2 skyrius

ALGORITMŲ KONSTRAVIMO METODAI

2.1 Metodas “Skaldyk ir valdyk”

Metodą “skaldyk ir valdyk” (lot. *divide et impera*; angl. *divide-and-conquer*) nuo senovės Romos laikų sėkmingai naudojo šimtai valdovų ir karvedžių. Pasirodo, kad šis principas yra naudingas ne tik politikoje, bet ir algoritmų kūrime. Šį principą jau keletą kartų naudojome ir mes (žr. binariosios paieškos, didžiausio ir mažiausio aibės elemento paieškos ir rūšiavimo sąlaja algoritmus).

Dažnai pradinį uždavinį galima suskaidyti į keletą mažesnių tos pačios klasės uždavinių, kuriuos išsprendę, nesunkiai randame ir pradinio uždavinio sprendinį. Rekursyviai tęsdami šį skaldymo procesą, mes pagaliau gauname mažus uždavinius, kuriems išspręsti pakanka kelių operacijų. Bendras algoritmo sudėtingumas priklauso nuo mažesnių uždavinių kiekio, jų dydžio ir nuo skaičiaus papildomų operacijų, kurios reikalingos iš mažesnių uždavinių sprendinių formuoti didesnių uždavinių sprendinius. Naudojant šį metodą, algoritmo sudėtingumas $L(n)$ rekurenčiai išsireiškia per to paties algoritmo sudėtingumą mažesnėms parametro n reikšmėms. Pasirodo, galima įrodyti teoremą, kuri duoda bendrą tokių rekurenčių sąsajų sprendinį. Norint rasti konkretaus rekursyvaus algoritmo sprendinį, pakanka šio algoritmo parametrus įstatyti į bendrą sprendinį, gaunamą pagal šią teoremą.

2.1.1 Teorema “skaldyk ir valdyk”

Teorema 2.1.1. *Tarkime, $n = b^k$, ir mums pavyko uždavinį dydžio n suskaidyti į a to paties tipo uždavinių, kurie yra b kartų mažesni už pradinį uždavinį. Jei tokiam skaidymui ir pradinio uždavinio sprendinio formavimui iš šių mažesnių uždavinių sprendinių reikia cn^d operacijų, tai*

tokio algoritmo sudėtingumas išsireiškia rekurenčiąja sąsaja

$$L(n) = aL\left(\frac{n}{b}\right) + cn^d,$$

kur $a \geqslant 1,\ b > 1,\ c, d > 0$ *yra sveiki skaičiai.*

Tada algoritmo sudėtingumas bus

$$L(n) = \begin{cases} \mathrm{O}(n^d), & \textit{jei } a < b^d, \\ \mathrm{O}(n^d \log_b n), & \textit{jei } a = b^d, \\ \mathrm{O}(n^{\log_b a}), & \textit{jei } a > b^d. \end{cases}$$

Įrodymas. Kadangi n yra sveikojo skaičiaus b laipsnis, tai galime pratęsti rekurenčiąją formulę:

$$\begin{aligned} L(n) &= aL\left(\frac{n}{b}\right) + cn^d \\ &= a\left(aL\left(\frac{n}{b^2}\right) + c\left(\frac{n}{b}\right)^d\right) + cn^d \\ &= a^2\left(aL\left(\frac{n}{b^3}\right) + c\left(\frac{n}{b^2}\right)^d\right) + ac\left(\frac{n}{b}\right)^d + cn^d \\ &= a^3 L\left(\frac{n}{b^3}\right) + cn^d\left(1 + \frac{a}{b^d} + \frac{a^2}{b^{2d}}\right) \\ &= \cdots \\ &= a^k L\left(\frac{n}{b^k}\right) + cn^d\left(1 + \frac{a}{b^d} + \cdots + \frac{a^{k-1}}{b^{(k-1)d}}\right) \\ &= a^k L(1) + cn^d\left(1 + \frac{a}{b^d} + \cdots + \frac{a^{k-1}}{b^{(k-1)d}}\right). \end{aligned}$$

Kadangi $L(1) = \mathrm{const}$, tai belieka susumuoti skliaustuose stovinčią geometrinę progresiją su progresijos vardikliu a/b^d. (Kai kuriems uždaviniams dydis $L(1)$ gali būti neapibrėžtas, nes uždavinys dydžio n gali neturėti prasmės. Tokiu atveju sustojame ne po k rekursijos žingsnių, o po $k_0 < k$ žingsnių, kur $k_0 < k$ yra didžiausias natūralusis skaičius, kuriam uždavinys dydžio n/b^{k_0} turi prasmę. Kadangi $L(n/b^{k_0})$ yra konstanta, tai tokiu atveju pirmasis dėmuo paskutinėje lygybėje gali padidėti tik konstantą kartų, o antrasis dėmuo, t.y., geometrinės progresijos suma, gali tik sumažėti, nes visi progresijos nariai yra teigiami. Kadangi mes įrodinėjame viršutinį įvertį su tikslumu iki konstantos, tai gausime tą patį.)

Nagrinėsime 3 atvejus.

1. Kai $a < b^d$, geometrinė progresija yra mažėjanti. Kadangi progresijos nariai yra teigiami, tai šios baigtinės progresijos suma bus mažesnė už analogiškos begalinės progresijos narių sumą, o be galo mažėjančios geometrinės progresijos suma visada yra konstanta. Gauname

$$L(n) = \mathrm{O}(a^{\log_b n}) + \mathrm{O}(n^d) = \mathrm{O}(n^d),$$

nes

$$a^{\log_b n} = (b^{\log_b a})^{\log_b n} = (b^{\log_b n})^{\log_b a} = n^{\log_b a}$$

ir $\log_b a < d$.

2. Kai $a = b^d$, kiekvienas geometrinės progresijos narys yra lygus 1, todėl jos suma yra lygi $k = \log_b n$. Taigi, šiuo atveju

$$L(n) = \mathrm{O}(n^{\log_b a}) + \mathrm{O}(n^d \log_b n) = \mathrm{O}(n^d \log_b n).$$

3. Kai $a > b^d$, taikome geometrinės progresijos sumos formulę $S_m = b_1 \frac{1-q^m}{1-q}$:

$$L(n) = \mathrm{O}(n^{\log_b a}) + cn^d \frac{\frac{a^k}{b^{dk}} - 1}{\frac{a}{b^d} - 1} = \mathrm{O}(n^{\log_b a}) + \mathrm{O}\left(n^d \frac{a^{\log_b n}}{n^d}\right) = \mathrm{O}(n^{\log_b a}).$$

Teorema įrodyta. $\square$

Pavyzdys 2.1.1. Binariosios paieškos algoritmui (žr. 1.6) gauname

$$L(n) = L\left(\frac{n}{2}\right) + 1,$$

taigi $a = 1$, $b = 2$, ir $d = 0$ (konstanta c nesvarbu kokia). Kadangi $1 = 2^0$, tai pagal teoremą $L(n) = \mathrm{O}(\log_2 n)$.

Pavyzdys 2.1.2. Rūšiavimo sąlaja algoritmui (žr. 1.8.1) gauname

$$L(n) = 2L\left(\frac{n}{2}\right) + n,$$

taigi $a = 2$, $b = 2$, ir $d = 1$. Kadangi $2 = 2^1$, tai pagal teoremą $L(n) = \mathrm{O}(n \log_2 n)$.

Pavyzdys 2.1.3. Rekursyviam aibės didžiausio ir mažiausio elemento paieškos algoritmui (žr. 1.7.1) gauname

$$L(n) = 2L\left(\frac{n}{2}\right) + 2,$$

taigi $a = 2$, $b = 2$, ir $d = 0$. Kadangi $2 > 2^0$, tai pagal teoremą $L(n) = \mathrm{O}(n)$. Skyrelyje 1.7.1 mes gavome tikslesnį viršutinį šio algoritmo sudėtingumo įvertį $L(n) = \frac{3}{2}n - 2$. Šis pavyzdys rodo, kad tuo atveju, kai mus domina ir koeficientų dydžiai algoritmo sudėtingumo išraiškoje, šios teoremos taikyti negalima, nes ji nustato sudėtingumą tik su tikslumu iki pastovaus daugiklio.

Dabar pateiksime dar du metodo “skaldyk ir valdyk” panaudojimo pavyzdžius.

2.1.2 Sveikųjų dvejetainių skaičių daugyba

Kiek operacijų su bitais reikalinga, norint sudauginti du sveikuosius dvejetainius skaičius ilgio n? Įprastas daugybos "stulpeliu" būdas reikalauna $\mathrm{O}(n^2)$ operacijų, nes reikia sudėti n dvejetainių skaičių ilgio n:

$$\begin{array}{r} 1101 \\ 1010 \\ \hline 0000 \\ 1101 \\ 0000 \\ 1101 \\ \hline 10000010 \end{array}$$

Pažymėkime šį uždavinį $\mathrm{MULT\_INT}$. Įrodysime, kad $L^{\mathrm{MULT\_INT}}(n) = \mathrm{O}(n^{\log_2 3})$, kur $\log_2 3 \approx 1.59$. Šį rezultatą 1962 m. įrodė Karacuba ir Ofman.

Tarkime, kad mums reikia sudauginti dvejetainius skaičius x ir y vienodo ilgio n. Pirmiausia nagrinėsime atvejį, kai $n = 2^k$. Tada x ir y galime suskaidyti į vienodo ilgio dalis:

$$x = \boxed{a}\boxed{b}$$
$$y = \boxed{c}\boxed{d}$$

Tada skaičių x ir y sandaugą xy galėsime užrašyti pavidalu

$$xy = (a2^{n/2} + b)(c2^{n/2} + d) = a \cdot c \cdot 2^n + (a \cdot d + b \cdot c) \cdot 2^{n/2} + b \cdot d.$$

Taigi, šiai sandaugai rasti reikia 4 daugybos operacijų su dvejetainiais skaičiais ilgio $n/2$, o taip pat kelių sudėties ir postūmio (t.y., daugybos iš 2 laipsnio) operacijų. Pasirodo, pakanka ir 3 daugybos operacijų. Pažymėję

$$u := (a + b) \cdot (c + d),$$
$$v := a \cdot c,$$
$$w := b \cdot d,$$

gauname

$$xy = v \cdot 2^n + (u - v - w) \cdot 2^{n/2} + w.$$

Kadangi visoms sudėties, atimties ir postūmio operacijoms tereikia $\mathrm{O}(n)$ operacijų su bitais, tai gauname rekurenčią sudėtingumo sąsają

$$L(n) = 3L\left(\frac{n}{2}\right) + \mathrm{O}(n),$$

iš kurios pagal "skaldyk ir valdyk" teoremą ($a = 3,\ b = 2,\ d = 1$) išplaukia $L(n) = \mathrm{O}(n^{\log_2 3})$.

Mūsų įrodymas turi vieną trūkumą: mes visur skaičiavome operacijas su dvejetainiais skaičiais ilgio $n/2$, tuo tarpu sumos $a+b$ ir $c+d$, įeinančios į vieną iš sandaugų, galėjo būti ir ilgio $n/2+1$. Šiuo atveju užrašome

$$a + b = a_1 \cdot 2^{n/2} + b_1,$$
$$c + d = c_1 \cdot 2^{n/2} + d_1,$$

kur $a_1, c_1 \in \{0, 1\}$, b_1 ir d_1 yra ilgio $n/2$. Kadangi

$$(a + b)(c + d) = a_1 c_1 \cdot 2^n + (a_1 d_1 + b_1 c_1) \cdot 2^{n/2} + b_1 \cdot d_1,$$

tai iš paskutinės lygybės matyti, kad ir sandaugai $(a+b)\cdot(c+d)$ pakanka 1 daugybos tarp skaičių ilgio $n/2$, papildomai panaudojus $\mathrm{O}(n)$ operacijų su bitais sudėčiai ir postūmiams.

Liko išnagrinėti atvejį, kai parametras n nėra 2 laipsnis. Šiuo atveju $\exists k$: $2^{k-1} < n < 2^k = n'$. Papildę skaičius x ir y iš priekio nuliais, gausime skaičius ilgio n', kuriems teorema jau įrodyta. Kadangi $n' < 2n$, gauname $L(n) < L(n') = \mathrm{O}((n')^{\log_2 3}) = \mathrm{O}(n^{\log_2 3})$.

2.1.3 Matricų daugyba Strassen'o metodu

Nagrinėsime kvadratinių n-osios eilės matricų daugybos uždavinį MATRIX_MULTIPLICATION. Skaičiuosime, kiek tokių matricų daugybai reikia aritmetinių, priskyrimo ir kitokių operacijų. Šio uždavinio sudėtingumą pažymėkime $M(n)$. Pasinaudoję standartiniu matricų daugybos algoritmu, gauname trivialų viršutinį įvertį $M(n) = \mathrm{O}(n^3)$. Iš tiesų, jei $C = AB$, tai matricos C elementai gaunami pagal formulę

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j},$$

taigi kiekvienam matricos C elementui rasti pakanka $2n - 1$ aritmetinių operacijų, o tokių elementų skaičius yra lygus n^2.

1969 m. Strassen pasiūlė matricų daugybos algoritmą, kurio sudėtingumas yra $\mathrm{O}(n^{\log_2 7})$, kur $\log_2 7 < 2.81$. Pagrindinė šio algoritmo idėja buvo ta, kad vietoje standartinio matricų 2×2 daugybos būdo, naudojančio 8 daugybos ir 4 sudėties operacijas, Strassen pasiūlė formules, kurios leidžia tokias matricas sudauginti, panaudojus 7 daugybos ir 18 sudėties operacijų. Pirmiausia įrodysime dvi lemas.

