolult felé azt jelenti, hogy egyre nagyobb részfákra oldjuk meg a feladatot. Általában mélységi bejárást alkalmazunk, és a mélységi fa (mint gyökeres fa) csomópontjai képviselte részfeladatokat postorder momentumokban oldjuk meg (egy adott csomópont képviselte részfeladatot az illető csomópont gyökerű részfa ábrázolja).

Izoláld a gócpontokat: Legyen egy n pontú fa (körmentes, összefüggő gráf), amelynek csomópontjai vagy kékek (tiszta), vagy pirosak (fertőzött). Ismerve minden kapcsolat felszámolásának költségét, izoláld egymás között a gócpontokat (pirosak), minimális költséggel (a 7.32. ábra egy példát mutat be).



7.32. ábra. *Példa 5 pontú fára. Pirossal jelöltük azon élek költségét, amelyek felszámolása jelenti az optimális megoldást*

Ötlet: A mélységi fa részfáit postorder sorrendben tekintjük, és minden részfára meghatározzuk a minimális költségű megoldást. A kurrens részfára vonatkozó optimumot a már rendelkezésre álló fiúrészfák optimumaiból építjük fel. Ahogy az előző példában minden részhalmaz kapcsán annyi optimumot kellett eltároljunk, ahány nullától különböző eleme volt, úgy e feladat esetében is célszerű minden részfa kapcsán két optimumot eltárolni. Mi az optimum arra az esetre vonatkozóan, ha az illető részfa gyökere tiszta/fertőzött komponensbe kerül? A további implementációs részleteket az olvasóra hagyjuk.

8. ÖSSZKÉP: AZ UTAZÓ ÜGYNÖK PROBLÉMÁJA

Az előző fejezetekben már számos felülnézettel találkozhatott az olvasó. Például a 6. fejezetben az egér–sajt feladatot megoldottuk backtracking, oszd-meg-és-uralkodj, valamint DF és BF alapú branch-and-bound stratégiákkal is. Sőt utaltunk egy lehetséges mohó megközelítésre is. A 7. fejezetből kiderült, hogy az a megoldás, amit BF-branch-and-bound módszernek neveztünk, dinamikus programozásnak is tekinthető, hiszen a már elért cellákhoz tartozó optimumokból építi fel a szomszédos cellák optimumait. Erre az algoritmusra gyakran utalnak úgy is, mint Lee algoritmusa (Lee 1961).

Egy másik példa a Pénzérmék feladat, amely egy adott összeg optimális kifizetéséről szól, adott értékű pénzérmékkel. Az 5. fejezet bemutatott egy mohó megközelítést, amely általános esetben nem vezet optimális megoldáshoz (ha a pénzérmék értékei k^0, k^1, k^2, … alakúak, akkor bizonyítható, hogy a mohó stratégiai célba vezet). A 7. fejezet dinamikus programozásos algoritmust adott az érmék minimális számának meghatározására. Az *Algoritmusok felülnézetből* jegyzet (Kátai 2007) 2. fejezete backtracking algoritmussal generálja az összes megoldást, a 6. fejezete pedig ötleteket ad ahhoz, hogy miként lehetne hatékonyabbá tenni a backtracking és mohó megoldásokat e két módszer kombinálása révén.

És megemlíthetjük a híres hátizsákproblémát is. A 6. fejezet bemutatta, hogy mi lenne a mohó és backtracking megoldás a feladatra, illetve hogy miként lehet kombinálni e módszereket branch-and-bound szellemben. Az *Algoritmusok felülnézetből* című jegyzet (Kátai 2007) 9. fejezete váll váll mellett mutatja be a mohó és dinamikus programozásos megoldásokat e feladatra.

Az alábbiakban a 7.6.5. alfejezetben bemutatott utazóügynök-problémára alapozva igyekszünk összképet nyújtani az előző fejezetekben tárgyalt algoritmustervezési stratégiákról.

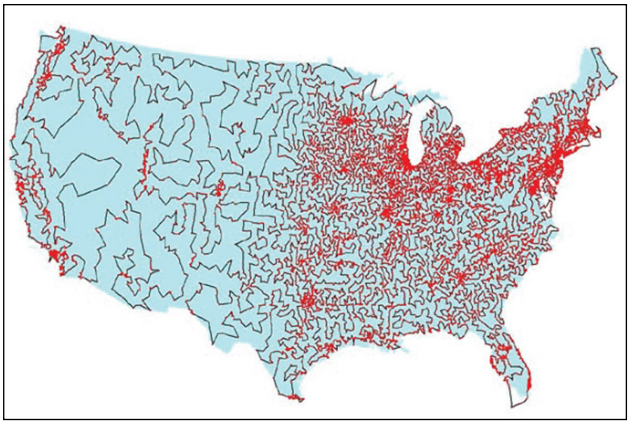
8.1. Az utazó ügynök problémája

Utazóügynök: Adva van n város, illetve az útiköltség bármely két város között, keressük a legolcsóbb utat egy adott városból indulva, amely minden várost pontosan egyszer érint, majd a kiindulási városba ér vissza (lásd a 8.1. ábrát, amely 13 509 egyesült államokbeli városra adja meg az optimális Hamilton-kört).

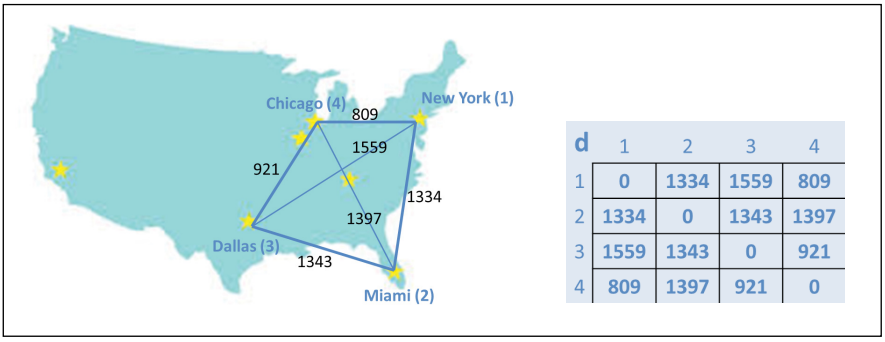
A továbbiakban a városokat az 1, 2, …, n számokkal azonosítjuk, a direkt utak költségeit pedig a $d[1..n][1..n]$ tömb tárolja (8.2.a. ábra).

Ha leszögezzük, hogy az 1-es városból indulunk, akkor ezt a többi város $(n-1)!$ különböző sorrendben követheti. Egy n szintes fa elevenedik meg előttünk,

amelynek gyökerében az 1-es város található, és az n. szintjén $(n-1)!$ levele van (8.2.b. ábra). Ha kizárjuk, hogy ugyanazon körutat mindkét irányból tekintsük, akkor $((n-1)!)/2$ levél (ez éppen a Hamilton-körök száma) közül vagyunk érdekeltek az optimálisban, pontosabban az ehhez vezető optimális gyökér–levél útban.

