ĮVADAS

Algoritmų analizės objektas yra *algoritmai*. Nors algoritmo sąvoka yra laikoma pirmine matematikos sąvoka, nereikalaujančia apibrėžimo, dažnai algoritmą apibūdina kaip *baigtinę seką tikslių komandų (instrukcijų), nurodančių kaip rasti nagrinėjamo uždavinio sprendinį*.

Beveik visus algoritmus galima suskirstyti į dvi dideles klases: *kombinatorinius algoritmus* ir *skaitinius algoritmus*. Kombinatoriniai algoritmai operuoja su diskrečiais (= kombinatoriniais) objektais: sveikaisiais skaičiais, baigtinėmis aibėmis, grafais, matricomis ir pan. Skaitiniai algoritmai paprastai yra skaičiavimo metodų realizacijos, t.y., algoritmai sprendžiantys įvairaus pavidalo matematines lygtis su realiais koeficientais arba optimizuojantys realaus argumento funkcijas. Nagrinėdami algoritmus, mes pagrindinį dėmesį skirsime kombinatoriniams algoritmams. Taigi, algoritmų analizės kursą galima laikyti skaičiavimo metodų kurso analogu diskrečioje matematikoje: skaičiavimo metodai taiko matematinę analizę (pvz., diferencialines lygtis) tolydiems uždaviniams spręsti, o kombinatoriniai algoritmai taiko diskrečią matematiką diskretiems uždaviniams spręsti. Išvardinsime tik keletą kombinatorinių uždavinių pavyzdžių:

1. Reikia sudaryti optimalų paskaitų tvarkaraštį Vilniaus Universiteto MIF fakultete.

2. Reikia rasti trumpiausią maršrutą, praeinantį po 1 kartą per kiekvieną iš n miestų, kai duoti atstumai tarp tų miestų (taip vadinamas *keliaujančio pirklio* uždavinys; trumpiau: KPU).

3. Reikia parašyti programą, gerai žaidžiančią šachmatais.

Diskretūs uždaviniai dažniausiai yra susiję su variantų perrinkimo problema. Kadangi galimų duoto uždavinio sprendinių skaičius dažniausiai būna baigtinis, tai išnagrinėję visus galimus variantus ir juos įvertinę (t.y., priskirę kiekvienam variantui jo vertę), mes galėtume išsirinkti geriausią sprendinį. Taip elgiasi taip vadinami brutalios jėgos (angl. *brute force*) algoritmai. Deja, praktikoje mes dažnai susiduriame su taip vadinama *kombinatorinio sprogimo* problema: jei uždavinys yra pakankamai didelis (toks, kurio mes be kompiuterio pagalbos nebegalime išspręsti), tai galimų variantų skaičius labai greitai auga ir pasiekia tokį kiekį, kurio negalima perrinkti ir su geriausiu pasaulyje kompiuteriu. Pavyzdžiui, ieškant optimalaus keliaujančio pirklio maršruto, norint apkeliauti n miestų, gali tekti nagrinėti $(n-1)!$ skirtingų maršrutų (plg. $20! = 2\,432\,902\,008\,176\,640\,000$); norint išnagrinėti visas galimas šachmatų partijos pozicijas n ėjimų į priekį, gali tekti perrinkti apie $20^{2n} = 400^n$ variantų, nes kiekviename ėjime tiek baltieji, tiek juodieji gali

turėti po 20 skirtingų galimybių (plg. $400^{10} = 1048576 \times 10^{20}$). Todėl kombinatoriniai uždaviniai natūraliai suskyla į dvi grupes:

- uždaviniai, kuriems yra žinomas polinominio sudėtingumo algoritmas, arba *paprasti* (angl. *tractable*) uždaviniai, pavyzdžiui, trumpiausio kelio tarp dviejų miestų radimo problema, ir
- uždaviniai, kuriems nėra žinoma jokio polinominio sudėtingumo algoritmo, arba *sunkūs* (angl. *intractable*) uždaviniai, pavyzdžiui, aukščiau minėtas KPU.

Pagrindinės algoritmų analizės nagrinėjamos problemos yra šios: *algoritmo sustojimo* problema, *algoritmo korektiškumo* problema, *algoritmo sudėtingumo* problema ir *algoritmo efektyvumo* problema (smulkiau žr. 1.1 skyrelį). Šioje mokymo priemonėje pagrindinis dėmesys yra skiriamas dviems paskutinėms problemoms: sudėtingumui ir efektyvumui. Pirmame skyriuje mes apibrėžiame algoritmus, jų savybes, algoritmų ir uždavinių sudėtingumą, ir pateikiame algoritmų analizės pavyzdžių. Antrajame skyriuje išvardijame pagrindinius kombinatorinius objektus ir nagrinėjame, kaip efektyviau juos vaizduoti kompiuterio atmintyje. Trečias skyrius yra skirtas algoritmų konstravimo metodams. Ketvirtame skyriuje pateikiame algoritmų grafuose pavyzdžių, įvertindami tų algoritmų sudėtingumą. Dauguma nagrinėjamų algoritmų grafuose naudoja vieną ar kitą metodą iš trečiojo skyriaus. Pagaliau, penktas skyrius yra skirtas uždavinių sudėtingumo klasėms. Jame apibrėžiamos sudėtingumo klasės P, NP, NPC, nagrinėjami jų tarpusavio ryšiai bei vienų uždavinių polinominė redukcija į kitus uždavinius.

Iš aukščiau pateiktos apžvalgos matyti, kad pagrindinis šios mokymo priemonės tikslas yra supažindinti studentus su efektyviais algoritmų konstravimo metodais ir efektyviais klasikiniais algoritmais bei išmokyti atpažinti uždavinių sudėtingumą ir mokėti atskirti paprastus uždavinius nuo sunkių.

LITERATŪRA

[CLR] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, The MIT Press/ McGraw-Hill, Cambridge, MA/New York, 1990. (Yra taip pat rusiškas vertimas: T. Kormen, Č. Leizerson, R. Rivest, *Algoritmy: Postroenije i Analiz*, MCNMO, Moskva, 2001.)

[RND] E.M. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1977. (Yra taip pat rusiškas vertimas: E. Reingol'd, Ju. Nivergel't, N. Deo, *Kombinatornyje Algoritmy: Teorija i Praktika*, Mir, Moskva, 1980.)

[CH] N. Christofides, *Graph Theory: An Algorithmic Approach*, Academic Press, New York, 1975. (Yra taip pat rusiškas vertimas: N. Kristofides, *Teorija Grafov: Algoritmičeskij Podchod*, Mir, Moskva, 1978.)

1 skyrius

ALGORITMAI IR KOMBINATORINIAI OBJEKTAI

1.1 Algoritmų analizės problemos ir pavyzdžiai

Pagrindinės algoritmų analizės nagrinėjamos problemos yra šios:

(1) *algoritmo sustojimo* problema: reikia nustatyti, ar konkretus algoritmas, pritaikytas konkretiems pradiniams duomenims, baigs darbą ar dirbs be galo;

(2) *algoritmo korektiškumo* (teisingumo) problema: reikia nustatyti, ar konkretus algoritmas išduos atsakymą, sutampantį su tikruoju nagrinėjamo uždavinio sprendiniu;

(3) *algoritmo sudėtingumo* problema: reikia nustatyti, kiek žingsnių daugiausia atliks konkretus algoritmas iki sustojimo, ar jis užbaigs darbą per mums priimtiną laiką, ir ar šiam algoritmui užteks turimų atminties resursų;

(4) *algoritmo efektyvumo* problema: nustačius algoritmo sudėtingumą, reikia įvertinti, kiek jis yra efektyvus, t.y., ar tai yra pats geriausias galimas algoritmas nagrinėjamam uždaviniui spręsti, ar galima rasti geresnį algoritmą.

Pavyzdys 1.1. Duotas sveikasis teigiamas skaičius n. Reikia rasti jo faktorialą $n!$. Panagrinėkime tokį algoritmą:

function $\text{faktorialas} = \text{fact1}(n)$
$i := 1;$
$\text{faktorialas} := 1;$
while $i < n$ **do**
  $i := i + 1;$
  $\text{faktorialas} := \text{faktorialas} \cdot i;$
end;

1. Pirmiausia įrodysime, kad algoritmas fact1 baigia darbą (*sustojimo* problema). Pakanka įrodyti, kad ciklas **while** nėra begalinis. Ciklo pradžioje kintamasis $i = 1$. Kadangi

pirmoji vidinė ciklo komanda prie i prideda 1, o antroji komanda kintamojo i nekeičia, tai po $n-1$ ciklo iteracijos i tampa lygus n, ir ciklo vykdymas yra nutraukiamas.

2. Dabar įrodysime algoritmo *korektiškumą*, t.y., kad atsakymas tikrai bus $\mathrm{faktorialas} = n!$. Programų verifikacijos principus pasiūlė anglų informatikas Hoare[1]. Pažymėkime raide p teiginį "$\mathrm{faktorialas} = i!$ ir $i \leqslant n$". Pirmiausia įrodysime, kad šis teiginys yra invariantiškas ciklo atžvilgiu, t.y., ciklo vykdymas nekeičia teiginio teisingumo.

Programoje $\mathrm{fact1}$ naudojamą ciklą schematiškai galime pažymėti "**while** *sąlyga* **do** S". Teiginį p vadina *invariantišku* tokio ciklo atžvilgiu, jei teiginys "$(p \,\&\, \textit{sąlyga})\{S\}p$" yra teisingas. Užrašas $p\{S\}q$ reiškia, kad programos segmentas S yra *teisingas pradinės prielaidos p ir galutinės prielaidos q atžvilgiu*, t.y., jei p yra teisingas prieš pradedant vykdyti S, tai ir q bus teisingas 1 kartą įvykdžius S. Tarkime, kad prieš prasidedant ciklui teisinga p & *sąlyga*, t.y., teisinga $\mathrm{faktorialas} = i!$ ir $i < n$. Naujos kintamųjų i ir faktorialas reikšmės bus $i_{\mathrm{new}} = i + 1$ ir $\mathrm{faktorialas}_{\mathrm{new}} = \mathrm{faktorialas} \cdot (i+1) = (i+1)! = i_{\mathrm{new}}!$. Kadangi $i < n$, tai $i_{\mathrm{new}} = i + 1 \leqslant n$. Taigi, teiginys p yra teisingas ciklo gale; vadinasi, p yra invariantiškas šio ciklo atžvilgiu.

Norėdami baigti įrodyti programos $\mathrm{fact1}$ korektiškumą pasinaudosime programų verifikacijoje ciklui **while** naudojama *išvedimo taisyklė*

$$(p \,\&\, \textit{sąlyga})\{S\}p \vdash p\{\textbf{while } \textit{sąlyga } \textbf{do } S\}(\neg \textit{sąlyga} \,\&\, p),$$

kuri sako, kad iš to, kad teiginys p yra invariantiškas ciklo **while** atžvilgiu išplaukia, kad jei teiginys p yra teisingas prieš ciklo vykdymą, tai teiginys $\neg$*sąlyga* & p yra teisingas užbaigus ciklą.

Kadangi prieš pradedant ciklą turime $i = 1 \leqslant n$ ir $\mathrm{faktorialas} = 1 = i!$, tai prieš ciklo vykdymą teiginys p yra teisingas. Pagal aukščiau pateiktą taisyklę užbaigus ciklą bus teisingas teiginys ($\neg$*sąlyga* & p), t.y., bus teisinga konjunkcija $i \geqslant n \,\&\, \mathrm{faktorialas} = i! \,\&\, i \leqslant n$. Gauname, kad $i = n$ ir $\mathrm{faktorialas} = n!$. Taigi, programa $\mathrm{fact1}$ yra korektiška.

3. Apskaičiuosime algoritmo $\mathrm{fact1}$ *sudėtingumą*. Laikydami vienoje eilutėje užrašytos komandos vykdymą 1 žingsniu, gausime, kad algoritmas $\mathrm{fact1}$ sustoja po $L_1(n) = 3n$ žingsnių. Pavyzdžiui, kai $n = 2$, bus įvykdytos 6 komandos:

$i := 1;$
$\mathrm{faktorialas} := 1;$
$1 < 2?$
$i := 1 + 1 = 2;$
$\mathrm{faktorialas} := 1 \cdot 2 = 2;$
$2 < 2?$

4. Panagrinėkime, ar galima rasti *efektyvesnį* algorimą skaičiaus faktorialui skaičiuoti. Pabandykime tą pačią programą užrašyti su ciklu **for**:

[1] **C. Anthony R. Hoare (g. 1934).** Hoare įnešė svarbų indėlį į programavimo kalbų teoriją ir programavimo metodologiją. Jis pirmasis apibrėžė programavimo kalbą, leidžiančią įrodinėti programų korektiškumą jų specifikacijų atžvilgiu. Hoare pasiūlė algoritmą *Quicksort*, kuris dabar yra vienas iš plačiausiai naudojamų ir ištyrinėtų rūšiavimo algoritmų.

function faktorialas $=$ fact2(n)
faktorialas $:= 1$;
if $n > 1$ **then**
  for $i := 2$ **to** n **do** faktorialas $:=$ faktorialas $\cdot\, i$; **end**;
end;

Kadangi realizuojant ciklą **for** kiekvienoje ciklo iteracijoje prie ciklo kintamojo i bus pridedama po 1 ir kiekvieną kartą bus tikrinama ciklo pabaigos sąlyga $i = n$?, tai ir šio algoritmo sudėtingumas bus $L_2(n) = 3n$, kai $n > 1$, ir $L_2(1) = 2$, kai $n = 1$.

Panašų sudėtingumą gausime ir realizuodami faktorialo skaičiavimą rekursijos pagalba:

function faktorialas $=$ fact3(n)
if $n = 1$ **then** faktorialas $:= 1$ **else** faktorialas $:=$ fact3$(n-1) \cdot n$; **end**;

Kreipimąsi į funkciją fact3 laikant atskiru žingsniu, algoritmo fact3 sudėtingumas gaunasi $L_3(n) = 3n - 1$. Pavyzdžiui, kai $n = 2$, bus įvykdytos 5 komandos:

$2 = 1$?
call fact3(1);
$1 = 1$?
faktorialas $:= 1$;
faktorialas $:= 1 \cdot 2 = 2$;

Kadangi funkcijų iškvietimas praktiškai reikalauja daugiau laiko, negu paprastos priskyrimo komandos, reikėtų tikėtis, kad algoritmas fact3 praktiškai bus lėtesnis už algoritmus fact1 ir fact2. Pastarųjų vykdymo laikas priklausys nuo pasirinkto kompiuterio ir kompiliatoriaus.

Visų trijų aukščiau pateiktų algoritmų sudėtingumas tiesiškai priklauso nuo skaičiaus n, todėl šių algoritmų vykdymo laikas skirsis labai nežymiai. Pridėkime dar vieną "kvailą" algoritmą, tinkantį kompiuteriui, kuris moka tik sudėti sveikus skaičius, bet nemoka jų dauginti:

function faktorialas $=$ fact4(n)
faktorialas $:= 1$;
if $n > 1$ **then**
  for $i := 2$ **to** n **do**
    sumfakt $:=$ faktorialas;
    for $j := 2$ **to** i **do** sumfakt $:=$ sumfakt $+$ faktorialas; **end**;
    faktorialas $:=$ sumfakt
  end;
end;

Kadangi šis algoritmas turi dvigubą ciklą, tai jo sudėtingumas bus $L_4(n) = \mathrm{O}(n^2)$.[2]

[2]Primename, kad žymėjimas $f(n) = \mathrm{O}(g(n))$ reiškia, kad $\exists N \in \mathbb{N}$ ir $\exists c > 0$: $f(n) \leqslant cg(n)\ \forall n \geq N$. Smulkiau žr. 1.3 skyrelį.

Visi 4 algoritmai buvo realizuoti su Matlab programa 700 MHz kompiuteriu. Papildomai $n!$ dar buvo skaičiuojamas su vidine Matlab funkcija $\mathrm{prod}(1:n)$, kuri randa masyvo $[1, 2, 3, \ldots, n]$ elementų sandaugą ir su Matlab funkcija $\mathrm{gamma}(n+1)$, kur $\mathrm{gamma}(x) = \int_0^\infty t^{x-1}\mathrm{e}^{-t}\,\mathrm{d}t$ ir yra žinoma, kad $\mathrm{gamma}(n+1) = n!$. Lentelėje pateikiame visų 6 algoritmų išnaudotą CPU laiką mikrosekundėmis (1 mikrosekundė $= 10^{-6}$ sek.). Kadangi Matlab laiką matuoja tik šimtųjų sekundės dalių tikslumu, tai kiekvienas algoritmas buvo kartojamas cikle $100\,000$ kartų.

Algoritmas	$n = 5$	$n = 10$	$n = 20$	$n = 50$
fact1	38.464	80.488	162.127	404.050
fact2	24.577	42.690	76.018	175.392
fact3	87.946	179.047	359.362	906.222
fact4	85.803	258.583	844.149	4438.480
prod	10.796	11.105	12.424	13.540
gamma	437.719	581.236	298.690	298.150

Gauti rezultatai rodo, kad vidinė Matlab funkcija prod neilgus masyvus daugina praktiškai per pastovų laiką, nepriklausomai nuo masyvo ilgio. Ši funkcija sprendžia nagrinėjamą uždavinį efektyviausiai. Iš šių paskaitų autoriaus pateiktų algoritmų "nugalėjo" algoritmas fact2, naudojantis ciklą **for**. Rekursyvus algoritmas, kaip ir reikėjo tikėtis, veikia ilgiau už iteracinius. Keistoką funkcijos $\mathrm{gamma}(x)$ laiko kitimą nesunku paaiškinti: kai $x < 12$, ši funkcija yra skaičiuojama iteratyviai, o kai $x > 12$, yra naudojamos apytikslės formulės. Todėl didesnėms argumento reikšmėms funkcijos reikšmė apskaičiuojama greičiau.

Išnagrinėtas pavyzdys rodo, kad algoritmo sustojimo ir korektiškumo nagrinėjimas yra gana nuobodus ir varginantis užsiėmimas. Gal būt galima šį užsiėmimą pavesti kompiuteriui, t.y., sukurti universalią programą, kuri pagal duotą programą P ir jos įėjimo duomenis I atsakytų, ar programa kada nors sustos. Įrodysime, kad tokios programos neegzistuoja, t.y., **sustojimo problema yra algoritmiškai neišsprendžiama**.

Visas galimas programas, parašytas kuria nors pasirinkta programavimo kalba, galima abipus vienareikšmiškai koduoti natūraliaisiais skaičiais, t.y., sugalvoti taisyklę, kaip kiekvienai programai P priskirti jos kodą $\langle P\rangle \in \mathbb{N}$ ir atvirkščiai, kaip pagal duotą natūralųjį skaičių n atkurti programą P, kurios kodas $\langle P\rangle = n$. Paprastumo dėlei laikysime, kad visų nagrinėjamų programų įėjimai I ir išėjimai O gali būti tik natūralieji skaičiai, t.y. programos realizuoja funkcijas $f : \mathbb{N}^k \to \mathbb{N}$. Toliau nagrinėsime programas, realizuojančias vieno kintamojo dalines funkcijas $f : \mathbb{N} \to \mathbb{N}$ (t.y., $k = 1$).

Tarkime, kad egzistuoja programa H su dviem įėjimais, skaičiuojanti funkciją

$$\mathrm{HALT}(\langle P\rangle, I) = \begin{cases} 1, & \text{jei } P(I) \text{ sustoja;} \\ 0, & \text{jei } P(I) \text{ dirba be galo.} \end{cases}$$

Tada galima sukonstruoti programą K, kuri programai H į abu įėjimus paduoda programos P kodą $\langle P\rangle$ ir tikrina, kam lygus atsakymas $\mathrm{HALT}(\langle P\rangle, \langle P\rangle)$. Jei atsakymas yra 1, programa K nukreipia į amžiną ciklą. Jei atsakymas yra 0, tai ir programa K sustoja ir

išduoda nulį. Taigi, programa $K(\langle P\rangle)$ sustoja tada ir tik tada, kai savo kodui pritaikyta programa $P(\langle P\rangle)$ nesustoja. Programa K skaičiuoja dalinę funkciją

$$K(\langle P\rangle) = \begin{cases} 0, & \text{jei } P(\langle P\rangle) \text{ nesustoja;} \\ ?, & \text{jei } P(\langle P\rangle) \text{ sustoja,} \end{cases}$$

kur ? reiškia, kad funkcijos reikšmė yra neapibrėžta. Dabar imame programos K kodą $\langle K\rangle$ ir pateikiame jį kaip įėjimą programai K. Gauname prieštaravimą: $K(\langle K\rangle)$ sustoja tada ir tik tada, kai $K(\langle K\rangle)$ nesustoja. Gautas prieštaravimas įrodo, kad neegzistuoja sustojimo problemą sprendžiančios programos H.

Analogiškai yra įrodoma, kad ir **algoritmo korektiškumo problema yra algoritmiškai neišsprendžiama**, t.y., neegzistuoja programos, kuri patikrintų ar pritaikius programą P įėjimui I, jos išėjimas bus O. Kitaip sakant, funkcija

$$\mathrm{CORRECT}(\langle P\rangle, I, O) = \begin{cases} 1, & \text{jei } P(I) = O; \\ 0, & \text{priešingu atveju} \end{cases}$$

nėra algoritminė.

Kadangi algoritmų korektiškumas yra nagrinėjamas programų verifikacijos kurse, algoritmų analizėje pagrindinį dėmesį skirsime *algoritmų sudėtingumui* ir *efektyvių algoritmų konstravimui*.

1.2 Algoritmai ir jų sudėtingumas

Nors *algoritmo* sąvoką paprastai sieja su programomis bei kompiuteriais, šis žodis jau yra žinomas daugiau kaip tūkstantį metų. Jis kilęs iš žymaus Rytų mokslininko al-Khowârizmî[3] vardo. XII amžiuje Europoje pasirodė šio mokslininko traktato apie aritmetiką vertimas iš arabų į lotynų kalbą, kuris vadinosi "Dixit algorizmi" ("Al Chorezmis pasakė"). Kadangi šiame traktate buvo aprašomi veiksmai su skaičiais pozicinėje skaičiavimo sistemoje (pavydžiui, sudėtis stulpeliu), tai šie veiksmai, o vėliau ir kitos įvairios procedūros buvo pradėta vadinti algoritmais. Vieną iš tokių procedūrų, kuri tebenaudojama iki šiol dviejų sveikų skaičių bendram didžiausiam dalikliui rasti, dar apie 300 m. prieš Kristų aprašė Euklidas[4].

"Algoritmas" yra pirminė matematikos bei informatikos sąvoka, panašiai kaip sąvokos "skaičius", "aibė" ir kitos. Neformaliai (intuityviai) algoritmą galima apibrėžti kaip objektą, turintį šias savybes:

[3] **Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî (783–850).** Astronomas ir matematikas Al Chorezmis yra kilęs iš Chorezmio miesto (dabartinė Chiva Uzbekistane). Jis buvo Bagdado mokslininkų akademijos, vadintos Išminčių Rūmais, nariu. Vakarų europiečiai algebros pradmenis sužinojo iš jo darbų, išverstų į lotynų kalbą. Be žodžio *algoritmas*, taip pat ir žodis *algebra* atsirado iš aukščiau tekste minimos knygos pavadinimo, nes jos pavadinimas arabų kalba skambėjo "Kitab al-jabr w'al muquabala".

[4] **Euclid (325–265 P.K.).** Euklidas parašė žymiausią matematinį veikalą pasaulio istorijoje. Jo knyga "Elementai" nuo seniausių iki dabartinių laikų įvairiomis kalbomis buvo išleista daugiau kaip 1000 kartų. Apie jo gyvenimą išliko mažai žinių, tačiau neabejotina, kad Euklidas dėstė žymiojoje Aleksandrijos akademijoje. Jis nesidomėjo matematikos taikymais. Kai vienas jo mokinių paklausė, kur jis galės pritaikyti geometrijos žinias, Euklidas jam atsakė, kad žinių įgyjimas turi būti vertinamas pats savaime, ir liepė savo tarnui duoti šiam mokiniui monetą, jei jis taip nori gauti pajamų iš to, ką jis mokosi.

(1) *Programiškumas*. Tai reiškia, kad algoritmas suprantamas kaip baigtinis taisyklių arba komandų rinkinys (dar vadinamas programa), nusakantis kaip vienus objektus (duomenis), atlikus baigtinį skaičių nesudėtingų operacijų perdirbti į kitus objektus (rezultatus).

(2) *Diskretumas*. Tai reiškia, kad algoritmas apibrėžia nuoseklų duomenų apdorojimo (skaičiavimo) procesą, suskirstytą į atskirus etapus (žingsnius).

(3) *Determinuotumas*. Paprastai reikalaujama, kad po kiekvieno žingsnio būtų vienareikšmiškai apibrėžtas kitas žingsnis arba būtų nurodyta, jog skaičiavimo procesas pasibaigė. Teorinėms reikmėms kartais naudojami ir nedeterminuoti algoritmai, leidžiantys keletą galimų kito žingsnio pasirinkimo variantų.

(4) *Žingsnių elementarumas (lokalumas)*. Taisyklė, nusakanti kaip pakeisti duomenis per 1 žingsnį turi būti paprasta ir lokali (nežymiai pakeičianti duomenis). Pavyzdžiui, algoritmas negali turėti tokios komandos kaip "Dabar įrodykite Didžiąją Ferma[5] Teoremą laipsnio rodikliui $n = 73$".