Lema 2.1.1 (Apie matricų daugybą blokais). *Jei n yra lyginis skaičius ir $C = AB$, tai padalijęs matricas A, B į vienodo dydžio $(n/2) \times (n/2)$ pomatrices, mes galėsime matricos C tokio pat dydžio pomatrices išreikšti per matricų A ir B pomatrices:*

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

kur

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22}. \end{aligned}$$

Įrodymas. Tegu c_{ij} yra bet kuris matricos C_{11} elementas. Tada

$$cij = \sum_{k=1}^{n} a_{ik}b_{kj} = \sum_{k=1}^{n/2} a_{ik}b_{kj} + \sum_{k=n/2+1}^{n} a_{ik}b_{kj},$$

taigi c_{ij} yra matricos A_{11} i-osios eilutės ir matricos B_{11} j-ojo stulpelio sandauga plius matricos A_{12} i-osios eilutės ir matricos B_{21} j-ojo stulpelio sandauga. Analogiškai yra įrodoma ir likusių pomatricių elementams. □

Lema 2.1.2. *Dvi matricas dydžio* 2×2 *galima sudauginti, panaudojus* 7 *daugybos ir* 18 *sudėties operacijų.*

Įrodymas. Turime

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix},$$

reikia elementus c_{ij} išreikšti per matricų A ir B elementus. Apibrėžiame papildomus kintamuosius m_i:

$$\begin{aligned} m_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) && (2 \text{ stulp.} \times 2 \text{ eil.}), \\ m_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) && (\text{įstriž.} \times \text{įstriž.}), \\ m_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) && (1 \text{ stulp.} \times 1 \text{ eil.}), \\ m_4 &= (a_{11} + a_{12})b_{22} && (1 \text{ eil.} \times b22), \\ m_5 &= a_{11}(b_{12} - b_{22}) && (a11 \times 2 \text{ stulp.}), \\ m_6 &= a_{22}(b_{21} - b_{11}) && (a22 \times -1 \text{ stulp.}), \\ m_7 &= (a_{21} + a_{22})b_{11} && (2 \text{ eil.} \times b11), \end{aligned}$$

kur skliaustuose "koduojame" formules, kad jas lengviau būtų įsiminti (eilutės elementus visada imame su pliusu, o stulpelio antrąjį elementą visada su minusu). Dabar matricos C elementus nesunku išreikšti per aukščiau išvardintus kintamuosius:

$$\begin{aligned} c_{11} &= m_1 + m_2 - m_4 + m_6, \\ c_{12} &= m_4 + m_5, \\ c_{21} &= m_6 + m_7, \\ c_{22} &= m_2 - m_3 + m_5 - m_7. \end{aligned}$$

Patikrinkime, pavyzdžiui, antrą lygybę:

$$c_{12} = m_4 + m_5 = (a_{11} + a_{12})b_{22} + a_{11}(b_{12} - b_{22}) = a_{11}b_{12} + a_{12}b_{22}.$$

Analogiškai įrodomos ir kitos lygybės. Nesunku suskaičiuoti, kad elementams c_{ij} rasti buvo panaudota 7 daugybos ir 18 sudėties ar atimties operacijų. $\square$

Teorema 2.1.2. *Dvi kvadratines matricas dydžio $n \times n$ galima sudauginti, panaudojus $\mathrm{O}(n^{\log_2 7})$ operacijų.*

Įrodymas. Pirmiausia tarkime, kad $n = 2^k$. Pagal 1 ir 2 lemas norint sudauginti dvi n-os eilės matricas, pakanka sudauginti 7 matricas dydžio $(n/2) \times (n/2)$ ir dar panaudoti $\mathrm{O}(n^2)$ sudėties bei atimties operacijų. Taigi,

$$M(n) = 7M\left(\frac{n}{2}\right) + \mathrm{O}(n^2),$$

iš kur pagal teoremą "skaldyk ir valdyk" gauname, kad $M(n) = \mathrm{O}(n^{\log_2 7})$ ($a = 7$, $b = 2$, $d = 2$).

Jei n nėra 2 laipsnis, tai $\exists k$: $2^{k-1} < n < 2^k = n'$. Papildę matricas A ir B iki eilės n' nuliais ir remdamiesi nelygybe $n' < 2n$, gauname $L(n) < L(n') = \mathrm{O}((n')^{\log_2 7}) = \mathrm{O}(n^{\log_2 7})$. $\square$

Naudojant tenzorinę algebrą, Strassen'o viršutinį įvertį vėliau pavyko pagerinti iki $\mathrm{O}(n^{2.376})$. Deja, išskyrus Strassen'o algoritmą, kurį galima taikyti ir praktiškai, kiti įverčiai yra daugiau "sportinio" tipo, nes prieš n laipsnį juose stovi milžiniškos konstantos. Trivialus apatinis šio uždavinio sudėtingumo įvertis yra $\Omega(n^2)$, nes algoritmo rezultatų skaičius (matricos C elementų skaičius) yra n^2. Todėl įdomu, kiek dar galima priartinti apatinį ir viršutinį įverčius vieną prie kito.

2.2 Dinaminis programavimas

Ankstesniame skyrelyje nagrinėtas "skaldyk ir valdyk" metodas remiasi rekursyviu uždavinio skaidymu "iš viršaus žemyn" į vis mažesnius uždavinius. "Skaldyk ir valdyk" metodas yra efektyvus tada, kai rekursija yra *išbalansuota*, t.y., uždaviniai yra skaidomi į kelis maždaug vienodo dydžio uždavinius. Tuo tarpu kai naudojame neišbalansuotą rekursiją, šis metodas gali tapti labai neefektyvus. Tai demonstruoja žemiau pateikiamas pavyzdys su Fibonacci skaičiais. Tokiais atvejais dažnai padeda *dinaminis programavimas*.

Dinaminis programavimas — tai uždavinio sprendimo metodas "iš apačios į viršų". Pirmiausia mes išsprendžiame visus paprasčiausius duoto uždavinio atvejus, t.y., mažiausius dalinius uždavinius ir įsimename gautus rezultatus. Remdamiesi gautais sprendiniais, randame didesnių dalinių uždavinių sprendinius ir t.t. Šis metodas yra efektyvus tada, kai pačių mažiausių dalinių

uždavinių nėra labai daug (t.y., kai jų skaičius polinomiškai priklauso nuo pradinio uždavinio dydžio) ir kai pradinio uždavinio sprendiniui rasti mums nereikia visų anksčiau gautų rezultatų, o pakanka tik tam tikros jų dalies.

Dinaminis programavimas dažniausiai taikomas tokiems uždaviniams, kurių objektas yra sutvarkyta aibė, ir kada pavyksta rasti rekurentinę priklausomybę tarp dalinių uždavinių sprendinių. Panagrinėsime keletą tokių uždavinių.

2.2.1 Fibonacci skaičiai

Dar XIII amžiuje išleistoje knygoje *Liber Abaci* italų pirklys Fibonacci[11] suformulavo įžymųjį uždavinį apie triušius. Tarkime, saloje apsigyveno porelė triušių (patinėlis ir patelė). Yra žinoma, kad sulaukę dviejų mėnesių amžiaus pora triušių kas mėnesį atveda porelę triušiukų: patinėlį ir patelę. Reikia nustatyti, kiek porų triušių bus saloje po n mėnesių. Pažymėję triušių porų skaičių po n mėnesių $F(n)$, nesunkiai randame rekurenčiąją sąsają $F(n) = F(n-1) + F(n-2)$ su pradine sąlyga $F(0) = 0$, $F(1) = 1$. Taigi, labai nesunku parašyti tokią rekursyvią programėlę skaičiui $F(n)$ rasti:

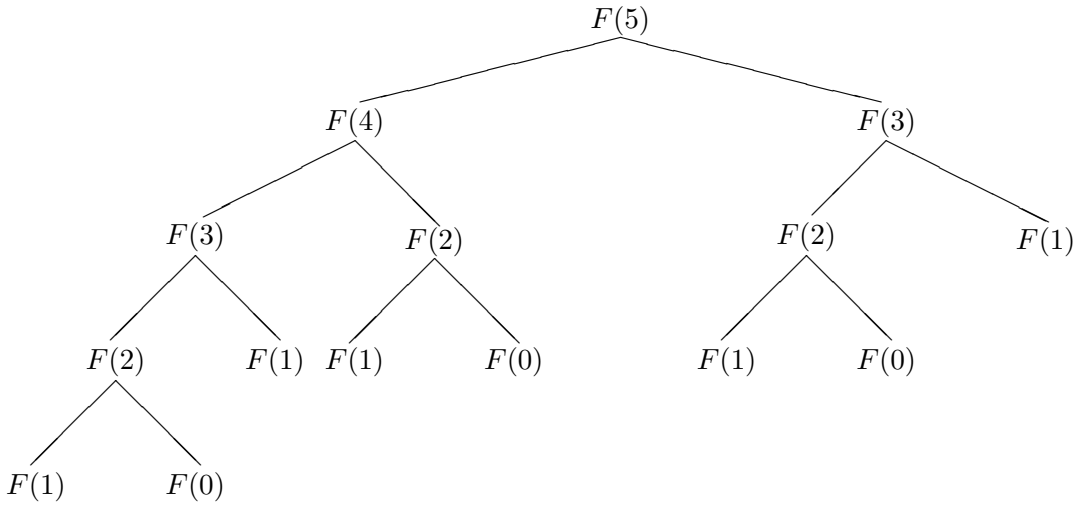
function $\mathrm{fib1}(n)$
if $n = 0$ **then** $\mathrm{fib1} := 0$
else if $n = 1$ **then** $\mathrm{fib1} := 1$
else $\mathrm{fib1} := \mathrm{fib1}(n-2) + \mathrm{fib1}(n-1)$

Deja, net ir nedidelėms n reikšmėms ($n \approx 40$) ši elementari programa dirbs labai ilgai. Tai lengva matyti iš rekursijos medžio, vaizduojamo 2.1 pav. Yra žinoma, kad Fibonacci skaičių santykis $F(n+1)/F(n)$ apytiksliai yra lygus $(1+\sqrt{5})/2 \approx 1.6$, taigi $F(n) \approx 1.6^n$. Kadangi rekursijos medžio lapuose stovi vienetai, tai lapų skaičius bus didesnis už $F(n)$, taigi programa fib1 daugiau negu $F(n) \approx 1.6^n$ kartų rekursyviai kreipiasi į save.

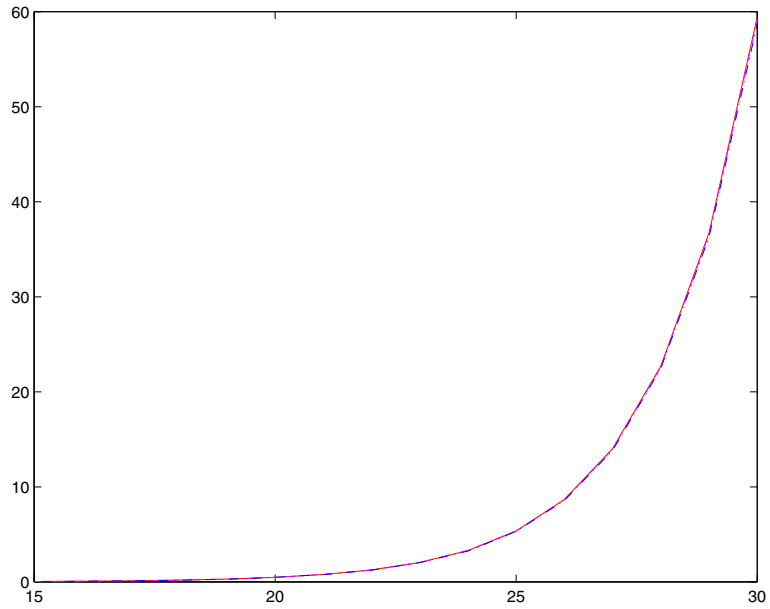
Pav. 2.2.1 matote tris grafikus, kurie absoliučiai sutampa. Pirmasis grafikas vaizduoja CPU laiką (sekundėmis), kurį sugaišo Matlab programa $\mathrm{fib1}$, skaičiuojant Fibonacci skaičius $F(15) = 610, \ldots, F(30) = 832040$, antrasis yra funkcijos $f(n) = F(n)/14000$ grafikas, ir trečiasis — funkcijos $g(n) = ((1+\sqrt{5})/2)^{n-2}/12000$ grafikas. Taigi, tokio algoritmo sudėtingumas auga eksponentiškai.

Akivaizdu, kad rekursyvią programą $\mathrm{fib1}$ galime pakeisti nerekursyvia programa, įsimindami tarpinius rezultatus masyve F (vietoje masyvo galime naudoti ir 2 kintamuosius, bet mes nebetaupysime):

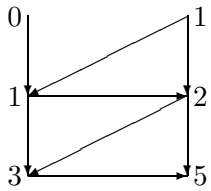
[11] **Fibonacci (1170–1250).** Fibonacci (sutrumpinta nuo *filius Bonacci*, t.y., "Bonacci sūnus") gimė Italijos mieste Pizoje, todėl dar yra žinomas Leonardo iš Pizos vardu. Jis buvo pirklys ir dažnai keliaudavo į Artimuosius Rytus, kur susipažino su arabų matematikais. Savo knygoje *Liber Abaci* jis supažindino europiečius su arabiškąja skaičiavimo sistema ir aritmetinių veiksmų algoritmais. Šioje knygoje Fibonacci ir suformulavo uždavinį apie triušius. Fibonacci taip pat parašė knygas apie geometriją, trigonometriją bei Diofanto lygtis.



Pav. 2.1: Programos fib1 rekursijos medis.



Pav. 2.2: CPU laiko sąnaudos (sekundėmis), rekursyviai ieškant Fibonacci skaičių $F(15)$–$F(30)$.



Pav. 2.3: Programos $\mathrm{fib2}$ skaičiavimo schema.

function $\mathrm{fib2}(n)$
$F(0) := 0$; $F(1) := 1$;
if $n > 1$ **then for** $i = 2$ **to** n **do** $F(i) := F(i-2) + F(i-1)$;
$\mathrm{fib2} := F(n)$;

Programa $\mathrm{fib2}$ pradeda nuo Fibonacci skaičių $F(0)$ ir $F(1)$ ir randa paeiliui visus Fibonacci skaičius $F(2), F(3), \ldots, F(n)$, kiekvieną kartą naudodama tik dvi paskutines reikšmes. Tai ir yra dinaminis programavimas. Algoritmas $\mathrm{fib2}$ yra tiesinio sudėtingumo. Realizavus jį Matlabe, šis algoritmas randa bet kurį iš skaičių $F(15)$–$F(30)$ greičiau, nei per 0.01 sek. Dar daugiau, per 0.01 sek. algoritmas $\mathrm{fib2}$ randa $F(1476) \approx 1.307 \cdot 10^{308}$. Vietoje rekursijos medžio, kurį naudoja programa $\mathrm{fib1}$, programa $\mathrm{fib2}$ naudoja tiesinio dydžio gardelę, pavaizduotą 2.3 pav.

2.2.2 Matricų daugybos tvarka

Tarkime, mums reikia sudauginti ilgą stačiakampių matricų seką:

$$M_1 \times M_2 \times \cdots \times M_n,$$

kur kiekviena M_i yra $r_{i-1} \times r_i$ dydžio matrica ($i = 1, 2, \ldots, n$). Naudojant standartinį matricų daugybos algoritmą, norint sudauginti $m \times n$ matricą A iš $n \times k$ matricos B, reikės $\mathrm{O}(mnk)$ aritmetinių operacijų. Šiame skyrelyje laikysime, kad jų reikės lygiai mnk. Matricų daugyba nėra komutatyvi, taigi matricų sukeisti vietomis negalime. Tačiau kadangi matricų daugyba yra asociatyvi, t.y., $A(BC) = (AB)C$, tai bendras operacijų skaičius priklausys nuo matricų daugybos tvarkos.