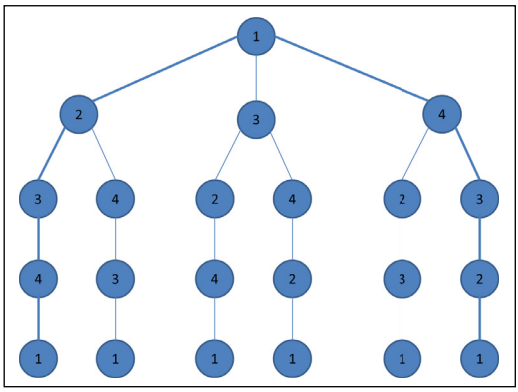


8.1. ábra. *Optimális utazóügynök-útvonal 13 509 egyesült államokbeli városra (CRPC 1998)*



d	1	2	3	4
1	0	1334	1559	809
2	1334	0	1343	1397
3	1559	1343	0	921
4	809	1397	921	0

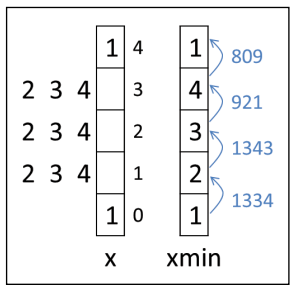
8.2. ábra. *(a) Szemléltető példa n=4 esetre (d: a megfelelő költségmátrix)*



8.2. ábra. *(b) A keresési tér, az n=4 esetre. Megvastagítottuk az optimális Hamilton-kört képviselő gyökér–levél utakat*

8.1.1. Backtracking megközelítés

A backtracking megközelítés ez esetben nyers erő módszernek számít: generáljuk az összes Hamilton-kört (azaz az összes gyökér–levél utat a 8.2.b. ábrán bemutatott fában), és kiválasztjuk közülük a legjobbat. Az alábbi algoritmus e stratégiát követi (lásd a 3. fejezet BTp eljárását is). A kiír eljárás helyén minimumszámolás zajlik. A min_költség paraméter az addig talált, per pillanat legjobb körút költségét tárolja (kezdetben ∞ az értéke), az xmin tömb pedig a kurrens legjobb utat (a frissít_min_ut eljárás másolatot készít xmin-ben, x-ről). Továbbá minden csomópontra nyilvántartjuk a gyökérből odavezető kurrens út költségét (kur_költség). Az *x* tömb 0. és *n*. pozícióiban lerögzítettük az 1-es várost (8.3. ábra).



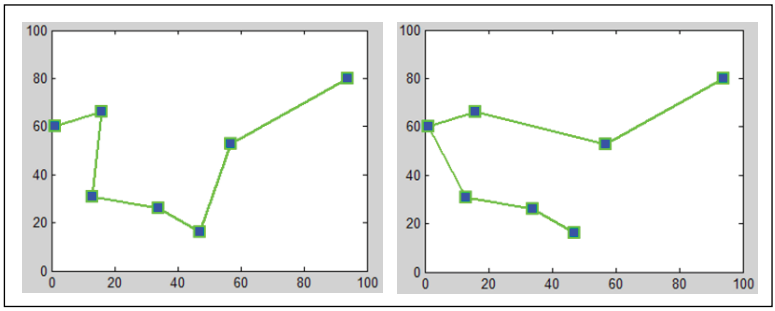
8.3. ábra. *Backtracking modell az utazó ügynök feladatra (n=4). Az x tömb 1, 2, …, (n-1) szintjein generáljuk a 2, 3, …, n számsor összes permutációját. Az xmin tömb a legjobb Hamilton-kört mutatja be a megadott példára*

```
BTp(x[],n,xmin[],min_költség,kur_költség,k)
   minden x[k] = 2,n végezd
      ha ígéretes(x,k) akkor
         kur_költség = kur_költség + d[x[k−1]][x[k]]
         ha k < n−1 akkor
            BTp(x, n,xmin,min_költség,kur_költség,k+1)
         különben
            ha kur_költség + d[x[k]][1] < min_költség akkor
               min_költség = kur_költség + d[x[k]][1]
               frissít_min_ut(x,xmin,n)
            vége ha
         vége ha
      kur_költség = kur_költség−d[x[k−1]][x[k]]
      vége ha
   vége minden
vége BTp

ígéretes(x[],k)
   minden i = 1,k−1 végezd
      ha x[i] == x[k] akkor
         return HAMIS
      vége ha
   vége minden
   return IGAZ
vége ígéretes
```

A fenti algoritmus hatékonysága könnyűszerrel növelhető. Ha valamely csomóponthoz tartozó kur_költség érték már nagyobb a min_költség értéknél, akkor az illető csomópont gyökerű részfa bejárása teljesen értelmetlen. E javítás úgy is tekinthető, mint ami branch-and-bound szellemben történik (lásd lentebb).

8.1.2. Mohó stratégia

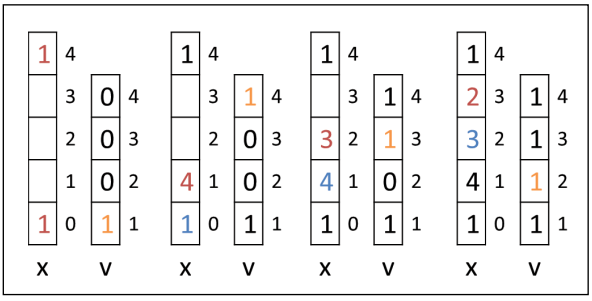
Egy mohó stratégia erre a feladatra az lehetne, hogy indulunk valamelyik városból, és minden lépésben a legközelebbi, még nem érintett város irányába haladunk tovább (az utolsó városból pedig visszatérünk az elsőbe). Ez a megközelítés a feladat struktúráját ábrázoló fa egyetlen gyökér–levél útján szalad le, az optimálisnak vélten. Abból is látszik, hogy ez az algoritmus nem garantálja az optimális megoldást, hogy attól függően, hogy melyik városból indul, más-más eredményhez jut (lásd a 8.4. ábrát).