(5) *Masiškumas*. Tai reiškia, kad algoritmas turėtų būti toks, kad jį galima būtų pritaikyti įvairiems duomenims iš tam tikros leistinų pradinių duomenų aibės (paprastai begalinės), o algoritmo darbo rezultatai taip pat priklausytų apibrėžtai leistinų rezultatų aibei. Pavyzdžiui, taisyklė "norint sudėti du skaičius 2 ir 3 reikia užrašyti, kad atsakymas lygus 5" nėra laikoma algoritmu, nes jos negalima pritaikyti norint sudėti du bet kokius sveikuosius skaičius. Paprastai algoritmuose duomenys ir rezultatai būna *konstruktyvūs objektai*. Konstruktyviu objektu vadiname baigtinį objektą, turintį diskrečią struktūrą ir vidinę koordinačių sistemą. Pavyzdžiui, sveikasis skaičius užrašytas pozicinėje skaičiavimo sistemoje yra konstruktyvus objektas. Tuo tarpu baigtinė aibė, kaipo tokia, dar nėra konstruktyvus objektas. Ji tampa konstruktyviu objektu, tik įvedus kokią nors tvarką tarp jos elementų.

Prie aukščiau išvardintų savybių dažnai dar yra pridedama *rezultatyvumo* savybė, reikalaujanti, kad algoritmas po baigtinio žingsnių skaičiaus sustotų ir išduotų kokį nors rezultatą. Tačiau algoritmų teorijoje paprastai nagrinėjami ir tie algoritmai, kurie kai kuriems pradiniams duomenims gali dirbti be galo arba jiems sustojus rezultatas būna neapibrėžtas. Tokie algoritmai realizuoja ne visur apibrėžtas funkcijas. Dabar pateiksime algoritmų pavyzdžių.

[5]**Pierre de Fermat (1601–1665).** Vienas žymiausių XVII a. matematikų Pjeras Ferma buvo teisininkas. Jis yra pats žymiausias pasaulio istorijoje matematikas mėgėjas. Apie jo darbus daugiau yra žinoma tik iš jo susirašinėjimo su kitais matematikais. Ferma buvo vienas iš analizinės geometrijos kūrėjų. Jis taip pat prisidėjo prie integralinio skaičiavimo ir tikimybių teorijos vystymo. Ferma suformulavo uždavinį, kuris ilgą laiką buvo tapęs žymiausia neišspręsta matematikos problema. Tai taip vadinama Didžioji Ferma Teorema, kuri teigia, kad lygtis $x^n + y^n = z^n$ visiems $n \geq 2$ neturi netrivialių sveikaskaitinių sprendinių. Senovės graikų matematiko Diofanto knygos paraštėse Ferma rašė, kad jis žino šios teoremos įrodymą, bet paraštėse neužtenka vietos įrodymui išdėstyti. Tūkstančiai viso pasaulio matematikų daugiau kaip 300 metų nesėkmingai bandė įrodyti šią teoremą, kol 1994 m. Andrew Wiles, remdamasis pačiais naujausiais ir sudėtingais šiuolaikinės matematikos metodais, ją įrodė. Todėl yra manoma, kad arba Ferma žinomas įrodymas buvo klaidingas, arba jis jo nežinojo, o tik bandė kitus paskatinti įrodyti šią teoremą.

Pavyzdys 1.2 (Paieška sąraše, uždavinys SEARCH**).** Duotas skirtingų objektų (pvz., skaičių) sąrašas (masyvas) $A = \{a_1, \ldots, a_n\}$ ir objektas b. Reikia rasti objekto b vietą sąraše A arba nustatyti, kad tokio objekto sąraše nėra, t.y., apskaičiuoti funkciją

$$\mathrm{location}(A, b) = \begin{cases} i, & \exists a_i = b, \\ 0, & a_i \neq b \ \ \forall i = 1, \ldots, n. \end{cases}$$

Pavyzdyje suformuluotam uždaviniui spręsti naudosime nuosekliosios (tiesinės) paieškos algoritmą LIN_SEARCH ir binariosios paieškos algoritmą BIN_SEARCH. Antrasis algoritmas tinka tik paieškai pilnai sutvarkytame sąraše, t.y., kai $a_1 < a_2 < \cdots < a_n$ ir objektą b taip pat galime palyginti su visais objektais a_i.

function $\mathrm{location} = \mathrm{LIN\_SEARCH}(A, b)$
$i := 1$; $a_{n+1} := b$;
while $b \neq a_i$ **do** $i := i + 1$; **end**;
if $i \leq n$ **then** $\mathrm{location} := i$ **else** $\mathrm{location} := 0$; **end**;

Pastebėkite, kad čia mes pateikiame šiek tiek "patobulintą" nuosekliosios paieškos algoritmą. Tam, kad nereikėtų tikrinti ciklo **while** viduje tikrinti jo pabaigos sąlygos $i \leqslant n$, mes prijungiame prie sąrašo patį ieškomą objektą b, ko pasėkoje ciklas visada užsibaigs po $\leqslant n + 1$ žingsnių.

Dabar įvertinsime algoritmo LIN_SEARCH sudėtingumą. Paprastai algoritme galima išskirti vieno ar kelių tipų esmines operacijas, nuo kurių kiekio priklauso bendras algoritmo sudėtingumas (bendras sudėtingumas paprastai būna konstantą kartų didesnis už esminių operacijų skaičių). Sprendžiant paieškos ar rūšiavimo uždavinius esminės operacijos yra objektų palyginimai tarpusavyje. Kai duoti objektai yra ne skaičiai bet sudėtingesnės struktūros objektai (pavyzdžiui, sąrašo elementas gali būti asmens vardas ir pavardė), tai šios esminės operacijos dažnai yra vykdomos ilgiau už paprastas aritmetines ar priskyrimo operacijas, todėl jų kiekis ir lemia bendrą algoritmo sudėtingumą. Taigi, spręsdami paieškos uždavinį skaičiuosime tik objekto b palyginimus su objektais a_i. Kai sąrašo A ilgis n artės į begalybę, bendras žingsnių skaičius skirsis nuo atliktų palyginimų skaičiaus ne daugiau kaip pastoviu daugikliu (tiksliau, jei palyginimų bus P, tai viso bus įvykdyta $2P + \mathrm{const}$ žingsnių).

Pažymėkime $\widetilde{L}(A, b)$ skaičių palyginimų, kuriuos reikės atlikti, kol rasime objekto b vietą sąraše A, naudodami nuoseklios paieškos algoritmą. Pavyzdžiui, $\widetilde{L}(\{1, 3\}, 1) = 1$, $\widetilde{L}(\{1, 3\}, 3) = 2$, $\widetilde{L}(\{1, 3\}, 5) = 3$. Jei sąrašo A ilgis yra n, tai geriausiu atveju reikės atlikti 1 palyginimą (kai $b = a_1$), o blogiausiu atveju reikės $n + 1$ palyginimo (kai objekto b sąraše A nėra). Kadangi algoritmo sudėtingumas uždaviniui dydžio n apibrėžiamas kaip sudėtingumas blogiausiems duomenims dydžio n, tai paieškos sąraše uždavinio sudėtingumas, sprendžiant šį uždavinį nuosekliosios paieškos algoritmu, yra

$$L^{\mathrm{SEARCH}}_{\mathrm{LIN\_SEARCH}}(n) \leqslant 2(n + 1) + \mathrm{const} = \mathrm{O}(n).$$

Dabar panagrinėkime *binariosios paieškos* algoritmą, kurio idėja yra sąrašą nuosekliai dalinti pusiau ir lyginti objektą x su viduriniu sąrašo objektu tam, kad nustatyti, kuriame iš dviejų gautų perpus trumpesnių sąrašų reikia tęsti paiešką.

function location $=$ BIN_SEARCH(A, b)
$i := 1$; $j := n$;
while $i < j$ **do**
  $k := \lfloor (i + j)/2 \rfloor$;
  if $b \leqslant a_k$ **then** $j := k$ **else** $i := k + 1$; **end**;
end;
if $b = a_i$ **then** location $:= i$ **else** location $:= 0$; **end**;

Kadangi po kiekvienos ciklo **while** iteracijos sąrašo, kuriame yra tęsiama paieška, ilgis $i + j - 1$ sumažėja maždaug perpus, tai nesunku įsitikinti, kad šis algoritmas atlikęs ne daugiau kaip $c \cdot \log_2 n$ žingsnių (kur c yra nedidelė konstanta) sustos ir išduos atsakymą (rezultatą) location. Taigi, jo sudėtingumas bus

$$L^{\text{SEARCH}}_{\text{BIN\_SEARCH}}(n) = \text{O}(\log_2 n).$$

Pavyzdys 1.3 (Sąrašo rūšiavimas, uždavinys SORT**).** Duotas objektų (pvz., skaičių) sąrašas (masyvas) $A = \{a_1, \ldots, a_n\}$, kuriame apibrėžtas pilnos tvarkos sąryšis $\leqslant$. Reikia duotą sąrašą surūšiuoti, t.y. išdėstyti jo elementus nemažėjančia tvarka: $A' = \{a_{i_1} \leqslant a_{i_2} \leqslant \cdots \leqslant a_{i_n}\}$ (kitaip sakant, reikia nurodyti kėlinį arba bijekciją $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ tokią, kad $\pi(j) = i_j$).

Trivialus rūšiavimo algoritmas (dar vadinamas brutalios jėgos algoritmu, BRUTE_FORCE_SORT gali būti toks: išrenkame mažiausią duoto sąrašo elementą, pašaliname jį iš pradinio sąrašo ir patalpiname į 1-ą naujo sąrašo vietą; po to iš likusių pradinio sąrašo elementų vėl išrenkame mažiausią ir patalpiname į 2-ą naujo sąrašo vietą ir t.t. Šio algoritmo sudėtingumas

$$L^{\text{SORT}}_{\text{BRUTE\_FORCE\_SORT}}(n) = \text{O}(n^2).$$

Yra žinomi asimptotiškai greitesni rūšiavimo algoritmai (t.y., greitesni pakankamai didelėms n reikšmėms). Tokie yra, pavyzdžiui, sąlajos rūšiavimo algoritmas MERGE_SORT ir "krūvą" (specialaus pavidalo duomenų struktūrą) naudojantis algoritmas HEAP_SORT. Yra įrodyta (žr. **??** skyrelį), kad jų sudėtingumas yra

$$L^{\text{SORT}}_{\text{MERGE\_SORT}}(n) = L^{\text{SORT}}_{\text{HEAP\_SORT}}(n) = \text{O}(n \log_2 n).$$

Bet kurio algoritmiškai išsprendžiamo uždavinio sudėtingumas priklauso nuo:

(1) uždavinio dydžio;

(2) pasirinkto algoritmo;

(3) konkrečių duomenų;

(4) vidinio problemos sudėtingumo (paieškos sąraše problema yra paprasta, sąrašo rūšiavimo problema yra sudėtingesnė, o visų galimų skirtingų sąrašų ilgio n generavimo problema dar sudėtingesnė, nes pareikalaus $\text{const} \cdot n!$ elementarių operacijų).

Tarkime, parametras n charakterizuoja uždavinio dydį palyginus jį su kitais tos pačios klasės uždaviniais. Iš pateiktų pavyzdžių matyti, kad konkretaus uždavinio U dydį tarp to paties tipo (pvz., SEARCH arba SORT) uždavinių, susijusių su sąrašais, charakterizuoja pradinio sąrašo ilgis n, nes kuo ilgesnis yra sąrašas, tuo ilgiau užtruks objekto paieška arba sąrašo rūšiavimas. Taigi, uždavinio dydis dažniausiai priklauso nuo jo pradinių duomenų dydžio. Pradiniai duomenys algoritmuose gali būti sveikieji skaičiai, aibės (masyvai), matricos, grafai ir įvairūs kiti objektai. Pavyzdžiui, kuo didesnis yra natūralusis skaičius, tuo ilgesnis yra jo dvejetainis kodas. Kuo yra didesnė matrica, tuo daugiau ji turi eilučių ir stulpelių. Kuo didesnis grafas, tuo daugiau jis turi viršūnių ir lankų. Taigi, jei A yra natūralusis skaičius, tai jo dydis $n = |A| = \lceil \log_2 A \rceil$; jei $A = \{a_1, \ldots, a_n\}$ yra aibė, tai jos dydis yra elementų skaičius $n = |A|$; jei A yra kvadratinė $n \times n$ matrica, tai jos dydis yra matricos eilė n. Grafo dydžiu, priklausomai nuo nagrinėjamo uždavinio, gali būti laikoma jo viršūnių skaičius n, jo briaunų skaičius m, abu šie parametrai m ir n, arba kokia nors jų kombinacija, pvz. $m+n$ arba $\max(m, n)$. Konkretaus uždavinio U iš klasės $\mathcal{U}$ dydį žymėsime $|U|$.

Tarkime, $\mathcal{U}$ yra uždavinių klasė ir A yra konkretus algoritmas šios klasės uždaviniams spręsti. Algoritmo A, sprendžiančio konkretų uždavinį $U \in \mathcal{U}$, sudėtingumu vadiname algoritmo A žingsnių skaičių iš pradinės konfigūracijos iki sustojimo ir žymime $L_A(U)$ (kartais tą patį dydį vadina konkretaus uždavinio $U \in \mathcal{U}$ sudėtingumu sprendžiant uždavinį U algoritmu A). Algoritmo žingsnių skaičių, sprendžiant uždavinį U, dar vadina laiku arba laiko sudėtingumu ir žymi $T_A(U)$, o panaudotos atminties kiekį vadina erdve arba erdvės sudėtingumu ir žymi $S_A(U)$. Algoritmo A, sprendžiančio klasės $\mathcal{U}$ uždavinius, sudėtingumu vadiname algoritmo A sudėtingumą sprendžiant sudėtingiausią uždavinį iš vienodo dydžio uždavinių iš klasės $\mathcal{U}$ ir žymime

$$L_A^{\mathcal{U}}(n) = \max_{U \in \mathcal{U}:\ |U|=n} L_A(U).$$

Naudojant du skirtingus algoritmus A ir B, gausime skirtingus sudėtingumus $L_A^{\mathcal{U}}(n)$ ir $L_B^{\mathcal{U}}(n)$. Todėl uždavinių klasės $\mathcal{U}$ sudėtingumu vadinsime dydį nepriklausantį nuo konkretaus algoritmo, o būtent pasirinksime paties geriausio algoritmo sudėtingumą:

$$L^{\mathcal{U}}(n) = \min_A L_A^{\mathcal{U}}(n).$$

Kai uždavinių klasė $\mathcal{U}$ yra numanoma, kartais jos sudėtingumą žymi tiesiog $L(n)$. Iš aukščiau pateiktų pavyzdžių matyti, kad paieškos *pilnai sutvarkytame* sąraše uždavinio SEARCH sudėtingumas yra $L^{\mathrm{SEARCH}}(n) = \mathrm{O}(\log_2 n)$, o sąrašo rūšiavimo uždavinio SORT sudėtingumas yra $L^{\mathrm{SORT}}(n) = \mathrm{O}(n \log_2 n)$.

Kai kurie algoritmai blogiausiu atveju dirba ilgai, tačiau vidutiniškai jie dirba trumpiau (juk blogiausias duomenų atvejis gali niekada ir nepasitaikyti!). Todėl kartais naudojamas *vidutinis algoritmo sudėtingumas*. Tarkime, kiekvienam n ir kiekvienam konkrečiam uždaviniui U dydžio n mes žinome tikimybę $p(U)$, su kuria šis uždavinys pasitaikys, sprendžiant uždavinius iš klasės $\mathcal{U}$. Jei mes iš anksto žinome uždavinio dydį n, tai su tikimybe 1 kuris nors konkretus uždavinys mums pasitaikys. Taigi tikimybės turi tenkinti

sąlygas

$$p(U) \geqslant 0 \ \ \forall U \quad \text{ir} \quad \sum_{U\in\mathcal{U}:\ |U|=n} p(U) = 1.$$

Algoritmo A, sprendžiančio klasės $\mathcal{U}$ uždavinius, vidutiniu sudėtingumu vadiname

$$\overline{L}_A^{\mathcal{U}}(n) = \sum_{U\in\mathcal{U}:\ |U|=n} p(U)L_A(U).$$

Pavyzdžiui, žymusis greito rūšiavimo algoritmas QUICK_SORT blogiausiu pradinių duomenų išsidėstymo atveju dirbs taip pat ilgai kaip ir paprastesni rūšiavimo algoritmai (pavyzdžiui, "burbulo" algoritmas), t.y., $L^{\mathrm{SORT}}_{\mathrm{QUICK\_SORT}}(n) = \mathrm{O}(n^2)$, tuo tarpu vidutinis šio algoritmo sudėtingumas yra $\overline{L}^{\mathrm{SORT}}_{\mathrm{QUICK\_SORT}}(n) = \mathrm{O}(n\log_2 n)$, todėl šis algoritmas ir yra plačiai naudojamas.

Panagrinėkime tiesinės paieškos algoritmo vidutinį sudėtingumą. Pažymėkime $\widetilde{L}(U)$ palyginimų (t.y., esminių operacijų) skaičių, taikant algoritmą LIN_SEARCH uždaviniui U. Tarkime, nepriklausomai nuo pradinių duomenų, ieškomas objektas b su vienoda tikimybe $1/(n+1)$ gali sutapti su bet kuriuo sąrašo objektu a_i ir su tokia pat tikimybe jo iš viso nebus sąraše A. (Pastebėkite, kad čia mes pateikiame supaprastintas pradines sąlygas, kurios skiriasi nuo sąlygų, pateiktų vidutinio sudėtingumo apibrėžime, nes mes apibrėžiame ne konkretaus uždavinio U tikimybę $p(U)$, o tikimybę p_i, kad konkretus uždavinys pasitaikys iš tam tikro klasės $\mathcal{U}$ poaibio $\mathcal{U}_i$.) Kadangi tuo atveju, kai $b = a_i$, algoritmas LIN_SEARCH atlieka i palyginimų, tai vidutinis jo sudėtingumas bus

$$\overline{\widetilde{L}}^{\mathrm{SEARCH}}_{\mathrm{LIN\_SEARCH}}(n) = \frac{1}{n+1}\sum_{i=1}^{n+1} i = \frac{1}{n+1}\frac{(n+2)(n+1)}{2} = \frac{n}{2}+1.$$

1.3 Funkcijų augimo greičiai ir kombinatorinis sprogimas

Sudėtingumas $L(n)$ yra funkcija $L:\mathbb{N}\to\mathbb{N}$. Kai uždavinys yra nedidelis, pavyzdžiui, $n \leqslant 10$, net ir eksponentinio sudėtingumo algoritmai baigia darbą labai greitai. Tačiau situacija visiškai pasikeičia, kai uždavinio dydis n auga. Kai $n \geqslant 50$, daug uždavinių, kuriems nežinomi polinominio sudėtingumo algoritmai praktiškai jau tampa sunkiai įveikiami. Todėl labai svarbu žinoti kaip algoritmų ir uždavinių sudėtingumas $L(n)$ elgiasi asimptotiškai, t.y., kai $n\to\infty$. Šiame skyrelyje pateiksime keletą apibrėžimų iš funkcijų teorijos, kurie dažnai naudojami algoritmų analizėje. Kai kuriuos iš čia apibrėžtų žymėjimų mes jau naudojome ankstesniuose skyreliuose.

Tegu $f, g:\mathbb{N}\to\mathbb{R}^+$. Žymėsime:

- $f(n) = \mathrm{O}(g(n))$ (arba $f(n) \preccurlyeq g(n)$) ir sakysime, kad "f asimptotiškai yra ne aukštesnės eilės dydis kaip g", jei $\exists N\in\mathbb{N}$ ir $\exists c>0$: $f(n)\leqslant cg(n)\ \forall n\geq N$;
- $f(n) = \Omega(g(n))$ (arba $f(n) \succcurlyeq g(n)$) ir sakysime, kad "f asimptotiškai yra ne žemesnės eilės dydis kaip g", jei $\exists N\in\mathbb{N}$ ir $\exists c>0$: $f(n)\geqslant cg(n)\ \forall n\geq N$; akivaizdu, kad jei $f(n)=\mathrm{O}(g(n))$, tai $g(n)=\Omega(f(n))$;

- $f(n) = \Theta(g(n))$ (arba $f(n) \asymp g(n)$) ir sakysime, kad "f ir g asimptotiškai yra tokios pat eilės dydžiai", jei $f(n) = \mathrm{O}(g(n))$ ir $f(n) = \Omega(g(n))$;
- $f(n) = \mathrm{o}(g(n))$ (arba $f(n) \prec g(n)$) ir sakysime, kad "f asimptotiškai yra žemesnės eilės dydis už g", jei
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0;$$
- $f(n) \lesssim g(n)$ ir sakysime, kad "f yra asimptotiškai mažesnė arba lygi g", jei
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq 1;$$
- $f(n) \sim g(n)$ ir sakysime, kad "f yra asimptotiškai lygi g", jei $f(n) \lesssim g(n)$ ir $g(n) \lesssim f(n)$.

Pavyzdys 1.4. (i) $1000n^2 + 1000000n \log_2 n = \Theta(n^2)$ ir $1000n^2 + 1000000n \log_2 n = \mathrm{o}(n^3)$,

(ii) $n^{100} = \mathrm{o}(1.1^n)$,

(iii) $10^n = \mathrm{o}(n!)$,

(iv) $1^2 + 2^2 + \cdots + n^2 = \Theta(n^3)$, nes
$$1^2 + 2^2 + \cdots + n^2 > \left(\frac{n}{2} + 1\right)^2 + \cdots + n^2 > \left(\frac{n}{2}\right)^2 \cdot \frac{n}{2} = \frac{n^3}{8}$$
ir $1^2 + 2^2 + \cdots + n^2 < n \cdot n^2 = n^3$.

Kai sudėtingumas $L(n)$ yra ne aukštesnės eilės nei kai kurios dažniau pasitaikančios algoritmų analizėje funkcijos, tokiam sudėtingumui apibūdinti yra naudojami specialūs terminai. Keletą tokių terminų čia ir išvardinsime (sudėtingumo didėjimo tvarka):

- jei $L(n) = \mathrm{O}(\log_2 n)$, tai sudėtingumas vadinamas *logaritminiu*;
- jei $L(n) = \mathrm{O}(n)$, tai sudėtingumas vadinamas *tiesiniu*;
- jei $L(n) = \mathrm{O}(n^2)$, tai sudėtingumas vadinamas *kvadratiniu*;
- jei $L(n) = \mathrm{O}(n^3)$, tai sudėtingumas vadinamas *kubiniu*;
- jei egzistuoja $k \geqslant 1$: $L(n) = \mathrm{O}(n^k)$, tai sudėtingumas vadinamas *polinominiu*;
- jei egzistuoja $a > 1$: $L(n) = \mathrm{O}(a^n)$, tai sudėtingumas vadinamas *eksponentiniu*.

Uždavinio dydis n	Algoritmo sudėtingumas					
	$\log_2 n$	n	n^2	n^3	2^n	$n!$
10	3×10^{-9} s	10^{-8} s	10^{-7} s	10^{-6} s	10^{-6} s	3×10^{-3} s
20	4.5×10^{-9} s	2×10^{-8} s	4×10^{-7} s	8×10^{-6} s	10^{-3} s	77 metai
100	7×10^{-9} s	10^{-7} s	10^{-5} s	10^{-3} s	4×10^{13} metų	$*$
1000	10^{-8} s	10^{-6} s	10^{-3} s	1 s	$*$	$*$
1000000	2×10^{-8} s	10^{-3} s	17 min.	32 metai	$*$	$*$

1.1 lentelė: Kompiuterinio laiko lentelė įvairaus dydžio ir sudėtingumo uždaviniams.

Kai kuriems uždaviniams spręsti nėra žinoma jokių geresnių algoritmų už visų galimų variantų perrinkimą (brutalios jėgos algoritmą). Jei didėjant uždaviniui variantų skaičius auga greičiau už bet kokį polinomą (pavyzdžiui, eksponentiškai), tai tokį reiškinį vadina *kombinatoriniu sprogimu*. Taip auga, pavyzdžiui, Fibonacci skaičiai (žr. 3.2 skyrelį), kėlinių ilgio n skaičius, Hamiltono ciklų skaičius pilname grafe, pilno grafo klikų (pilnų pografių)skaičius, ir daugelio kitokių kombinatorinių objektų kiekis. 1.1 lentelė parodo, kad kombinatorinio sprogimo negalės įveikti patys greičiausi kompiuteriai, kiek bedidėtų jų greitis ateityje. Šioje lentelėje pateikiame CPU laiką įvairaus sudėtingumo algoritmams ir įvairaus dydžio uždaviniams, darant prielaidą, kad kompiuteris vykdo 10^9 (t.y., 1 milijardą) operacijų per sekundę. Žvaigždutė žymi laiką ilgesnį nei 10^{100} metų. Iš šios lentelės matyti, kad tobulesnių kompiuterių sukūrimas gali padėti įveikti tik tuos atvejus, kai šiuo metu laikas yra lygus 32 ir 77 metams. Visais atvejais, kurie lentelėje pažymėti žvaigždute, gali padėti tik greitesnių algoritmų sukūrimas arba apytikslių algoritmų naudojimas vietoje tikslių.

Pademonstruosime, kad sugalvoti greitesnį algoritmą gali būti dar svarbiau, negu sugebėti iš esmės padidinti procesoriaus greitį.