Pirmiausia panagrinėkime, ar šio uždavinio negalima būtų išspręsti pilno variantų perrinkimo būdu ("brutalios jėgos" algoritmu). Pažymėkime $K(n)$ skaičių skirtingų daugybos tvarkų, kai duota n matricų. Akivaizdu, kad

$$K(1) = 1 \quad \text{ir} \quad K(n) = \sum_{k=1}^{n-1} K(k)K(n-k), \quad n = 2, 3, \ldots, \tag{2.1}$$

nes seką $M_1, M_2, \ldots, M_n$ bet kurioje vietoje $k = 1, 2, \ldots, n-1$ perskyrę į du posekius $M_1, \ldots, M_k$ ir $M_{k+1} \ldots, M_n$, mes galime abiejuose posekiuose matricas dauginti bet kuria tvarka, taip

gaudami $K(k)K(n-k)$ skirtingų daugybos tvarkų. Skaičius $K(n)$ vadina *Katalano skaičiais*. Yra žinoma, kad Katalano skaičiai auga eksponentiškai: $K(n) \sim 4^{n-1}/(n\sqrt{\pi n})$, taigi pilnas perrinkimas yra labai neefektyvus.

Rekurenčioji sąsaja (2.1) duoda mums idėją, kaip rasti geriausią matricų daugybos tvarką: perskyrus matricų seką $M_1, M_2, \ldots, M_n$ į du posekius $M_1, \ldots, M_k$ ir $M_{k+1} \ldots, M_n$, reikia pasirinkti optimalią daugybos tvarką pirmajame posekyje ir optimalią daugybos tvarką antrajame posekyje. Pažymėję m_{ij} mažiausią operacijų skaičių, reikalingą norint sudauginti bet kurias pradinės sekos matricas nuo i iki j, t.y., matricas $M_i, M_{i+1}, \ldots, M_j$ $(1 \leqslant i \leqslant j \leqslant n)$ gauname rekurenčiąją sąsają

$$\begin{cases} m_{ij} = \min\limits_{i \leqslant k < j} (m_{ik} + m_{k+1,j} + r_{i-1} r_k r_j), & i < j, \\ m_{ii} = 0. \end{cases} \tag{2.2}$$

Naudodami šią sąsają, mes galime iš pradžių rasti optimalią bet kurių dviejų iš eilės sekoje stovinčių matricų daugybos tvarką (šiuo atveju vienintelė galima tvarka ir bus optimali), po to optimalią bet kurių trijų iš eilės stovinčių matricų daugybos tvarką ir t.t., kol rasime optimalią iš eilės stovinčių n matricų daugybos tvarką. Visas indeksų poras i, j, kurias peržiūrės dinaminio programavimo algoritmas, galima pavaizduoti trikampe matrica:

$$\begin{pmatrix} 1,1 & 1,2 & \ldots & 1,n-1 & 1,n \\ & 2,2 & \ldots & 2,n-1 & 2,n \\ & & \ddots & \vdots & \vdots \\ & & & n-1,n-1 & n-1,n \\ & & & & n,n \end{pmatrix}.$$

Sunumeruokime šios matricos įstrižaines, pradedant nuo pagrindinės įstrižainės ir einant dešinio viršutinio matricos kampo link: $\mathrm{diag} = 0, 1, \ldots, n-1$. Kiekvienos įstrižainės $\mathrm{diag} = i$ elementų reikšmės priklauso įstrižainėse $0, 1, \ldots, i-1$ stovinčių reikšmių. Taigi, pradėję nuo pagrindinės įstrižainės ir judėdami dešinio viršutinio matricos kampo link, po $n-1$ iteracijos rasime optimalų operacijų skaičių m_{1n}. Tai daro procedūra $\mathrm{Matrix\_Order}$, pateikiama žemiau. Kitoje trikampėje matricoje best mes saugosime optimalias k reikšmes, kurios duoda minimumą formulėje (2.2). Naudodama šias reikšmes, procedūra $\mathrm{Show\_Order}(1, n)$ leis mums rekursyviai atstatyti optimalią matricų daugybos tvarką.

procedure $\mathrm{Matrix\_Order}(r)$
for $i := 1$ **to** n **do** $m[i,i] := 0$;
for $i := 1$ **to** n **do**
    for $j := 1$ **to** n **do** $\mathrm{best}[i,j] := 0$;
for $\mathrm{diag} := 1$ **to** $n-1$ **do**
    for $i := 1$ **to** $n - \mathrm{diag}$ **do**
        $j := i + \mathrm{diag}$;

```
        m[i, j] := maxinteger;
        for k := i to j − 1 do
            mnew := m[i, k] + m[k + 1, j] + r[i − 1] · r[k] · r[j];
            if mnew < m[i, j] then
              m[i, j] := mnew; best[i, j] := k;
            end
        end
    end
end
return m[1, n], best

procedure Show_Order(i, j)
if i = j write M_i;
else k := best(i, j);
     write "("; Show_Order(i, k); write "*"; Show_Order(k + 1, j); write ")"
end
```

Akivaizdu, kad bendras abiejų algoritmų sudėtingumas yra $\mathrm{O}(n^3)$.

Pavyzdys 2.2.1. Tarkime, duotos 4 matricos M_1, M_2, M_3, M_4 dydžio atitinkamai 10×20, 20×50, 50×1 ir 1×100. Gauname

$$\begin{aligned}
&m[1,2] = 10000, \quad m[2,3] = 1000, \quad m[3,4] = 5000,\\
&m[1,3] = \min\{m[2,3] + r[0]\cdot r[1]\cdot r[3], m[1,2] + r[0]\cdot r[2]\cdot r[3]\} = 1200,\\
&m[2,4] = \min\{m[3,4] + r[1]\cdot r[2]\cdot r[4], m[2,3] + r[1]\cdot r[3]\cdot r[4]\} = 3000,\\
&m[1,4] = \min\{m[2,4] + r[0]\cdot r[2]\cdot r[4], m[1,2] + m[3,4] + r[0]\cdot r[2]\cdot r[4],\\
&\qquad\qquad m[1,3] + r[0]\cdot r[3]\cdot r[4]\} = 2200,
\end{aligned}$$

o masyvas best atrodo taip: $\mathrm{best}[1,2] = 1$; $\mathrm{best}[2,3] = 2$; $\mathrm{best}[3,4] = 3$; $\mathrm{best}[1,3] = 1$; $\mathrm{best}[2,4] = 3$; $\mathrm{best}[1,4] = 3$. Tada procedūra $\mathrm{Show\_Order}(1, n)$ duoda tokią optimalią šių matricų daugybos tvarką: $(M_1 * (M_2 * M_3)) * M_4$.

2.2.3 Kuprinės užpildymo uždavinys

Vagis įsibrovė į sandėlį, kuriame yra N rūšių daiktų. Daiktų dydžiai yra $\mathrm{size}[1], \ldots, \mathrm{size}[N]$, o jų vertės $\mathrm{val}[1], \ldots, \mathrm{val}[N]$, kur $\mathrm{size}[i], \mathrm{val}[i] \in \mathbb{N}\ \forall i = 1, \ldots, N$. Vagis turi kuprinę, kurios talpa $M \in \mathbb{N}$. Kiek kiekvienos rūšies daiktų turi paimti vagis, kad jų dydžių suma neviršytų kuprinės talpos, o jų bendra vertė būtų maksimali ? Bendrą pripildytos kuprinės daiktų vertę vadinsime kuprinės kaina. Tokį pat uždavinį tenka spręsti ir į žygį išsiruošusiam turistui.

Šį uždavinį galime apsukti iš kito galo. Tarkime, kad mes kažkokiu būdu užpildėme kuprinę. O dabar pagalvokime, kurios rūšies paskutinį daiktą vertėjo išsirinkti. Tarkime, kad mes žinojome

optimalias visų mažesnių kuprinių kainas $\mathrm{cost}[0], \mathrm{cost}[1], \ldots, \mathrm{cost}[M-1]$. Tada kiekvienos rūšies i daiktams mes galėjome patikrinti, kokia gausis kuprinės kaina, jei prie $M - \mathrm{size}[i]$ talpos kuprinės optimalios kainos mes pridėsime i-osios rūšies daikto vertę $\mathrm{val}[i]$, ir išsirinkti daiktą, kuriam ši suma bus didžiausia.

Taip mes gauname rekurenčiąją sąsają kuprinės kainai cost:

$$\begin{cases} \mathrm{cost}[0] = 0, \\ \mathrm{cost}[i] = \max\limits_{j=1}^{N}\{\mathrm{cost}[i - \mathrm{size}[j]] + \mathrm{val}[j]\}, \end{cases}$$

kur maksimumas renkamas nagrinėjant tik tuos j, kurie tenkina sąlygą $i - \mathrm{size}[j] \geqslant 0$. Taigi, mes galime dinamiškai rasti optimalią kainą visoms kuprinėms, kurių talpa yra mažesnė už M, o tada galėsime gauti ir optimalią M talpos kuprinės kainą. Dinaminio programavimo algoritmas atrodo taip:

procedure $\mathrm{knapsack\_packing}(\mathrm{size}, \mathrm{val}, M, N)$
for $i := 1$ **to** M **do** $\mathrm{cost}[i] := 0$; **end**;
for $j := 1$ **to** N **do**
  for $i := 1$ **to** M **do**
    if $i - \mathrm{size}[j] \geqslant 0$ **then**
      if $\mathrm{cost}[i] < \mathrm{cost}[i - \mathrm{size}[j]] + \mathrm{val}[j]$ **then**
        $\mathrm{cost}[i] := \mathrm{cost}[i - \mathrm{size}[j]] + \mathrm{val}[j]$;
        $\mathrm{best}[i] := j$;
      end
    end
  end
end
return $\mathrm{cost}, \mathrm{best}$

Masyvas best yra naudojamas geriausiam kuprinės užpildymui įsiminti. $\mathrm{best}[i] = j$ reiškia, kad optimaliai užpildant kuprinę talpos i, paskutinį reikia dėti j-osios rūšies daiktą. Šio algoritmo sudėtingumas yra $\mathrm{O}(MN)$. Taigi, jei kuprinės talpa auga ne greičiau, kaip koks nors polinomas, t.y. $M = \mathrm{O}(N^k)$, tai dinaminio programavimo algoritmas taip pat bus polinominio sudėtingumo.

Pavyzdys 2.2.2. Tarkime, yra duota 5 rūšių daiktai (A, B, C, D ir E) ir yra žinomi jų dydžiai bei vertės:

i	1	2	3	4	5
$\mathrm{size}[i]$	3	4	7	8	9
$\mathrm{val}[i]$	4	5	10	11	13
$\mathrm{name}[i]$	A	B	C	D	E

Kuprinės talpa yra 17. Pritaikę algoritmą, po 5 iteracijų gauname masyvus cost ir best, vaizduojamus 2.1 lentelėje. Ši lentelė rodo, kad paskutinis daiktas turi būti rūšies C. Kadangi jo dydis

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
									$j = 1$								
$\mathrm{cost}[i]$	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
$\mathrm{best}[i]$			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
									$j = 2$								
$\mathrm{cost}[i]$	0	0	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
$\mathrm{best}[i]$			1	2	2	1	2	2	1	2	2	1	2	2	1	2	2
									$j = 3$								
$\mathrm{cost}[i]$	0	0	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
$\mathrm{best}[i]$			1	2	2	1	3	2	1	3	3	1	3	3	1	3	3
									$j = 4$								
$\mathrm{cost}[i]$	0	0	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
$\mathrm{best}[i]$			1	2	2	1	3	4	1	3	3	1	3	3	4	3	3
									$j = 5$								
$\mathrm{cost}[i]$	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
$\mathrm{best}[i]$			1	2	2	1	3	4	5	3	3	5	3	3	4	5	3
$\mathrm{name}[i]$			A	B	B	A	C	D	E	C	C	E	C	C	D	E	C

Lentelė 2.1: Dinaminis kuprinės pakavimas sveikaskaitiniu atveju.

yra 7, o $\mathrm{name}[17-7] = \mathrm{C}$, tai priešpaskutinis daiktas taip pat buvo rūšies C. Pagaliau priešprieš-paskutinis daiktas buvo rūšies A, nes $\mathrm{name}[3] = \mathrm{A}$. Taigi, jei kuprinės talpa yra 17, tai optimalus sprendimas yra dėti daiktus A, C, C, kurių bendra vertė yra 24.

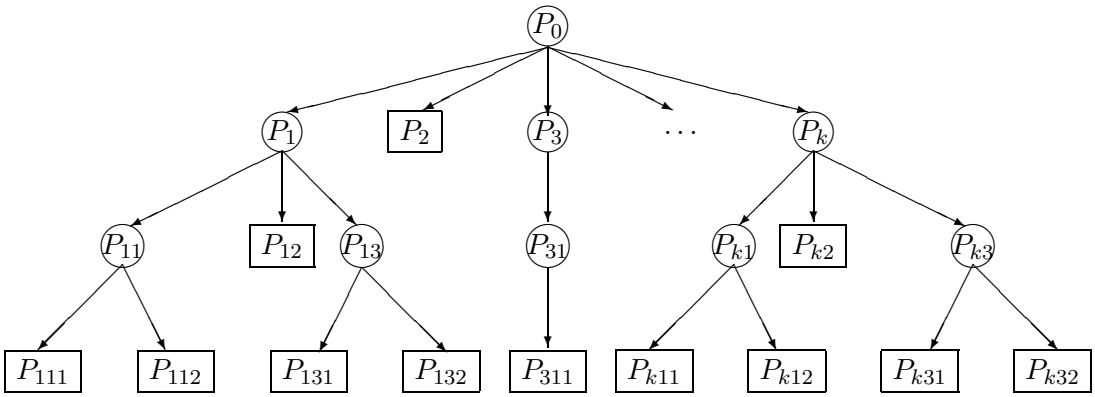
2.3 Paieška su grįžimu

Sprendžiant kai kuriuos kombinatorinius uždavinius du aukščiau išnagrinėti metodai (“skaldyk ir valdyk” bei dinaminis programavimas) gali būti nepritaikomi arba neefektyvūs. Tokiu atveju dažnai belieka perrinkti visus galimus uždavinio sprendinius ir išsirinkti tinkamą. Pilnas perrinkimas dar yra vadinamas *brutalios jėgos* (*brute force*, angl.) metodu.

Šiame skyrelyje panagrinėsime sprendinių perrinkimo metodą, kuris yra efektyvesnis už brutalios jėgos metodą. Naudojant šį metodą, blogiausiu atveju vis tiek tektų perrinkti visus variantus, tačiau vidutiniškai perrinkimas gaunasi mažesnis.

2.3.1 Sprendinių medis

Dažnai pradinį uždavinį P_0 galima suskaidyti į kelis dalinius uždavinius $P_1, \ldots, P_k$, kuriuos išsprendę gausime uždavinio P_0 sprendinį. Kiekvieną dalinį uždavinį P_i vėl galime suskaidyti į



Pav. 2.4: Sprendinių medis (neskaidūs uždaviniai pažymėti stačiakampiais).

mažesnius uždavinius $P_{il_1}, \ldots, P_{il_i}$ ir t.t. Uždavinį vadiname *neskaidžiu*, jei:

(a) galime lengvai rasti to uždavinio optimalų sprendinį;

(b) galime parodyti, kad to uždavinio optimalus sprendinys bus blogesnis už kitą, jau gautą, sprendinį;

(c) uždavinys yra *neleistinas*, t.y., jis neturi sprendinio.