8.4. ábra. *A „legközelebbi szomszéd” módszer 7 városra (a kör bezárását nem rajzoltuk be, ez implicit). Ha a legnyugatabbra lévő várostól indulunk, más eredményhez jutunk, mint ha a legdélebbre lévővel kezdenénk*

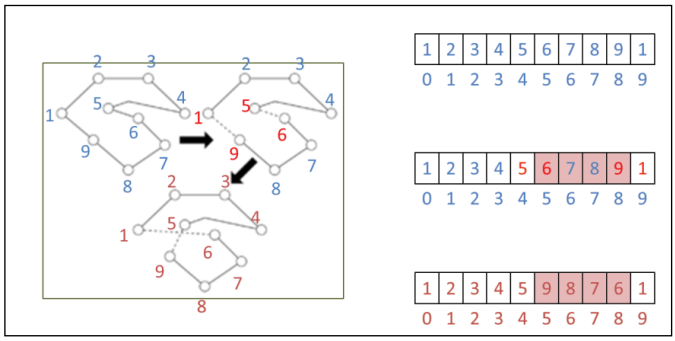
A következő algoritmusrészlet az $x[0..n]$ tömbben építi fel a vélt legrövidebb Hamilton-kört (lásd a 8.5. ábrát). Használja a $v[1..n]$ tömböt annak nyilvántartására, hogy mely városokat érintette már. Az NN (Nearest Neighbour = legközelebbi szomszéd) segédfüggvény egy adott városhoz legközelebb eső, még nem érintett város azonosítóját téríti vissza.

```
x[0] = 1                                  // az 1-es városból indulunk
x[n] = 1                                  // az 1-es városba érkezünk vissza
v[1] = 1                                  // az 1-es város érintett
minden i = 1,n−1 végezd
   x[i] = NN(x[i−1],v,d)                  // a következő legközelebbi város
   v[x[i]] = 1                            // bejelöljük mint meglátogatottat
vége minden
```



8.5. ábra. *Az „NN algoritmus” a megadott példára (x: a megtalált Hamilton-kör, lépésről lépésre; v: mely városok lettek már meglátogatva)*

A 8.6. ábra egy ötletet szemléltet arra, hogy miként lehet javítani a NN megközelítés találta megoldáson (Lin–Kernighan 1973). Véletlenszerűen kiválasztunk két nem szomszédos élet a mohó körről, és – amennyiben ez javít a hosszan – lecseréljük ezeket arra az élpárra, amely helyreállítja a kört. Addig ismételjük e lépést, amíg a módszer már nem eredményez javítást.



8.6. ábra. *Példa arra, hogy egy 1, 2, 3, 4, 5, **6**, **7**, **8**, **9**, 1 kör miként alakul át az 1, 2, 3, 4, 5, **9**, **8** , **7**, **6** ,1 körré, az (5,6) és (9,1) éleknek, az (5,9) és (6,1) élekre való lecserélése révén*

8.1.3. Egy branch-and-bound algoritmus

Továbbra is úgy tekintünk a feladat keresési terére, mint egy n szintes gyökeres fára, amelynek $(n-1)!$ levele van. Célunk az optimális levél megtalálása, illetve az idevezető minimális költségű gyökér–levél úté. Branch-and-bound szemszögből annál céltudatosabb a keresés, minél kisebb részét kell a fának effektíve bejárjuk/generáljuk e cél elérése érdekében. Kezdetben a generált fa az 1-es várost képviselő gyökércsomópontból áll. Minden lépésben a fa koronája peremén elhelyezkedő nyílt csomópontok egyikét (kurrens csomópont) elágaztatjuk („branching") a fiúcsomópontjaira, amelyekhez a kurrens ágon még nem szereplő városokat rendeljük. A „bound" fázis alapötlete, hogy amennyiben valamely fiúcsomópontról kideríthető, hogy a hozzá tartozó fiúrészfa garantáltan nem tartalmazza az optimális levelet, akkor lemondunk róla, azaz nem kerül fel a növekvő fa koronája peremére (száraz irányt képvisel).

Lévén szó minimalizálási problémáról, a fenti ötlet implementálása egy *alsó* korlát függvény definiálását feltételezi. Minden újonnan generált csomóponthoz társítunk egy $f=g+h$ értéket. A g komponens jelentse az illető pontig megtett útszakasz költségét, azaz legyen egyenlő a kurrens ágat alkotó élek költségeinek összegével. A gyökér esetében a g érték nyilván 0. A h komponens pedig legyen egy alulról becsült értéke a hátralévő útszakasz költségének (a kurrens pont gyö-

kerű részfában). A szakirodalomban számos javaslatot találunk a h heurisztikus függvényt illetően. Például választhatjuk a nullát is h értékének. Ebből a szemszögből tekintve, a 8.1.1. alfejezet végén közölt optimalizálási ötlet branch-and-bound szellemben javítja a backtracking algoritmust.

Az alábbiakban bemutatott megoldás azon a megfigyelésen alapszik, hogy minden Hamilton-körön n él szerepel. Jelöljünk egy ilyen kört a következőképpen: p_1, p_2, …, p_n, p_1. A megfelelő élsorozat a következő: (p_1,p_2), (p_2,p_3), …, (p_{n-1},p_n), (p_n,p_1). Minden (p_i,p_j) él költségére a körön nyilván igaz, hogy nagyobb vagy egyenlő, mint a p_i-re illeszkedő összes él költségeinek a minimuma. Más szóval, a költségmátrix soronkénti minimumainak összege alsó becslésnek számít a Hamilton-körök költségeire nézve (mivel a mátrix szimmetrikus, ugyanehhez az értékhez jutunk akkor is, ha az oszlopok minimumait tekintjük). Természetesen a teljes mátrixra kiszámított minimum a gyökér csomópontnak megfelelő h érték lesz. Az egyes ágakon megjelenő csomópontok h értékeinek kiszámításakor figyelembe kell vegyük, hogy mely csomópontok, illetve élek kerültek már fel a gyökértől az illető csomóponthoz vezető útszakaszra (ezek költségei a csomópont g értékébe épülnek be). A hátralevő útszakasz költségének becslésekor csak a többi csomópontnak (illetve élnek) megfelelő sorait/oszlopait kell a költségmátrixnak tekintsük. Ez a h érték alsó becslés lesz az illető csomóponthoz tartozó fiúrészfa gyökér–levél útjai költségeinek tekintetében.

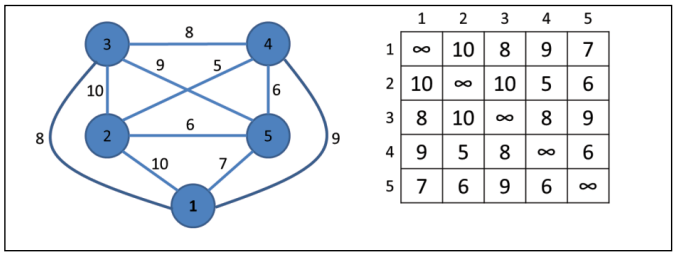
A következőkben a fentebb körvonalazott best-first-branch-and-bound stratégiát egy olyan példán szemléltetjük, amikor a városok száma $n=5$. A 8.7. ábra egy ilyen esetet ábrázol a hozzá tartozó költségmátrixszal. A 8.8. ábra lépésről lépésre mutatja be, hogy miként épül a fa. Kékkel ábrázoltuk a nyílt csomópontokat, és szürkével a zártakat. A piros keret azt jelzi, hogy melyik nyílt csomópont kerül elágaztatásra az adott lépésben, a best-first stratégia értelmében (amelyiknek a legkisebb az f értéke). Mindenik újonnan generált csomópont mellett megadtuk, hogy miként számolódik ki a megfelelő f érték (g: a kurrens ág menti élek súlyainak összege; h: a hátralévő útszakasz becsült költségét *dőlten* jelenítettük meg).