Pavyzdys 1.5. Tarkime, kokiam nors uždaviniui yra žinomas $\mathrm{O}(n^4)$ sudėtingumo algoritmas ir nėra jokio geresnio algoritmo (tokio sudėtingumo, pavyzdžiui, yra kai kurie maksimalaus srauto tinkle radimo algoritmai). Tarkime, konstanta, įeinanti į O apibrėžimą yra lygi 5. Jei superkompiuteris sugeba atlikti 1 milijardą operacijų per sekundę, o personalinis namų kompiuteris — 1 milijoną operacijų per sekundę, tai uždavinį dydžio $n = 1000$ pirmasis kompiuteris spręs $5 \cdot 10^{12}/10^9 = 5000$ sekundžių = 1 h 23 min. 20 sek., o antras kompiuteris — $5 \cdot 10^{12}/10^6 = 5000000$ sekundžių = 57 paros 20 h 53 min. 20 sek. Aišku, namų sąlygomis nepavyks išspręsti tokio uždavinio, nes per tą laiką bent kartą dings elektra arba namiškiams atsibos per naktis dūzgiantis kompiuteris.

Dabar tarkime, kad per keletą metų, investavus milijardines lėšas, pasaulyje pavyko 10 kartų pagreitinti superkompiuterį (iki 10 milijardų operacijų per sekundę) ir 10 kartų pagreitinti personalinius kompiuterius (iki 10 milijonų operacijų per sekundę). Tada superkompiuteris šį uždavinį išspręs per 8 min. 20 sek., na o paprastam mirtingajam su savo personaliniu kompiuteriu atsakymo reikės laukti 5 paras 18 h 53 min. 20 sek.

Tačiau gali būti ir kitaip. Vieną rytą koks nors genialus matematikos olimpiadininkas pliaukšteli sau per kaktą ir rėkia: "Radau algoritmą sudėtingumo $\mathrm{O}(n^3)$". Tarkime, kad konstanta vėl lygi 5. Tada senuoju superkompiuteriu šį uždavinį spręsime $5 \cdot 10^9/10^9 =$

5 sekundes, o senuoju geruoju namų PC-iuku — $5 \cdot 10^9/10^6 = 5000$ sekundžių = 1 h 23 min. 20 sek., t.y., tiek laiko kiek naudodamas seną algoritmą sugaišo superkompiuteris.

Išvada 1.1. *Geras algoritmas geriau už gerą kompiuterį!*

1.4 Viršutinis ir apatinis rūšiavimo uždavinio sudėtingumo įverčiai

Duotas objektų sąrašas $A = \{a_1, \ldots, a_n\}$, kuriame apibrėžtas pilnos tvarkos sąryšis $\leqslant$. Reikia duoto sąrašo elementus išdėstyti nemažėjančia tvarka: $A' = \{a_{i_1} \leqslant a_{i_2} \leqslant \cdots \leqslant a_{i_n}\}$ (žr. 1.2 skyrelį). Nagrinėsime tik tokius rūšiavimo algoritmus, kuriuos galima pavaizduoti *palyginimų medžiu*, t.y., mes kažkuriame algoritmo žingsnyje lyginame tarpusavyje du pasirinktus pradinio sąrašo elementus, po to, priklausomai nuo atsakymo, kuris iš tų elementų buvo didesnis, mes vėl lyginame du sąrašo elementus ir t.t. (žr. 1.1 pav.). Kadangi bendras rūšiavimo algoritmo sudėtingumas būna proporcingas tokių palyginimų skaičiui, tai mes skaičiuosime tik palyginimus. Taigi, $L_A^{\mathrm{SORT}}(n)$ reikš rūšiavimo algoritmo A atliktų palyginimų skaičių blogiausiu atveju, kai rūšiuojamų objektų skaičius yra n. Įrodysime, kad $L^{\mathrm{SORT}}(n) = \Theta(n \log_2 n)$, t.y., rūšiavimo uždavinio sudėtingumo viršutinis ir apatinis įverčiai skiriasi nuo $n \log_2 n$ tik pastoviu daugikliu.

Norėdami gauti viršutinį rūšiavimo uždavinio sudėtingumo įvertį, taikysime rūšiavimą sąlaja ($\mathrm{MERGE\_SORT}$). Tai rekursyvus algoritmas, naudojantis metodą "skaldyk ir valdyk". Tarkime, kad $n = 2^k$. Pradinį uždavinį $\mathrm{SORT}(A)$ suskaidome į du perpus mažesnius $\mathrm{SORT}(\{a_1, \ldots, a_{n/2}\})$ ir $\mathrm{SORT}(\{a_{n/2+1}, \ldots, a_n\})$, juos išsprendžiame, o po to du jau surūšiuotus masyvus suliejame į vieną surūšiuotą masyvą A'.

Pirmiausia pateikiame sąlajos algoritmą MERGE, kuris du surūšiuotus masyvus A ir B ilgio m ir n, atitinkamai, sulieja į naują masyvą C ilgio $m + n$.

function $C = \mathrm{MERGE}(A, B)$
$A[m+1] := \infty$;
$B[n+1] := \infty$;
$i := 1$;
$j := 1$;
for $k := 1$ **to** $m + n$ **do**
  if $A[i] < B[j]$ **then**
    $C[k] := A[i]$;
    $i := i + 1$;
  else
    $C[k] := B[j]$;
    $j := j + 1$;
  end;
end;

Akivaizdu, kad algoritmo MERGE sudėtingumas yra $\mathrm{O}(m+n)$. Pagrindinis algoritmas atrodo taip:

function $A' = \mathrm{MERGE\_SORT}(A)$
if $n = 1$ **then** $A' = A$
else $A' := \mathrm{MERGE}(\mathrm{MERGE\_SORT}(A[1 : n/2]), \mathrm{MERGE\_SORT}(A[n/2+1 : n]))$;
end;

Kaip šis algoritmas veikia pademonstruosime pavyzdžiu. Tarkime, $A = \{9, 1, 5, 4, 3, 7, 6, 2\}$. Po 3 rekursijos žingsnių šis masyvas bus suskaidytas į 8 masyvus ilgio 1, kurie grįžtant į aukštesnius rekursijos lygius bus palaipsniui suliejami į didesnius surūšiuotus masyvus:

$$\begin{aligned} A = \{9,1,5,4,3,7,6,2\} &\to \{9,1,5,4\}\{3,7,6,2\} \\ &\to \{9,1\}\{5,4\}\{3,7\}\{6,2\} \\ &\to \{9\}\{1\}\{5\}\{4\}\{3\}\{7\}\{6\}\{2\} \\ &\to \{1,9\}\{4,5\}\{3,7\}\{2,6\} \\ &\to \{1,4,5,9\}\{2,3,6,7\} \\ &\to A' = \{1,2,3,4,5,6,7,9\}. \end{aligned}$$

Algoritmo $\mathrm{MERGE\_SORT}$ sudėtingumas

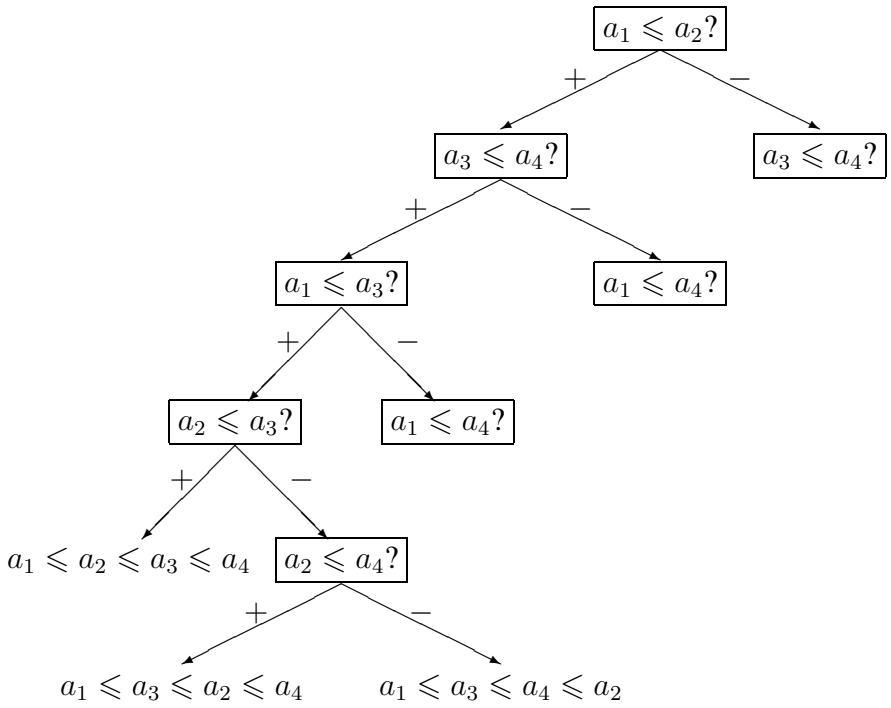
$$\begin{aligned} L(n) &\leqslant 2L\Big(\frac{n}{2}\Big) + cn \\ &= 2\Big(2L\Big(\frac{n}{4}\Big) + c\frac{n}{2}\Big) + cn \\ &= 4L\Big(\frac{n}{4}\Big) + 2cn = \cdots \\ &= 2^k L\Big(\frac{n}{2^k}\Big) + kcn = cn\log_2 n, \end{aligned}$$

nes $k = \log_2 n$ ir $L(1) = 0$. Taigi, kai n yra dvejeto laipsnis, viršutinį įvertį įrodėme. Jei $n \neq 2^k$, tai $\exists k$: $2^{k-1} < n < 2^k = n'$. Papildę masyvą A bet kokiais objektais, didesniais už patį didžiausią masyvo A objektą, iki ilgio n', galime pritaikyti algoritmą $\mathrm{MERGE\_SORT}$ šiam ilgesniam masyvui, o kai algoritmas baigs darbą, paimti tik pirmuosius n surūšiuoto masyvo elementų. Kadangi $n' < 2n$, gauname

$$L(n) \leqslant L(n') \leqslant cn'\log_2 n' < 2cn\log_2(2n) < c'n\log_2 n.$$

Viršutinį įvertį įrodėme, tačiau lieka nelabai aišku, kaip sąlajos algoritmą galima būtų vaizduoti palyginimų medžiu. Pav. 1.1 pateikiame tokio medžio fragmentą tuo atveju, kai $n = 4$. Taigi, algoritmas MERGE_SORT priklauso nagrinėjamų algoritmų klasei.

Norint gauti uždavinių klasės $\mathcal{U}$ sudėtingumo apatinį įvertį $L^{\mathcal{U}}(n) \geqslant f(n)$, reikia įrodyti, kad kiekvienas algoritmas, spręsdamas uždavinį $U \in \mathcal{U}$ dydžio n, darys ne mažiau kaip $f(n)$ žingsnių. Nesunku gauti aukštus apatinius įverčius tokiems uždaviniams, kurių



1.1 Pav.: Palyginimų medžio, vaizduojančio MERGE_SORT algoritmo veikimą, fragmentas.

pats sprendinys yra didelis. Dažniausiai tai yra įvairių kombinatorinių objektų generavimo ar paieškos uždaviniai, pavyzdžiui: (1) generuoti visus kėlinius ilgio n, (2) rasti visus duoto grafo karkasus, (3) rasti visas duoto grafo klikas. Akivaizdu, kad jei algoritmo išėjimas yra $f(n)$ skirtingų objektų, tai kadangi tie objektai yra skirtingi, tai kiekvienam iš jų reikia bent vieno algoritmo žingsnio, kuris nesutaps su kitais algoritmo žingsniais. Taigi, rezultatų kiekis yra trivialus apatinis įvertis algoritmo sudėtingumui. Pavyzdžiui, visų skirtingų kėlinių ilgio n generavimo uždavinio sudėtingumas yra $L(n) = \Theta(n!)$.[6] Viršutinis įvertis išplaukia iš to, kad nesunku nurodyti algoritmą, kuris generuoja visus kėlinius, pradėdamas nuo kėlinio $12 \ldots n$ ir kiekvieną kartą keisdamas vietomis tik 2 anksčiau gauto kėlinio elementus. Apatinis įvertis gaunamas, naudojantis tuo, kad rezultatų kiekis turi būti $n!$.

Deja, daugumos uždavinių sprendinys yra vienas ar keli skaičiai. Tokiems uždaviniams būna labai sunku gauti gerus apatinius įverčius, t.y., tokius apatinius įverčius, kurie būtų artimi viršutiniams. Netrivialūs apatiniai įverčiai buvo gauti tik nedaugeliui uždavinių. Kadangi algoritmo sąvoka yra neformali, norint gauti apatinį uždavinių klasės sudėtingumo įvertį, reikia pirmiausia formalizuoti algoritmus, t.y., griežtai apibrėžti klasę algoritmų, kuriuos taikysime pasirinktam uždaviniui. Pademonstruosime, kaip galima gauti tikslų apatinį įvertį (t.y., sutampantį su viršutiniu) rūšiavimo uždaviniui SORT.

Sąrašą ilgio n galima sutvarkyti $n!$ skirtingų būdų. Tai reiškia, kad bet kuris palygini-

[6] $f(n) = \Theta(g(n))$, jei $f(n) = \mathrm{O}(g(n))$ ir $f(n) = \Omega(g(n))$.

mų medis privalo turėti ne mažiau kaip $n!$ lapų (lapais vadiname medžio viršūnes, iš kurių neišeina nė vienas lankas). Jei lapų būtų mažiau, tada galima būtų parinkti du skirtingus skaičių $\{1, 2, \ldots, n\}$ kėlinius, kurie atvestų į tą patį lapą, t.y., jiems algoritmo atsakymas sutaptų. Tokiu atveju vienas iš tų kėlinių būtų surūšiuotas klaidingai. Primename, kad medžio aukščiu vadiname vidinių viršūnių (t.y., ne lapų) skaičių ilgiausioje jo šakoje. Kadangi konkretiems pradiniams duomenims rūšiavimo algoritmas praeina lygiai vieną medžio šaką, tai jo sudėtingumas (atliktų palyginimų skaičius blogiausiu atveju) sutampa su medžio aukščiu. Palyginimų medžiai yra binarieji medžiai, todėl palyginimų medis aukščio h gali turėti ne daugiau lapų, negu jų turės pilnas binarusis medis aukščio h, o toks medis turi 2^h lapų.

Tarkime, A yra bet kuris rūšiavimo algoritmas, kurį galima pavaizduoti palyginimų medžiu. Iš aukščiau pateiktų samprotavimų išplaukia, kad algoritmo A sudėtingumas $L(n)$ turi tenkinti nelygybę

$$2^{L(n)} \geqslant n!.$$

Pasinaudoję iš Stirlingo formulės gaunamu įverčiu

$$n! \geqslant \sqrt{2\pi n}\Big(\frac{n}{\mathrm{e}}\Big)^n,$$

gauname

$$L(n) \geqslant \log_2\left(\sqrt{2\pi n}\Big(\frac{n}{\mathrm{e}}\Big)^n\right) = n\log_2 n + \frac{1}{2}\log_2(2\pi n) - n\log_2 \mathrm{e} \sim n\log_2 n.$$

Matome, kad rūšiavimo uždavinio sudėtingumo viršutinis ir apatinis įverčiai skiriasi tik pastoviu daugikliu. Taigi, $L^{\mathrm{SORT}}(n) = \Theta(n\log_2 n)$.

1.5 Sveikųjų skaičių vaizdavimas kompiuterio atmintyje

Paprasčiausi kombinatoriniai objektai yra *sveikieji skaičiai*, *aibės*, *sekos*, *medžiai* ir *grafai*. Panagrinėsime galimus jų vaizdavimo būdus. Nuo pasirinktos duomenų struktūros ir nuo jos programinės realizacijos dažnai priklauso realizuojamo algoritmo žingsnių skaičius. Realizuojant konkretų algoritmą efektyviausias nagrinėjamų objektų klasės realizacijos būdas priklauso nuo:

(1) kam mes tuos objektus naudosime, ir

(2) kokias operacijas su jais atlikinėsime.

Kombinatoriniai algoritmai dažniausiai operuoja sveikaisiais skaičiais bei įvairiomis kombinatorinėmis jų konfigūracijomis (kėliniais, deriniais, gretiniais ir t.t.). Kadangi visada 1 bitą galima paskirti skaičiaus ženklo kodavimui, tai laikysime, kad sveikieji skaičiai yra neneigiami.

Sveikųjų skaičių vaizdavimas skaičiavimo sistemoje su pagrindu r. Labiausiai paplitęs sveikųjų skaičių vaizdavimo būdas yra skaičiaus vaizdavimas *pozicinėje skaičiavimo sistemoje su pagrindu* r:

$$N = (d_k d_{k-1} \dots d_1 d_0)_r = d_0 + d_1 r + d_2 r^2 + \dots + d_k r^k,$$

kur $0 \leqslant d_i < r$ ir $d_k \neq 0$, jei $N \neq 0$. Natūralusis skaičius $r > 1$ vadinamas *sistemos pagrindu*. Kasdieniame gyvenime naudojame pagrindą 10, paveldėtą iš arabų, o kompiuterio atmintis ir programavimo kalbos naudoja pagrindus 2, 8 ir 16. Senovės babiloniečiai naudojo šešiasdešimtainę, o indėnai majai — dvidešimtainę skaičiavimo sistemas. Pateiksime gerai žinomą algoritmą kaip rasti skaičiaus N išraišką r-ėje skaičiavimo sistemoje:

$d_0 := 0$;
$q := N$;
$k := 0$;
while $q \neq 0$ **do**
  $d_k := q \bmod r$;
  $q := \lfloor q/r \rfloor$;
  $k := k + 1$;
end;
if $k \neq 0$ **then** $k := k - 1$; **end**;

Pavyzdžiui, $13_{10} = 1101_2$, nes $13 = 6 \cdot 2 + 1$, $6 = 3 \cdot 2 + 0$, $3 = 1 \cdot 2 + 1$ ir $1 = 0 \cdot 2 + 1$.

Sveikųjų skaičių vaizdavimas mišrioje skaičiavimo sistemoje. Kartais sveikieji skaičiai yra vaizduojami *mišrioje skaičiavimo sistemoje su pagrindais* $r_0, r_1, \dots, r_{k-1}$:

$$N = d_0 + d_1 r_0 + d_2 r_0 r_1 + d_3 r_0 r_1 r_2 + \dots + d_k \prod_{i=0}^{k-1} r_i,$$

kur $0 \leqslant d_i < r_i$ ir $d_k \neq 0$, jei $N \neq 0$. Perėjimo nuo dešimtainės skaičiavimo sistemos prie mišrios skaičiavimo sistemos su pagrindais $r_0, r_1, \dots, r_{k-1}$ algoritmas yra visiškai panašus į aukščiau pateiktą algoritmą:

$d_0 := 0$;
$q := N$;
$k := 0$;
while $q \neq 0$ **do**
  $d_k := q \bmod r_k$;
  $q := \lfloor q/r_k \rfloor$;
  $k := k + 1$;
end;
if $k \neq 0$ **then** $k := k - 1$; **end**;

Pavyzdys 1.6. Skaičiuojant laiką, naudojame mišrią skaičiavimo sistemą su pagrindais $60, 60, 24, 7, 52$, pavyzdžiui 1000000 sek. $=$ 1 sav. 4 d. 13 val. 46 min. 40 sek., nes

$1000000 = 16666 \cdot 60 + 40$, $16666 = 277 \cdot 60 + 46$, $277 = 11 \cdot 24 + 13$, $11 = 1 \cdot 7 + 4$ ir $1 = 0 \cdot 52 + 1$. Beje, šią skaičiavimo sistemą jau naudojome įvado 1.5 pavyzdyje, skaičiuodami CPU laiką!

Pavyzdys 1.7. Kartais sveikieji skaičiai yra išreiškiami per faktorialus:

$$N = d_0 \cdot 0! + d_1 \cdot 1! + d_2 \cdot 2! + \cdots + d_k \cdot k!,$$

t.y., mišrioje skaičiavimo sistemoje su pagrindais $r_0 = 1, r_1 = 2, \ldots, r_{k-1} = k$. Kadangi $0 \leq d_i < r_i$, tai gauname, kad visada $d_0 = 0$.

Sveikųjų skaičių vaizdavimas liekanų vektoriais. Kai sveikieji skaičiai yra dideli, dvejetainėje skaičiavimo sistemoje jų išraiškos tampa ilgos, todėl veiksmai su tokiais skaičiais reikalauna daug dvejetainių operacijų. Pasirodo, veiksmams su dideliais sveikaisiais skaičiais galima sukonstruoti efektyvesnius algoritmus, operuojančius ne su pačiais skaičiais, o su liekanomis, gautomis dalinant tuos sveikus skaičius iš pasirinktų mažesnių sveikųjų skaičių. Tai, kad pagal liekanų vektorių galima vienareikšmiškai rasti jas atitinkantį sveikąjį skaičių, kiniečiai jau žinojo daugiau kaip prieš 2000 metų. Todėl veiksmai su liekanomis yra vadinama *moduline aritmetika*, arba *kiniečių aritmetika*.

Teorema 1.1 (Kiniečių teorema apie liekanas). *Tegu $p_0, p_1 \ldots, p_{k-1}$ yra poromis tarpusavyje pirminiai natūralieji skaičiai. Lyginių sistema*

$$\begin{aligned} u &\equiv r_0 \pmod{p_0}, \\ u &\equiv r_1 \pmod{p_1}, \\ &\vdots \\ u &\equiv r_{k-1} \pmod{p_{k-1}} \end{aligned}$$

turi vienintelį sprendinį $u \in \mathbb{Z}$: $0 \leqslant u < p_0 p_1 \cdots p_{k-1}$.

Taigi, kiekvieną sveikąjį skaičių u: $0 \leqslant u < p_0 p_1 \cdots p_{k-1}$ vienareikšmiškai atitinka jo liekanų vektorius $(r_0, r_1, \ldots, r_{k-1})$. Tai reiškia, kad vietoje to, kad atlikinėti veiksmus su dideliais sveikais skaičiais, mes galime juos koduoti liekanų vektoriais, po to atlikti tuos veiksmus su liekanomis ir gautą rezultatą (liekanų vektorių) vėl paversti sveikuoju skaičiumi. Skaičiavimus galima dar labiau pagreitinti veiksmus su liekanų vektoriais atliekant lygiagrečiai su k procesorių.

Pavyzdys 1.8. Mes galime sudaryti daugybos lentelę visiems natūraliesiems skaičiams, mažesniems už šimtą (t.y., matricą M dydžio 100×100). Tada tokių skaičių daugyba bus vykdoma labai greit: $i \cdot j = M(i, j)$. Pasirinkę, pavyzdžiui, tarpusavyje pirminius skaičius $99, 98, 97$ ir 95, mes galėsime greitai atlikinėti veiksmus su sveikais teigiamais skaičiais mažesniais už $99 \cdot 98 \cdot 97 \cdot 95 = 89\,403\,930$. Tarkime, reikia atlikti veiksmą $9999 \cdot 6666 + 12345678$. Nesunku patikrinti, kad šiuos skaičius atitiks tokie liekanų vektoriai (atitinkamai moduliu $99, 98, 97$ ir 95): $9999 \sim (0, 3, 8, 24)$, $6666 \sim (33, 2, 70, 16)$

ir $12345678 \sim (81, 30, 3, 48)$. Sudauginę pasirinktais moduliais vektorius $(0, 3, 8, 24)$ ir $(33, 2, 70, 16)$, gauname vektorių $(0, 6, 75, 4)$. Taigi, atsakymas bus liekanų vektorius $(0, 6, 75, 4) + (81, 30, 3, 48) = (81, 36, 78, 52)$. Vienintelis sistemos

$$\begin{aligned} u &\equiv 81 \pmod{99}, \\ u &\equiv 36 \pmod{98}, \\ u &\equiv 78 \pmod{97}, \\ u &\equiv 52 \pmod{95} \end{aligned}$$

sprendinys yra $u = 78\,999\,012$. Tai ir yra ieškomas atsakymas.

1.6 Sekos

Priminsime keletą apibrėžimų iš aibių teorijos. *Multiaibe* arba *šeima* vadiname aibę su pasikartojančiais elementais, t.y., rinkinį bet kokių objektų, kur vienodi objektai gali pasikartoti keletą kartų. Pilnai sutvarkytą baigtinę multiaibę vadiname *baigtine seka* arba *vektoriumi*. Pilnai sutvarkytą begalinę multiaibę vadiname *seka*. Terminas "pilnai sutvarkytą" reiškia, kad kiekvienam sekos elementui yra priskirta jo vieta; sukeitę du nelygius sekos elementus vietomis, gausime jau kitą seką. Baigtinės sekos pavyzdys gali būti žodis bet kokioje abėcėlėje A: $u = u_1 u_2 \dots u_m$, kur $u_i \in A$. Begalinės sekos pavyzdys yra pirminių skaičių aibė

$$P = \{2, 3, 5, 7, 11, 13, 17, 19, \dots\}$$

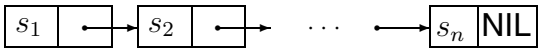
arba visų galimų žodžių abėcėlėje $A = \{a_1, a_2, \dots, a_n\}$ aibė

$$A^* = \{a_1, \dots, a_n, a_1a_1, a_1a_2, \dots, a_na_n, a_1a_1a_1, \dots\},$$

kur trumpesni žodžiai stovi prieš ilgesnius, o vienodo ilgio žodžiai yra išdėstyti leksikografine tvarka, t.y., taip, kaip žodyne. Kombinatoriniai algoritmai operuoja su baigtinėmis sekomis arba baigtiniais begalinių sekų fragmentais. Todėl toliau žodis "seka" reikš baigtinę seką.