Taigi, uždavinį P_0 atitinka medis, kurio šaknis yra pradinis uždavinys P_0, o lapai yra neskaidūs uždaviniai (žr. Pav. 2.4). Naudojant šį medį, iš kai kurių dalinių uždavinių sprendinių mes gauname pradinio uždavinio sprendinį. Todėl šis medis yra vadinamas *sprendinių medžiu*. Priklausomai nuo uždavinio, galima naudoti įvairias sprendinio paieškos sprendinių medyje strategijas. Pavyzdžiui, sprendžiant keliaujančio pirklio uždavinį arba ieškant išėjimo iš labirinto yra naudojama *paieška gylyn*. Ieškant grafo minimalaus karkaso arba trumpiausio kelio tarp dviejų grafo viršūnių yra naudojama *paieška platyn*. Brutalios jėgos algoritmas peržiūri visas sprendinių medžio viršūnes ir randa optimalų sprendinį. *Godus* algoritmas kiekvienoje viršūnėje renkasi lokaliai geriausią medžio briauną. Tokiu būdu godus algoritmas peržiūri tik vieną sprendinių medžio šaką ir gauna sprendinį, kuris nebūtinai yra optimalus. Paieška su grįžimu yra tarpinis metodas tarp šių dviejų kraštutinumų. Paieška su grįžimu peržiūri dalį sprendinių medžio ir randa optimalų sprendinį. Geriausiu atveju pakaks peržiūrėti vieną medžio šaką, blogiausiu atveju teks peržiūrėti visą sprendinių medį.

Pastaba 2.3.1. Sprendinių medis ir paieškos tokiame medyje efektyvumas priklauso nuo pasirinkto pradinio uždavinio skaidymo į mažesnius būdo. Pavyzdžiui, spręsdami keliaujančio pirklio uždavinį iš pirmo miesto, galime visus galimus sprendinius suskirstyti į $n-1$ grupę: sprendiniai, į kuriuos įeina briauna $(1,2)$, sprendiniai, į kuriuos įeina briauna $(1,3)$, $\ldots$, sprendiniai, į kuriuos įeina briauna $(1,n)$. Tačiau galimus maršrutus galima skaidyti ir į dvi grupes: sprendiniai, į kuriuos įeina briauna $(1,2)$, ir sprendiniai, į kuriuos ši briauna neįeina.

2.3.2 Paieškos su grįžimu algoritmas

Paieška su grįžimu — tai paieškos sprendinių medyje būdas, kurį galime taikyti kai sprendiniai tenkina tam tikras savybes.

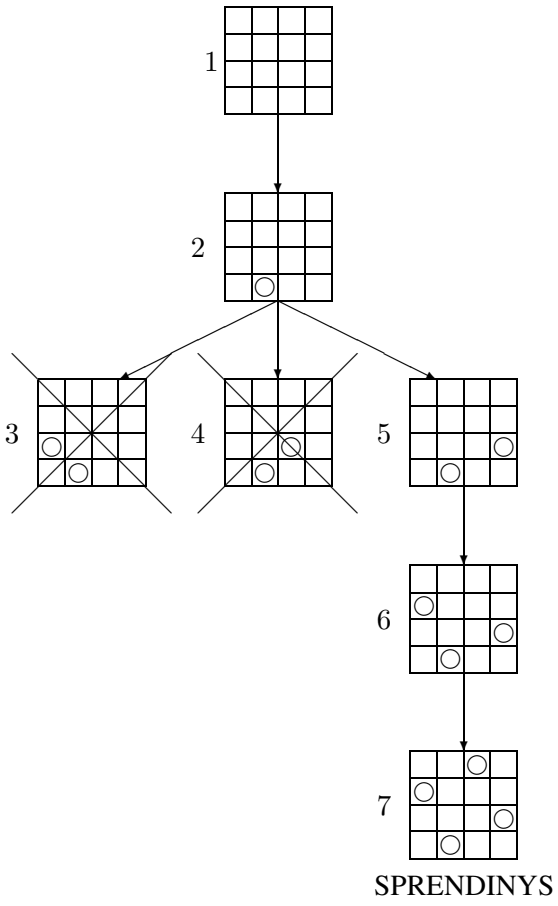
Tarkime, kad sprendinį galima užrašyti vektoriumi $(a_1, a_2, \ldots)$, kur $a_i \in A_i$ ir A_i yra pilnai sutvarkytos aibės. Pradėję nuo tuščio vektoriaus $(\,)$ ir pažymėję $S_1 \subseteq A_1$ aibę galimų kandidatų į a_1, imame $a_1 = \min S_1$ (t.y., pirmąjį aibės S_1 elementą) ir gauname dalinį sprendinį (a_1). Toliau nagrinėjame $S_2 \subseteq A_2$ ir t.t., kol randame neskaidaus dalinio uždavinio sprendinį $(a_1, \ldots, a_n)$ arba uždavinys tampa neleistinas. Jei sprendinys $(a_1, \ldots, a_n)$ nėra optimalus, grįžtame sprendinių medyje vienu lygiu aukštyn ir renkamės kitą kandidatą į a_n vietą. Jei perrinkus visus aibės S_n elementus, mes vis dar nerandame optimalaus sprendinio, tada grįžtame į $n-1$ lygį, renkamės naują kandidatą į a_{n-1} vietą ir vėl leidžiamės į lygį n. Šį paieškos su grįžimu metodą galima aprašyti tokia procedūra:

procedure $\mathrm{backtracking}(A_1, \ldots, A_n)$
$k := 1$;
$S_1 := \mathrm{geri}(A_1)$; /* Medžio šakų atkirtimas */
while $k > 0$ **do**
  while $(S_k \neq \emptyset$ **do**
    $a_k := \min(S_k)$;
    $S_k := S_k \setminus \{a_k\}$;
    if $((a_1, \ldots, a_k)$ yra leistinas galutinis sprendinys **then** save $(a_1, \ldots, a_k)$;
    $k := k + 1$;
    $S_k := \mathrm{geri}(A_k)$; /* Medžio šakų atkirtimas */
  end while
  $k := k - 1$; /* Grįžimas */
end while

2.3.3 n valdovių uždavinys

Turime šachmatų lentą $n \times n$ $(n > 1)$. Reikia joje sustatyti n valdovių taip, kad nė viena valdovė negrąsintų nė vienai kitai valdovei (valdovė grąsina visiems tos pačios vertikalės, kurioje ji stovi, laukeliams, o taip pat visiems tos pačios horizontalės ir visiems dviejų įstrižainių laukeliams). Atlikus pilną perrinkimą nesunku įsitikinti, kad kai $n = 2, 3$, uždavinys neturi sprendinio. Kai $n = 4$, gana nesunkiai rasime galimą sprendinį. Tačiau kai $n = 8$ (klasikinė šachmatų lenta), uždavinys jau tampa sunkiai įveikiamas be kompiuterio pagalbos.

Pabandykime 8 valdovių uždaviniui pritaikyti paieškos su grįžimu algoritmą. Brutalios jėgos metodas duoda $\mathrm{C}_{64}^{8} \approx 4,4 \cdot 10^9$ variantų. Akivaizdu, kad dvi valdovės negali stovėti vienoje horizontalėje bei vienoje vertikalėje. Tai reiškia, kad kiekvienoje vertikalėje ir kiekvienoje horizontalėje bus lygiai po 1 valdovę! Taigi, sprendinius galime vaizduoti vektoriais $(v_1, v_2, \ldots, v_8)$. Toks sprendinys reiškia, kad 1-oji valdovė stovi laukelyje $(1, v_1)$, 2-oji valdovė laukelyje $(2, v_2)$



Pav. 2.5: 4 valdovių uždavinys.

ir t.t. Be to, kadangi $v_i \neq v_j$, kiekvienas sprendinys yra skaičių $1, 2, \ldots, 8$ kėlinys. Taigi, lieka $8! = 40320$ galimų sprendinių, kuriuos galime pavaizduoti sprendinių medžiu (medis turės $8!$ lapų). To medžio šaknis bus tuščias sprendinys $()$, t.y. tuščia šachmatų lenta. Pirmoje horizontalėje galime pastatyti 1-ą valdovę į bet kurį laukelį $v_1 = 1, 2, \ldots, 8$. Kadangi radus poziciją, kur 8 valdovės negrąsina viena kitai, mes gauname, kad ir šiai pozicijai simetriškos lentos vidurio linijos atžvilgiu pozicijos (o taip pat pozicijos, gaunamos pasukus lentą 90, 180 arba 270 laipsnių kampu) tenkina šią savybę, tai pakanka nagrinėti 4 galimus kandidatus į v_1 vietą: $v_1 = 1, 2, 3, 4$. Kadangi lentos kampe (jų yra 4) gali stovėti tik viena valdovė, tai mes galime pirmos valdovės nestatyti į laukelį $v_1 = 1$, nes tokį sprendinį mes gausime pasukę reikiamu kampu poziciją, kur valdovė stovi kitame lentos kampe. Lieka 3 kandidatai į v_1 vietą, taigi sprendinių medžio lapų skaičius sumažėja iki $3 \cdot 7! = 15120$.

Tarkime, 1-ąją valdovę statome į laukelį $(1,2)$ (t.y., $v_1 = 2$). Bandydami 2-ą valdovę statyti į laukelius $(2,1)$ arba $(2,3)$, iš karto gauname neleistinus sprendinius. Lieka laukelis $(2,4)$. Tada 3-iai valdovei tinka laukelis $(3,1)$ ir t.t. Nukirsdami sprendinių medžio pomedžius su šaknimis $(2,1)$ ir $(2,3)$, mes atkirtome dvi dideles šakas su $2 \cdot 6! = 1440$ lapų. Taigi, paieška su grįžimu šį uždavinį sprendžia labai efektyviai.

Pav. 2.5 matome sprendinių medį, kuris gaunasi sprendžiant 4 valdovių uždavinį paieškos su grįžimu metodu. Skaičiai greta pozicijų žymi medžio viršūnių apėjimo tvarką. Naudojanti simetrijomis vidurio linijos atžvilgiu ir posūkiais, 1-ai valdovei lieka vienintelis laukelis $(1,2)$. Pilnas sprendinių medis turi $1+1+3+6+6 = 17$ viršūnių. Sprendinį gauname, peržiūrėję tik 7 viršūnes.

2.3.4 Aibių skaidiniai

Aibės $A = \{a_1, \ldots, a_n\}$ *denginiu* vadiname netuščią jos poaibių šeimą $\mathcal{B} = \{B_1, \ldots, B_k\}$ ($B_i \subseteq A$), dengiančią aibę A: $A \subseteq B_1 \cup \cdots \cup B_k$. Jei aibės B_i poromis nesikerta, t.y., $B_i \cap B_j = \emptyset$, tai šeimą $\mathcal{B}$ vadiname aibės A *skaidiniu*.

Minimalaus skaidinio uždavinys. Duotas aibės A denginys $\mathcal{B} = \{B_1, \ldots, B_k\}$ ir poaibių B_i kainos c_i. Išrinkti iš denginio $\mathcal{B}$ pigiausią aibės A skaidinį, t.y. rasti $\mathcal{B}_0 \subseteq \mathcal{B}$: $\mathcal{B}_0$ — aibės A skaidinys ir

$$\sum_{i:\ B_i \in \mathcal{B}_0} c_i \leqslant \sum_{i:\ B_i \in \mathcal{C}} c_i$$

kiekvienam A skaidiniui $\mathcal{C} \subseteq \mathcal{B}$.

Pavyzdys 2.3.1. Tegu $A = \{a,b,c,d,e,f\}$, $B_1 = \{a,b\}$, $B_2 = \{c,d\}$, $B_3 = \{e,f\}$, $B_4 = \{a,c,e\}$, $B_5 = \{a,c,f\}$, $B_6 = \{b,d,e\}$, $B_7 = \{b,d,f\}$, $\mathcal{B} = \{B_1, \ldots, B_7\}$, ir kiekvieno poaibio B_i kaina yra lygi 1. Nesunku įsitikinti, kad minimalus aibės A skaidinys bus $\mathcal{B}_0 = \{B_4, B_7\}$, kurio kaina yra lygi 2.

Minimalaus skaidinio uždavinys turi įvairius pritaikymus. Pateiksime tik vieną iš jų. Tarkime, kažkoks sandėlys ar didmeninės prekybos bazė aptarnauna n užsakovų, kuriems reikia pristatyti įvairius produktus, laikomus šiame sandėlyje. Kadangi tų produktų kiekiai nėra dideli, tai keliems užsakovams juos galime pristatyti viena mašina. Yra žinoma k tokių maršrutų, kai mašina išvyksta iš sandėlio, aplanko keletą užsakovų ir vėl grįžta į sandėlį. Šių maršrutų ilgiai yra $c_1, \ldots, c_k$. Reikia surasti, kiek mašinų ir kokiais maršrutais reikia pasiųsti, kad būtų aptarnauti visi užsakovai ir kad maršrutų ilgių suma būtų minimali. Pažymėję užsakovų aibę raide A: $A = \{u_1, \ldots, u_n\}$, kiekvieną maršrutą, aplankantį užsakovus $u_{i_1}, \ldots, u_{i_m}$, galime laikyti aibės A poaibiu $\{u_{i_1}, \ldots, u_{i_m}\}$, ir gauname minimalaus skaidinio uždavinį.

Kiekvieną n-aibės k-denginį atitinka dvejetainė matrica, turinti n eilučių ir k stulpelių. Šios matricos stulpeliai yra poaibių B_j charakteringieji vektoriai $\kappa(B_j) \in \{0,1\}^n$, tokie, kad

$$\kappa_i = \begin{cases} 1, & \text{jei } a_i \in B_j, \\ 0, & \text{jei } a_i \notin B_j. \end{cases}$$

Pavyzdžiui, aukščiau pateikto pavyzdžio 2.3.1 denginį $\mathcal{B}$ atitinka matrica

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}.$$

Denginį $\mathcal{B}$ atitinkančiai dvejetainei matricai minimalaus skaidinio uždavinys skamba taip: išrinkti mažiausią skaičių stulpelių tokių, kad kiekvienai eilutei $i = 1, \ldots, n$ lygiai vieno iš šių stulpelių i-asis elementas būtų lygus 1 (o kituose stulpeliuose šioje eilutėje stovėtų visi nuliai).

Taikysime minimalaus skaidinio uždaviniui paiešką su grįžimu. Pirmiausia denginį $\mathcal{B}$ atitinkančios matricos stulpelius suskaidome į n blokų (po vieną kiekvienam aibės A elementui) taip, kad i-ojo bloko stulpelius atitinkantys poaibiai turėtų elementą a_i, bet neturėtų nė vieno iš elementų $a_1, \ldots, a_{i-1}$. Kai kurie iš šių blokų gali būti tušti. Bloko viduje stulpelius išrikiuojame juos atitinkančių poaibių kainų augimo tvarka. Po šių matricos pertvarkymų poaibius B_i pernumeruojame tvarka, atitinkančia naują stulpelių tvarką. Tarkime, kad denginys $\mathcal{B} = \{S_1, \ldots, S_k\}$, kur $\{S_1, \ldots, S_k\}$ yra poaibiai $\{B_1, \ldots, B_k\}$ išdėstyti nauja tvarka.