A kapcsolódó mátrixok magyarázattal szolgálnak a h komponensek kiszámítása tekintetében. A gyökér h értéke közvetlenül a költségmátrixból számolódik ki (8.8.a. ábra). Amint haladunk lefele egy ágon, a mátrix redukálódik a generált éleknek megfelelően. Egy (p_i,p_j) él p_i kezdőpontjához (apapont) tartozó mátrixból úgy kapjuk a p_j végponthoz (fiúpont) tartozó mátrixot, hogy töröljük (minden elemét ∞-re állítjuk) a p_i-edik sort, a p_j-edik oszlopot, valamint a [p_j][1] elemet (az 1-es városba való időnap előtti visszatérés kizárása). Szürke háttérrel jeleztük a kurrens törléseket.

Az ($n-1$)-dik szinti csomópontok leveleknek tekinthetők, hiszen már csak egy város vár meglátogatásra, egyértelmű az, amelyik még nem volt érintve az illető ágon (az utolsó városból az elsőbe való visszatérés ugyancsak egyértelmű). Ezért e levél-csomópontok f értéke a megfelelő Hamilton-kör pontos költségértéke lesz.

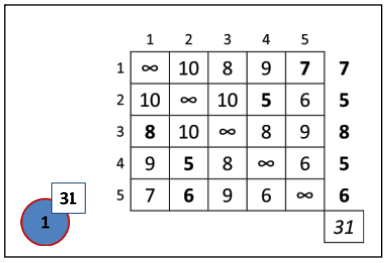
Lévén szó minimalizálási problémáról, a kurrens optimumértéket kezdetben ∞-re állítjuk, majd frissítjük, valahányszor levélcsomópont kerül generálásra. Minden nyílt csomópont, amelynek f értéke nem kisebb, mint a kurrens opti-

mum, lemetszhető a fáról (8.8.e. és 8.8.f. ábrák). Akkor ér véget az algoritmus, ha a nyílt csomópontok halmaza kiürült.



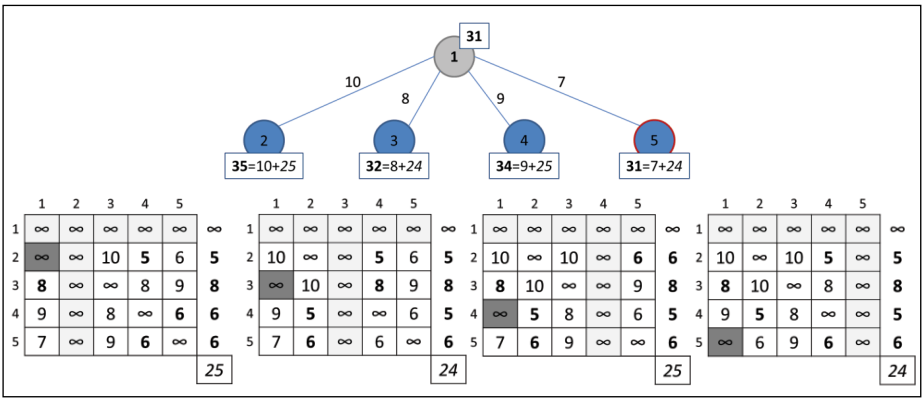
	1	2	3	4	5
1	∞	10	8	9	7
2	10	∞	10	5	6
3	8	10	∞	8	9
4	9	5	8	∞	6
5	7	6	9	6	∞

8.7. ábra. *Példa n=5 városra és a megfelelő költségmátrix*

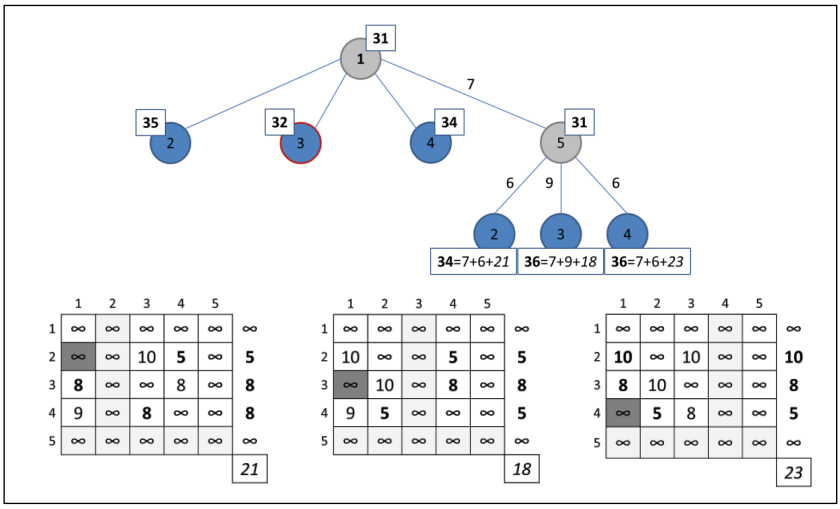


	1	2	3	4	5	
1	∞	10	8	9	**7**	**7**
2	10	∞	10	**5**	6	**5**
3	**8**	10	∞	8	9	**8**
4	9	**5**	8	∞	6	**5**
5	7	**6**	9	6	∞	**6**
						31

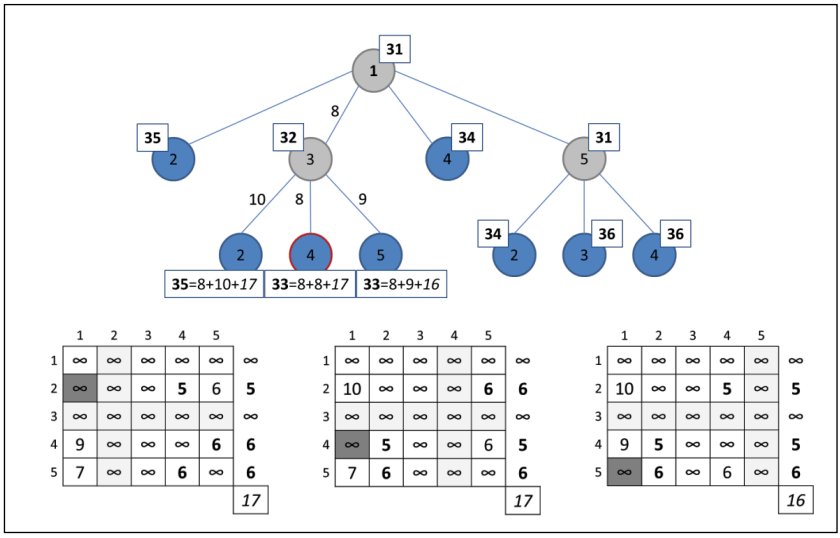
8.8.a. ábra. *A generált fa gyökere. A megfelelő f=g+h érték: 31=0+31. A h komponens a kapcsolódó mátrix soronkénti minimumainak összege*