Nuoseklus sekų vaizdavimas. Paprasčiausias sekų vaizdavimo būdas yra sekos $S = \{s_1, s_2, \dots, s_n\}$ elementus saugoti nuosekliai išdėstytus masyve ilgio n. Šis būdas leidžia lengvai surasti sekos elementą pagal jo numerį, tačiau yra nepatogus, kai tenka į seką įtraukti naujus elementus arba šalinti elementus iš sekos. Tada tenka perstumti ir kitus sekos elementus.

Sekų vaizdavimas sąrašais. Dinaminę seką $S = \{s_1, s_2, \dots, s_n\}$ patogu vaizduoti sąrašu. Kiekvieną sąrašo elementą sudaro informacinė dalis, kur talpiname pačius sekos elementus, ir adresinė dalis, kuri nurodo kokiu adresu rasime kitą sekos elementą. Pradiniu momentu toks sąrašas atrodo taip:



Sekos vaizdavimas sąrašais leidžia greitai vykdyti elementų įterpimą ir šalinimą iš sekos. Iš kitos pusės, šis būdas nėra patogus, kai norime rasti i-ąjį sekos elementą s_i. Be aukščiau pavaizduoto paprasto sąrašo dar naudojami dvigubai susieti sąrašai, kurių elementus sudaro ankstesnio elemento adresas, informacinė dalis ir sekančio elemento adresas.

Sekų vaizdavimas charakteringaisiais vektoriais. Kai nagrinėjamos sekos yra didesnės žinomos sekos $A = \{a_1, a_2, \ldots, a_n\}$ posekiai, tai seką $S = \{s_1, s_2, \ldots, s_m\}$ galima vaizduoti jos charakteringuoju vektoriumi $\kappa(S) = (\kappa_1, \kappa_2, \ldots, \kappa_n)$, kur

$$\kappa_i = \begin{cases} 1, & \text{jei } s_i \in A; \\ 0, & \text{jei } s_i \notin A. \end{cases}$$

Kadangi charakteringojo vektoriaus koordinatėms užtenka 1 bito atminties, tai šis sekų vaizdavimo būdas leidžia sutaupyti atmintį, jei nagrinėjami posekiai yra tankūs, t.y., jiems priklauso didelė dalis sekos A elementų.

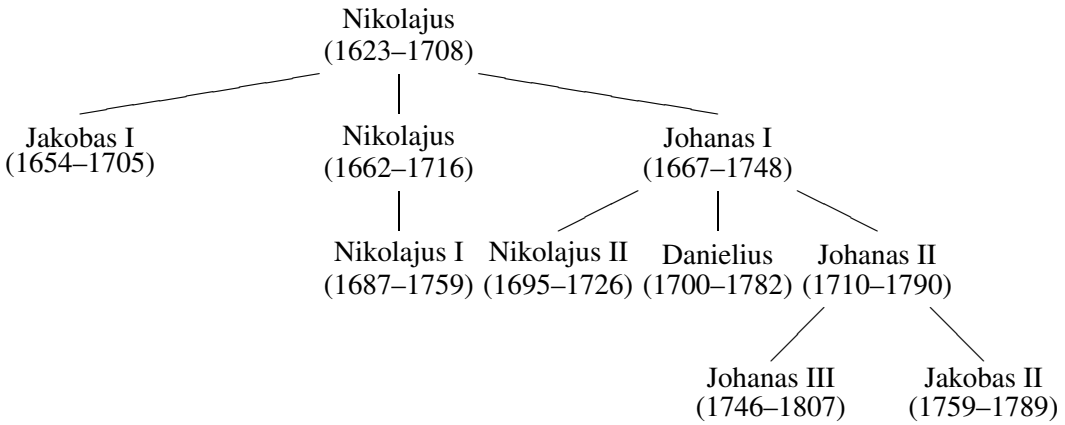
Pavyzdys 1.9. Pirminių skaičių, mažesnių už milijoną yra 78498. Kiekvienam skaičiui skiriant po 1 žodį iš 4 baitų, tokių skaičių seka, vaizduojant ją nuosekliai, užims 78498 žodžius. Kadangi, išskyrus skaičių 2, visi kiti pirminiai skaičiai yra nelyginiai, tai pirminių skaičių, mažesnių už milijoną, seką P galima vaizduoti kaip nelyginių natūraliųjų skaičių sekos $\{1, 3, 5, 7, 9, \ldots, 999999\}$ posekį su charakteringuoju vektoriumi $\kappa(P) = (0, 1, 1, 1, 0, 1, 1, 0, \ldots, 0\}$. Tokiam vektoriui reikės $500000/32 = 15625$ žodžių atminties, t.y., apie 5 kartus mažiau, negu pirmuoju būdu.

1.7 Medžiai

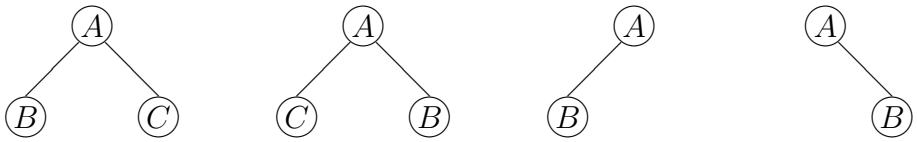
Medžiais yra vadinami neorientuoti jungūs grafai be ciklų (žr. 1.9 skyrelį). *Šakniniais medžiais* vadiname medžius, kurių viena viršūnė yra išskirta iš kitų ir vadinama *šaknimi*. Kadangi medžiai yra jungūs ir neturi ciklų, tai šakniniame medyje iš medžio šaknies r į bet kurią jo viršūnę v egzistuoja vienintelis kelias (ta pačia briauna galime eiti tik vieną kartą). Visos šiame kelyje sutinkamos medžio viršūnės u yra vadinamos viršūnės v *protėviais*. Jei viršūnė u yra viršūnės v protėvis, tai viršūnę v vadiname viršūnės u *palikuoniu*. Jei (u, v) yra paskutinė kelio iš šaknies r į viršūnę v briauna, tai viršūnė u yra vadinama viršūnės v *tėvu*, o viršūnė v yra vadinama viršūnės u *vaiku*. Jei kelios viršūnės turi bendrą tėvą, tai tos viršūnės yra vadinamos *broliais*.

Kadangi tikslų grafo apibrėžimą mes pateikiame tik 1.9 skyrelyje, tai čia pateiksime rekursyvų šakninio medžio apibrėžimą:

(i) Viena viršūnė r yra šakninis medis su šaknimi r.



1.2 Pav.: Bernoulli matematikų giminės medis.



1.3 Pav.: Binarieji medžiai aukščio 1.

(ii) Jei $T_1, T_2, \ldots, T_n$ yra šakniniai medžiai su šaknimis $r_1, r_2, \ldots, r_n$, tai prijungę naują viršūnę r ir sujungę ją briaunomis su kiekviena viršūne $r_1, r_2, \ldots, r_n$, vėl gauname šakninį medį su šaknimi r.

Kadangi algoritmuose paprastai naudojami tik šakniniai medžiai, tai toliau šakninį medį vadinsime tiesiog medžiu. Aukščiau pateikti terminai rodo, kad medžiais yra patogu vaizduoti giminystės ryšius (tokie medžiai yra vadinami *genealoginiais medžiais*). Pavyzdžiui, 1.2 pav. vaizduoja Bernoulli matematikų giminės medį.

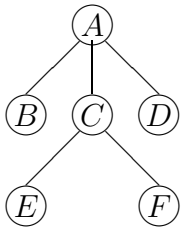
Binariuoju medžiu vadiname medį, kurio kiekviena viršūnė turi ne daugiau kaip 2 vaikus, ir kiekvienam vaikui yra žinoma, ar jis kairysis, ar dešinysis vaikas (taigi, viršūnė gali turėti ir vieną vaiką, ir tas vienas vaikas gali būti ir dešinysis!). Pav. 1.3 matome 4 skirtingus binariuosius medžius. Rekursyviai binarieji medžiai apibrėžiami taip:

(i) Tuščia aibė yra binarusis medis.

(ii) Jei T_1, T_2 yra binarieji medžiai, tai prijungę naują viršūnę r ir sujungę ją briaunomis su kairiojo pomedžio šaknimi r_1 ir dešiniojo pomedžio šaknimi r_2, vėl gauname binarųjį medį su šaknimi r (jei kuris nors pomedis tuščias, tada su juo nejungiame).

Vaizduojant medžius kompiuterio atmintyje, kiekvienai viršūnei yra skiriamas įrašas, sudarytas iš informacinės dalies ir vienos ar kelių nuorodų. Pagrindiniai medžių vaizdavimo būdai yra šie:

1. *Tėvų nuorodomis*, t.y., kiekvienai viršūnei nurodant jos tėvą. Šis būdas yra naudojamas rekursyviuose algoritmuose, kai norime išsaugoti informaciją apie tai, iš kurios



1.4 Pav.: Medžio pavyzdys.

viršūnės mes atėjome. Tačiau jis yra nepatogus, kai reikia surasti viršūnės palikuonius. Be to, jis netinka binariesiems medžiams, nes nurodant tik tėvą, neaišku, ar vaikas yra kairysis, ar dešinysis.

2. *Vaikų nuorodomis*, t.y., kiekvienai viršūnei nurodant visus jos vaikus. Šiuo atveju reikalinga žinoti, kiek daugiausia vaikų gali turėti medžio viršūnės. Jei maksimalus vaikų skaičius lygus m, tai kiekvienas įrašas turės m nuorodų. Kadangi daugelis nuorodų gali būti tuščios (NIL), tai šis būdas reikalauna daug atminties. Tačiau jis labai tinka binariesiems medžiams, kur $m = 2$.

3. Kiekvienai viršūnei nurodant jos *kairyjį vaiką ir dešinyjį brolį* (angl. left-child, right-sibling). Vaizduojant medžius šiuo būdu, kiekvienai viršūnei reikia tik dviejų nuorodų.

Pavyzdys 1.10. Visais 3 išvardintais būdais pavaizduosime medį iš 1.4 paveikslėlio. Vietoje nuorodų naudosime masyvo indeksus.

1. Tėvų nuorodos:

Indeksas	INFO	Tėvas
1	A	NIL
2	B	1
3	C	1
4	D	1
5	E	3
6	F	3

2. Vaikų nuorodos:

Indeksas	INFO	1 vaikas	2 vaikas	3 vaikas
1	A	2	3	4
2	B	NIL	NIL	NIL
3	C	5	6	NIL
4	D	NIL	NIL	NIL
5	E	NIL	NIL	NIL
6	F	NIL	NIL	NIL

3. Nurodant kairyjį vaiką ir dešinyjį brolį:

Indeksas	INFO	Kairysis vaikas	Dešinysis brolis
1	A	2	NIL
2	B	NIL	3
3	C	5	4
4	D	NIL	NIL
5	E	NIL	6
6	F	NIL	NIL

1.8 Aibės

Kadangi algoritmai operuoja tik su baigtinėmis aibėmis, tai sunumeravus kuria nors tvarka duotos aibės elementus, ši aibė virsta seka. Taigi aibėms tinka visi vaizdavimo būdai, kurie yra naudojami sekoms vaizduoti:

1. Nuoseklus vaizdavimas.
2. Vaizdavimas sąrašais.
3. Vaizdavimas charakteringaisiais vektoriais.

Kartais būna patogu aibes vaizduoti dar vienu būdu:

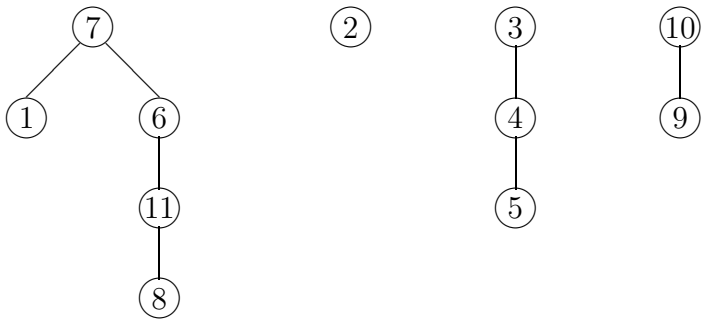
4. Mišku.

Mišku vadiname vieną ar keletą medžių. Tarkime, kad visos nagrinėjamos aibės yra poromis nesusikertantys didesnės aibės A poaibiai. Kiekvienam poaibiui identifikuoti išskiriame iš kitų bet kurį to poaibio elementą ir jį vadiname poaibio *vardu*. Tarkime, kad mums dažnai reikia rasti, kuriame poaibyje yra duotas aibės A elementas x (operacija $\mathrm{FIND}(x)$), o taip pat dažnai reikia sujungti du poaibius su vardais x ir y į vieną naują poaibį (operacija $\mathrm{UNION}(x, y)$). Tada šiuos poaibius galime vaizduoti medžiais, kurių šaknys yra poaibių vardai. Realizuojant tokią struktūrą kompiuterio atmintyje kiekvienam aibės elementui (medžio viršūnei) pakanka saugoti nuorodą į jos tėvą. Pradiniu momentu laikome, kad kiekvienas aibės elementas sudaro poaibį iš vieno elemento, t.y., jis yra medžio šaknis. Operacija $\mathrm{UNION}(x, y)$ reikš dviejų medžių su šaknimis x ir y sujungimą į vieną naują medį su šaknimi x arba y, o operacija $\mathrm{FIND}(x)$ reikš paiešką miške. Toks aibės vaizdavimo būdas leis atlikinėti minėtas dvi operacijas greičiau, negu tai leidžia kiti aibės vaizdavimo būdai.

Pavyzdys 1.11. Duota aibė $A = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$, suskaidyta į 4 poaibius

$$\{1, 6, \boxed{7}, 8, 11\}, \quad \{\boxed{2}\}, \quad \{\boxed{3}, 4, 5\}, \quad \{9, \boxed{10}\},$$

kur kvadratėliais pažymėjome poaibių vardus. Tokia struktūra galėjo susidaryti, pavyzdžiui, vykdant štai tokią UNION operacijų seką aibėje A:



1.5 Pav.: Aibės A vaizdavimas mišku.

UNION$(8, 11)$;
UNION$(6, 11)$;
UNION$(6, 7)$;
UNION$(1, 7)$;
UNION$(4, 5)$;
UNION$(3, 4)$;
UNION$(9, 10)$;

Tada šią aibę atitiks miškas, pavaizduotas 1.5 pav. Tarkime, kad dabar reikia sujungti poaibius, į kuriuos pateko elementai 11 ir 5. Tai atliks tokia programėlė:

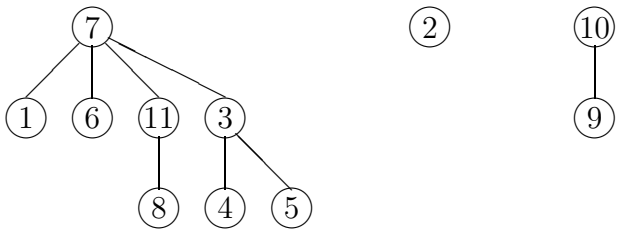
$x :=$ FIND(11);
$y :=$ FIND(5);
if $x \neq y$ **then** UNION(x, y); **end**;

Operaciją FIND(x) galima realizuoti, naudojant *kelių suspaudimą*. Tai reiškia, kad kuriame nors medyje eidami viena šaka iš elemento x į medžio šaknį, mes įsimename visas praeitas viršūnes, o radę medžio šaknį y keičiame visų jų nuorodas į y. Tai leidžia nuolat riboti medžių gylį ir tuo pačiu pagreitinti operacijų FIND vykdymą. Taigi, naudojant kelių suspaudimą, trijų aukščiau nurodytų operacijų, pritaikytų miškui iš 1.5 pav. rezultatas bus miškas, vaizduojamas 1.6 pav. Galima įrodyti, kad naudojant kelių suspaudimą ir *medžio šakų balansavimą* (atliekant operaciją UNION) m FIND ir UNION operacijų galima įvykdyti per O$(m\alpha(m, m))$ žingsnių, kur $\alpha(m, n)$ yra "atvirkštinė Akermano funkcija". Ši funkcija auga taip lėtai, kad praktikoje galime laikyti, kad $\alpha(m, n) \leq 4$, t.y., gauname praktiškai tiesinį sudėtingumą (žr. [CLR, 22.3]).

1.9 Grafai ir jų vaizdavimas

1.9.1 Grafo apibrėžimas ir pagrindinės sąvokos

Grafu vadiname porą $G = (V, E)$, kur V yra bet kokia netuščia aibė, o E yra bet kuris aibės porų iš V elementų multipoaibis. Jei tos poros yra vektoriai, tai grafą vadiname



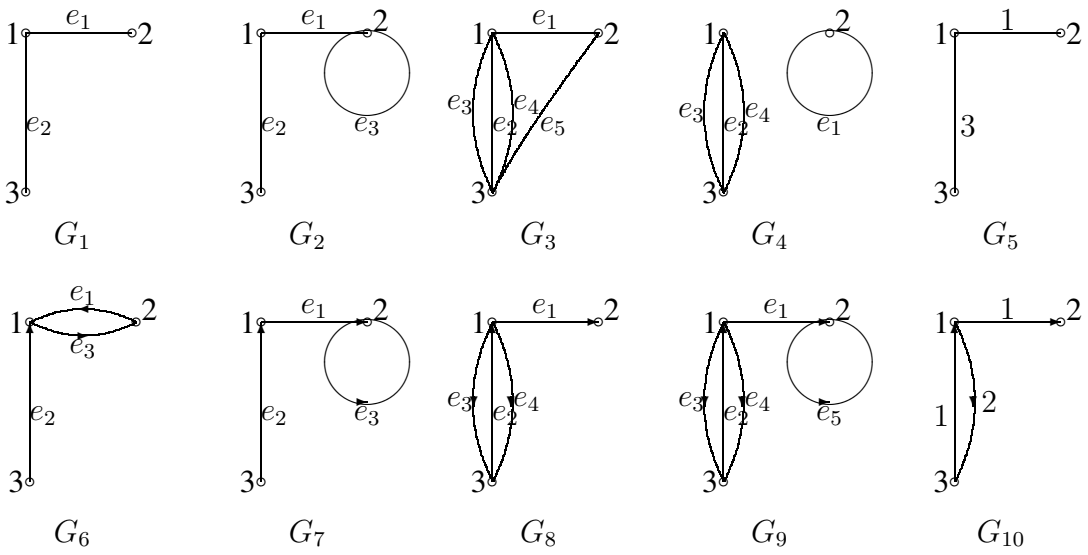
1.6 Pav.: Pertvarkyta aibė A.

orientuotu grafu arba *orgrafu*. Jei poros yra tiesiog aibės V multipoaibiai, tai grafą vadiname *neorientuotu* arba tiesiog grafu ("multi" čia pridėjome todėl, kad gali būti ir poros $\{v, v\} \in E$, kur $v \in V$). Aibė V vadinama grafo G *viršūnių aibe*. Orgrafuose porą $e = (u, v) \in E$ vadiname *lanku* ir sakome, kad lankas $e = (u, v)$ jungia orgrafo G viršūnes u ir v. Dvi poros (u, v) ir (v, u) orgrafe reiškia du skirtingus lankus. Neorientuotuose grafuose viršūnių $u, v \in V$ porą taip pat žymėsime (u, v). Jei $e = (u, v) \in E$, tai porą (u, v) vadiname grafo G *briauna* ir sakome, kad briauna $e = (u, v)$ jungia grafo G viršūnes u ir v. Taip pat sakoma, kad briauna (lankas) $e = (u, v)$ yra *incidentinė* (*-is*) viršūnėms u ir v. Neorientuotuose grafuose dvi poros (u, v) ir (v, u) reiškia tą pačią briauną.

Aukščiau apibrėžti grafai ir orgrafai gali turėti kelias vienodas briaunas (lankus), kurios vadinamos kartotinėmis briaunomis (kartotiniais lankais). Taip pat tokie grafai gali turėti ir *kilpas*, t.y., briaunas (lankus) pavidalo $e = (v, v) \in E$. Kai kuriuose vadovėliuose grafais vadinami tik grafai, neturintys nei kartotinių briaunų, nei kilpų (mes tokius grafus vadinsime *paprastaisiais grafais*), tuo tarpu mūsų apibrėžti grafai ten vadinami *pseudografais*. Grafai su kartotinėmis briaunomis, bet be kilpų, yra vadinami *multigrafais*. Taigi, sprendžiant bet kurį uždavinį, susijusį su grafais, visada reikia pasitikslinti, ar kalbama apie orgrafus ar neorientuotus grafus ir ar grafai gali turėti kartotinių briaunų bei kilpų. Apibendrindami, gauname iš viso 8 galimus variantus: grafai gali būti 2 tipų (orientuoti ir neorientuoti), o kiekvieno tipo grafai dar gali būti 4 rūšių: be kartotinių briaunų ir kilpų, be kartotinių briaunų bet su kilpom, be kilpų bet su kartotinėmis briaunomis ir pagaliau su kartotinėmis briaunomis ir kilpomis.

Be aukščiau išvardintų grafų, grafų teorija taip pat nagrinėja taip vadinamus *svorinius grafus*. Svorinis grafas — tai trejetas $G = (V, E, \omega)$, kur V yra viršūnių aibė, E yra briaunų (lankų) aibė ir $\omega : E \to \mathbb{R}$ yra svorinė funkcija, kiekvienai briaunai (lankui) e priskirianti svorį $\omega(e)$. Vietoje realiųjų skaičių aibės $\mathbb{R}$, briaunų svoriai gali būti iš kitokios aibės, pavyzdžiui $\mathbb{R}^+ = [0, \infty)$ arba $\mathbb{N} = \{0, 1, 2, \ldots\}$. Briaunos $e = (u, v)$ svoris $\omega(e)$ dažniausiai reiškia atstumą tarp viršūnių u ir v, tačiau jis gali turėti ir kitą prasmę. Pridėję prie aukščiau išvardintų 8 grafų variantų svorinius grafus bei orgrafus, viso gauname jau 10 galimų variantų. Visi jie yra pavaizduoti 1.7 pav.

Orgrafo viršūnės v *įėjimo laipsniu* $\mathrm{indg}(v)$ vadiname į šią viršūnę įeinančių lankų skaičių, o *išėjimo laipsniu* $\mathrm{outdg}(v)$ — iš šios viršūnės išeinančių lankų skaičių. Neorientuoto grafo viršūnės v *įėjimo laipsniu* $\mathrm{dg}(v)$ vadiname į šią viršūnę įeinančių briaunų skaičių. *Keliu, jungiančiu dvi orgrafo viršūnes* u *ir* v, vadiname lankų seką $K(u, v) =$



1.7 Pav.: Skirtingų rūšių grafai bei orgrafai. G_1 — paprastas grafas, G_2 — grafas su kilpomis, G_3 — multigrafas, G_4 — grafas, G_5 — svorinis grafas, G_6 — paprastas orgrafas, G_7 — orgrafas su kilpomis, G_8 — multiorgrafas, G_9 — orgrafas, G_{10} — svorinis orgrafas.

$\{(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\}$, kurioje $v_0 = u$, $v_k = v$ ir du gretimi sekos lankai turi bendrą viršūnę. Kelią $K(u, v)$ sudarančių lankų skaičių k vadiname kelio $K(u, v)$ ilgiu. Vietoje lankų išvardijimo kelią $K(u, v)$ dažnai apibrėžia išvardijant tik viršūnes, per kurias eina šis kelias: $K(u, v) = \{u, v_1, \ldots, v_{k-1}, v\}$. Netuščią kelią $K(u, u)$ vadiname *ciklu*. Pav. 1.7 pavaizduotas orgrafas G_6 turi ciklą $C = \{121\}$. Analogiškai keliai ir ciklai yra apibrėžiami ir neorientuotiems grafams. *Oilerio*[7] *ciklu* vadiname ciklą, praeinantį kiekviena grafo briauna (lanku) lygiai po vieną kartą. *Hamiltono*[8] *ciklu* vadiname ciklą, praeinantį per kiekvieną grafo viršūnę lygiai po vieną kartą. Pavyzdžiui, 1.7 pav. pavaiz-

[7]**Leonhard Euler (1707–1783).** Leonardas Oileris gimė kalvinų dvasininko šeimoje netoli Bazelio, Šveicarija. Būdamas 13 metų, jis įstojo į Bazelio universitetą studijuoti teologijos, tačiau vėliau ėmė studijuoti matematiką ir būdamas 16 metų gavo filosofijos magistro laipsnį. 1727 m. Petras Didysis jį pakvietė į Sankt Peterburgą, kur Oileris gyveno iki 1741 m. 1741–1766 m. jis gyveno Berlyne ir dirbo Berlyno Akademijoje, o likusį gyvenimą praleido Sankt Peterburge. Oileris buvo nepaprastai produktyvus mokslininkas, parašęs virš 1100 knygų ir straipsnių. Po savo mirties jis paliko tiek neatspausdintų rankraščių, kad prireikė 47 metų išleisti visiems jo darbams! Oileris įnešė savo įnašą tiek įvairiose matematikos srityse (skaičių teorijoje, kombinatorikoje, matematinėje analizėje), tiek ir jos taikymuose muzikoje bei laivų statyboje. Jis turėjo 13 vaikų, ir dažnai savo mokslinį darbą dirbdavo su vienu ar dviem vaikais sėdinčiais ant jo kelių. Paskutinius 17 gyvenimo metų Oileris buvo aklas, tačiau jo fenomenalios atminties dėka tai nė kiek nesumažino jo mokslinio produktyvumo.