Pavyzdžiui, aukščiau pateiktą pavyzdį 2.3.1 atitinka pertvarkyta matrica

$$M = \left(\begin{array}{ccc|cc|c|c} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}\right),$$

sudaryta iš blokų M_1, M_2, M_3, M_5. Bloką M_1 sudaro stulpeliai, atitinkantys poaibius $S_1 = B_1$, $S_2 = B_4$ ir $S_3 = B_5$, bloką M_2 sudaro stulpeliai, atitinkantys poaibius $S_4 = B_6$ ir $S_5 = B_7$, bloką M_3 sudaro stulpelis, atitinkantis poaibį $S_6 = B_2$, ir bloką M_5 sudaro stulpelis, atitinkantis poaibį $S_7 = B_3$.

Pažymėkime dalinį minimalaus aibės skaidinio uždavinio sprendinį tam tikru algoritmo vykdymo momentu $\mathcal{S}$, jo kainą — c, geriausią aibės A skaidinį, rastą iki šio momento, — $\widehat{\mathcal{S}}$, o jo kainą — $\widehat{c}$. Aibė $E \subseteq A$ bus aibė visų poaibių $S_i \in \mathcal{S}$ padengtų aibės A elementų.

1. Konstruojame blokinę matricą M ir priskiriame pradines reikšmes:
   $\mathcal{S} := \emptyset$; $E := \emptyset$; $c := 0$; $\widehat{c} := \infty$.

2. $p := \min\{i \mid r_i \notin E\}$;
   Aibę S_1^p pažymime žyme, rodančia, kad ši aibė yra panaudota.

3. Pradedant nuo pažymėtos aibės bloke p, nagrinėjame aibes S_i^p jų indekso i didėjimo tvarka.

(a) Jei randame aibę S_j^p tokią, kad $S_j^p \cap E = \emptyset$ ir $c + c_j^p < \widehat{c}$ (kur c_j^p yra aibės S_j^p kaina), tai pereiname į žingsnį 5.

(b) Priešingu atveju (t.y. jei blokas p užsibaigė arba radome aibę S_j^p tokią, kad $c + c_j^p \geqslant \widehat{c}$), pereiname į žingsnį 4.

4. Dalinio sprendinio $\mathcal{S}$ pagerinti nebegalima. Jei $\mathcal{S} = \emptyset$ (t.y. baigėsi 1-asis blokas), tai STOP; geriausias sprendinys yra $\widehat{\mathcal{S}}$.
Priešingu atveju pašaliname iš sprendinio $\mathcal{S}$ paskutinę aibę S_k^l, pakeičiame jo kainą: $c := c - c_k^l$, pašaliname aibės S_k^l elementus iš E: $E := E \setminus S_k^l$, pažymime aibę S_{k+1}^l, pašaliname ankstesnę žymę bloke $p := l$ ir pereiname į žingsnį 3.

5. Papildome sprendinį: $\mathcal{S} := \mathcal{S} \cup \{S_j^p\},\ E := E \cup S_j^p,\ c := c + c_j^p$.
Jei rastas pilnas sprendinys, t.y. $E = A$, tada atnaujiname geriausias reikšmes $\widehat{\mathcal{S}} := \mathcal{S}$, $\widehat{c} := c$ ir pereiname į žingsnį 4.
Priešingu atveju pereiname į žingsnį 2.

Pastaba 2.3.2. Jei šio algoritmo 4-ame žingsnyje aibė S_k^l bloke l buvo paskutinė, tai algoritmas grįžęs į žingsnį 3 neranda aibės S_{k+1}^l ir vėl pereina į žingsnį 4, t.y., iš sprendinio $\mathcal{S}$ pašalina dar vieną aibę.

Pritaikysime paieškos su grįžimu algoritmą pavyzdžiui 2.3.1. Gauname:
$\mathcal{S} := \emptyset,\ E := \emptyset,\ c := 0,\ \widehat{c} := \infty$;
$p := 1,\ \mathcal{S} := \{S_1\},\ E := \{a, b\},\ c := 1$;
$p := 3,\ \mathcal{S} := \{S_1, S_6\},\ E := \{a, b, c, d\},\ c := 2$;
$p := 5, \mathcal{S} := \{S_1, S_6, S_7\},\ E := \{a, b, c, d, e, f\} = A,\ c := 3,\ \widehat{\mathcal{S}} := \{S_1, S_6, S_7\},\ \widehat{c} := 3$;
$\mathcal{S} := \{S_1, S_6\},\ c := 2,\ E := \{a, b, c, d\},\ p := 3$;
$\mathcal{S} := \{S_1\},\ c := 1,\ E := \{a, b\},\ p := 1$;
$\mathcal{S} := \emptyset,\ c := 0,\ E := \emptyset$;
$\mathcal{S} := \{S_2\},\ E := \{a, c, e\},\ c := 1$;
$p := 2,\ \mathcal{S} := \{S_2, S_5\},\ E := \{a, b, c, d, e, f\} = A,\ c := 2,\ \widehat{\mathcal{S}} := \{S_2, S_5\},\ \widehat{c} := 2$;
$\mathcal{S} := \{S_2\},\ c := 1,\ E := \{a, c, e\},\ p := 1$;
$\mathcal{S} := \emptyset,\ c := 0,\ E := \emptyset$;
$\mathcal{S} := \{S_3\},\ E := \{a, c, f\},\ c := 1$;
$\mathcal{S} := \emptyset,\ c := 0,\ E := \emptyset$; STOP.
Taigi, optimalus sprendinys yra $\widehat{\mathcal{S}} := \{S_2, S_5\}$, kurio kaina lygi 2.

2.4 Šakų ir rėžių metodas

Šakų ir rėžių metodas — tai paieška su grįžimu, kai mes optimizuojame *tikslo funkciją* $Cost$, t.y., sprendžiame optimizavimo uždavinį $Cost \to \min$ ir mokame sprendinių paieškos medžio

pomedžiuose gaunamų sprendinių kainą iš anksto aprėžti iš apačios *aprėžiančios funkcijos* $Bound$ pagalba. Taigi, šakų ir rėžių metodą charakterizuoja 4 požymiai:

1. Sprendiniams $S = \{a_1, \ldots, a_k\}$ yra apibrėžta jų kaina $Cost$, turinti šias savybes:
   - $Cost \geqslant 0$;
   - $Cost(a_1, \ldots, a_{k-1}) \leqslant Cost(a_1, \ldots, a_k)$, pavyzdžiui, dažnai funkcija $Cost$ tenkina lygybę $Cost(a_1, \ldots, a_k) = Cost(a_1, \ldots, a_{k-1}) + Cost(a_k)$.
2. Sprendiniams $S = \{a_1, \ldots, a_k\}$ yra apibrėžtas jų kainos apatinis rėžis $Bound$, turintis šias savybes:
   - $Bound(a_1, \ldots, a_k) \geqslant Cost(a_1, \ldots, a_k)$ vidinėms sprendinių medžio viršūnėms, t.y. daliniams (dar ne pilniems) sprendiniams ir
   - $Bound(a_1, \ldots, a_k) = Cost(a_1, \ldots, a_k)$ medžio lapams, t.y., galutiniams sprendiniams.
3. Yra pateiktas sprendinių paieškos išsišakojimo į naujas šakas būdas, t.y., aibių A_i, naudojamų paieškos su grįžimu algoritme, parinkimo būdas.
4. Yra pasirinkta kokia nors medžio viršūnių perrinkimo strategija. Galimos strategijos yra DFS (paieška gylyn), BFS (paieška platyn), BeFS (geriausios viršūnės prioritetinio pasirinkimo strategija) ir kitos. Strategija BeFS šakų ir rėžių metode rekomenduoja kiekvieną kartą rinktis tą medžio viršūnę, kurios rėžis yra mažiausias iš dar nenagrinėtų medžio viršūnių.

Žinoma, šakų ir rėžių metodas tinka ir tiems uždaviniams, kur reikia rasti maksimalios vertės sprendinį ($Cost \to \max$), tik tada reikia naudoti viršutinius rėžius ir rinktis viršūnę su didžiausiu rėžiu.

Pateiksime bendrą šakų ir rėžių metodo algoritmą, laikydami, kad naudojamės mišria DFS–BeFS strategija, t.y., sprendinių medyje vykdome paiešką gylyn, tik pasirenkant kurią nors iš k galimų šakų renkamės ne iš eilės, o pagal mažiausią apatinį rėžį.

procedure $\mathrm{branch\&bound}(A_1, \ldots, A_n)$
$MinCost := \infty$;
$Cost := 0$;
$k := 1$;
for $a \in A_1$ **do** find $Bound(a)$;
$S_1 := A_1$;
while $k > 0$ **do**
  while $(S_k \neq \emptyset$ **and** $Cost < MinCost)$ **do**
    $a_k := a \in S_k$: $Bound(a_1, \ldots, a_{k-1}, a) = \min_{b \in S_k} Bound(a_1, \ldots, a_{k-1}, b)$;
    $S_k := S_k \setminus \{a_k\}$;

$Cost := Cost(a_1, \ldots, a_k)$;
if $((a_1, \ldots, a_k)$ yra leistinas galutinis sprendinys **and** $Cost < MinCost)$ **then**
$MinCost := Cost$; save $(a_1, \ldots, a_k)$ **end**;
$k := k + 1$;
for $a \in A_k$ **do** find $Bound(a_1, \ldots, a_{k-1}, a)$;
$S_k := \{a \in A_k$: $Bound(a_1, \ldots, a_{k-1}, a) < MinCost\}$;
end while
$k := k - 1$;
$Cost := Cost(a_1, \ldots, a_k)$;
end while

Pademonstruosime šakų ir rėžių metodo taikymą keliaujančio pirklio ir darbų paskirstymo uždaviniams.

2.4.1 Keliaujančio pirklio uždavinys (KPU)

Keliaujančio pirklio uždavinys (KPU) (angl. TSP — traveling salesman problem). Duota n miestų $\{1, 2, \ldots, n\}$ ir atstumų tarp tų miestų matrica $C = (C[i, j])$, kur $C[i, j] = \mathrm{dist}(i, j) \geqslant 0$. Reikia rasti trumpiausią maršrutą per visus miestus, kuris per kiekvieną miestą praeina lygiai vieną kartą. Tokį uždavinį tenka spręsti keliaujančiam pirkliui, kuris nori aplankyti keletą miestų, parduoti ten savo prekes ir grįžti į pradinį miestą, kuriame jis gyvena. Tai bene labiausiai išgarsėjęs kombinatorinis uždavinys, kuriam daug metų buvo bandoma surasti polinominio sudėtingumo algoritmą arba įrodyti, kad tokio algoritmo neegzistuoja. Deja, niekam nepavyko padaryti nei viena, nei antra.

Šį uždavinį spręsime bendriausiu atveju, laikydami, kad atstumų matrica nėra simetrinė, ir atstumai netenkina trikampio taisyklės $C[i, j] \leqslant C[i, k] + C[k, j]$. Tokia situacija, kai atstumas iš miesto i į miestą j skiriasi nuo atstumo iš miesto j į miestą i, praktiškai gali pasitaikyti dėl vienos krypties kelių ir kitų priežasčių. Taigi, perfrazuojant KPU uždavinį grafų kalba, mums reikia orientuotame svoriniame grafe rasti trumpiausią Hamiltono ciklą. Fiksuojant pradinį miestą, kuriame gyvena mūsų pirklys, gauname $(n-1)!$ skirtingų Hamiltono ciklų (iš pirmo miesto galima eiti į bet kurį iš miestų $2, 3, \ldots, n$, iš kiekvieno iš šių miestų galima eiti į bet kurį iš likusių $n - 2$ miestų ir t.t.). Taigi, sprendžiant šį uždavinį brutalios jėgos algoritmu, sudėtingumas būtų $L(n) = \mathrm{O}(n!)$ (kai kiekvienas miestas su kiekvienu kitu miestu yra sujungti keliais, neinančiais per kitus miestus), nes maršrutų skaičių dar reikėtų dauginti iš $\mathrm{O}(n)$ operacijų, reikalingų 1 maršruto ilgiui apskaičiuoti.

Taikysime KPU uždaviniui šakų ir rėžių metodą:

1. Maršruto $M = i_1 \to i_2 \to \cdots \to i_k$ kaina $Cost(i_1, \ldots, i_k) = C[i_1, i_2] + C[i_2, i_3] + \cdots + C[i_{k-1}, i_k]$ yra neneigiama funkcija ir tenkina nelygybę

$$Cost(i_1, \ldots, i_{k-1}) \leqslant Cost(i_1, \ldots, i_k) = Cost(i_1, \ldots, i_{k-1}) + C[i_{k-1}, i_k].$$

2. Kadangi iš kiekvieno miesto turime kažkuriuo keliu išeiti, tai kiekvienas miestas i į optimalaus maršruto kainą įneš indėlį, ne mažesnį už trumpiausio kelio iš miesto i ilgį $\min_{j \neq i} C[i,j]$. Taip gauname preliminarų apatinį rėžį bet kurio maršruto ilgiui:

$$Cost(M) \geqslant \sum_{i=1}^{n} \min_{j \neq i} C[i,j].$$

Tačiau mes turime ne tik išeiti iš kiekvieno miesto, bet turime ir pro kažkur į jį ateiti. Todėl mes turime įvertinti ir įeinančių kelių ilgius. Kadangi bet kuris kelias $i \to j$ miestui j yra įeinantis, o miestui i išeinantis, tai mes negalime iš karto įvertinti įeinančių kelių indėlio per matricos C stulpelių minimumus. Pirmiausia reikia matricą C *suprastinti*, iš kiekvieno jos elemento atimant eilutės, kurioje stovi tas elementas, mažiausią elementą (kad mums netrukdytų nuliai, atstumą iš bet kurio miesto į save mes laikome begaliniu: $C[i,i] = \infty \ \forall i = 1, \ldots, n$). Suprastinę, gauname matricą C'. Dabar jau galime rasti apatinį įvertį, kuris tinka bet kuriam maršrutui. Kadangi jokio maršruto dar neturime, tai yra sprendinys $(a_1, \ldots, a_k)$ (žr. branch_and_bound algoritmą) dar yra tuščias, tai šį rėžį žymėsime $Bound(\emptyset)$:

$$Bound(\emptyset) = \sum_{i=1}^{n} \min_{j} C[i,j] + \sum_{j=1}^{n} \min_{i} C'[i,j].$$

Vėliau, kai maršrutas ilgės, atstumų matricoje vėl atsiras kelių, kurių ilgį reikės įskaityti į ieškomo maršruto ilgį, ir apatinis rėžis tam maršrutui augs.