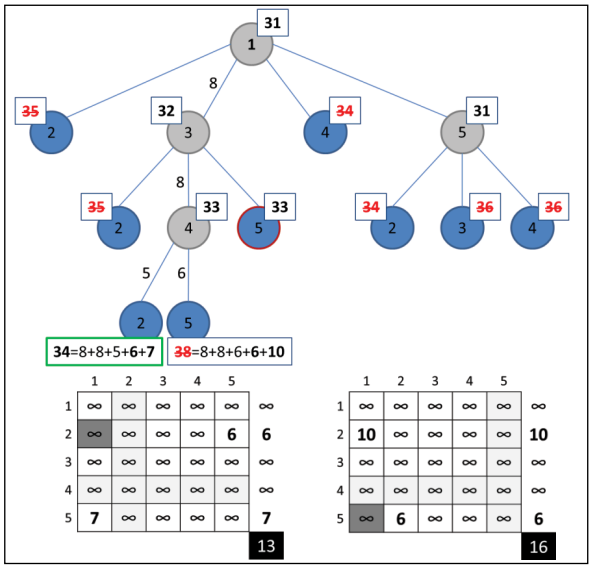
8.8.b. ábra. *A generált fa az első „branching” fázist követően. A mátrixokban szürke háttér jelzik, hogy mely elemek kerülnek törlésre az (1,2), (1,3), (1,4) és (1,5) élek nyomán*



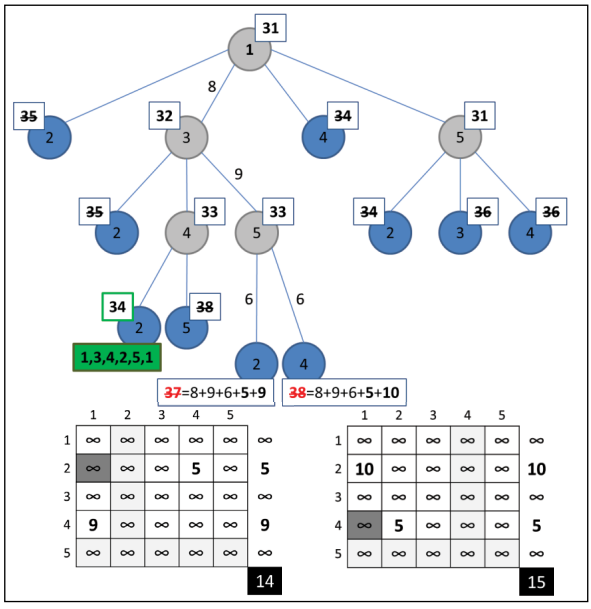
8.8.c. ábra. *A generált fa a második „branching” fázist követően*



8.8.d. ábra. *A generált fa a harmadik „branching” fázist követően*



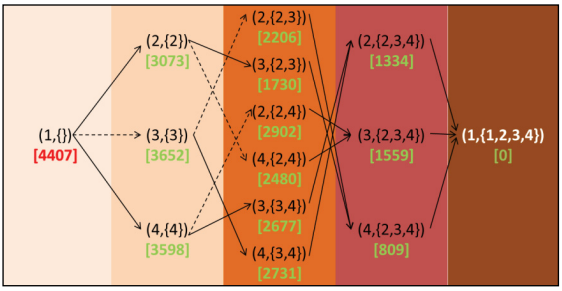
8.8.e. ábra. *A generált fa a negyedik „branching” fázist követően. A levelek f értékébe a becsült h komponens helyett a hátralévő két él hossza kerül be. A kurrens optimumot képviselő levelet zölddel kereteztük. A lemetszhető csomópontok f értékét áthúztuk*



8.8.f. ábra. *A generált fa az ötödik „branching” fázist követően. Az optimális Hamilton-kör: 1,3,4,2,5,1 (összköltsége 34)*

8.1.4. Dinamikus programozásos megoldás

A dinamikus programozás teljesen más perspektívából közelíti meg a feladatot. A feladat egy állapota a következőképpen is jellemezhető: (1) mely városban tartózkodik per pillanat az utazó ügynök, és (2) mely városok lettek már meglátogatva. A startállapot az, amikor az ügynök útra kész (az 1-es városban), és még az összes várost meg kell hogy látogassa (az 1-est is, utolsó állomásként): (1,{}). Akkor vagyunk célállapotban, ha az ügynök újra az 1-es városban van, és már minden várost meglátogatott: (1,{1,2,…,n}). A 8.9. ábra a megadott példára, szintekre bontva mutatja be az állapotgráfot. A kurrens szint valamely csomópontjának annyi fiúcsomópontja lesz a következő szinten, amennyi a még meglátogatásra váró városok száma. A dinamikus programozás nézőpontjából az állapotgráf összevont döntési faként is felfogható. A városok számának növekedésével ez a gráf is nagyon kiszélesedik. Például 20 város esetén a 10. szinten az állapotok száma már több mint 1 millió, illetve 30 város esetén, a 15. szinten az állapotok száma már több mint 1 billió. E jelenséget nevezte Bellman a dimenziók átkának.



8.9. ábra. *Szintről szintre, lentről felfele (jobbról balra). Bejelöltük az adott szintekhez tartozó apafeladatok előző szinti fiúrészfeladatait. A folytonos vonallal ábrázolt nyilak optimális apa-fiú kapcsolatokat jelölnek.*

A 7.6.5. alfejezetben már felvezettük a dinamikus programozásos ötletet az utazó ügynök feladatra. Az optimumértékek eltárolására a $c[1..n][0..2^n-1]$ tömböt használjuk (lásd a 8.10. ábrát). A $c[i][s]$ érték azt tárolja, hogy mennyi az 1-es városba való visszajutás minimális költsége, ha jelenleg az i városban vagyunk, és az s érték kódolta városok lettek már meglátogatva. Egészen pontosan az s index-érték n pozíción való bináris ábrázolásában a bitek a városokat képviselik, az 1-esek már meglátogatott városokat, a 0-sok a még meg nem látogatottakat (a j helyértékű bit a $j+1$ azonosítójú várost képviseli).

c	{} 0000 0	{2} 0010 2	{3} 0100 4	{4} 1000 8	{2,3} 0110 6	{2,4} 1010 10	{3,4} 1100 12	{2,3,4} 1110 14	{1,2,3,4} 1111 15
1	4407								0
2		3073			2206	2902		1334	
3			3652		1730		2677	1559	
4				3598		2480	2731	809	

8.10. ábra. *Az optimumok tömbje a 4 városos szemléltető példára (c[1..4][0..2^4-1])*

Az alábbi algoritmus rekurzívan implementálja a dinamikus programozásos stratégiát. A DP_utazóügynök függvényt a startállapotra hívjuk meg (még a teljes feladat megoldásra vár): DP_utazóügynök(1,0,n,d,c). Azokat a részfeladatokat, amikor már csak az 1-es városba való visszatérés maradt hátra, triviálisan egyszerűnek tekintjük. A rekurzióból adódó mélységi bejárás posztorder momentumaiban valósul meg a lentről felfele építkezés. Egy kurrens (i,s)-feladat optimuma a közvetlen fiúoptimumokból kerül kiszámításra, a $c[i][s]$ cellában zajló minimum számítás révén.