[8]**William Rowan Hamilton (1805–1865).** Žymiausias airių mokslininkas Wiljamas Hamiltonas gimė Dubline teisininko šeimoje. Būdamas 3 metų, jis jau skaitė ir skaičiavo, o sulaukęs 8 metų mokėjo lotynų, graikų ir hebrajų kalbas. Turėdamas 17 metų jis susidomėjo astronomija ir matematika. Dar būdamas studentu, jis tapo Airijos Karališkuoju Astronomu ir juo buvo iki pat mirties. Hamiltonas pasiekė svarbių rezultatų optikoje, algebroje ir dinamikoje. Algebroje jis pasiūlė taip vadinamus *kvaternionus*. 1857 m. jis sukūrė geometrinį žaidimą, kurio idėją pardavė prekybos agentui. Vienoje iš šio žaidimo versijų reikėjo rasti ciklą, praeinantį per dodekaedro viršūnes lygiai po 1 kartą.

duotame grafe G_3 egzistuoja tiek Oilerio ciklas $\{e_3, e_2, e_1, e_5, e_4\}$ (prasidedantis viršūnėje 1), tiek Hamiltono ciklas $\{e_1, e_5, e_2\}$.

Bet kurį grafą $G' = (V', E')$, kurio visos viršūnės ir briaunos taip pat priklauso ir grafui $G = (V, E)$ (t.y., $V' \subseteq V$ ir $E' \subseteq E$), vadinsime grafo G *pografiu*. Grafą G vadiname *jungiu*, jei tarp bet kurių dviejų skirtingų jo viršūnių u ir v grafe G egzistuoja kelias $K(u, v)$. Grafą, sudarytą iš vienos izoliuotos viršūnės taip pat laikome jungiu. Orgrafą vadiname jungiu, jei jį atitinkantis grafas (t.y., grafas, gaunamas iš orgrafo "pašalinus" briaunų orientaciją) yra jungus. Maksimalius[9] jungius grafo G pografius vadiname grafo G *jungumo komponentėmis* arba tiesiog *komponentėmis*. Visi 1.7 pav. vaizduojami grafai yra iš 1 komponentės, išskyrus grafą G_4, sudarytą iš 2 komponenčių.

Toliau nagrinėsime baigtinių grafų vaizdavimo būdus. Grafo $G = (V, E)$ viršūnių skaičių žymėsime raide n, o briaunų (lankų) skaičių raide m. Taigi, $|V| = n$ ir $|E| = m$, kur $n = 1, 2, \ldots$ ir $m = 0, 1, 2, \ldots$. Kai $m = 0$, grafo G briaunų aibė yra tuščia. Toks grafas yra sudarytas iš n izoliuotų viršūnių. Jis vadinamas *tuščiuoju grafu* ir žymimas O_n. Jei grafas yra paprastasis, tai bet kurios dvi jo viršūnės yra sujungtos ne daugiau kaip viena briauna. Be to, briaunos jungia tik skirtingas viršūnes. Taigi paprastasis grafas turi ne daugiau kaip C_n^2 briaunų, t.y., $0 \leqslant m \leqslant C_n^2 = n(n-1)/2$. Grafas, kuriame kiekviena viršūnė yra sujungta su kiekviena kita viršūne, yra vadinamas *pilnuoju grafu* ir žymimas K_n. Matome, kad paprastieji grafai visada turi $m = \mathrm{O}(n^2)$ briaunų. Jei $m = \mathrm{o}(n^2)$, tai grafą vadiname *retu*. Jei $m = \Omega(n^2)$, tai grafą vadiname *tankiu*. Taupant kompiuterio atmintį, priklausomai nuo to ar grafas yra retas ar tankus, dideliems grafams vaizduoti gali būti taikomi skirtingi būdai.

1.9.2 Grafų vaizdavimo būdai

Tegu $G = (V, E)$, kur $V = \{v_1, v_2, \ldots, v_n\}$ ir $E = \{e_1, e_2, \ldots, e_m\}$. Vaizduojant grafus kompiuterio atmintyje įvairiomis stuktūromis, laikysime, kad sveikieji skaičiai yra vaizduojami žodžiais ilgio l. Pvz., jei žodį sudaro 4 baitai, tai $l = 32$.

Grafinis vaizdavimo būdas. Nedidelius grafus patogu vaizduoti grafiškai plokščia diagrama, kurioje kiekviena viršūnė $v \in V$ vaizduojama tašku (arba mažu apskritimu) su greta prirašyta žyme v, o kiekviena briauna $e = (u, v)$ vaizduojama tiesės atkarpa ar kitokia linija, jungiančia taškus pažymėtus u ir v (ši linija negali daugiau eiti per jokią kitą grafo viršūnę). Jei grafas orientuotas, tai lanką (u, v) atitinkanti linija savo antrame gale turi rodyklę, nukreiptą į viršūnę v (žr. 1.7 pav.). Kadangi ne kiekvieną grafą galima pavaizduoti plokščia diagrama taip, kad briaunos nesusikirstų, tai reikia atkreipti dėmesį, kad paprasti briaunų susikirtimo taškai nelaikomi grafo viršūnėmis. Sprendžiant su grafais susijusius uždavinius, dažnai būna patogu pradinį grafą grafiškai vaizduoti kompiuterio ekrane ir su pelyte kaitalioti šio grafo viršūnes bei briaunas.

[9] Aibę A vadiname *maksimalia* kokios nors savybės S atžvilgiu, jei aibė A turi savybę S, o prie jos prijungus dar kokį nors elementą jau gausime aibę, kuri nebeturės šios savybės. Kadangi grafas yra viršūnių ir briaunų aibė, tai žodis "maksimalus" grafams reiškia, kad nebegalime prijungti nė vienos viršūnės ar briaunos.

Gretimumo matrica. Grafo (arba orgrafo) $G = (V, E)$ *gretimumo matrica* vadiname matricą $A = (a_{ij})$, kur

$$a_{ij} = \begin{cases} 1, & \text{jei } (v_i, v_j) \in E; \\ 0, & \text{priešingu atveju.} \end{cases}$$

Taigi, gretimumo matricos A elementas a_{ij} yra vienetas, jei viršūnes v_i ir v_j jungia briauna (t.y., jos yra *gretimos*), ir nulis priešingu atveju. Gretimumo matrica kartais dar yra vadinama sujungimų arba jungumo matrica. Multigrafams gretimumo matricoje vietoje 1 ir 0 tiesiog imame dvi viršūnes jungiančių briaunų skaičių. Svoriniuose grafuose gretimumo matricos elementai yra briaunų svoriai (jei dvi skirtingos viršūnės v_i ir v_j nėra sujungtos briauna, tai laikoma $a_{ij} = \infty$). Tokia matrica dar vadinama *svorine* arba *atstumų* matrica.

Pateiksime 1.7 pav. vaizduojamų grafų gretimumo matricas:

$$A_1 = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}, \qquad A_6 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix},$$
$$A_2 = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}, \qquad A_7 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix},$$
$$A_3 = \begin{pmatrix} 0 & 1 & 3 \\ 1 & 0 & 1 \\ 3 & 1 & 0 \end{pmatrix}, \qquad A_8 = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix},$$
$$A_4 = \begin{pmatrix} 0 & 0 & 3 \\ 0 & 1 & 0 \\ 3 & 0 & 0 \end{pmatrix}, \qquad A_9 = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix},$$
$$A_5 = \begin{pmatrix} 0 & 1 & 3 \\ 1 & 0 & \infty \\ 3 & \infty & 0 \end{pmatrix}, \quad A_{10} = \begin{pmatrix} 0 & 1 & 2 \\ \infty & 0 & \infty \\ 1 & \infty & 0 \end{pmatrix}.$$

Išvardinsime akivaizdžias paprasto grafo gretimumo matricos savybes:

1. Matrica A dvejetainė, t.y., $A \in \{0, 1\}^{n^2}$.
2. Matrica A simetriška, t.y., $a_{ij} = a_{ji}\ \forall i, j = 1, 2, \ldots, n$.
3. Matricos A įstrižainė yra sudaryta iš nulių, t.y., $a_{ii} = 0\ \forall i = 1, 2, \ldots, n$.
4. Matricos A i-osios eilutės (stulpelio) suma yra lygi viršūnės v_i laipsniui:

$$\sum_{j=1}^{n} a_{ij} = \sum_{j=1}^{n} a_{ji} = \mathrm{dg}(v_i) \quad \forall i = 1, 2, \ldots, n.$$

Nesunku pastebėti, kad tokiai matricai reikės n^2 bitų arba $n]n/l[$ žodžių atminties (naują eilutę talpiname nuo naujo žodžio pradžios).

Gretimumo struktūra. Grafo (arba orgrafo) $G = (V, E)$ *gretimumo struktūra* vadiname n sąrašų pavidalo

$$v_i \to v_{i1} \to v_{i2} \to \cdots \to v_{ik_i}, \quad i = 1, 2, \ldots, n,$$

kur $v_{i1}, v_{i2}, \ldots, v_{ik_i}$ yra tos viršūnės, kurios su viršūne v_i yra sujungtos briaunomis (lankais). Pavyzdžiui, grafo G_1 gretimumo struktūra bus

$$\begin{array}{l} 1 \to 2 \to 3 \to \mathsf{nil} \\ 2 \to 1 \to \mathsf{nil} \\ 3 \to 1 \to \mathsf{nil} \end{array}$$

Matome, kad gretimumo struktūrai reikės $2(2m + n)l$ bitų arba $2(2m + n)$ žodžių atminties (pusė atminties bus skirta rodyklėms, nurodančioms kito sąrašo elemento adresą).

Incidencijų matrica. Grafo be kilpų $G = (V, E)$ *incidencijų matrica* vadiname $n \times m$ matricą $B = (b_{ij})$, kur

$$b_{ij} = \begin{cases} 1, & \text{jei viršūnė } v_i \text{ yra incidentiška briaunai } e_j; \\ 0, & \text{priešingu atveju.} \end{cases}$$

Taigi, incidencijų matricos B elementas b_{ij} yra vienetas tada ir tik tada, kai viršūnė v_i yra vienas iš briaunos e_j galų.

Orgrafo be kilpų $G = (V, E)$ *incidencijų matrica* vadiname $n \times m$ matricą $B = (b_{ij})$, kur

$$b_{ij} = \begin{cases} 1, & \text{jei } e_j = (v_i, v_k); \\ -1, & \text{jei } e_j = (v_k, v_i); \\ 0, & \text{jei viršūnė } v_i \text{ nėra incidentiška lankui } e_j; \end{cases}$$

Pavyzdžiui, grafų G_1 ir G_6 incidencijų matricos bus

$$B_1 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \text{ir} \quad B_6 = \begin{pmatrix} -1 & 1 & -1 \\ 0 & -1 & 1 \\ 1 & 0 & 0 \end{pmatrix}.$$

Išvardinsime paprasto grafo incidencijų matricos savybes:

1. Matrica B dvejetainė, t.y., $A \in \{0, 1\}^{nm}$.
2. Matricos B i-osios eilutės suma yra lygi viršūnės v_i laipsniui:

$$\sum_{j=1}^{n} a_{ij} = \deg(v_i) \quad \forall i = 1, 2, \ldots, n.$$

3. Matricos B bet kurio stulpelio suma yra lygi 2.

Incidencijų matricai reikės nm bitų arba $n]m/l[$ žodžių atminties.

Briaunų masyvas. Vienas iš paprasčiausių (or)grafo vaizdavimo būdų yra išvardinti visas jo briaunas. Kadangi grafas gali turėti ir izoliuotų viršūnių, tai reikia ne tik išvardinti visas briaunas, bet ir nurodyti grafo viršūnių skaičių n. Taigi *briaunų masyvu* vadinsime vektorių $\vec{b} = (n, v_{11}, v_{12}, v_{21}, v_{22}, \ldots, v_{m1}, v_{m2})$, kur $e_i \in E \Rightarrow e_i = (v_{i1}, v_{i2})$ $(i = 1, \ldots, m)$. Pavyzdžiui, grafus G_1 ir G_6 atitiks vektoriai $\vec{b}_1 = (3, 1, 3, 1, 2)$ ir $\vec{b}_6 = (3, 3, 1, 1, 2, 2, 1)$. Dar patogiau yra lankų (briaunų) pradžias saugoti viename masyve ($\vec{p}$), o galus kitame ($\vec{g}$).

Briaunų masyvui reikia $(2m + 1)l$ bitų arba $2m + 1$ žodžių atminties.

Grafų vaizdavimo būdo pasirinkimas priklauso nuo sprendžiamo uždavinio ir nuo to, kiek grafas gali turėti briaunų, t.y., ar jis yra tankus ar retas. Jei mes taupome atmintį, tai retiems grafams ekonomiškiausi būdai yra grafo vaizdavimas briaunų masyvu arba gretimumo struktūra, o tankiems grafams — gretimumo matrica. Kadangi pereiti nuo vieno būdo prie kito pakanka $\mathrm{O}(n^2)$ operacijų, tai algoritmų, kurių sudėtingumas yra $\geqslant \mathrm{const} \cdot n^2$, vykdymo laikas nepriklauso nuo grafo vaizdavimo būdo!

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. *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 $a \geq 1$, $b > 1$, $c, d > 0$ 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:

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

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. □

Pavyzdys 2.1. Binariosios paieškos algoritmui (žr. 1.2 skyrelį) gauname

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

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

Pavyzdys 2.2. Rūšiavimo sąlaja algoritmui (žr. 1.4 skyrelį) gauname

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

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

Pavyzdys 2.3. Rekursyviam aibės didžiausio ir mažiausio elemento paieškos algoritmui (žr. **??** skyrelį) gauname

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

taigi $a = 2$, $b = 2$, $c = 2$ ir $d = 0$. Kadangi $2 > 2^0$, tai pagal teoremą $L(n) = \mathrm{O}(n)$. **??** skyrelyje 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

$$(a + b)(c + d) = a_1c_1 \cdot 2^n + (a_1d_1 + b_1c_1) \cdot 2^{n/2} + b_1 \cdot d_1,$$

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

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \quad (i, j = 1, \ldots, n),$$

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 (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

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

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 2.2. *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:

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

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. □

Teorema 2.2. *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 aukščiau įrodytas 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 $L(n) < L(n') = \mathrm{O}((n')^{\log_2 7}) = \mathrm{O}(n^{\log_2 7})$. □

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 *subalansuota*, t.y., uždaviniai yra skaidomi į kelis maždaug vienodo dydžio uždavinius. Tuo tarpu kai naudojame nesubalansuotą rekursiją, šis metodas gali tapti labai neefektyvus. Tai demonstruoja žemiau pateikiamas pavyzdys su Fibonacci skaičiais. Tokiais atvejais dažnai padeda *dinaminis programavimas*.

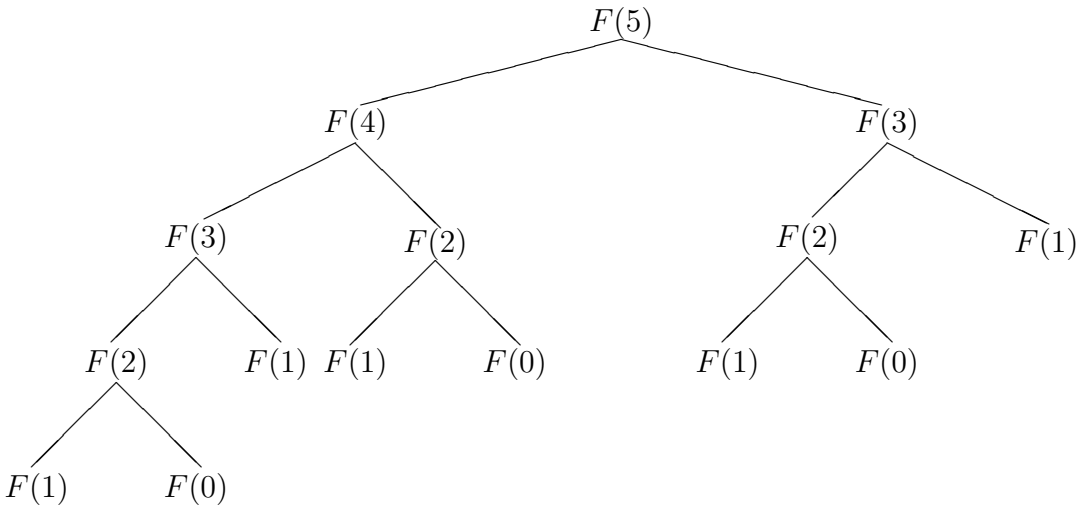
Dinaminis programavimas — tai uždavinio sprendimo metodas "iš apačios į viršų". Pirmiausia mes išsprendžiame visus paprasčiausius duoto uždavinio atvejus, t.y., mažiausius dalinius uždavinius ir įsimename gautus rezultatus. Remdamiesi gautais sprendiniais, randame didesnių dalinių uždavinių sprendinius 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[1] 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:

[1] **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.



2.1 Pav.: Programos fib1 rekursijos medis.

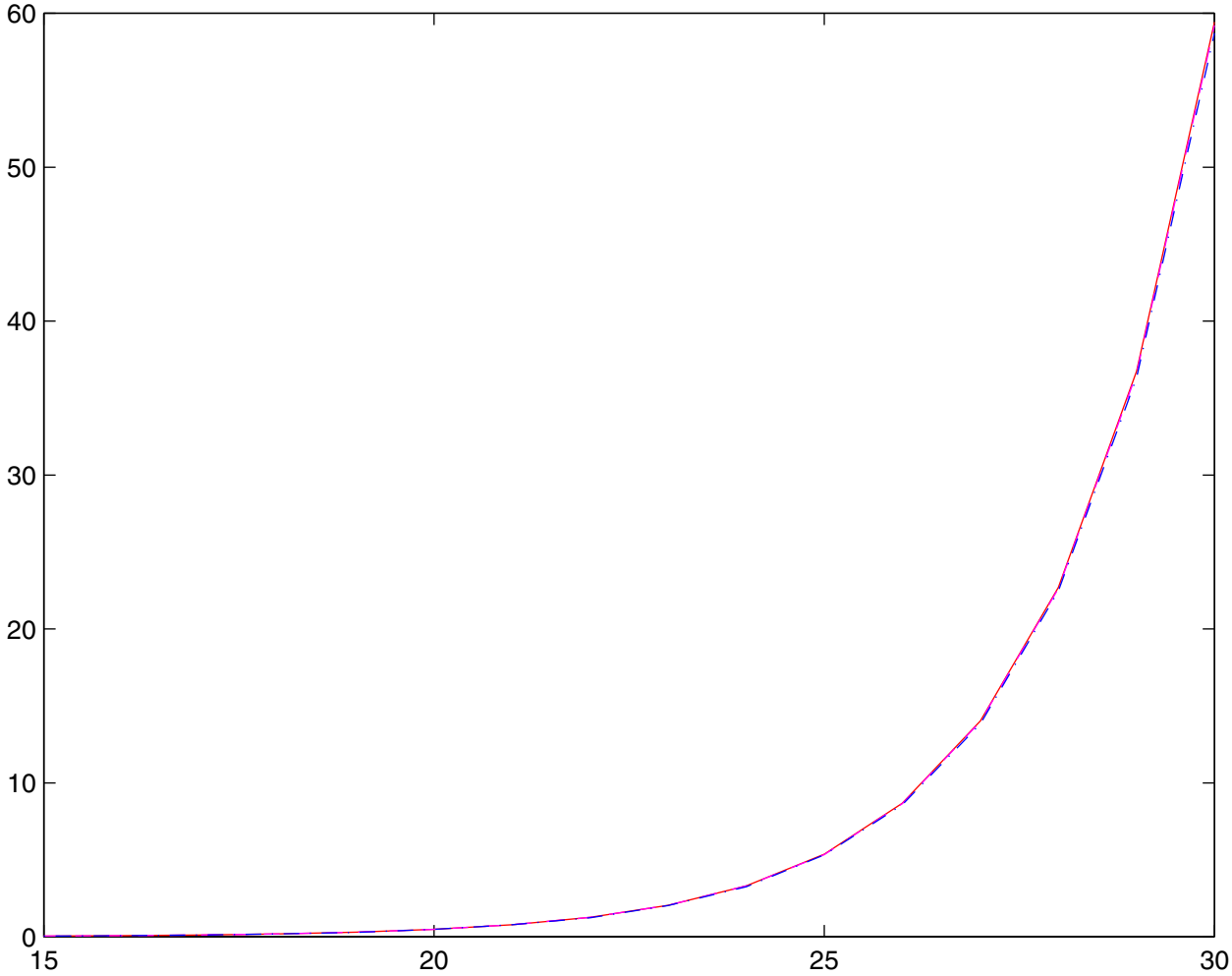
function fib1(n)
if $n = 0$ **then** fib1 $:= 0$
else if $n = 1$ **then** fib1 $:= 1$
else fib1 $:=$ fib1$(n-2)$ + fib1$(n-1)$
end;
end;

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.

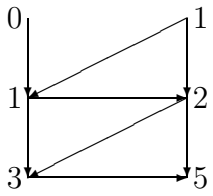
3.2 paveikslėlyje matote tris grafikus, kurie absoliučiai sutampa. Pirmasis grafikas vaizduoja CPU laiką (sekundėmis), kurį sugaišo Matlab programa fib1, skaičiuojant Fibonacci skaičius $F(15) = 610, \ldots, F(30) = 832040$, antrasis yra funkcijos $f(n) = F(n)/14000$ grafikas, ir trečiasis — funkcijos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

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

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



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



2.3 Pav.: Programos fib2 skaičiavimo schema.

Programa 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 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 fib2 randa $F(1476) \approx 1.307 \cdot 10^{308}$. Vietoje rekursijos medžio, kurį naudoja programa fib1, programa 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 $M_i, M_{i+1}, \ldots, M_j$ $(1 \le i \le j \le n)$ gauname rekurenčiąją sąsają

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

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 Matrix_Order(r)
n := size(r);
for i := 1 to n do m[i,i] := 0; end;
for i := 1 to n do
    for j := 1 to n do best[i,j] := 0; end;
end;
for diag := 1 to n - 1 do
    for i := 1 to n - diag do
        j := i + 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 then 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 2.4. 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: $(M_1 * (M_2 * M_3)) * M_4$.

2.3 Paieška su grįžimu

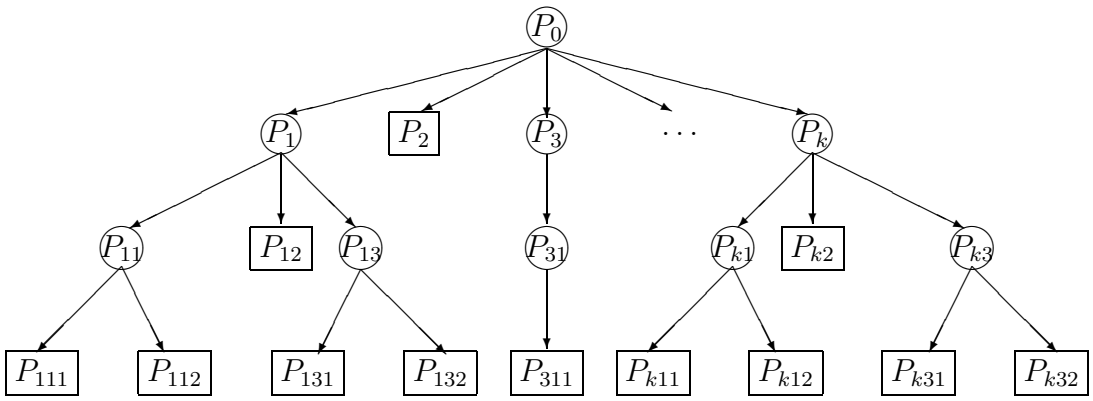
Sprendžiant kai kuriuos kombinatorinius uždavinius du aukščiau išnagrinėti metodai ("skaldyk ir valdyk" bei dinaminis programavimas) gali būti nepritaikomi arba neefektyvūs. Tokiu atveju dažnai belieka perrinkti visus galimus uždavinio sprendinius ir išsirinkti tinkamą. Pilnas perrinkimas dar yra vadinamas *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 mažesnius uždavinius $P_{il_1}, \ldots, P_{il_i}$ ir t.t. Uždavinį vadiname *neskaidžiu*, jei:

(a) galime lengvai rasti to uždavinio optimalų sprendinį;



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

(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. 2.4 pav.). 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 gilyn*. 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 $(1,3)$, ..., 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.