3. Sprendinių medį šakosime atsižvelgiant į tai, ar mes į ieškomą maršrutą įtraukiame konkretų kelią $i \to j$, ar ne. Taigi, pirmame sprendinių medžio lygyje visi galimi maršrutai pasidalins į dvi grupes: maršrutai, į kuriuos įtrauktas pasirinktas kelias $i \to j$ ir maršrutai, kuriuose nėra šio kelio. Kiekviena grupė vėl dalinsis į dvi grupes, priklausomai nuo to, ar į maršrutą įtrauksime kažkokį kitą kelią ir t.t.

   Apibrėšime taisyklę, pagal kurią mes renkamės šakojimui naudojamą kelią $i \to j$. Suprastinus pradinę atstumų matricą C pagal eilutes ir stulpelius, gauname matricą C'', kuri kiekvienoje eilutėje ir kiekviename stulpelyje turi nulinių elementų. Tarkime, $C[i,j]$ yra toks elementas. Tada tuose maršrutuose, kuriuose nebus kelio $i \to j$, būtinai turės būti du keliai $i \to k$ ir $l \to j$, kur $k \neq j$ ir $l \neq i$. Todėl visiems tokiems maršrutams jų apatinis rėžis dar padidės dydžiu

$$D[i,j] = \min_{k \neq j} C''[i,k] + \min_{k \neq i} C''[k,j].$$

   Kadangi mes stengiamės atkirsti kuo daugiau pomedžių, tai siekiame, kad apatiniai rėžiai būtų kuo didesni. Taigi, šakojimui naudojamo kelio pasirinkimo taisyklė yra tokia: imame

porą (i, j), tokią, kad $C''[i, j] = 0$ ir

$$D[i, j] = \max_{k,l:C''[k,l]=0} D[k, l].$$

4. Naudosime BeFS strategiją: iš visų dar nenagrinėtų, bet jau turinčių rėžius, sprendinių medžio viršūnių mes rinksimės viršūnę su mažiausiu rėžiu.

Pavyzdys 2.4.1. Duota atstumų tarp miestų matrica

$$C = \begin{pmatrix} \infty & 25 & 40 & 31 & 27 \\ 5 & \infty & 17 & 30 & 25 \\ 19 & 15 & \infty & 6 & 1 \\ 9 & 50 & 24 & \infty & 6 \\ 22 & 8 & 7 & 10 & \infty \end{pmatrix}.$$

Rasti trumpiausią Hamiltono ciklą M_{opt}.

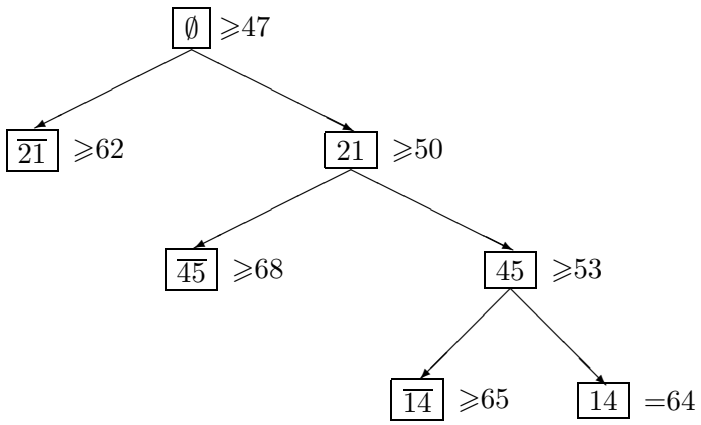
Pirmiausia prastiname matricą:

$$C = \left(\begin{array}{ccccc} \infty & 25 & 40 & 31 & 27 \\ 5 & \infty & 17 & 30 & 25 \\ 19 & 15 & \infty & 6 & 1 \\ 9 & 50 & 24 & \infty & 6 \\ 22 & 8 & 7 & 10 & \infty \end{array}\right| \begin{array}{c} -25 \\ -5 \\ -1 \\ -6 \\ -7 \end{array} \longrightarrow C' = \begin{pmatrix} \infty & 0 & 15 & 6 & 2 \\ 0 & \infty & 12 & 25 & 20 \\ 18 & 14 & \infty & 5 & 0 \\ 3 & 44 & 18 & \infty & 0 \\ 15 & 1 & 0 & \underline{3} & \infty \\ & & & \underline{-3} & \end{pmatrix}$$

$$\longrightarrow C'' = \begin{pmatrix} \infty & 0 & 15 & 3 & 2 \\ 0 & \infty & 12 & 22 & 20 \\ 18 & 14 & \infty & 2 & 0 \\ 3 & 44 & 18 & \infty & 0 \\ 15 & 1 & 0 & 0 & \infty \end{pmatrix}.$$

Gavome apatinį rėžį $Bound(\emptyset) = 47$. Norėdami išsirinkti optimalų kelią šakojimui, matricos C'' nuliniams elementams apskaičiuojame rėžio pokyčius $D[i, j]$: $D[1, 2] = 3$, $D[2, 1] = 15$, $D[3, 5] = 2$, $D[4, 5] = 3$, $D[5, 3] = 13$, $D[5, 4] = 2$. Taigi, visus maršrutus skaidome į dvi grupes: (1) maršrutai, į kuriuos įeina kelias $2 \to 1$, ir (2) maršrutai, į kuriuos šis kelias neįeina. Atitinkamas sprendinių medžio viršūnes pažymime 21 ir $\overline{21}$ (žr. pav. 2.6).

Viršūnė $\overline{21}$ gauna rėžį $Bound(\overline{21}) = Bound(\emptyset) + D[2, 1] = 62$. Apskaičiuosime viršūnės 21 rėžį. Kadangi kelias $2 \to 1$ tikrai priklauso visiems šio pomedžio maršrutams, galime išbraukti matricos C antrą eilutę ir pirmą stulpelį. Gauname matricą C_1 (šalia eilučių ir stulpelių rašome



Pav. 2.6: Sprendinių medis, gaunamas taikant KPU uždaviniui šakų ir rėžių metodą.

likusių miestų žymes, kurias toliau naudosime matricos elementų indeksavimui):

$$C_1 = \begin{array}{c} \\ 1 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccc} 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} \infty & 15 & 3 & 2 \\ 14 & \infty & 2 & 0 \\ 44 & 18 & \infty & 0 \\ 1 & 0 & 0 & \infty \end{pmatrix} \end{array}.$$

Matricos C_1 elementas $C_1[1,2] = \infty$, nes jis atitinka kelią $1 \to 2$, kurio nebegalima naudoti, nes jau buvo panaudotas kelias $2 \to 1$. Atėmę iš pirmos eilutės 2 ir iš antro stulpelio 1, randame suprastintą matricą

$$C_1'' = \begin{array}{c} \\ 1 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccc} 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} \infty & 13 & 1 & 0 \\ 13 & \infty & 2 & 0 \\ 43 & 18 & \infty & 0 \\ 0 & 0 & 0 & \infty \end{pmatrix} \end{array}$$

ir viršūnės 21 rėžį $47 + 3 = 50$. Iš matricos C_1'' randame, kad toliau sprendinių medį šakojame, naudodami kelią $4 \to 5$. Viršūnė $\overline{45}$ gauna rėžį $50 + D[4,5] = 68$. Išbraukę matricos C_1'' ketvirtą eilutę ir penktą stulpelį, randame naują matricą C_2, ją prastiname:

$$C_2 = \begin{array}{c} \\ 1 \\ 3 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccc} 2 & 3 & 4 \end{array} \\ \begin{pmatrix} \infty & 13 & 1 \\ 13 & \infty & 2 \\ 0 & 0 & 0 \end{pmatrix} \end{array} \longrightarrow C_2'' = \begin{array}{c} \\ 1 \\ 3 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccc} 2 & 3 & 4 \end{array} \\ \begin{pmatrix} \infty & 12 & 0 \\ 11 & \infty & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

ir randame viršūnės 45 rėžį $50 + 3 = 53$. Toliau šakojame, naudodami kelią $1 \to 4$. Viršūnė $\overline{14}$ gauna rėžį $53 + D[1,4] = 65$. Išbraukę matricos C_2'' pirmą eilutę ir ketvirtą stulpelį, gauname matricą

$$C_3 = \begin{array}{c} \\ 3 \\ 5 \end{array} \begin{array}{c} \begin{array}{cc} 2 & 3 \end{array} \\ \begin{pmatrix} 11 & \infty \\ 0 & 0 \end{pmatrix} \end{array},$$

iš kurios matyti, kad iš miesto 3 privalome eiti į miestą 2, o iš miesto 5 tada lieka eiti tik į miestą 3. Kadangi radome gauto trivialaus dalinio uždavinio sprendinį, tai viršūnė 14 yra nebeskaidoma, t.y., ji yra sprendinių medžio lapas. Mūsų rastas maršruto $M_1 = 2 \to 1 \to 4 \to 5 \to 3 \to 2$ ilgis yra $Cost(M_1) = 53 + C_3[3,2] = 64$.

Kadangi $Bound(\overline{45}) > 64$ ir $Bound(\overline{14}) > 64$, tai atkertame sprendinių medžio pomedžius su šaknimis $\overline{45}$ ir $\overline{14}$. Liko išnagrinėti pomedį su šaknimi $\overline{21}$. Tai paliekame kaip užduotį skaitytojui. Šiame pomedyje ir slepiasi optimalus maršrutas $M_{\text{opt}} = 2 \to 3 \to 5 \to 4 \to 1 \to 2$, kurio ilgis yra 62.

Koks šakų ir rėžių metodo sudėtingumas KPU uždaviniui? Aišku, tai priklauso nuo duomenų ir blogiausiu atveju gali tekti perrinkti visus $(n-1)!$ maršrutų. Eksperimentiškai buvo apskaičiuota, kad atsitiktinei atstumų tarp miestų matricai vidutinis sudėtingumas yra $\overline{L}^{\text{KPU}}_{\text{B\&B}}(n) = \text{O}(1.26^n)$ (žr. [RND, p. 145]).

2.4.2 Darbų paskirstymo uždavinys (DPU)

Darbų paskirstymo uždavinys (DPU) (angl. JAP — job assignment problem). Duota n asmenų $A = \{a_1, a_2, \ldots, a_n\}$ ir n darbų $D = \{d_1, d_2, \ldots, d_n\}$ bei darbų kainos matrica C, kurios elementai $C[i,j]$ nurodo sumą, kurią reikės sumokėti asmeniui a_i, jei jam paskirsime darbą d_j. Reikia rasti optimalų darbų paskirstymą, t.y., tokią bijekciją $f : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$, kurios kaina

$$Cost(f) = \sum_{i=1}^{n} C[i, f(i)] = \min_{g: I \to I \text{ bijekcija}} Cost(g),$$

kur indeksų aibę $\{1, 2, \ldots, n\}$ pažymėjome I. Kitaip sakant, kiekvienai matricos C eilutei i reikia priskirti vienintelį matricos C stulpelį $f(i)$, taip, kad skirtingoms eilutėms būtų priskirti skirtingi stulpeliai ir tokio priskyrimo kaina būtų mažiausia. Aišku, kad visų galimų tokių bijekcijų yra $n!$, taigi brutalios jėgos algoritmas atliktų $L(n) = \text{O}(n \cdot n!)$ operacijų. Patikrinsime, ar šis uždavinys tenkina savybes, reikalingas norint taikyti šakų ir rėžių metodą:

1. Daliniai šio uždavinio sprendiniai bus injekcijos $\{1, 2, \ldots, k\} \to \{1, 2, \ldots, n\}$, kurias žymėsime $J = (j_1, j_2, \ldots, j_k)$, kur j_i reiškia stulpelio numerį, priskirtą eilutei i. Tokių injekcijų kaina $Cost(J) = \sum_{i=1}^{k} C[i, j_i]$ yra neneigiama funkcija ir tenkina nelygybę

$$Cost(j_1, \ldots, j_{k-1}) \leqslant Cost(j_1, \ldots, j_k) = Cost(j_1, \ldots, j_{k-1}) + C[j_{k-1}, j_k].$$

2. Panagrinėsime dvi aprėžiančias funkcijas, kurias galima taikyti šiam uždaviniui:

$$Bound_1(j_1, \ldots, j_k) = Cost(j_1, \ldots, j_k) + \sum_{i=k+1}^{n} \min_j C[i, j]$$

ir

$$Bound_2(j_1, \ldots, j_k) = Cost(j_1, \ldots, j_k) + \sum_{i=k+1}^{n} \min_{j \notin J} C[i, j],$$

kur pirmasis rėžis grindžiamas tuo, kad priskyrus darbus pirmiesiems k asmenų, likusiems $n - k$ asmenų taip pat turėsime priskirti darbus, ir tai mums mažiausiai kainuos tiek, kiek gautųsi kiekvienam asmeniui priskirus pigiausią darbą. Antrasis rėžis gaunamas panašiai, tik iš visų galimų darbų jau atmetame pirmųjų k asmenų užimtus darbus su numeriais iš aibės $J = (j_1, \ldots, j_k)$. Akivaizdu, kad šios aprėžiančios funkcijos tenkina nelygybes

$$Bound_2(j_1, \ldots, j_k) \geqslant Bound_1(j_1, \ldots, j_k) \geqslant Cost(j_1, \ldots, j_k).$$

3. Sprendinių medį lygyje $k = 1, 2, \ldots, n - 1$ šakosime pagal tai, kurį iš likusių $n + 1 - k$ darbų mes priskirsime darbininkui su numeriu k. Sprendinių medžio viršūnes žymėsime $j_k|C[k, j_k]$, kur $j_k \in I \setminus \{j_1, \ldots, j_{k-1}\}$ — k-ajam darbininkui priskiriamo darbo numeris.

4. Naudosime mišrią DFS–BeFS strategiją: sprendinių medyje vykdome paiešką gylyn, o lygyje k pasirenkant kurią nors iš $n + 1 - k$ galimų šakų renkamės ne iš eilės, o pagal mažiausią apatinį rėžį.

Pavyzdys 2.4.2. Duota darbų kainų matrica

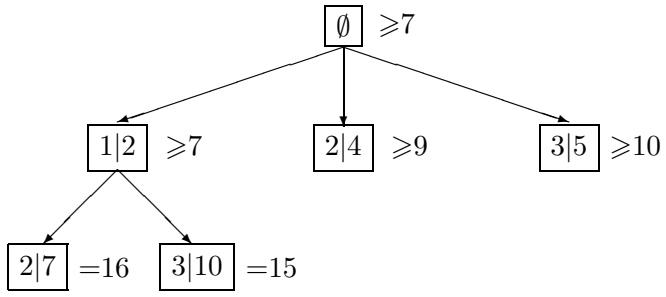
$$C = \begin{pmatrix} 2 & 4 & 5 \\ 2 & 7 & 10 \\ 5 & 3 & 7 \end{pmatrix}.$$

Rasti optimalų darbų paskirstymą $J_{\text{opt}} = (j_1, j_2, j_3)$.