Kezdetben a c tömb minden cellája a (-1) értéket tárolja, aminek jelentése az, hogy a megfelelő részfeladat még nem lett megoldva. Valahányszor megoldásra kerül egy részfeladat, a megfelelő optimumot eltároljuk, és ha később, egy másik ágon újra találkoznánk ugyanazzal a részfeladattal, akkor csak elővesszük az eltárolt értéket. A bit(s,j) függvény visszatéríti az s érték j. helyértékű bitjét. A bit1(s,j) függvény egy olyan értéket térít vissza, amelyben s-nek, a j. helyértékű bitje, 1-esre lett állítva.

```
DP_utazóügynök(i, s, n, d[][],c[][])
  ha c[i][s] ≠ −1 akkor                                  // identikus részfeladat
    return c[i][s]                          // visszatérítjük a korábban eltárolt megoldást
  vége ha
  ha s == 2^n−2 akkor                  // triviális: már csak az 1-es város maradt hátra
    c[i][s] = d[i][1]                                    // eltároljuk a megoldást
    return c[i][s]
  vége ha
  c[i][s] = ∞                                  // kezdőérték a minimumszámításhoz
  minden j = 1,n−1 végezd                        // minden 1..(n−1) helyértékű bitre
    ha bit(s,j) == 0 akkor                 // a j+1 város még nem lett meglátogatva
      temp = DP_utazóügynök(j+1,bit1(s,j),n,d,c)                  // fiúoptimum
      ha d[i][j+1] + temp < c[i][s] akkor
        c[i][s] = d[i][j+1] + temp
      vége ha
    vége ha
```

```
    vége minden
return c[i][s]
vége DP_utazóügynök
```

A backtracking megoldás nyers erő változatának komplexitása nyilván $\theta(n!)$. Ennél jóval hatékonyabb a dinamikus programozásos algoritmus, melynek időbonyolultsága $\theta(n^2 2^n)$. Bellman, illetve Held és Karp egymástól függetlenül publikálták e megoldást 1962-ben. Az általunk implementált rekurzív változat akár egy olyan oszd-meg-és-uralkodj stratégiaként is felfogható, amely eltárolja a már megoldott részfeladatok optimumait, hogy elkerülje ezek többszöri megoldását. A mohó megközelítés polinomiális bonyolultságú algoritmust eredményez, de nem garantálja az optimális megoldást. Egy 2006-ban közzétett branch-and-bound algoritmus (Concorde TSP Solver) 85 900 városra oldotta meg az utazó ügynök problémát (Applegate et al. 2006), ami sokat elmond ennek a módszernek a hatékonyságáról e feladat tekintetében. Mivel mind az öt módszerrel megközelíthető e híres feladat, ezért méltán tekinthető úgy, hogy egy sajátos felülnézetet tesz lehetővé a megvizsgált algoritmustervezési stratégiák világára.

9. KÖNYVÉSZET

AlgoRythmics
2020 https://www.youtube.com/user/AlgoRythmics
APPLEGATE, D. L. et alii
2006 *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, Princeton, USA.
BELLMAN Richard
1962 Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, 9. 1. 61–63.
CAMPBELL, Paul J.
1977 Gauss and the eight queens problem: A study in miniature of the propagation of historical error, *Historia Mathematica,* 4. 4. 397–404.
CENTER FOR RESEARCH ON PARALLEL COMPUTATION (CRPC)
1998 CRPC Researchers Solve Traveling Salesman Problem for Record-Breaking 13509 Cities, *Parallel Computing Research Newsletter*, 6. 2. https://www.crpc.rice.edu/CRPC/newsletters/sum98/news_tsp.html
CORMEN, Thomas H. et alii
2003 *Új algoritmusok*. Budapest, Scolar Kiadó.
GALE, D.–SHAPLEY, L. S.
1962 College Admissions and the Stability of Marriage, *American Mathematical Monthly*, 69. 1. 9–14.
GREGORICS Tibor
2014 *Heurisztikus útkereső algoritmusok*. Budapest, ELTE IK.
https://regi.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0052_32_heurisztikus_utkereso_algoritmusok/esas2004.htm?sco=lecke5_lap1.html
HANOI TORNYAI
2020 https://hu.wikipedia.org/wiki/Hanoi_tornyai
HELD, Michael–KARP, Richard
1962 A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied mathematics,* 10. 1. 196–210.
KÁTAI Zoltán
2007 *Algoritmusok felülnézetből*. Kolozsvár, Scientia Kiadó.
KÁTAI Zoltán
2010 Modelling dynamic programming problems by generalized d-graphs, *Acta Universitatis Sapientaiae, Informatica,* 2. 2. 210–230.
KÁTAI Zoltán–KOVÁCS Lehel István
2009 Towers of Hanoi–where programming techniques blend, *Acta Universitatis Sapientaiae, Informatica,* 1. 1. 89–108.
LAND, A. H.–DOIG, A. G.

1960 An automatic method of solving discrete programming problems, *Econometrica*, 28. 3. 497–520. doi:10.2307/1910129.

LEE, C. Y.

1961 An Algorithm for Path Connections and Its Applications, *IRE Transactions on Electronic Computers, EC*, 10. 2. 346–365.

LEVITIN, Anany

2008 *Introduction To Design And Analysis Of Algorithms*, Pearson Education India.

LIN, Shen–KERNIGHAN, B. W.

1973 An Effective Heuristic Algorithm for the Traveling-Salesman Problem, *Operations Research*, 21. 2. 498–516.

LITTLE, John D.C. et alii

1963 An algorithm for the traveling salesman problem, *Operations Research*, 11. 6: 972–989. doi:10.1287/opre.11.6.972.

NAU, D. S.–KUMAR, V.–KANAL, L.