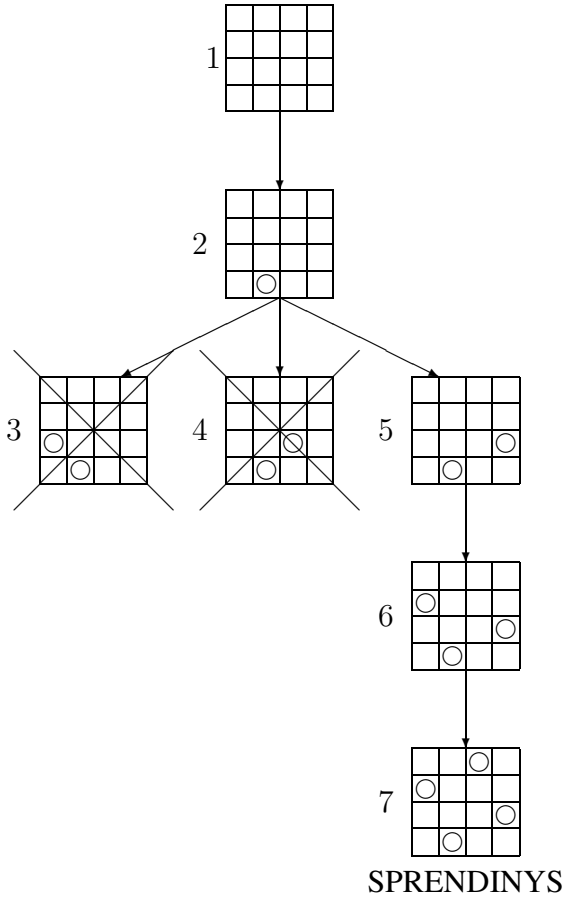
Tarkime, kad sprendinį galima užrašyti vektoriumi $(a_1, a_2, \ldots)$, kur $a_i \in A_i$, ir A_i yra pilnai sutvarkytos aibės. Pradėję nuo tuščio vektoriaus ( ) ir pažymėję $S_1 \subseteq A_1$ aibę galimų kandidatų į a_1, imame $a_1 = \min S_1$ (t.y., pirmąjį aibės S_1 elementą) ir gauname dalinį sprendinį (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** išsaugoti $(a_1, \ldots, a_k)$; **end**;
    $k := k + 1$;
    $S_k := \mathrm{geri}(A_k)$; /* Medžio šakų atkirtimas */
  end
  $k := k - 1$; /* Grįžimas */
end

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 $(v_1, v_2, \ldots, v_8)$, kur $v_i \in A_i = \{1, 2, \ldots, 8\}$. Toks sprendinys reiškia, kad 1-oji valdovė stovi laukelyje $(1, v_1)$, 2-oji valdovė laukelyje $(2, v_2)$ 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



2.5 Pav.: 4 valdovių uždavinys.

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 ($S_1 := \text{geri}(A_1) = \{2, 3, 4\}$), 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. Todėl renkamės laukelį $(2, 4)$. 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.

Naudojantis 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.

2.4 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:
   - $Cost \geq 0$;
   - $Cost(a_1, \ldots, a_{k-1}) \leq Cost(a_1, \ldots, a_k)$, 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:
   - $Bound(a_1, \ldots, a_k) \geq Cost(a_1, \ldots, a_k)$ 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 gilyn, *depth-first search* angl.), BFS (paieška platyn, *breadth-first search* angl.), BeFS (geriausios viršūnės prioritetinio pasirinkimo strategija, *best-first search* angl.) 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 gilyn, 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** rasti $Bound(a)$; **end**;
$S_1 := A_1$;
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**
      $MinCost := Cost$; išsaugoti $(a_1, \ldots, a_k)$
    end;
    $k := k + 1$;
    for $a \in A_k$ **do** rasti $Bound(a_1, \ldots, a_{k-1}, a)$; **end**;
    $S_k := \{a \in A_k$: $Bound(a_1, \ldots, a_{k-1}, a) < MinCost\}$;
  end
  $k := k - 1$;
  $Cost := Cost(a_1, \ldots, a_k)$;
end

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

2.4.1 Keliaujančio prekeivio 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 $C[i, j] = \mathrm{dist}(i, j) \geq 0$. 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 $C[i, j] \leq C[i, k] + C[k, j]$. 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

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

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:

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

   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. $\mathrm{branch\&bound}$ 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 2.5. 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}.

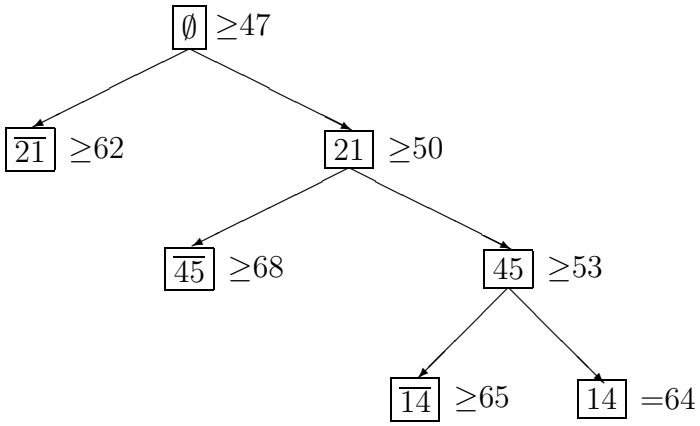
Pirmiausia prastiname matricą:

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

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

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. 2.6 pav.).

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 likusių miestų žymes, kurias toliau naudosime matricos elementų



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

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]).

3 skyrius

ALGORITMAI GRAFUOSE

3.1 Paieška platyn

Daugelyje algoritmų reikia sistemingai peržiūrėti visas grafo viršūnes lygiai po vieną kartą. Šis uždavinys vadinamas *paieška grafe*. Yra žinomi keli jo sprendimo metodai. Kurį iš jų pasirinkti priklauso nuo to, kuris metodas labiau atitinka sprendžiamo uždavinio algoritmą. Kai kuriuos iš šių metodų jau taikėme, vykdydami paiešką sprendinių medyje.

Paieškos platyn (angl. breadth-first search, BFS) idėja yra ta, kad pradėję paiešką iš kurios nors viršūnės iš pradžių aplankome visas jos kaimynes, po to jos kaimynių kaimynes (išskyrus tas, kuriose jau buvome) ir t.t. Taigi, paieškos platyn algoritmas apeina grafo viršūnes jų atstumų nuo pradinės viršūnės didėjimo tvarka. Čia *atstumu tarp viršūnių* u *ir* v vadiname briaunų skaičių trumpiausiame kelyje $K(u, v)$. Jei grafas jungus, paieškos platyn metu mes aplankysime visas viršūnes. Jei ne, teks pasirinkti bet kurią dar neaplankytą viršūnę ir vykdyti paiešką platyn iš jos. Taigi, jei grafas turi k komponenčių, tai reikės k kartų pradėti paiešką iš bet kurios dar neaplankytos viršūnės. Akivaizdu, kad paiešką platyn galima naudoti, norint rasti grafo komponentes arba viršūnių atstumus nuo fiksuotos viršūnės. Svoriniame grafe šį metodą naudoja Deikstros algoritmas trumpiausiems keliams iš duotos viršūnės rasti bei Primo algoritmas, konstruojantis minimalų grafo karkasą (žr. 3.3 skyrelį).

Paieškos platyn procedūra BFS naudoja grafo vaizdavimą gretimumo struktūromis. Viršūnei $v \in V$ gretimų viršūnių sąrašą žymėsime $\mathrm{GRET}[v]$. Viršūnėms, vienodai nutolusioms nuo pradinės viršūnės, saugoti naudojama duomenų struktūra *eilė* (angl. FIFO queue), kurią žymėsime kintamuoju EILE. Naujos viršūnės v įtraukimą į eilę žymėsime $v \Rightarrow \mathrm{EILE}$, o viršūnės v pašalinimą iš eilės žymėsime $v \Leftarrow \mathrm{EILE}$. Loginį masyvą NAUJAS naudosime pažymėti jau aplankytoms viršūnėms, t.y.,

$$\mathrm{NAUJAS}[v] = \begin{cases} \mathbf{true}, & \text{jei } v \text{ yra nauja viršūnė,} \\ \mathbf{false}, & \text{jei } v \text{ yra jau aplankyta viršūnė.} \end{cases}$$

Kartais vietoje loginio masyvo yra naudojamas viršūnių spalvinimas 3 spalvomis: neaplankytos viršūnės spalvinamos balta spalva, viršūnės, esančios eilėje — pilka, o pašalintos iš eilės viršūnės — juoda spalva.

Paieškos platyn grafe algoritmas atrodo taip:

for $v \in V$ **do** NAUJAS$[v] :=$ **true**; **end**;
iniciacija;
for $v \in V$ **do**
  if NAUJAS$[v]$ **then** *apdoroti* v; BFS(v);
  end;
end;

procedure BFS(v)
EILE $:= \emptyset$; $v \Rightarrow$ EILE; NAUJAS$[v] :=$ **false**;
while EILE $\neq \emptyset$ **do**
  $t \Leftarrow$ EILE; *apdoroti* t;
  for $u \in$ GRET(t) **do**
    if NAUJAS$[u]$ **then** $u \Rightarrow$ EILE; *apdoroti* t *ir* u; NAUJAS$[u] :=$ **false**;
    end;
  end;
end;

Čia komandos *iniciacija*, *apdoroti* v, *apdoroti* t, *apdoroti* t ir u reiškia veiksmus, priklausančius nuo sprendžiamo uždavinio.

Paieškos platyn algoritmas kiekvieną viršūnę lygiai vieną kartą įtrauks į eilę ir lygiai vieną kartą ją pašalins iš eilės. Be to, šis algoritmas lygiai vieną kartą peržiūrės kiekvieną grafo briauną. Taigi, jo sudėtingumas yra $\mathrm{O}(n + m)$.

Dabar pademonstruosime tris paprasčiausius paieškos platyn taikymus.

Grafo karkaso paieška. Duotas grafas $G = (V, E)$. Reikia rasti jo karkasą $K = (V, T)$.

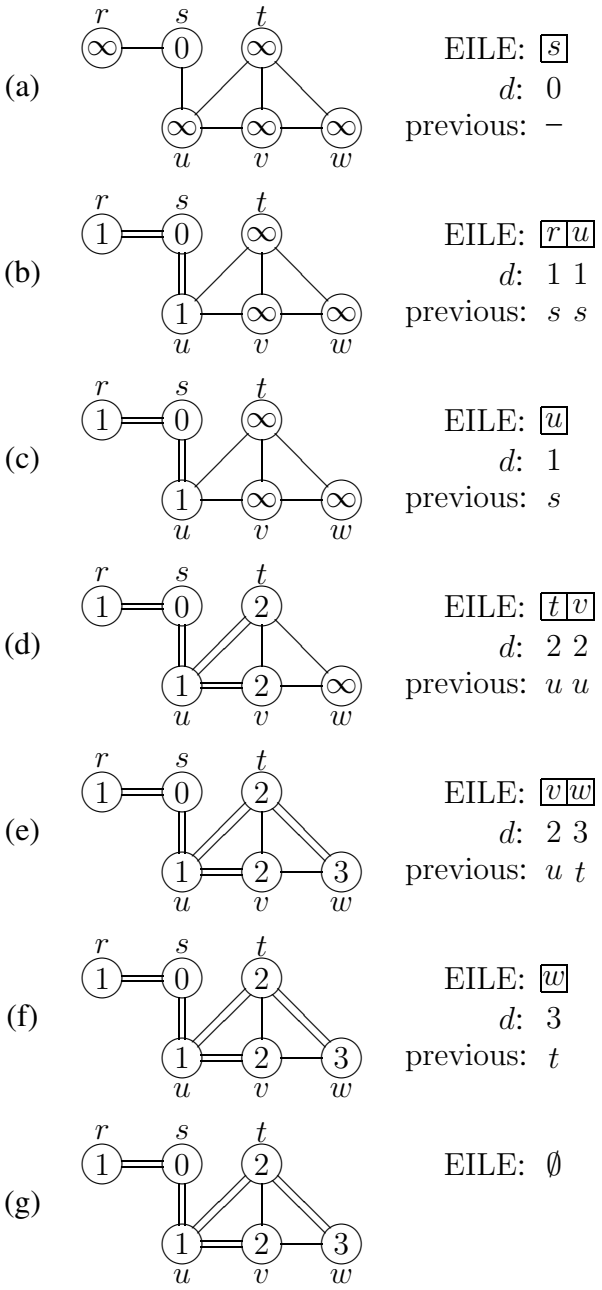
Grafo $G = \{V, E\}$ *karkasu* arba *atraminiu medžiu* (*spanning tree*, angl.) vadiname tokį grafo G pografį, kuris yra medis (arba miškas, jei grafas G nejungus), apimantis visas grafo G viršūnes. Jungiamė grafe paieškos platyn iš pradinės viršūnės s algoritmas konstruoja taip vadinamą *paieškos platyn medį* (angl. BFS tree) su šaknimi s. Jei einant briauna iš jau aplankytos viršūnės u algoritmas randa naują viršūnę v, tai viršūnė v ir briauna (u, v) yra įtraukiami į paieškos platyn medį, o viršūnė u vadinama viršūnės v *tėvu*. Tai galima padaryti su paieškos platyn algoritmu, atlikus šiuos pakeitimus:

1. Pagrindinėje programoje komandą *iniciacija* keičiame į $T := \emptyset$.

2. Procedūroje BFS(v) komandą *apdoroti* t *ir* u keičiame į $T := T \cup \{(t, u)\}$.

3.1(g) paveikslėlyje matote grafą ir jo karkasą, kurį randa paieškos platyn algoritmas. Karkaso briaunos pažymėtos dvigubomis linijomis.

Kodėl paieška platyn konstruojamas karkasas bus medis, t.y., neturės ciklų? Ciklas gautųsi tik tada, jei mes briauna sujungtume dvi "senas" (jau aplankytas) viršūnes, tuo tarpu paieškos platyn algoritmas kiekvieną kartą sujungia briauna "seną" ir "naują" viršūnę.



3.1 Pav.: Paieška platyn neorientuotame grafe. Kiekvienos ciklo **while** iteracijos pradžioje pateikiame grafą G, eilę EILE, atstumus d ir tėvų masyvą previous. Formuojamo karkaso briaunos vaizduojamos dvigubomis linijomis.

Trumpiausi keliai besvoriniuose grafuose. Duotas grafas $G = (V, E)$ ir jo viršūnė $s \in V$. Kiekvienai viršūnei $v \in V$ reikia rasti jos atstumą $d[v]$ nuo viršūnės s ir trumpiausią kelią $K(s, v)$ iš s į v.

Trumpiausią kelią $K(s, v)$ galėsime rasti paieškos platyn medyje su šaknimi s grįždami iš viršūnės v į šaknį s. Todėl pakanka kiekvienai viršūnei v įsiminti jos tėvą $\mathrm{previous}(v)$, rodantį, iš kurios viršūnės mes pirmą kartą patekome į viršūnę v paieškos platyn metu. Šiam uždaviniui išspręsti paieškos platyn algoritme reikia atlikti šiuos pakeitimus:

1. Pagrindinėje programoje komandą *iniciacija* keičiame į tokias komandas:

   for $v \in V$ **do** $\mathrm{previous}[v] := \mathrm{NIL}$; **end**;

   for $v \in V$ **do** $d[v] := \infty$; **end**;

   $d[s] := 0$;

2. Pernumeruojame aibės V elementus taip, kad viršūnė s pagrindinės programos komandose **for** $v \in V$ ... būtų pasirinkta pirmąja.

3. Procedūroje $\mathrm{BFS}(v)$ komandą *apdoroti* t *ir* u keičiame į

   $d[u] := d[t] + 1$; $\mathrm{previous}[u] := t$;

Pritaikę paieškos platyn algoritmą grafui, pavaizduotam 3.1(a) pav., gauname viršūnių trumpiausius atstumus, nurodytus 3.1(g) pav. matomo grafo viršūnėse. Trumpiausius kelius lengva rasti iš masyvo $\mathrm{previous}$. Pavyzdžiui, kadangi $\mathrm{previous}[w] = t$, $\mathrm{previous}[t] = u$, ir $\mathrm{previous}[u] = s$, tai trumpiausias kelias iš s į w yra $sutw$.

Grafo komponentės. Duotas grafas $G = (V, E)$. Reikia rasti jo komponentes, t.y., kiekvienai viršūnei $v \in V$ nurodyti komponentės, kuriai priklauso viršūnė v, numerį $\mathrm{COMP}[v]$.

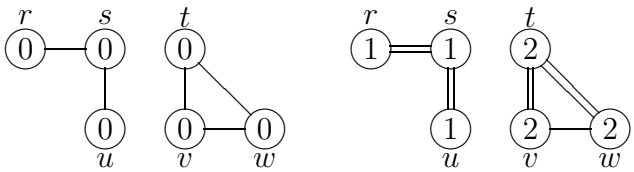
Grafo komponentes galima rasti su paieškos platyn algoritmu, atlikus šiuos pakeitimus:

1. Pagrindinėje programoje komandą *iniciacija* keičiame į $k := 0$.

2. Pagrindinėje programoje komandą *apdoroti* v keisti komanda $k := k + 1$.

3. Procedūroje $\mathrm{BFS}(v)$ komandą *apdoroti* t keičiame į $\mathrm{COMP}[t] := k$.

Pavyzdžiui, 4.2 pav. grafo viršūnėse nurodyta, kuriai grafo komponentei jos priklauso. Pav. (a) matome pradinį grafą, o pav. (b) matome tą patį grafą, atlikus jo komponenčių paiešką, naudojant paiešką platyn.



3.2 Pav.: Grafo komponenčių paieška, naudojant paiešką platyn: (a) prieš algoritmo vykdymą; (b) įvykdžius algoritmą. Kiekvienoje viršūnėje nurodytas komponentės, kuriai priklauso ši viršūnė, numeris.

3.2 Paieška gilyn

Paieškos gilyn (angl. depth-first search, DFS) idėja yra ta, kad iš pradinės viršūnės iš pradžių einame į bet kurią jai gretimą, iš šios į jai gretimą ir t.t. Taigi, paieškoje gilyn visą laiką renkamės naują viršūnę (t.y., tokią, kurioje dar nebuvome), formuodami viršūnių grandinę. Kai iš gautos grandinės paskutinės viršūnės nebėra kelio į naujas viršūnes, tada grįžtame į priešpaskutinę viršūnę ir einame į naują jai gretimą viršūnę.

Paiešką gilyn grafe lengva realizuoti rekursyvia procedūra:

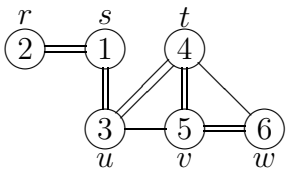
for $v \in V$ **do** $\mathrm{NAUJAS}[v] :=$ **true**; **end**;
iniciacija;
for $v \in V$ **do**
  if $\mathrm{NAUJAS}[v]$ **then** *apdoroti* v; $\mathrm{DFS}(v)$;
  end;
end;

procedure $\mathrm{DFS}(v)$
apdoroti v; $\mathrm{NAUJAS}[v] :=$ **false**;
for $u \in \mathrm{GRET}(v)$ **do**
  if $\mathrm{NAUJAS}[u]$ **then** *apdoroti* v *ir* u; $\mathrm{DFS}(u)$;
  end;
end;

Kaip ir paieška platyn, paieškos gilyn algoritmas peržiūri visas grafo viršūnes ir visas briaunas. Taigi, jo sudėtingumas yra $\mathrm{O}(n+m)$. Kadangi rekursyvios programos paprastai yra vykdomos ilgiau už analogiškas nerekursyvias, tai dideliems grafams patartina vietoje rekursijos naudoti *steką*, į kurį talpinamos paieškos gilyn metu rastos naujos viršūnės. Kiekvieną kartą imame viršutinę steko viršūnę ir lankome jai gretimas viršūnes. Jei nagrinėjamai viršūnei neberandame nė vienos naujos jos kaimynės, tada šią viršūnę šaliname iš steko.

Pademonstruosime paieškos gilyn taikymus.

Grafo karkaso paieška. Duotas grafas $G = (V, E)$. Reikia rasti jo karkasą $K = (V, T)$.



3.3 Pav.: Grafo karkasas, kurį suranda paieška gilyn iš viršūnės s. Karkaso briaunos vaizduojamos dvigubomis linijomis. Grafo viršūnėse įrašyti numeriai rodo naujų viršūnių apėjimo tvarką paieškoje gilyn.

Paieškos gilyn algoritmas konstruoja taip vadinamą *paieškos gilyn medį* (angl. DFS tree), jei grafas yra jungus, arba *paieškos gilyn mišką*, jei grafas nėra jungus. Tai galima padaryti su paieškos gilyn algoritmu, atlikus šiuos pakeitimus:

1. Pagrindinėje programoje komandą *iniciacija* keičiame į $T := \emptyset$.
2. Procedūroje $\mathrm{BFS}(v)$ komandą *apdoroti* v *ir* u keičiame į $T := T \cup \{(v, u)\}$.

3.3 paveikslėlyje matote grafą ir jo karkasą, kurį randa paieškos gilyn algoritmas. Karkaso briaunos pažymėtos dvigubomis linijomis.

Grafo komponentės. Duotas grafas $G = (V, E)$. Reikia rasti jo komponentes, t.y., kiekvienai viršūnei $v \in V$ nurodyti komponentės, kuriai priklauso viršūnė v, numerį $\mathrm{COMP}[v]$.

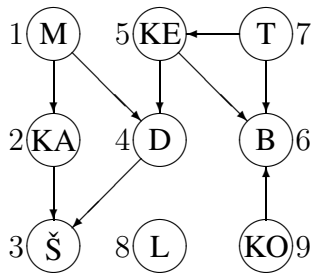
Grafo komponentes rasti galima su paieškos gilyn algoritmu, atlikus šiuos pakeitimus:

1. Pagrindinėje programoje komandą *iniciacija* keičiame į $k := 0$.
2. Pagrindinėje programoje komandą *apdoroti* v keisti komanda $k := k + 1$.
3. Procedūroje $\mathrm{DFS}(v)$ komandą *apdoroti* v keičiame į $\mathrm{COMP}[v] := k$.

Topologinis rūšiavimas. Duotas orgrafas $G = (V, E)$. Reikia rasti jo viršūnių topologinį sutvarkymą.

Orgrafo viršūnių *topologiniu sutvarkymu* vadiname tokią jo viršūnių numeraciją skaičiais nuo 1 iki $|V|$, priskiriant kiekvienai viršūnei $v \in V$ jos numerį $\mathrm{label}(v)$, kad bet kuriam lankui $(u, v) \in E$ galiotų nelygybė $\mathrm{label}(u) < \mathrm{label}(v)$. Galima įrodyti, kad orgrafo viršūnes galima topologiškai sutvarkyti tada ir tik tada, kai duotasis orgrafas neturi ciklų. Topologinis rūšiavimas reikalingas, pavyzdžiui, nustatant darbų eilę dideliuose projektuose (pvz., namo statyboje), kai yra žinoma, koks darbas po kurio turi sekti, sprendžiant dalinius projekto uždavinius, bet būna sunku aprėpti viso projekto darbų eilę.

Topologinį rūšiavimą galima realizuoti su paieška gilyn, atlikus tokius algoritmo pakeitimus:

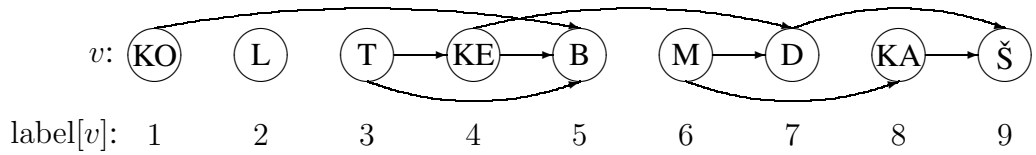


3.4 Pav.: Dalinė drabužių rengimosi tvarka. Kiekvienas lankas (u, v) reiškia, kad drabužį u būtina apsirengti anksčiau, negu drabužį v. Šalia grafo viršūnių pažymėta jų pirmojo aplankymo eilės tvarka.

1. Pagrindinėje programoje komandą *iniciacija* keičiame į tokias komandas:
   for $v \in V$ **do** $\mathrm{label}[v] := 0$; **end**;
   $j := |V|$;
2. Procedūroje $\mathrm{DFS}(v)$ komandą *apdoroti v ir u* keičiame į
   $\mathrm{label}[v] := j$;
   $j := j - 1$;

Pavyzdys 3.1 (Profesorius rengiasi). Išsiblaškęs profesorius kas rytą turi spręsti nelengvą uždavinį, kokia tvarka jis turėtų rengtis savo drabužius. Kaip žinoma, kai kuriuos drabužius būtina apsirengti anksčiau, negu tam tikrus kitus (pavyzdžiui, kojines būtina apsimauti anksčiau negu batus). Kai keletą kartų teko nusirengti ir vėl iš naujo apsirengti dėl to, kad buvo pažeista drabužių rengimosi tvarka, profesorius nutarė sėsti prie stalo ir visiems laikams apskaičiuoti, kokia gi tvarka jis turi rengtis savo drabužius. Profesorius nusibraižė 3.4 pav. vaizduojamą orgrafą, kuris nurodo dalinės tvarkos ryšius tarp įvairių drabužių. Grafo viršūnės atitinka šiuos drabužius: B — batai, D — diržas, KA — kaklaraištis, KE — kelnės, KO — kojinės, L — laikrodis, M — marškiniai, Š — švarkas, T — trumpikės.

Pradėję iš viršūnės M grafui iš 3.4 pav. taikome paiešką gilyn. Kai patenkame į viršūnę, iš kurios nebėra lanko nė į vieną naują viršūnę, šiai viršūnei priskiriame žymę label. Pavyzdžiui, iš viršūnės M eisime į viršūnę KA, iš šios į viršūnę Š. Kadangi iš viršūnės Š neišeina nė vienas lankas, tai jai priskirsime rengimosi eilės numerį $\mathrm{label}(\text{Š}) = 9$. Kai galime pasirinkti kelias viršūnes, renkamės tą, kuri stovi aukščiau ir kairiau už kitas. Šalia grafo viršūnių pažymėta jų pirmojo aplankymo eilės tvarka. 3.5 paveikslėlyje matome grafo viršūnes, išdėstytas jų žymių label didėjimo tvarka. Tai ir yra ieškoma drabužių rengimosi tvarka.



3.5 Pav.: Topologiškai sutvarkytos grafo iš 3.4 pav. viršūnės. Jų tvarka atitinka drabužių rengimosi tvarką. Atkreipkite dėmesį, kad kiekvienas lankas nukreiptas iš kairės į dešinę.