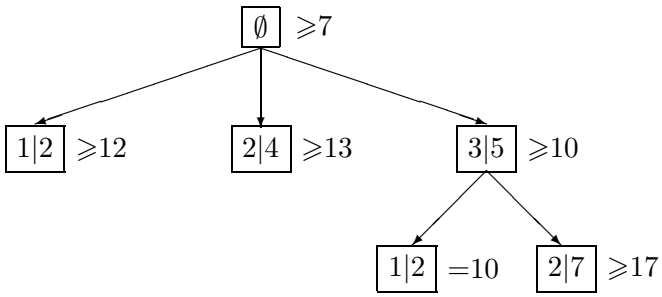
1. Sprendžiame duotą uždavinį, taikydami pirmąją aprėžiančią funkciją $Bound_1$. Turime:

$$Bound_1(\emptyset) = \sum_{i=1}^{3} \min_j C[i, j] = 7.$$

Pirmame sprendinių medžio lygyje turime 3 viršūnes $1|2$, $2|4$ ir $3|5$ su rėžiais atitinkamai 7, 9 ir 10 (žr. 2.7 pav.). Pagal BeFS strategiją renkamės viršūnę $1|2$. Ši viršūnė turės du pomedžius su šaknimis $2|7$ ir $3|10$, kurių rėžiai bus 16 ir 15. Pagal BeFS strategiją renkamės viršūnę $3|10$ ir joje gauname pirmą sprendinį $J_1 = (1, 2, 3)$, kurio kaina yra $Cost(J_1) = 15$. Kadangi viršūnės $2|7$ rėžis yra didesnis už 15, tai ją atkertame. Toliau grįžtame į pirmą lygį, renkamės viršūnę $2|4$



Pav. 2.7: Sprendinių medžio, gaunamo naudojant DPU uždaviniui aprėžiančią funkciją $Bound_1$, fragmentas.



Pav. 2.8: Sprendinių medis, gaunamas naudojant DPU uždaviniui aprėžiančią funkciją $Bound_2$.

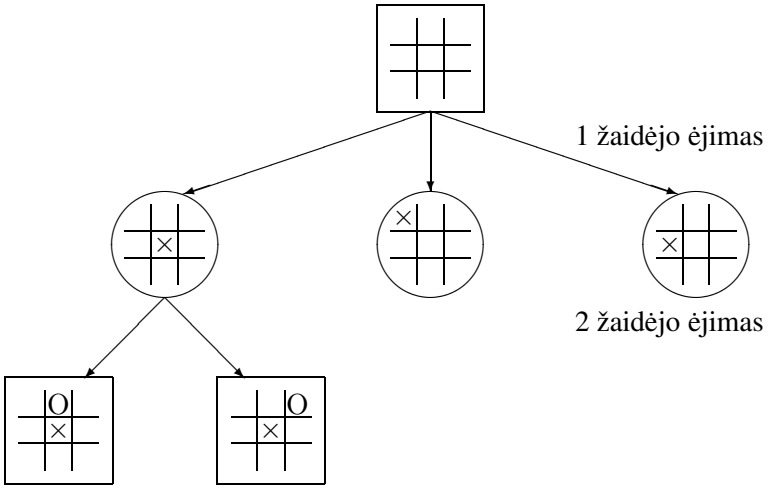
ir t.t. Paliekame skaitytojui kaip užduotį įsitikinti, kad šiam uždaviniui teks peržiūrėti beveik visą sprendinių medį, kol rasime optimalų darbų paskirstymą J_{opt}.

2. Naudojant antrąją aprėžiančią funkciją $Bound_2$, turime:

$$Bound_2(\emptyset) = \sum_{i=1}^{3} \min_j C[i,j] = 7.$$

Dabar pirmame sprendinių medžio lygyje turime 3 viršūnes $1|2$, $2|4$ ir $3|5$ su rėžiais atitinkamai 12, 13 ir 10 (žr. 2.8 pav.), todėl renkamės viršūnę $3|5$. Ši viršūnė turės du pomedžius su šaknimis $1|2$ ir $2|3$, kurių rėžiai bus 10 ir 17. Taigi, renkamės viršūnę $1|2$ ir joje gauname pirmą sprendinį $J_1 = (3,1,2)$, kurio kaina yra $Cost(J_1) = 10$. Kadangi rasto darbų paskirstymo kaina yra mažesnė už visų šiuo metu turimų medžio viršūnių rėžius, tai paieška baigta: pats pirmasis rastas sprendinys ir bus optimalus: $J_{\text{opt}} = J_1 = (3,1,2)$.

Šis pavyzdys parodo, kaip svarbu yra parinkti gerą aprėžiančią funkciją, kad ji leistų atkirsti kuo didesnę sprendinių medžio dalį.



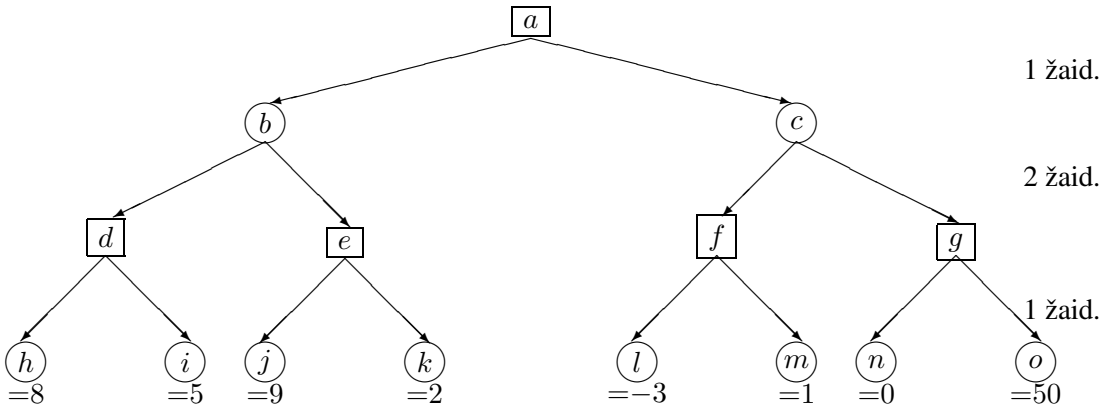
Pav. 2.9: Žaidimo kryžiukai–nuliukai medžio fragmentas.

2.5 Paieška žaidimų medžiuose

Nagrinėsime žaidimus, kuriuos žaidžia du žaidėjai, pradėdami iš pradinės pozicijos ir paeiliui darydami po 1 ėjimą. Abu žaidėjai mato vienas kito ėjimus, ir žaidimas yra determinuotas, t.y., jame nenaudojami tokie atsitiktiniai elementai, kaip kauliukų mėtimas. Tokį žaidimą galima vaizduoti medžiu, kurio viršūnės yra galimos žaidimo pozicijos (žr. 2.9 pav.). Simetriškas pozicijas galime vaizduoti viena viršūne. Jei kuris nors žaidėjas 1 ėjimu gali patekti iš pozicijos a į poziciją b, tai medžio viršūnes a ir b sujungiame lanku. Pozicijas, susidarančias po pirmo žaidėjo ėjimo medyje žymėsime skrituliukais, o po antro žaidėjo ėjimo — stačiakampiais. Medžio lapai yra galutinės žaidimo pozicijos, kuriose vienas iš priešininkų laimi arba žaidimas baigiasi lygiomis. Jei laimi 1-asis žaidėjas, tokiai pozicijai priskiriame vertę 1, jei laimi 2-asis žaidėjas, priskiriame vertę -1 (t.y., mes save laikome 1-uoju žaidėju), o jei žaidimas baigiasi lygiomis, pozicijai priskiriame skaičių 0.

Tuose žaidimuose, kur galime rasti pilną žaidimo medį, mes galime nustatyti kokiu rezultatu baigsis žaidimas, jei abu priešininkai darys geriausius ėjimus, t.y., 1-asis žaidėjas iš visų galimų ėjimų rinksis tokį ėjimą, kuris veda į vertingiausią poziciją, o 2-asis žaidėjas rinksis tokį ėjimą, kuris veda į poziciją, mažiausiai vertingą 1-ajam žaidėjui. Pavyzdžiui, nesunku parašyti programėlę, kuri nustatys, kad abiems priešininkams darant geriausius ėjimus, žaidimas kryžiukai–nuliukai visada baigsis lygiomis.

Deja (o iš tikrųjų, laimei, nes priešingu atveju išnyktų populiarūs žaidimai), dažnai žaidimų medžiai būna tokio dydžio, kokį sunku net įsivaizduoti. Imant, pavyzdžiui, kad vidutinė šachmatų partija trunka 50 ėjimų, o kiekvienoje pozicijoje baltieji turi apie 20 ėjimų, o juodieji į kiekvieną



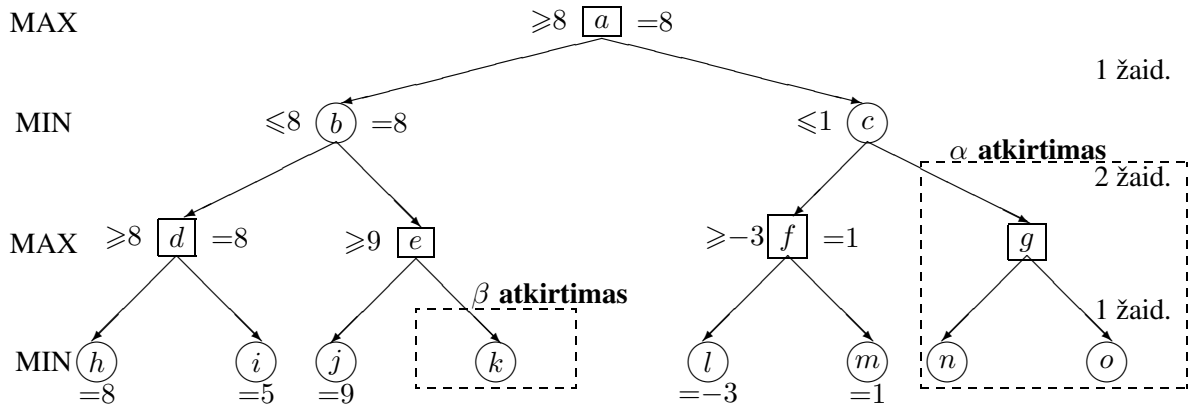
Pav. 2.10: Žaidimo medis, kai žinomi laimėjimai $gain$ visuose medžio lapuose.

baltųjų ėjimą apie 20 atsakymų, gauname, kad tokios partijos žaidimo medis turės 20^{100} viršūnių! Tokiu atveju vietoje galutinės pozicijos vertės yra naudojamos funkcijos, nusakančios tos pozicijos gerumą 1-ojo žaidėjo atžvilgiu. Tarkime, $gain$ yra tokia funkcija, bet kuriai pozicijai (medžio viršūnei) priskirianti realų skaičių. Jei abu žaidėjai naudosis tokia pat poziciją įvertinančia funkcija $gain$, tada jie naudos taip vadinamą MAXMIN strategiją: 1-asis žaidėjas rinksis ėjimą, vedantį į viršūne su didžiausia $gain$ reikšme, o 2-asis žaidėjas rinksis ėjimą, vedantį į viršūnę su mažiausia $gain$ reikšme.

Panagrinėkime žaidimo medį, pavaizduotą 2.10 pav. Tarkime, 1-asis žaidėjas apskaičiavo du ėjimus į priekį (t.y., savo ėjimą, priešininko atsakymą, ir savo 2-ąjį ėjimą) ir rado, kokį laimėjimą jis gautų kiekvienoje pozicijoje, kuri gali susidaryti po 2 ėjimų. Kurį ėjimą, b ar c, jis turi pasirinkti?

Jei 1-asis žaidėjas daro ėjimą b, o priešininkas atsako ėjimu d, tada 1-asis žaidėjas iš dviejų galimybių renkasi h, nes pozicijoje h jo laimėjimas yra didesnis. Taigi, jei antrasis žaidėjas atsakys d, tai 1-asis žaidėjas turės persvarą $+8$. Analogiškai gauname, kad antrajam žaidėjui į ėjimą b atsakius ėjimu e 1-asis žaidėjas turės persvarą $+9$. Kadangi tai žino ir antrasis žaidėjas, tai iš dviejų $gain$ reikšmių jis rinksis mažesnę, ir viršūnė b gaus reikšmę $gain(b) = 8$. Analogiškai MAXMIN strategijos pagalba randame $gain(c) = 1$. Taigi, 1-asis žaidėjas rinksis ėjimą b.

Iš tikrųjų šiame pavyzdyje mums nevertėjo peržiūrėti viso žaidimo medžio. Žaidimų medžiuose taip pat galima taikyti šakų ir rėžių metodą. Žaidimo medžio pomedžių atkirtimus, gaunamus šakų ir rėžių metodo pagalba, vadina *α-atkirtimais* ir *β-atkirtimais*. Dar kartą panagrinėkime medį pavaizduotą 2.10 pav. Radę laimėjimo dydį pozicijoje d, $gain(d) = 8$, mes gauname viršutinį įvertį laimėjimui viršūnėje b: $gain(b) \leqslant 8$, nes priešininkas rinksis ėjimą d arba dar blogesnį. Todėl pozicijoje j radę reikšmę 9, mes galime nebenagrinėti pozijos k, nes po priešininko ėjimo e mes ir taip jau laimėtume 9, todėl jis šio ėjimo nesirinks, nes turi geresnį ėjimą d. Tokią situaciją



Pav. 2.11: Žaidimo medis su α- ir β-atkirtimais.

vadina β-atkirtimu (žr. 2.11 pav.).

- Taigi, β-atkirtimą turime, kai:

  1. jūs esate viršūnėje, kurioje yra jūsų eilė daryti ėjimą (viršūnė e mūsų pavyzdyje);
  2. jūs radote šios viršūnės (e) įvertį ir jos tėvo (b) įvertį;
  3. viršūnės tėvo įvertis yra *mažesnis arba lygus* jos pačios įverčiui;
  4. ši viršūnė turi vaikų, kurie dar nenagrinėti.

  Jei išpildytos šios keturios sąlygos, visus pomedžius, kurių šaknys yra minėti nagrinėjamos viršūnės vaikai, galime atkirsti.

Atkirtę poziciją k, mes randame $gain(b) = 8$, todėl gauname apatinį įvertį viršūnei a: $gain(a) \geqslant 8$. Nusileidę šaka c žemyn ir aplankę viršūnes l ir m, randame laimėjimą viršūnėje f: $gain(f) = 1$. Šis rezultatas duoda viršutinį įvertį viršūnei c: $gain(c) \leqslant 1$. Šis įvertis iš karto rodo, kad ėjimo c mes nesirinksime, nes jau radome geresnį ėjimą b, taigi pomedį su šaknimi g galime atkirsti. Tokį atkirtimą vadina α-atkirtimu (žr. 2.11 pav.).