1984 General branch and bound, and its relation to A* and AO*, *Artificial Intelligence*, 23. 1. 29–58.

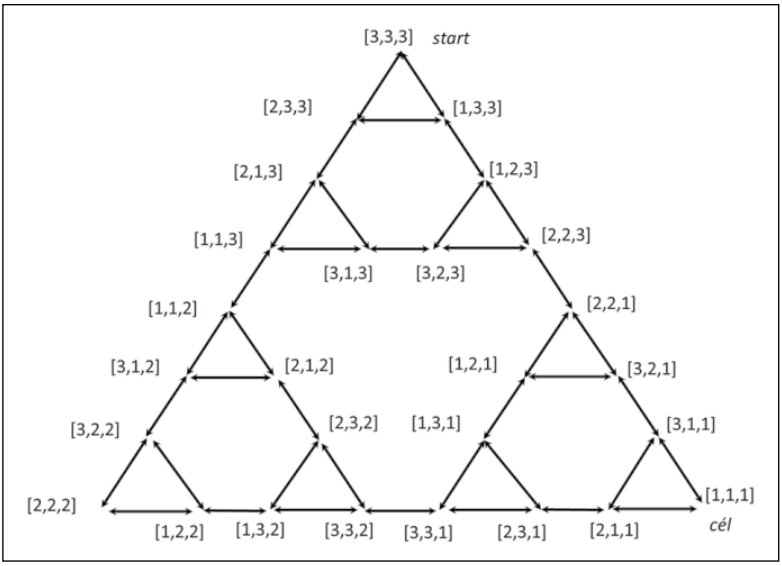
SADLER, John Edward

2013 *JA Comenius and the concept of universal education*. Routledge.

10. FÜGGELÉK

Az alábbiakban bemutatjuk három híres probléma állapottér-reprezentációját, ahogyan az megjelent a Gregorics 2014 tananyagban (lásd továbbá az 1. fejezet *A technikák mint útkereső stratégiák* című betétjét).

Hanoi tornyai: Elsőként következzen a Hanoi tornyai probléma állapotgráfja $n=3$ korong esetén.



9.1. ábra. *A Hanoi tornyai probléma állapotgráfja három korong esetén (Gregorics 2014)*

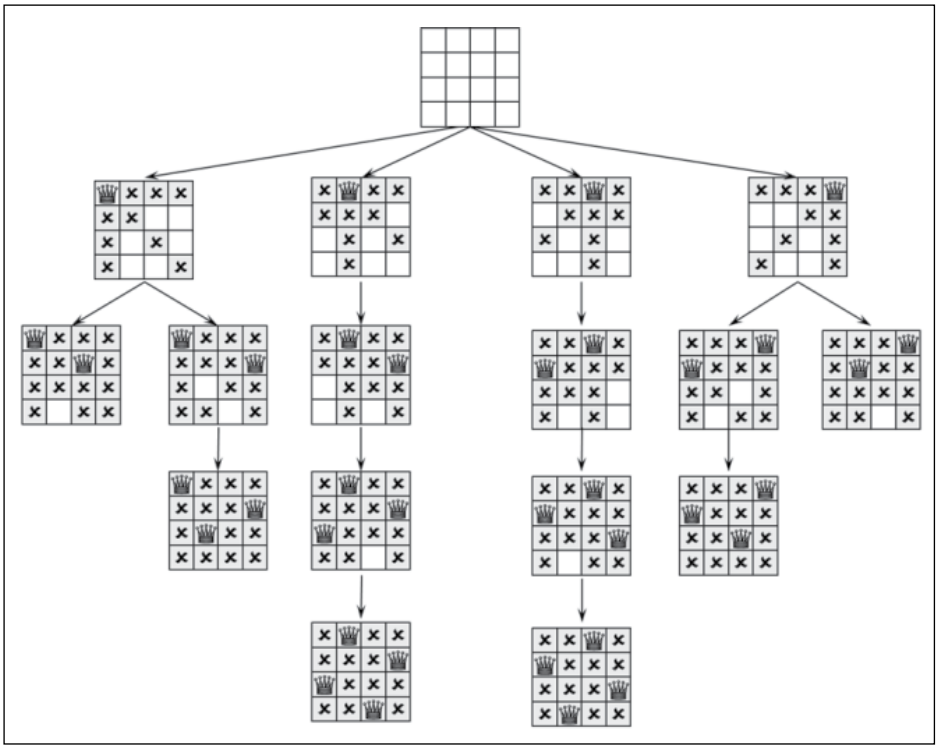
Az ábra azt a felállást szemlélteti, amikor kezdetben a korongok a 3-as rúdon vannak, és a célrúd az 1-es. A probléma egy-egy állapotát egy n hosszú vektorral tudjuk leírni. A vektor i-edik eleme azt képviseli, hogy az i-korong mely rúdon található éppen (startállapotban mindenik korong a 3-as rúdon, célállapotban pedig az 1-es rúdon található).

Észrevehető, hogy az állapotgráf csomópontjainak száma megegyezik $\{1,2,3\}$ halmaz n-szeres Descartes-szorzata elemeinek számával (3^n). Mivel minden állapotban tipikusan 3 megengedett lépés van, ezért a csomópontok szomszédjainak

száma tipikusan 3. Ez alól kivételek azok az állapotok, amikor mindenik korong ugyanazon rúdon található (ilyen esetben a szomszédok száma 2).

A 4. fejezetben bemutatott „oszd meg és uralkodj” algoritmus a legrövidebb start–cél útvonalat generálta, amely 7 lépéses megoldásnak felel meg $n=3$ korong esetén.

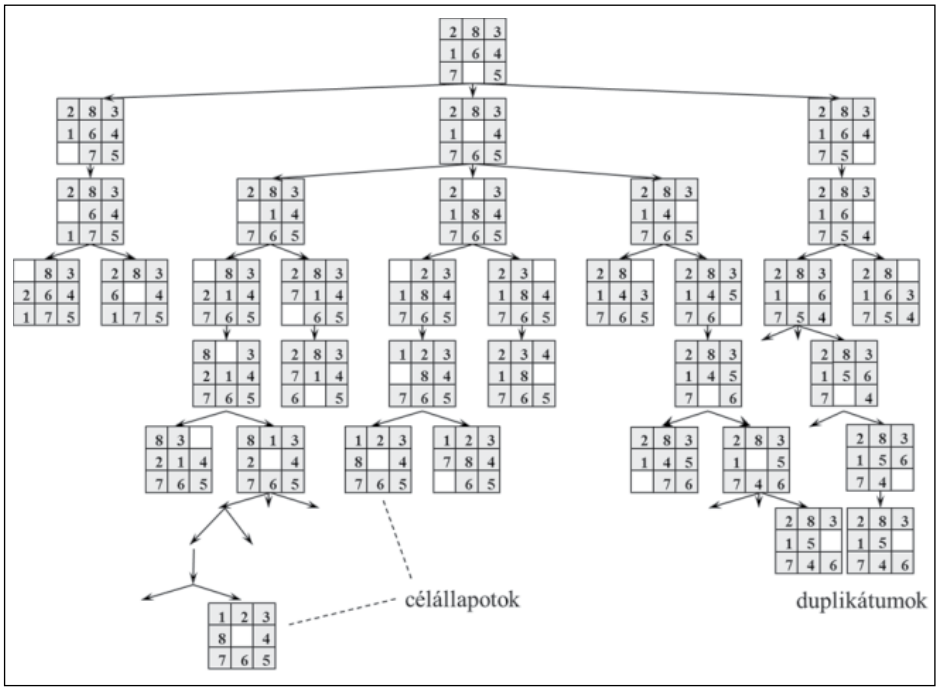
***n* királynő:** A 3. fejezetben tárgyalt n királynő probléma állapotgráfja $n=4$ esetre az alábbi:



9.2. ábra. *A 4-királynő probléma állapotgráfja (Gregorics 2014)*

A probléma állapotait egy-egy $n \times n$ méretű mátrix ábrázolja. Műveletnek azt választottuk, hogy „királynő felhelyezés”. Azt is kikötöttük, hogy sorról sorra haladunk, és minden sorba csak egy királynőt teszünk. A 3. fejezetben bemutatott algoritmus javítható, amennyiben folyamatosan nyilvántartjuk, hogy mely oszlopok, illetve átlók foglaltak már (az ábrán x-ek jelölik a kizárt négyzeteket). Ez esetben konstans időben eldönthető, hogy a kurrens sor egy adott pozíciója szabad-e az éppen elhelyezendő királynő számára.

Tili-toli: A 6. fejezetben tárgyalt kirakójáték állapotgráfja $n=3$ esetre az alábbi. Egészen pontosan az ábra a keresési tér egy részét mutatja be. Emlékezzünk, hogy keresési térnek nevezzük azt a fát, amelyet a „keresés az állapotgráfból lát" (az állapotgráf kiegyenesedik egy irányított fává). A keresési tér alapvetően ugyanazt a problémateret jelöli ki, mint az állapotgráf. Ha a keresési tér identikus állapotokat képviselő csúcsait (duplikátumok) egymásra csúsztatjuk, akkor visszanyerjük az állapotgráfot.



9.3. ábra. A 8-as tili-toli játék keresési terének részlete (Gregorics 2014)

Az ábra a játék azon változatát mutatja be, amikor kirakott állapotnak az számít, ha az üres cella középen van, és az 1, 2, …, 8 számok a bal felső saroktól induló órajárás szerinti sorrendben követik egymást.

Az ábrán megjelenítettük az állapotgráf startcsúcsból kivezető fontos útjait. Megfigyelhető, hogy ugyanaz az állapot több csúcsban is megjelenik, merthogy az illető állapothoz az eredeti állapotgráfban több különböző úton is el lehet jutni. Egy további sajátossága e keresési térnek, hogy az állapotgráfbeli körök miatt a fának végtelen hosszú ágai is vannak.

ABSTRACT

Algorithm Design Techniques

This book presents the five basic algorithm design strategies: greedy, backtracking, divide and conquer, dynamic programming, and branch and bound. The goal of the suggested syllabus is, beyond the presentation of the techniques, to offer the students a view that reveals them the basic and even the slight differences and similarities between the analysed programming technics.

REZUMAT

Tehnici de proiectare a algoritmilor

Cartea prezintă cele cinci tehnici de programare de bază: greedy, backtracking, divide et impera, programarea dinamică şi branch and bound. Scopul didactic al cărţii este de a transmite cititorului o vedere de ansamblu, şi să ajute în discernerea asemănărilor şi a diferenţelor – chiar şi de nuanţă – între metodele menţionate.

A SZERZŐRŐL

Kátai Zoltán 1968. március 13-án született Nagyváradon. Középiskolai tanulmányait a marosvásárhelyi Bolyai Farkas Elméleti Líceumban végezte 1982–1986 között, egyetemi tanulmányait pedig a Kolozsvári Műszaki Egyetem Automatizálás és Számítógépek Karán 1987–1992 között.

1992–2005 között informatikatanár a marosvásárhelyi Bolyai Farkas Elméleti Líceumban, 2002-től pedig a Sapientia Erdélyi Magyar Tudományegyetem Marosvásárhelyi Kara Matematika–Informatika Tanszékének oktatója és kutatója. Fő kutatási területe a programozási technikák (főleg ezek gráfelméleti háttere), kiemelten a dinamikus programozás. Oktatástudományi vonalon az érzékszervek bevonása a programozásoktatás folyamatába témakörben mélyült el. A szerző kezdeményezte az AlgoRythmics projektet, amelynek keretében született meg a youtube-on is elhíresült tíz algoritmus-tánckoreográfia. Az AlgoRythmics projekt 2013-ban elnyerte az Informatics Europe által felajánlott „Best practice in education” díjat.

JEGYZETEK

Scientia Kiadó
400112 Kolozsvár (Cluj-Napoca)
Mátyás király (Matei Corvin) u. 4. sz.
Tel./fax: +40-364-401454
E-mail: scientia@kpi.sapientia.ro
www.scientiakiado.ro

Korrektúra:
Szenkovics Enikő

Borító:
Tipotéka Kft.
A borítóképet rajzolta: Takács Kinga, Abodi Norbert

Műszaki szerkesztés:
Metaforma Kft.

Tipográfia:
Könczey Elemér

Nyomdai munkálatok:
F&F INTERNATIONAL Kft.
Felelős vezető: Ambrus Enikő igazgató

sapientia
tankönyvek

Comenius, akit a modern oktatás létrehozójának tartanak, a következő kijelentést tette a tanítási módszereket illetően: „Tanítani szinte nem is jelent mást, mint megmutatni, miben különböznek egymástól a dolgok a különböző céljukat, megjelenési formájukat és eredetüket illetően... Ezért aki jól megkülönbözteti egymástól a dolgokat, az jól is tanít".

Jelen könyv elsősorban erre a didaktikai alapelvre épül. Egy olyan tanítási, illetve tanulási módszert ajánl, amely segít a tanulóknak úgymond felülnézetből látni a megvizsgált öt programozási módszert (algoritmustervezési stratégiát): mohó keresés, visszalépéses keresés, oszd-meg-és-uralkodj, elágazás és korlátozás, illetve dinamikus programozás.

Tehát nem csak az a célunk, hogy bemutassuk e módszereket, hanem hogy olyan nézőpontba juttassuk az olvasót, amelyből feltárulnak előtte a technikák közötti elvi, alapvető, sőt árnyalatbeli különbségek, illetve hasonlóságok. A comeniusi alapelvvel összhangban ez nélkülözhetetlen, ha uralni szeretnénk a programozás e területét.

ISBN 978-606-975-037-7