3.2.1 Trumpiausi keliai grafuose

Kiekvienam ne kartą teko spręsti uždavinį, kaip rasti trumpiausią kelią iš taško A į tašką B. A ir B gali būti du miestai, du pastatai viename mieste ir t.t. Geometrine prasme šis uždavinys atrodo trivialus: tereikia žemėlapyje sujungti taškus A ir B tiesės atkarpa, pasiimti kompasą ir drožti nosies tiesumu. Deja, gyvenime mes naudojamės egzistuojančiu kelių tinklu. Geometriškai trumpiausias maršrutas gali būti nerealus dėl įvairių kliūčių ir kitų priežasčių. Kadangi bet kurį kelių tinklą galime vaizduoti svoriniu orgrafu, tai gauname tokį grafų teorijos uždavinį:

(i) Duotas svorinis orgrafas $G = (V, E, \omega)$ ir dvi grafo viršūnės u ir v. Reikia rasti trumpiausią kelią iš u į v ir to kelio ilgį.

Dažnai mums tenka spręsti bendresnius trumpiausio kelio paieškos uždavinius:

(ii) Duotas svorinis orgrafas $G = (V, E, \omega)$ ir to grafo viršūnė u. Reikia rasti trumpiausius kelius iš u į visas kitas to grafo viršūnes.

(iii) Duotas svorinis orgrafas $G = (V, E, \omega)$. Reikia rasti trumpiausius kelius iš kiekvienos grafo viršūnės į kiekvieną kitą to grafo viršūnę.

Atrodytų, jei grafas turi $n = |V|$ viršūnių, tai antrasis uždavinys yra n kartų sudėtingesnis už pirmąjį, o trečiasis uždavinys savo ruožtu yra n kartų sudėtingesnis už antrąjį. Tačiau iš tikrųjų, norint rasti trumpiausią kelią tarp dviejų fiksuotų grafo viršūnių u ir v, tenka apžiūrėti visas grafo viršūnes, nes jei bent vieną praleisime, tai gali pasirodyti, kad kaip tik eidami per šią viršūnę mes trumpiausiu keliu pateksime iš u į v. Žemiau pateikiame Floyd–Warshall algoritmą, kuris sprendžia trečiąjį uždavinį, o tuo pačiu ir pirmuosius du uždavinius. Nors yra žinoma daug kitų algoritmų (pvz., Deikstros ir Fordo–Belmano) dviems pirmiesiems uždaviniams spręsti, tačiau įdomu tai, kad bendruoju atveju nė vienas iš tų algoritmų nėra paprastesnis už Floyd–Warshall algoritmą.

Taigi, duotas orgrafas $G = (V, E)$ su viršūnių aibe $V = \{v_1, \ldots, v_n\}$, briaunų aibe E ir svorių (atstumų) matrica $A = (\omega(v_i, v_j))$. Reikia rasti trumpiausių kelių ilgių matricą D, t.y.

$$D[i, j] = \min_{K(i,j)} \sum_{e \in K(i,j)} \omega(e).$$

Galime laikyti, kad G yra pilnas grafas, nes jei dvi viršūnės nėra sujungtos briauna, laikome, kad tokios briaunos svoris yra begalybė (∞). Svoriai (atstumai) gali būti ir neigiami, tačiau grafas G negali turėti neigiamo svorio ciklų, t.y., kelių $K(i,i)$ tokių, kad $\sum_{e \in K(i,i)} \omega(e) < 0$ (priešingu atveju mes be galo suktumėmės tokiu ciklu, o nueitas kelias artėtų į $-\infty$).

Kodėl mes ieškome tik trumpiausių kelių ilgių, o ne pačių kelių ? Pasirodo, kad pačius trumpiausius kelius lengvai galima rasti, naudojant matricas D ir A. Norėdami rasti priešpaskutinę kelio $K(i,j)$ viršūnę v_k, ieškome tokio k, kuriam būtų teisinga lygybė $D[i,j] = D[i,k] + A[k,j]$, t.y., sudedame matricos D i-osios eilutės ir matricos A j-ojo stulpelio atitinkamus elementus, kol gausime lygybę. Aišku, kad $\forall l\ D[i,j] \leq D[i,l] + A[l,j]$. Kadangi trumpiausias kelias $K(i,j)$ turi praeiti per kažkurią viršūnę $k \neq j$, tai būtinai atsiras k tokia, kad $D[i,j] = D[i,k]+A[k,j]$. Suradę priešpaskutinę kelio viršūnę, ieškome priešpriešpaskutinės ir t.t., kol grįšime į i.

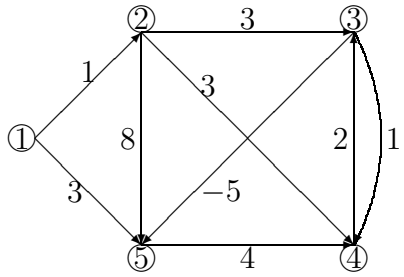
Pagrindinė Floyd–Warshall algoritmo idėja yra paeiliui įterpinėti naujas tarpines viršūnes į visus tuo metu rastus trumpiausius kelius ir tikrinti, ar naudojant naują tarpinę viršūnę gausime naują kelią, trumpesnį už jau turimą kelią. Kadangi grafas neturi neigiamų ciklų, tai bet kuriame trumpiausiame kelyje kiekviena viršūnė pasitaikys ne daugiau kaip 1 kartą. Tarkime, $i, j \in V$ ir K yra trumpiausias kelias iš i į j tarp tokių kelių, kurių visos tarpinės viršūnės priklauso aibei $N_k = \{1, \ldots, k\}$, kur $k \leq n$. Jei viršūnė k priklauso keliui K, tai kelio K pirmoji dalis $K_1 = K(i,k)$ bus trumpiausias kelias tarp viršūnių i ir k su tarpinėmis viršūnėmis iš aibės N_{k-1}, o kelias $K_2 = K(k,j)$ bus trumpiausias kelias tarp viršūnių k ir j su tarpinėmis viršūnėmis iš aibės N_{k-1} (nes priešingu atveju egzistuotų kelias iš i į j su tarpinėmis viršūnėmis iš N_k, trumpesnis už K). Jei viršūnė k nepriklauso keliui K, tai kelias K bus trumpiausias kelias tarp viršūnių i ir j su tarpinėmis viršūnėmis iš aibės N_{k-1}.

Pažymėję trumpiausio kelio iš i į j su tarpinėmis viršūnėmis iš aibės N_k ilgį $D^{(k)}[i,j]$, gauname rekurenčią sąsają trumpiausių kelių ilgiams:

$$D^{(k)}[i,j] = \begin{cases} A[i,j], & \text{jei } k = 0, \\ \min\{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}, & \text{jei } k \geq 1. \end{cases}$$

Floyd–Warshall algoritmas realizuojamas trigubu ciklu:

function $D = \text{shortest\_paths}(A)$
for $i := 1$ **to** n **do**
    for $j := 1$ **to** n **do** $D[i,j] := A[i,j]$
    end
end
for $k := 1$ **to** n **do**
    for $i := 1$ **to** n **do**
        for $j := 1$ **to** n **do** $D[i,j] := \min\{D[i,j], D[i,k] + D[k,j]\}$
        end
    end
end



3.6 Pav.: Rasti trumpiausius kelius.

Šio algoritmo sudėtingumas yra $\mathrm{O}(n^3)$.

Pavyzdys 3.2. Duota grafas (žr. 3.6 pav.) su atstumų matrica

$$A = \begin{pmatrix} 0 & 1 & \infty & \infty & 3 \\ \infty & 0 & 3 & 3 & 8 \\ \infty & \infty & 0 & 1 & -5 \\ \infty & \infty & 2 & 0 & \infty \\ \infty & \infty & \infty & 4 & 0 \end{pmatrix}.$$

Rasti trumpiausių kelių tarp bet kurių dviejų grafo viršūnių ilgius bei trumpiausią kelią iš 1 į 4 viršūnę.

Pademonstruosime kaip keičiasi trumpiausių kelių ilgių matrica D (pradiniu momentu $D^{(0)} = A$):

$$D^{(1)} = \begin{pmatrix} 0 & 1 & \infty & \infty & 3 \\ \infty & 0 & 3 & 3 & 8 \\ \infty & \infty & 0 & 1 & -5 \\ \infty & \infty & 2 & 0 & \infty \\ \infty & \infty & \infty & 4 & 0 \end{pmatrix},$$

$$D^{(2)} = \begin{pmatrix} 0 & 1 & 4 & 4 & 3 \\ \infty & 0 & 3 & 3 & 8 \\ \infty & \infty & 0 & 1 & -5 \\ \infty & \infty & 2 & 0 & \infty \\ \infty & \infty & \infty & 4 & 0 \end{pmatrix},$$

$$D^{(3)} = \begin{pmatrix} 0 & 1 & 4 & 4 & -1 \\ \infty & 0 & 3 & 3 & -2 \\ \infty & \infty & 0 & 1 & -5 \\ \infty & \infty & 2 & 0 & -3 \\ \infty & \infty & \infty & 4 & 0 \end{pmatrix},$$

$$D^{(4)} = \begin{pmatrix} 0 & 1 & 4 & 4 & -1 \\ \infty & 0 & 3 & 3 & -2 \\ \infty & \infty & 0 & 1 & -5 \\ \infty & \infty & 2 & 0 & -3 \\ \infty & \infty & 6 & 4 & 0 \end{pmatrix},$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & 4 & 3 & -1 \\ \infty & 0 & 3 & 2 & -2 \\ \infty & \infty & 0 & -1 & -5 \\ \infty & \infty & 2 & 0 & -3 \\ \infty & \infty & 6 & 4 & 0 \end{pmatrix} = D.$$

Liko rasti trumpiausią kelią iš 1 į 4. Kadangi $D[1,4] = 3 = D[1,5] + A[5,4]$, tai priešpaskutinė šio kelio viršūnė yra 5. Kadangi $D[1,5] = -1 = D[1,3] + A[3,4]$, tai prieš 5 viršūnę eina viršūnė 3. Kadangi $D[1,3] = 4 = D[1,2] + A[2,3]$, tai prieš 3 viršūnę eina viršūnė 2. Gauname kelią $1 \to 2 \to 3 \to 5 \to 4$ ilgio 3.

3.3 Minimalaus karkaso uždavinys

Didelė kompanija, turinti savo centrus įvairiuose miestuose, nutarė didelio pralaidumo kabeliais visus kompiuterius sujungti į tinklą taip, kad iš bet kurio kompiuterio siunčiami duomenys galėtų pakliuti į bet kurį kitą kompiuterį. Kabelis bus tiesiamas šalia kelių, taigi jo tiesimo kaina yra proporcinga atstumams tarp miestų. Aišku, kad nebūtina sujungti kiekvieną miestą su kiekvienu: pakanka įtraukti tiek kelių, kad iš kiekvieno miesto signalas galėtų pakliūti į kiekvieną kitą miestą per tarpinius miestus. Kuriuos kelius reikia išsirinkti, kad tokio tinklo kaina būtų mažiausia ? Tokį pat uždavinį žiemą gali tekti spręsti kaimynams kaimo vietovėje. Vasarą jie buvo įpratę lankytis vieni pas kitus, pasirinkdami trumpiausius kelius. Žiemą visus kelius užpustė, todėl jie norėtų rasti trumpiausią kelių tinklą, kad nuvalius tik to tinklo kelius, jie galėtų nueiti vieni pas kitus.

Suformuluosime aukščiau aprašytą praktinį uždavinį grafų kalba. Duotas svorinis grafas $G = (V, E, \omega)$ su neneigiamais svoriais. Reikia rasti jungų šio grafo pografį $T = (V_T, E_T, \omega)$, kuris apimtų visas pradinio grafo viršūnes ($V_T = V$) ir kurio briaunų svorių suma būtų minimali tarp visų galimų tokių pografių. Aišku, kad toks minimalus pografis negali turėti ciklų, nes priešingu atveju išmetus bet kurią ciklo teigiamo svorio briauną, pografis liktų jungus, o jo svoris sumažėtų. Taigi, tai turi būti medis. 3.1 skyrelyje medį, apimantį visas grafo viršūnes, vadinome karkasu. Taigi, reikia rasti duoto grafo minimalų karkasą (*minimum spanning tree* (MST), angl.).

Yra žinomi du pagrindiniai algoritmai minimalaus karkaso uždaviniui spręsti: Kraskalo (Joseph Kruskal, g. 1928) ir Primo (Robert Prim, g. 1921). Jie abu naudoja godų metodą: visą laiką rinktis pigiausią (mažiausio svorio) dar nepaimtą grafo briauną, jei ji tenkina tam tikras taisykles. Minimalaus karkaso uždavinys yra klasikinis pavyzdys, kai godus algoritmas randa optimalų sprendinį.

Pirmiausia pateiksime "žodines" abiejų algoritmų formuluotes. Jos yra labai paprastos. Priminsime, kad jei grafas G turi n viršūnių, tai pagal medžių savybes minimalus karkasas T visada turės $n-1$ briauną. Kadangi a priori yra žinoma, kad ciklų negali būti, tai galime laikyti, kad briaunų svoriai yra bet kokie (t.y., gali būti ir neigiami).

procedure Kruskal
$V_T := V$; $E_T := \emptyset$;
for $i = 1 : n - 1$ **do**
    $e :=$ pigiausia grafo G briauna, kurią prijungę prie T, mes negausime ciklo;
    $E_T := E_T \cup \{e\}$;
end;

procedure Prim
$V_T := \{v_0\}$; $E_T := \emptyset$;
for $i = 1 : n - 1$ **do**
    $e = (u, v) :=$ pigiausia grafo G briauna, incidentinė kuriai nors karkaso T
        viršūnei u, kurią prijungę prie T, mes negausime ciklo;
    $E_T := E_T \cup \{e\}$; $V_T := V_T \cup \{v\}$;
end;

Vienintelis šių algoritmų skirtumas, kad antrasis konstruoja medį siekdamas, kad tas medis visą laiką būtų jungus, tuo tarpu pirmasis ima paeiliui pigiausias briaunas tik tikrindamas, kad neatsirastų ciklo. Nors šis skirtumas iš pažiūros atrodo nedidelis, abu algoritmai yra realizuojami labai skirtingai.

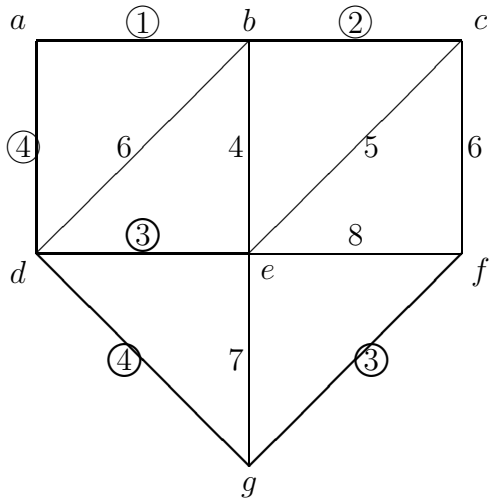
Abiejuose algoritmuose reikia užtikrinti, kad prijungiant prie karkaso naują briauną neatsiras ciklo. Ciklas gali atsirasti tada ir tik tada, kai prijungiamos briaunos abu galai priklauso tai pačiai jau turimo karkaso komponentei. Kad taip neatsitiktų, Kraskalo algoritme kiekvieną karkaso komponentę saugosime kaip medį (apie aibių vaizdavimą mišku žr. 2.4 skyrelį). Tada prijungiant naują briauną pakaks patikrinti, ar jos galai priklauso skirtingiems medžiams. Kadangi Primo algoritmas visą laiką bando praplėsti vieną komponentę, tai pakanka saugoti karkasui T jau panaudotų viršūnių aibę V_T ir nagrinėti tik tokias naujas briaunas, kurios jungia viršūnes iš V_T su viršūnėmis iš $V \setminus V_T$.

Panagrinėsime Kraskalo ir Primo algoritmus detaliau.

3.3.1 Kraskalo algoritmas

Pateikiame detalesnį Kraskalo algoritmą. Kaip jau minėjome, karkaso T viršūnių aibė yra vaizduojama mišku. M_v reiškia medį, kuriame yra viršūnė v, o $r(v)$ reiškia medžio M_v šaknį.

procedure Kruskal(V, E, w)
$E :=$ SORT(E); /* Rūšiuojame briaunų aibę briaunų svorių $w(e)$ didėjimo tvarka */
$V_T := V$; $E_T := \emptyset$;
for $v \in V$ **do** $M_v := \{v\}$; $r(v) := v$; **end**; /* Inicializuojame mišką */
$i := 1$;
while $|E_T| < n - 1$ **do**
  $e := e_i = (u, v)$; /* Pasirenkame pigiausią iš eilės briauną */
  if $r(u) \neq r(v)$ **then**
    $E_T := E_T \cup \{e\}$;
    for $t \in M_v$ **do** $r(t) = r(u)$; **end**;



3.7 Pav.: Minimalus grafo karkasas, kurį randa Kraskalo algoritmas (karkaso briaunų svoriai pažymėti skrituliukais).

$M_u := M_u \cup M_v$; /* Sujungiame medžius, kuriuose buvo viršūnės u ir v */
end;
$i := i + 1$;
end;

Koks šio algoritmo sudėtingumas? m briaunų rūšiavimas reikalauna $\mathrm{O}(m \log_2 m)$ operacijų. Dvigubo ciklo **while-for** sudėtingumas blogiausiu atveju bus $\mathrm{O}(n^2)$. Gauname $L^{\mathrm{MST}}_{\mathrm{Kruskal}} = \mathrm{O}(m \log_2 m + n^2)$. Veiksmams su medžiams naudojant šakų suspaudimą, minėtą 1.4 skyrelyje, galima sumažinti dvigubo ciklo sudėtingumą iki $\mathrm{O}(m \log_2 n)$. Kadangi iš $m \leq n^2$ išplaukia $\log_2 m = \mathrm{O}(\log_2 n)$, tai optimaliai realizuoto Kraskalo algoritmo sudėtingumas yra $\mathrm{O}(m \log_2 n)$.

Pavyzdys 3.3. Pritaikykime Kraskalo algoritmą grafui, pavaizduotam 3.7 paveikslėlyje. Surūšiavę duoto grafo briaunų aibę gauname

$$E = \{((a,b),(b,c),(d,e),(f,g),(a,d),(b,e),(d,g),(c,e),(b,d),(c,f),(e,g),(e,f)\}.$$

Imame paeiliui briaunas, tikrindami, ar jos nesudaro ciklo. Pradinės briaunos (a,b), (b,c), (d,e), (f,g), (a,d) visos tinka. Briauna (b,e) sudarytų ciklą su briaunomis (a,b), (a,d) ir (d,e), todėl ją praleidžiame. Po jos einanti briauna (d,g) tinka. Kadangi jau turime 6 briaunas, o pradinis grafas turėjo 7 viršūnes, baigiame darbą. Gauto karkaso briaunų svoriai 3.7 paveikslėlyje yra pažymėti skrituliukais.

3.3.2 Primo algoritmas

Kaip jau minėjome, Primo algoritmas pradeda konstruoti karkasą T iš bet kurios pradinės grafo viršūnės v_0 ir kiekvieną kartą pasirenka pigiausią briauną iš visų briaunų, jungiančių aibės V_T viršūnes su aibės $V \setminus V_T$ viršūnėmis. Tam, kad rasti tokią briauną, reikia peržiūrėti viršūnių iš V_T kaimynes grafe G. Kaip ir anksčiau, viršūnės v kaimynių (viršūnių,

sujungtų briauna su v) aibę žymime $\mathrm{GRET}(v)$. Tam, kad pasirenkant pigiausią briauną nereikėtų kiekvieną kartą iš naujo paržiūrėti visų karkaso viršūnių ir visų jų kaimynių (t.y., kad išvengti trigubo ciklo), mes kiekvienai grafo viršūnei v saugosime jos trumpiausią atstumą β_v iki karkaso T. Be to, įsiminsime ir tą karkaso T viršūnę, iki kurios nagrinėjama viršūnė v yra arčiausiai, t.y., saugosime $\alpha_v := u \in V_T$: $w(u, v) = \beta_v$. Pradiniu momentu, jei algoritmas pradeda darbą iš viršūnės v_0, tai visoms viršūnėms $v \in V \setminus \{v_0\}$ jų trumpiausi atstumai iki karkaso T bus $\beta_v = w(v, v_0)$, o artimiausia karkaso viršūnė joms visoms bus v_0. Jei pradiniame grafe viršūnės v ir v_0 nėra sujungtos briauna, laikome, kad trumpiausias atstumas β_v yra begalybė. Algoritmo vykdymo metu trumpiausi atstumai bus dinamiškai perskaičiuojami.

procedure $\mathrm{Prim}(V, E, w)$
$V_T := \{v_0\}$; $E_T := \emptyset$;
for $v \in V \setminus V_T$ **do** $\beta_v := w(v_0, v)$; $\alpha_v := v_0$; **end**; /* Iniciacija */
while $|V_T| < n$ **do**
  $\beta^* := \min_{v \in V \setminus V_T} \beta_v = \beta_{v^*}$; /* Randame artimiausią dar neprijungtą viršūnę v^* */
  $V_T := V_T \cup \{v^*\}$; $E_T := E_T \cup \{(v^*, \alpha_{v^*})\}$;
  for $v \in \mathrm{GRET}(v^*)$ **do**
    if $\beta_v > w(v^*, v)$ **then** $\beta_v := w(v^*, v)$; $\alpha_v := v^*$; **end**;
  end;
end;

Kadangi sudėtingiausia algoritmo dalis yra dvigubas ciklas **while-for**, tai Primo algoritmo sudėtingumas $L^{\mathrm{MST}}_{\mathrm{Prim}} = \mathrm{O}(n^2)$. Buvo įrodyta, kad saugant viršūnes iš $v \in V \setminus V_T$ *prioritetinėje eilėje* ir naudojant *Fibonačio krūvos* (*Fibonacci heap*, angl.) duomenų struktūrą, Primo algoritmo sudėtingumą galima sumažinti iki $\mathrm{O}(m + n \log_2 n)$. Taigi, tik retiems grafams (jei $m = \mathrm{O}(n)$) Kraskalo algoritmas gali būti greitesnis už Primo algoritmą. Tankesniems grafams Primo algoritmas asimptotiškai yra greitesnis už Kraskalo. Tačiau tai priklauso nuo duomenų struktūrų, panaudotų abiejų algoritmų realizacijai.

Pavyzdys 3.4. Pritaikykime Primo algoritmą grafui, pavaizduotam 3.7 paveikslėlyje. Tarkime, kad grafo briaunos kompiuterio atmintyje yra išdėstytos leksikografine tvarka:

$$E = \{(a, b), (a, d), (b, c), (b, e), (b, d), (c, e), (c, f), (d, e), (d, g), (e, f), (e, g), (f, g)\}.$$

Pradėjus iš viršūnės a, artimiausia viršūnė yra b, taigi briauną (a, b) įtraukiame į karkaso briaunų aibę E_T. Tarp viršūnių a ir b kaimynių joms artimiausia yra viršūnė c, taigi įtraukiame ir briauną (b, c). Perskaičiavus likusių viršūniu trumpiausius atstumus, masyvai α ir β atrodo taip:

$$\alpha_d = a,\ \alpha_e = b,\ \alpha_f = c,\ \alpha_g = a$$

ir

$$\beta_d = 4,\ \beta_e = 4,\ \beta_f = 6,\ \beta_g = \infty.$$

Taigi, renkamės briauną (a, d). Galų gale gauname minimalų karkasą $T = (V, E_T)$, kur

$$E_T = \{(a, b), (b, c), (a, d), (d, e), (d, g), (f, g)\},$$

t.y., tą patį, kurį rado ir Kraskalo algoritmas.

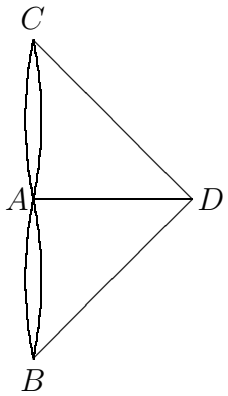
3.4 Oilerio ciklo paieška

Priminsime, kad Oilerio ciklu vadiname ciklą, praeinantį kiekviena grafo briauna lygiai po vieną kartą. Grafai, kuriuose egzistuoja Oilerio ciklas, yra vadinami *Oilerio grafais*. Dar 1736 m. Leonardas Oileris (žr. 2.5.1 skyrelį) gyvendamas Karaliaučiuje (kuris tuomet vadinosi Kionigsbergu, o dabar Kaliningradu) ir vaikščiodamas tiltais per Priegliaus upę suformulavo vieną pirmųjų grafų teorijos uždavinių: ar jis galėtų išėjęs iš namų pereiti kiekvienu iš 7 tiltų per upę vienintelį kartą ir vėl grįžti namo. 3.8 paveikslėlyje matote multigrafą, atitinkantį Oilerio suformuluotą "7 tiltų" uždavinį. Oileriui pavyko rasti būtiną ir pakankamą sąlygą tokio ciklo egzistavimui bet kokiame duotame grafe.

Teorema 3.1 (Oilerio teorema). *Jungus multigrafas yra Oilerio grafas tada ir tik tada, kai visų jo viršūnių laipsniai yra lyginiai.*

Įrodymas. *Būtinumas*. Tarkime, turime Oilerio grafą. Kadangi jis yra jungus, tai iš kiekvienos viršūnės išeina bent viena briauna. Nagrinėjame bet kurią grafo viršūnę v_0. Kadangi bet kuri iš šios viršūnės išeinanti briauna priklauso Oilerio ciklui, tai pasirenkame bet kurią briauną ir keliauname Oilerio ciklu. Praeinant per bet kurią grafo viršūnę, mes viena briauna į ją ateiname, o kita išeiname, taigi vieną kartą praeidamas per bet kurią grafo viršūnę, nesutampančią su viršūne v_0, Oilerio ciklas panaudoja lygiai dvi briaunas. Kadangi judame ciklu, tai kažkada mes grįšime į pradinę viršūnę v_0, tokiu būdu taip pat panaudodami lygiai dvi briaunas, incidentines viršūnei v_0 (pirmą ir paskutinę ciklo briaunas). Jei apėjome dar ne visą ciklą, vėl išeiname iš viršūnės v_0 ir vėl turėsime į ją grįžti. Kadangi briaunos nesikartoja, tai visų viršūnių laipsniai privalo būti lyginiai, kitaip mes negalėsime išeiti iš kažkurios viršūnės ir sugrįžti į v_0.