- Taigi, α-atkirtimą turime, kai:

  1. jūs esate viršūnėje, kurioje yra priešininko eilė daryti ėjimą (viršūnė c mūsų pavyzdyje);
  2. jūs radote šios viršūnės (c) įvertį ir jos tėvo (a) įvertį;
  3. viršūnės tėvo įvertis yra *didesnis arba lygus* jos pačios įverčiui;
  4. ši viršūnė turi vaikų, kurie dar nenagrinėti.

Jei išpildytos šios keturios sąlygos, visus pomedžius, kurių šaknys yra minėti nagrinėjamos viršūnės vaikai, galime atkirsti.

Nesunku pastebėti, kad α-atkirtimai yra sutinkami lyginiuose žaidimų medžio sluoksniuose, o β-atkirtimai — nelyginiuose. Pabaigai išnagrinėkime dar vieno žaidimo medį.

Žaidimas nim

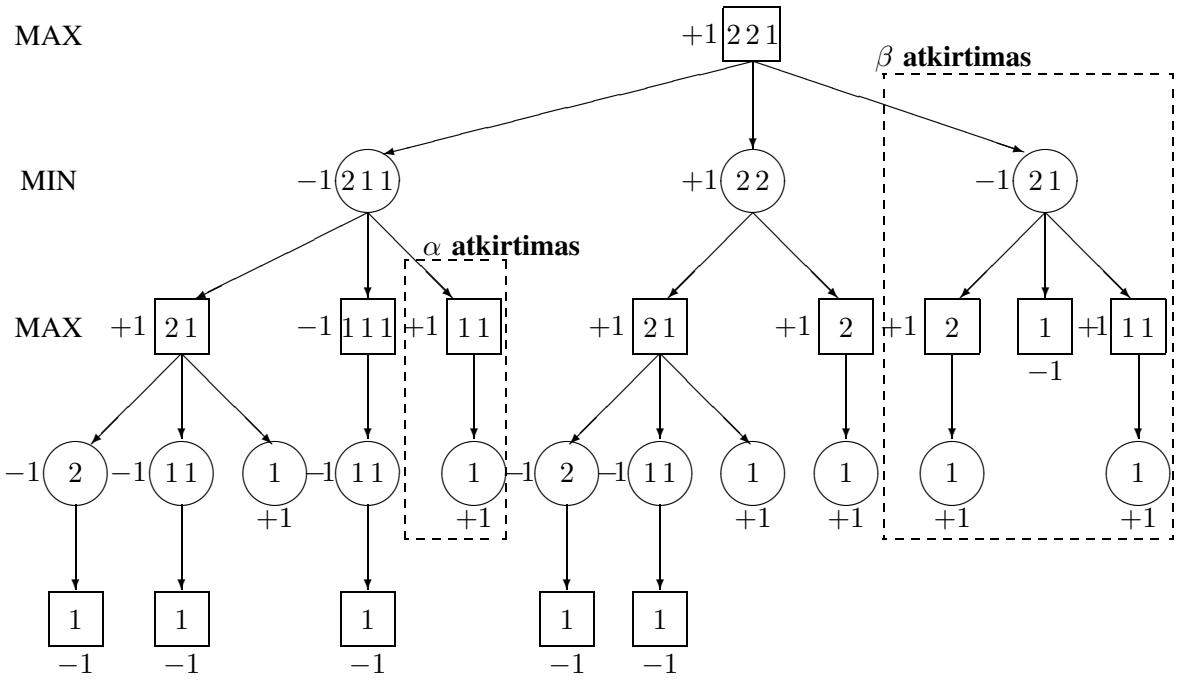
Žaidimo nim pradžioje yra duota keletas krūvelių akmenukų. Kiekvienas žaidėjas paeiliui gali paimti iš bet kurios krūvelės (bet tik iš vienos!) bet kokį skaičių akmenukų. Tas, kuris paims paskutinį akmenuką, pralaimi. (Tai viena iš šio žaidimo versijų. Dar yra žinoma versija, kai paėmęs paskutinį akmenuką laimi, ir kitokie variantai.) Kiekviena galutinė šio žaidimo pozicija gauna vertę $+1$ (laimėjo 1-asis žaidėjas) arba -1 (laimėjo 1-asis žaidėjas), nes lygiųjų žaidime nim negali būti. Negalutinėms žaidimo pozicijoms galima priskirti įverčius, naudojantis MAXMIN principu. Tarkime, turime poziciją P, kurioje ėjimo teisę turi 1-asis žaidėjas. Tada $gain(P) = +1$, jei egzistuoja toks 1-ojo žaidėjo ėjimas, kad kaip beatsakytų jo priešininkas, 1-asis žaidėjas, ir toliau rinkdamasis geriausius ėjimus, laimės. Jei į bet kurį 1-ojo žaidėjo ėjimą priešininkas gali atsakyti tokiu ėjimu, po kurio abiems priešininkams besirenkant geriausius ėjimus, 1-asis žaidėjas pralaimės, tai $gain(P) = -1$.

Panagrinėkime, kokį ėjimą turėtų daryti pirmasis žaidėjas, kai pradinė padėtis yra $2\,2\,1$, t.y., žaidžiama su 3 krūvelėm akmenukų. Jis turi 5 galimus ėjimus, po kurių gali susidaryti 3 skirtingos pozicijos $2\,1\,1$, $2\,2$ ir $2\,1$ (žr. 2.12 pav.). Nagrinėjame galimus 2-ojo žaidėjo ėjimus kiekvienoje iš šių pozicijų. Jei iš pozicijos $2\,1\,1$ 2-as žaidėjas paeina į poziciją $2\,1$, tai 1-asis žaidėjas paima iš 1-os krūvelės abu akmenukus ir laimi. Todėl 2-as žaidėjas rinksis ėjimą, vedantį į poziciją $1\,1\,1$, iš kurios 1-as žaidėjas bus priverstas pereiti į poziciją $1\,1$ ir pralaimės. Kadangi iš priešininko reikia tikėtis blogiausio (MIN), tai pozicija $2\,1\,1$ gauna įvertį -1. Tuo pačiu mes galime nebenagrinėti 2-o žaidėjo ėjimo, vedančio iš padėties $2\,1\,1$ į padėtį $1\,1$, nes jau radome geriausią ėjimą 2-am žaidėjui (α-atkirtimas, žr. pav.2.12; čia nebereikia žinoti viršūnės $2\,1\,1$ tėvo įverčio: kadangi tėra tik dvi galimos reikšmės $+1$ ir -1, tai tėvo įvertis tikrai bus didesnis arba lygus už viršūnės $2\,1\,1$ įvertį, lygų -1).

Taigi, pirmajam žaidėjui geriau nesirinkti ėjimo, vedančio į poziciją $2\,1\,1$. Panašiai išnagrinėję visus variantus, gaunamus pozicijoje $2\,2$, randame, kad šioje pozicijoje po bet kurio 2-o žaidėjo ėjimo 1-asis žaidėjas laimi. Taigi, pozicija $2\,2$ gauna įvertį $+1$, ir mes radome geriausią 1-ojo žaidėjo ėjimą pradinėje pozicijoje, kuri taip pat gauna įvertį $+1$ (MAX). Tuo pačiu mes galime nebenagrinėti žaidimo pomedžio su šaknimi $2\,1$ (β-atkirtimas).

2.6 Godūs algoritmai

Spręsdami kombinatorinius uždavinius, mes dažnai susiduriame su situacija, kai galimų sprendinių skaičius yra baigtinis, ir teoriškai brutalios jėgos algoritmo pagalba mes galėtume šį uždavinį



Pav. 2.12: Pilnas nimo žaidimo medis.

išspręsti, tačiau perrinkimo variantų skaičius yra toks didelis, kad praktiškai pritaikyti brutalios jėgos algoritmą nėra jokios vilties. Jei nepavyksta šiam uždaviniui rasti polinominio sudėtingumo algoritmo, taikant dinaminį programavimą arba metodą "Skaldyk ir valdyk", tada dažnai tenka atsisakyti vilties rasti tikslų (optimalų) šio uždavinio sprendinį. Tokiais atvejais galima bandyti rasti apytikslį sprendinį, taikant euristinius algoritmus. Algoritmą vadiname *euristiniu*, jei: (a) jis randa nebūtinai optimalų, bet gana gerą sprendinį ir (b) jį realizuoti praktiškai galima paprasčiau už bet kurį žinomą tikslų algoritmą. Vienas iš dažniausiai taikomų ir paprasčiausių euristinių algoritmų yra *godus algoritmas*.

Pagrindinė godaus algoritmo idėja — kiekviename žingsnyje pasirenkame lokaliai optimalų sprendinį. Jei brutalios jėgos algoritmai apeina visą sprendinių medį, o šakų ir rėžių algoritmai peržiūri dalį sprendinių medžio, tai godus algoritmas paprastai praeina tik viena sprendinių medžio šaka. Godūs algoritmai randa gana gerą sprendinį, tačiau labai dažnai šis sprendinys būna neoptimalus. Tam, kad algoritmas galėtų pasirinkti lokaliai optimalų sprendinį, reikia:

(a) mokėti įvertinti, kuris sprendinys tam tikru momentu yra geriausias; tam paprastai naudojama funkcija, nusakanti sprendinio vertę;

(b) mokėti patikrinti, ar pasirinktas dalinis uždavinio sprendinys yra leistinas, t.y., ar jį bus galima praplėsti iki pilno sprendinio.

Pažymėję kandidatų į sprendinius sąrašą raide C, o dalinį sprendinį raide S, galime suformuluoti bendro pavidalo godų algoritmą:

procedure greedy(C)
$S := \emptyset$; /* Pradžioje sprendinys yra tuščias */
while (**not** solution S **and** $C \neq \emptyset$) **do**
  $x :=$ select(C);
  $C := C \setminus \{x\}$;
  if feasible$(S \cup \{x\})$ **then** $S := S \cup \{x\}$;
end while;
if (solution(S)) **then return** S **else return** $\emptyset$;

2.6.1 Minimalaus denginio uždavinys

Aibės *denginys* buvo apibrėžtas skyrelyje 2.3.4, kur naudodami paiešką su grįžimu sprendėme minimalaus aibės skaidinio uždavinį.

Minimalaus denginio uždavinys. Duotas aibės A denginys $\mathcal{B} = \{B_1, \ldots, B_k\}$. Išrinkti iš denginio $\mathcal{B}$ trumpiausią aibės A denginį, t.y. rasti $\mathcal{B}_0 \subseteq \mathcal{B}$: $\mathcal{B}_0$ — aibės A denginys ir $|\mathcal{B}_0| \leqslant |\mathcal{C}|$ kiekvienam A denginiui $\mathcal{C} \subseteq \mathcal{B}$. Galima spręsti ir bendresnį minimalaus denginio uždavinį, kai poaibio B_j kaina yra ne vienetas, o c_j, ir reikia rasti pigiausią aibės A denginį.

Pavyzdys 2.6.1. Tegu $A = \{a, b, c, d, e, f\}$, $B_1 = \{a, b, c\}$, $B_2 = \{a, d\}$, $B_3 = \{b, e\}$, $B_4 = \{c, f\}$, $B_5 = \{e\}$, $\mathcal{B} = \{B_1, \ldots, B_5\}$. Nesunku įsitikinti, kad minimalus aibės A denginys bus $\mathcal{B}_0 = \{B_2, B_3, B_4\}$.

Suformuluosime du praktikoje sutinkamus minimalaus denginio uždavinius:

1. Gamybos procese gamyklai prireikė n detalių, laikomų k sandėliuose. Reikia rasti mažiausią kiekį sandėlių, iš kurių galima atsigabenti visas reikiamas detales.

2. Į ekspediciją nori vykti k žmonių, iš kurių kiekvienas yra įvaldęs keletą profesijų, reikalingų ekspedicijoje (geologo, metereologo, gydytojo, virėjo ir t.t.). Reikia išrinkti mažiausią grupę žmonių, sugebančių dirbti visus n reikiamų darbų.

Minimalaus denginio uždavinį galima išspręsti *brutalios jėgos* algoritmo pagalba: imame kiekvieną iš 2^k aibės $\mathcal{B}$ poaibių $\mathcal{C}$ ir tikriname, ar $\mathcal{C}$ yra A denginys; po to iš visų denginių išrenkame trumpiausią. Deja, praktiniuose uždaviniuose skaičius k gali būti gana didelis, todėl šis algoritmas turi daugiau teorinę reikšmę.

Dabar suformuluosime *godų* algoritmą. Taikant šį algoritmą minimalaus denginio uždaviniui išsirenkame poaibį $B_{i_1} \in \mathcal{B}$, uždengiantį daugiausia aibės A elementų, ir įtraukiame į būsimą sprendinį. Pašalinę jau uždengtus elementus iš aibės A ir iš visų poaibių B_i, vėl renkamės daugiausia likusių aibės A elementų uždengiantį poaibį $B_{i_2} \in \mathcal{B}$ ir t.t., kol uždengsime visus aibės A elementus.

function $\mathcal{B}_{\mathrm{g}} = \mathrm{set\_cover}(A, \mathcal{B})$
$\mathcal{B}_{\mathrm{g}} := \emptyset$;
while $A \neq \emptyset$ **do**
  find $B_j \in \mathcal{B}$: $|B_j| = \max\{|B_1|, \ldots, |B_k|\}$;
  $\mathcal{B}_{\mathrm{g}} := \mathcal{B}_{\mathrm{g}} \cup \{B_j\}$; $A := A \setminus B_j$;
  for $i = 1 : k$ **do** $B_i := B_i \setminus B_j$ **end for**;
end while;

Nesunku įvertinti, kad šio algoritmo sudėtingumas yra $L_{\mathrm{greedy}}(n, k) = \mathrm{O}(nk \min(k, n))$, nes ciklas **while** bus vykdomas $\min(n, k)$ kartų, o cikle **for** operacija $B_i := B_i \setminus B_j$ gali pareikalauti n elementarių žingsnių.

Pritaikę šį algoritmą pavyzdžiui 2.6.1 gauname neoptimalų sprendinį $\mathcal{B}_{\mathrm{g}} = \{B_1, B_2, B_3, B_4\}$ dydžio 4. Šio uždavinio optimalus sprendinys yra $\{B_2, B_3, B_4\}$ dydžio 3. Galima įrodyti, kad blogiausiu atveju godaus algoritmo rastas denginys gali būti $\mathrm{O}(\frac{1}{k} \log \frac{n}{k})$ kartų didesnis (kai $n, k \to \infty$) už optimalų denginį.

2.6.2 Kuprinės pakavimo uždavinys realių svorių atveju

2.6.3 Minimalus grafo karkasas

2.6.4 KPU sprendimas, taikant euristinius algoritmus

Keliaujančio pirklio uždavinys (KPU) buvo aprašytas skyrelyje 2.4.1. Išvardinsime keletą euristinių algoritmų, kuriuos galima taikyti šio uždavinio sprendimui.