Pakankamumas. Tarkime, kad duotas multigrafas yra jungus, ir visų jo viršūnių laipsniai yra lyginiai. Pasirinkę bet kurią pradinę viršūnę v_0, sukonstruosime Oilerio ciklą. Iš viršūnės v_0 einame bet kuria grafo briauna (v_0, u), iš viršūnės u vėl bet kuria briauna (u, w) ir t.t., kiekvieną jau panaudotą briauną pašalindami iš briaunų aibės. Kadangi grafo viršūnių laipsniai lyginiai, tai atėjus briauna į bet kurią grafo viršūnę visada dar turėsime bent vieną briauną išėjimui iš šios viršūnės. Galų gale mes pakliusime į pradinę viršūnę



3.8 Pav.: Karaliaučiaus tiltų uždavinys.

(nes iš visų kitų galėjome išeiti). Jei dar liko briaunų, kuriomis galime išeiti iš šios viršūnės (t.y., jei jos laipsnis pradžioje buvo didesnis už 2), tai vėl einame bet kuria briauna ir vėl kažkada grįšime į v_0. Taip tęsiame tol, kol grįžę į v_0 iš jos išeiti nebegalime. Gavome ciklą C. Šis ciklas nebūtinai yra Oilerio ciklas, nes dar gali būti likusių neapeitų briaunų. Nagrinėjame pradinio grafo G pografį G', kuris gavosi iš pradinio grafo pašalinus visas ciklui C priklausančias briaunas ir izoliuotas viršūnes (pografis G' nebūtinai jungus). Šis pografis būtinai turės bent vieną bendrą viršūnę su ciklu C (nes priešingu atveju ciklas C būtų buvęs atskira pradinio grafo komponentė, o taip negali būti). Tarkime, kad tai yra viršūnė v_1. Kadangi pografio G' viršūnių laipsniai yra lyginiai, tai išėję iš viršūnės v_1 ir vaikščiodami pografio G' briaunomis mes apeisime ciklą C_1 ir vėl sugrįšime į viršūnę v_1.

Dabar iš ciklų C ir C_1 sukonstruosime vieną ciklą, kurį vėl pavadinsime C. Einame ciklo C dalimi, jungiančia viršūnes v_0 ir v_1, po to apeiname visą ciklą C_1 ir grįžtame į viršūnę v_1, po to einame likusia ciklo C dalimi, jungiančia viršūnes v_1 ir v_0:

$$C := C(v_0, v_1) \cup C_1 \cup C(v_1, v_0).$$

Jei ciklas C yra Oilerio ciklas, teorema įrodyta. Jei ne, jis vėl turės viršūnę v_2, iš kurios dar galima išeiti ir rasti ciklą C_2. Tada ciklus C ir C_2 vėl galime apjungti į bendrą ciklą C. Taip tęsiant galų gale į ciklą C paklius visos duoto grafo briaunos. □

Pavyzdys 3.5. Panagrinėkime grafą, pavaizduotą 3.9 pav. Pradėję iš pradinės viršūnės c einame per viršūnes d, b, a, kol grįžtame į viršūnę c. Kadangi dar yra briaunų, kuriomis galima išeiti, tęsiame maršrutą, t.y., einame į f, iš f į e ir grįžtame į c. Gavome ciklą

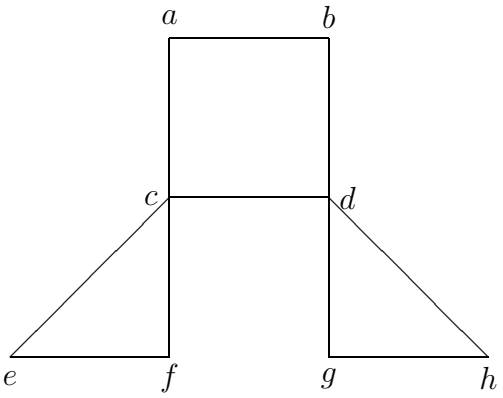
$$C = ((c, d), (d, b), (b, a), (a, c), (c, f), (f, e), (e, c)).$$

Liko neapeitas pografis, sudarytas iš viršūnių d, g, h ir jas jungiančių briaunų. Tame pografyje randame ciklą $C_1 = ((d, g), (g, h), (h, d))$, einantį per ciklo C viršūnę d. Pagaliau iš ciklų C ir C_1 konstruojame galutinį Oilerio ciklą, įterpdami ciklą C_1 į atitinkamą ciklo C vietą:

$$C = ((c, d), (d, g), (g, h), (h, d), (d, b), (b, a), (a, c), (c, f), (f, e), (e, c)).$$

Pavyzdys 3.6. Iš Oilerio teoremos išplaukia, kad grafas, vaizduojantis 7 Karaliaučiaus tiltus (žr. 3.8 pav.), nėra Oilerio grafas, nes jo visų viršūnių laipsniai yra nelyginiai. Taigi, Oileriui nepavyko pasivaikščioti taip, kaip jis buvo suplanavęs.

Oilerio teoremos įrodyme naudotas Oilerio ciklo konstravimo algoritmas yra vadinamas *iteratyviu*, nes jis iš pradžių randa bet kokį ciklą, o po to iteracija po iteracijos jį didina iki Oilerio ciklo. Iš teoremos įrodymo ir 3.4 pavyzdžio matyti, kad tokio algoritmo realizacijai tinka paieška gilyn, kai vietoje rekursijos mes naudojame steką. Iš pradinės viršūnės vykdome paiešką gilyn, talpindami praeitas briaunas į steką, kol niekur nebegalėsime išeiti. Taip steke atsidurs pradinio ciklo C iš teoremos įrodymo visos briaunos. Grįždami atgal antrą kartą praeinamas briaunas mes jau galime išimti iš ciklo ir įsiminti, nes jos ir sudarys ieškomą Oilerio ciklą.



3.9 Pav.: Iteratyvus Oilerio ciklo konstravimas.

Iteratyvųjį algoritmą galima užrašyti taip:

procedure $\mathrm{Euler}(G)$
$\mathrm{STEKAS} := \emptyset$; $\mathrm{CIKLAS} := \emptyset$;
$v :=$ bet kuri grafo G viršūnė;
$\mathrm{STEKAS} \Leftarrow v$; /* Įtraukiame viršūnę v į steką*/
while $\mathrm{STEKAS} \neq \emptyset$ **do**
  $v := \mathrm{top}(\mathrm{STEKAS})$;
  if $\mathrm{GRET}[v] \neq \emptyset$ **then**
    $u \Leftarrow \mathrm{GRET}[v]$;
    $\mathrm{STEKAS} \Leftarrow u$;
    $\mathrm{GRET}[v] := \mathrm{GRET}[v] \setminus \{u\}$;
    $\mathrm{GRET}[u] := \mathrm{GRET}[u] \setminus \{v\}$;
    $v := u$;
  else
    $v \Leftarrow \mathrm{STEKAS}$; /* Šaliname viršūnę v iš steko */
    $\mathrm{CIKLAS} \Leftarrow v$;
  end;
end;

Pavyzdys 3.6 (tęsinys). Taikome procedūrą Euler grafui iš 3.9 pav. Pradėjus iš viršūnės c ir apeinant grafo briaunas leksikografine tvarka, stekas pamažu pildysis, kol atrodys taip: $\mathrm{STEKAS} = [c, f, e, c, d, b, a, c]$. Kadangi praeidami briauna mes jos galus šalinome iš gretimumo sąrašų, tai šiuo metu bus sąrašas $\mathrm{GRET}[c]$ bus tuščias. Taigi, viršūnę c išimame iš steko ir įtraukiame į masyvą CIKLAS. Analogiškai elgiamės su viršūnėmis f, e, c. Gauname $\mathrm{STEKAS} = [d, b, a, c]$, $\mathrm{CIKLAS} = [c, e, f, c]$. Kadangi viršūnė d dar turi kaimyninių viršūnių, tai tęsiame paiešką gilyn, įtraukdami viršūnes g, h ir d į steką. Stekas tampa lygus $[d, h, g, d, b, a, c]$. Kadangi visos briaunos jau praeitos bent vieną kartą, tai bet kurios steko viršūnės gretimų viršūnių sąrašas yra tuščias, todėl jas po vieną išimame iš steko ir dedame į ciklą. Galų gale gauname Oilerio ciklą $\mathrm{CIKLAS} = [c, e, f, c, d, h, g, d, b, a, c]$.

Kadangi paieška gilyn lygiai du kartus praeina visas grafo briaunas, tai Oilerio ciklo paieškos uždavinio sudėtingumas yra $L(m, n) = \mathrm{O}(m)$.

4 skyrius

Sudėtingumo klasės ir jų hierarchija

4.1 Kalbų ir uždavinių ryšys

Abstrakčiu uždaviniu vadiname sąryšį $Q \subset I \times O$, kur I yra įėjimų (duomenų) aibė, o O yra išėjimų (galimų sprendinių) aibė.

Pavyzdys 4.1 (Uždavinys SHORTEST-PATH**).** Duota grafas $G = (V, E)$ ir dvi to grafo viršūnės $v_1, v_2 \in V$. Rasti trumpiausią kelią $K(v_1, v_2)$ (kelio ilgiu laikome jį sudarančių briaunų skaičių). Čia įėjimų aibė yra grafų aibės ir jų viršūnių porų aibės Dekarto sandauga, t.y., $I = \{(V, E, v_1, v_2)\}$, o išėjimų aibė yra aibė grafo viršūnių baigtinių sekų: $O = \{\{u_1, \ldots, u_n\}$:
$u_i \in V\}$.

Trumpiausio kelio radimo uždavinys SHORTEST-PATH $\subset I \times O$ yra *optimizavimo* uždavinio pavyzdys.

Pavyzdys 4.2 (Uždavinys PATH**).** Duota grafas $G = (V, E)$, dvi to grafo viršūnės $v_1, v_2 \in V$ ir natūralusis skaičius $k \in \mathbb{N}$. Rasti, ar grafe G tarp viršūnių v_1 ir v_2 egzistuoja kelias $K(v_1, v_2)$, ne ilgesnis už k, t.y., toks, kad $|K(v_1, v_2)| \leq k$.

Tai yra *egzistavimo* uždavinio pavyzdys. Kadangi atsakymą "taip" galima koduoti 1, o atsakymą "ne" — 0, tai egzistavimo uždavinių išėjimų aibė yra aibė $\{0, 1\}$. Pažymėję $I = \{(V, E, v_1, v_2, k)\}$, kur $k \in \mathbb{N}$, gauname, kad PATH $\subset I \times \{0, 1\}$ yra egzistavimo uždavinys.

Iš šių pavyzdžių matyti, kad bet kurį optimizavimo uždavinį Q atitinka egzistavimo uždavinys E, o atskirą optimizavimo uždavinio atvejį $q = (i, o) \in Q$ atitinka atskirų egzistavimo uždavinio atvejų aibė $\{e(k) = (i, k, o')$: $k \in \mathbb{N}\} \subset E$. Čia abstrakčiu uždaviniu vadiname to paties tipo uždavinių klasę (tai ką mes 1 skyriuje žymėjome $\mathcal{U}$), atskirais uždavinio atvejais vadiname konkrečius tos klasės uždavinius ($U \in \mathcal{U}$).

Parodysime (ne visai griežtai), kad nagrinėjant uždavinių sudėtingumą, galima apsiriboti egzistavimo uždaviniais. Kaip ir anksčiau, uždavinio Q atskiro atvejo $q = (i, o) \in Q$ dydžiu $|q|$ vadiname parametrą n, apibūdinantį q dydį klasėje Q.

Tegu $f : \mathbb{N} \to \mathbb{N}$. Uždavinį Q vadiname *f-sunkiu*, jei egzistuoja algoritmas A, kurio sudėtingumas sprendžiant uždavinį Q

$$\mathrm{TIME}_A(q) = \Omega(f(|q|)) \quad \forall q \in Q.$$

Lema 4.1. *Nagrinėjant uždavinių sudėtingumą, galima apsiriboti egzistavimo uždaviniais, t.y., jei $f : \mathbb{N} \to \mathbb{N}$ ir egzistavimo uždavinys E yra f-sunkus, tai ir jį atitinkantis optimizavimo uždavinys Q yra f-sunkus.*

Įrodymas. Tarkime, Q ir E yra vienas kitą atitinkantys optimizavimo ir egzistavimo uždaviniai, ir uždavinys E yra f-sunkus. Įrodysime, kad tada ir uždavinys Q yra f-sunkus.

Tarkime, kad uždavinys Q yra paprastesnis už uždavinį E, t.y., egzistuoja algoritmas A toks, kad $\mathrm{TIME}_A(q) = \mathrm{o}(f(|q|))\ \forall q \in Q$. Kaip jau aukščiau minėjome, kiekvieną $q \in Q$ atitinka aibė $\{e(k)\} \subset E$. Be to, $|q| = |e(k)|$, nes egzistavimo uždaviniui reikia tų pačių duomenų, kaip ir optimizavimo uždaviniui, papildomai dar nurodant natūralųjį skaičių k. Pridėję dar vieną žingsnį prie algoritmo A mes gautume algoritmą B, sprendžiantį bet kurį uždavinį $e(k_0)$: tam, kad išspręsti $e(k_0)$, reikia pirma išspręsti Q, o po to gautą sprendinį palyginti su skaičiumi k_0. Aišku, kad algoritmo B sudėtingumas tada būtų

$$\mathrm{TIME}_B(e(k_0)) = \mathrm{o}(f(|Q|)) = \mathrm{o}(f(|e(k_0)|)),$$

o tai prieštarautų prielaidai, kad uždavinys E yra f-sunkus. □

Remdamiesi šia lema, toliau nagrinėsime tik abstrakčius uždavinius $Q \subset I \times \{0,1\}$.

Konkrečiu uždaviniu vadiname sąryšį $\mathcal{U} \subset \{0,1\}^* \times \{0,1\}$. Taigi, konkretaus uždavinio duomenys yra nulių bei vienetų seka, o jo sprendinys yra skaičius 0 arba 1. Vietoje abstrakčių uždavinių galima nagrinėti konkrečius uždavinius, naudojant *kodavimą* $e : I \to \{0,1\}^*$. Tada bet kurį abstraktų uždavinį $Q \subset I \times \{0,1\}$ atitiks konkretus uždavinys $\mathcal{U} = e(Q) \subset \{0,1\}^* \times \{0,1\}$. Naudodami abstrakčių uždavinių kodavimą, galėsime lyginti įvairaus tipo uždavinių sudėtingumą (uždavinių apie grafus, logines formules bei schemas, veiksmų su matricomis ir pan.).

Sakysime, kad algoritmas A sprendžia konkretų uždavinį $\mathcal{U} = e(Q) \subset \{0,1\}^* \times \{0,1\}$ per laiką $\mathrm{O}(f(n))$ ir žymėsime $T_A^{\mathcal{U}}(n) = \mathrm{O}(f(n))$ jei $\forall i \in \{0,1\}^*$,

$$T_A(i) = \mathrm{O}(f(|i|)),$$

kur $|i|$ yra žodžio $i \in \{0,1\}^*$ ilgis. *Sudėtingumo klase* P vadinsime aibę konkrečių uždavinių, kuriems egzistuoja algoritmai, sprendžiantys tuos uždavinius per polinominį laiką, t.y.,

$$\mathrm{P} = \{\mathcal{U}\text{: } \exists A \exists k \text{ tokie, kad } T_A^{\mathcal{U}}(n) = \mathrm{O}(n^k)\}.$$

Kadangi naudodami įėjimų aibės kodavimą $e : I \to \{0,1\}^*$ abstrakčius uždavinius galime koduoti konkrečiais uždaviniais, tai galime kalbėti ir apie polinominio sudėtingumo abstrakčius uždavinius. Tačiau reikia susitarti, kokį kodavimą galima naudoti. Pasirodo,

koduojant aibę I skirtingais būdais, vienu atveju konkretus uždavinys $e(Q)$ gali priklausyti aibei P, o kitu atveju gali nepriklausyti. Taip, pavyzdžiui, gali atsitikti, natūraliuosius skaičius koduojant dvejetainiu pavidalu ir "vienetainiu" pavidalu (t.y., skaičių n koduojant $n+1$ vienetu), nes antrasis kodas bus eksponentiškai ilgesnis už pirmąjį.

Šiame skyriuje koduodami abstrakčius uždavinius laikysimės *standartinio kodavimo* e_0 arba jam ekvivalentaus:

(1) natūraliuosius skaičius koduoja dvejetainiu pavidalu, t.y., $e_0(n) = \alpha_1\alpha_2\ldots\alpha_k$, kur $k = \lfloor \log_2 n \rfloor + 1$, jei $n > 0$, ir $k = 1$, jei $n = 0$;

(2) baigtinę aibę A koduoja žodžiu, sudarytu iš tos aibės elementų kodų:

$$e_0(A) = e_0(a_1)\#e_0(a_2)\#\ldots\#e_0(a_n);$$

(3) grafą koduoja žodžiu, sudarytu iš to grafo viršūnių aibės ir briaunų aibės kodų: $e_0(G) = e_0(V)\#e_0(E)$ ir t.t.

Toks kodavimas leis laisvai operuoti su abstrakčių uždavinių sudėtingumu. Uždavinio duomenų $i \in I$ kodą $e(i) \in \{0,1\}^*$ toliau žymėsime kampuotais skliausteliais, t.y. vietoje $e(i)$ rašysime tiesiog $\langle i \rangle$.

Kadangi kiekvienas konkretus uždavinys yra atvaizdavimas $u : \{0,1\}^* \to \{0,1\}$, tai kiekvieną konkretų uždavinį U atitinka kalba $L_U = u^{-1}(1) \subset \{0,1\}^*$, tai yra aibė žodžių, kuriems uždavinio U sprendinys yra lygus 1. Pavyzdžiui, kalba

$$\mathrm{PRIME} = \{10, 11, 101, 111, 1011, \ldots\}$$

atitinka abstraktų egzistavimo uždavinį, ar duotas natūralusis skaičius yra pirminis.

Taigi šiame skyrelyje mes pademonstravome, kad kalbant apie uždavinių sudėtingumą, vietoje optimizavimo uždavinių galima apsiriboti egzistavimo uždaviniais, pastaruosius galima koduoti konkrečiais uždaviniais, kurių duomenys sudaryti tik iš nulių ir vienetų, o konkrečius uždavinius atitinka kalbos abėcėlėje $\{0,1\}$. Tai mums leidžia vietoje uždavinių sudėtingumo klasių toliau nagrinėti kalbų sudėtingumo klases

4.2 Sudėtingumo klasės P ir NP

Algoritmas A *išsprendžia* kalbą L, jei $\forall x \in \{0,1\}^*$

$$A(x) = \begin{cases} 1, & x \in L, \\ 0, & x \notin L. \end{cases}$$

Algoritmas A *priima* kalbą L, jei

$$A(x) = 1 \iff x \in L.$$

Taigi, jei žodis nepriklauso kalbai L, tai išsprendžiantis algoritmas išduoda atsakymą 0, tuo tarpu priimantis algoritmas išduoda atsakymą 0 arba niekada nesustoja.

Sudėtingumo klase P vadiname polinomiškai išsprendžiamų kalbų klasę:

$$\mathrm{P} = \{L \subseteq \{0,1\}^* \colon \exists \text{ algoritmas } A \text{ toks, kad } A \text{ išsprendžia } L \text{ per polinominį laiką}\}.$$

Pavyzdys 4.3. Riboto ilgio kelio tarp dviejų grafo viršūnių egzistavimo uždavinį atitinka kalba

$$\mathrm{PATH} = \{\langle G, u, v, k\rangle\colon\ G = (V, E) \text{ — grafas}, u, v \in V,\ k \geq 0 \text{ — sveikasis skaičius,} \\ \text{ir egzistuoja kelias } K(u, v) \text{ iš } u \text{ į } v \text{ ilgio} \leq k\}.$$

Kadangi yra žinoma nemažai polinominių trumpiausio kelio radimo grafe algoritmų (pvz., Deikstros), tai pritaikę tokį algoritmą duomenims G, u, v ir jo gautą rezultatą (trumpiausio kelio ilgį) palyginę su skaičiumi k, gauname polinominį algoritmą, išsprendžiantį kalbą PATH, taigi $\mathrm{PATH} \in \mathrm{P}$.

Turingo mašinos sustojimo problemą atitinka kalba

$$\mathrm{HALT} = \{\langle M, x\rangle\colon \text{Turingo mašina } M \text{ duomenims } x \text{ sustoja, t.y., } M(x)\downarrow\}.$$

Yra žinoma, kad ši problema algoritmiškai neišsprendžiama, tačiau egzistuoja (nepolinominis) algoritmas, priimantis kalbą HALT. Šis algoritmas — tai universali Turingo mašina U, modeliuojanti $M(x)$ darbą. Jei $M(x)\downarrow$, tai ir mašina $U(\langle M, x\rangle)$ sustos, t.y., priims žodį $\langle M, x\rangle$.

Pavyzdys 4.3 rodo, kad algoritmiškai išsprendžiamų ir priimamų kalbų klasės skiriasi. Tačiau jei mes apsiribosime tik polinominio sudėtingumo algoritmais, tai šios klasės sutaps.

Lema 4.2.

$$\mathrm{P} = \{L \subseteq \{0,1\}^*\colon \exists \textit{ algoritmas } A \textit{ toks, kad } A \textit{ priima } L \textit{ per polinominį laiką}\}.$$

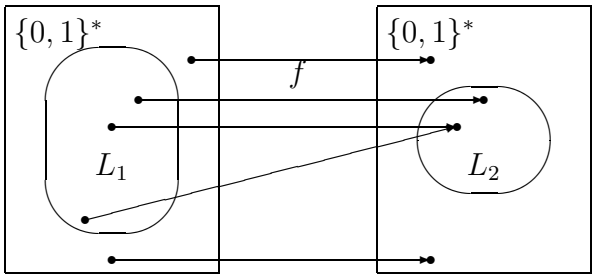
Įrodymas. Kadangi kokią nors kalbą L išsprendžiantis polinominis algoritmas yra kartu ir kalbą L priimantis algoritmas, tai pakanka tik parodyti, kad jei egzistuoja kalbą L primantis polinominis algoritmas A, tai egzistuoja ir kalbą L išsprendžiantis polinominis algoritmas A'. Kadangi A polinominis, tai atsiras konstantos $c > 0$ ir $k \in \mathbb{Z}$ tokios, kad $A(x)$ sustoja po ne daugiau kaip $T = c|x|^k$ žingsnių kiekvienam $x \in L$. Algoritmas A' bus universalaus algoritmo modifikacija. Jis modeliuoja algoritmo $A(x)$ darbą ir skaičiuoja žingsnius. Jei po T žingsnių algoritmas $A(x)$ dar nesustojo, tai algoritmas $A'(x)$ sustoja ir išduoda atsakymą 0. Jei $A(x)$ sustoja kuriame nors žingsnyje $t \leq T$, tai $A'(x)$ irgi sustoja ir išduoda atsakymą sutampantį su $A(x)$ atsakymu. □

Funkciją $f : \{0,1\}^* \to \{0,1\}^*$ vadiname *polinomiškai apskaičiuojama*, jei egzistuoja polinominio sudėtingumo algoritmas A, kuris randa $f(x)$.

Kalbą L_1 vadinsime *polinomiškai redukuojama* į kalbą L_2 ir žymėsime $L_1 \leq_{\mathrm{p}} L_2$ jei egzistuoja polinomiškai apskaičiuojama funkcija f tokia, kad

$$x \in L_1 \iff f(x) \in L_2.$$

Polinominę redukciją iliustruoja 4.1 pav. Norint nustatyti, ar $x \in L_1$ pakanka nustatyti ar $f(x) \in L_2$.



4.1 Pav.: Polinominė redukcija.

Pavyzdys 4.4. Apibrėžkime kalbas

$$\text{EULER-CYCLE} = \{\langle G\rangle\colon G \text{ — Euler'io grafas}\}$$

ir

$$\text{EVEN-SEQUENCE} = \{\langle(k_1,\ldots,k_n)\rangle\colon k_1,\ldots,k_n \text{ yra lyginiai natūralieji skaičiai}\}.$$

Kadangi yra žinoma teorema, kad grafas yra Euler'io tada ir tik tada, kai visų jo viršūnių laipsniai yra lyginiai, tai polinominis algoritmas F, randantis duoto grafo visų viršūnių laipsnius, polinomiškai redukuos kalbą EULER-CYCLE į kalbą EVEN-SEQUENCE: $\text{EULER-CYCLE} \leq_{\text{p}} \text{EVEN-SEQUENCE}$.

Polinominė redukcija leidžia lengvai įrodyti kalbų priklausomybę klasei P redukuojant vienas kalbas į kitas (t.y., nereikia konstruoti tiesioginio išsprendžiančio algoritmo).

Lema 4.3. *Jei $L_1 \leq_{\text{p}} L_2$ ir $L_2 \in \text{P}$, tai ir $L_1 \in \text{P}$.*

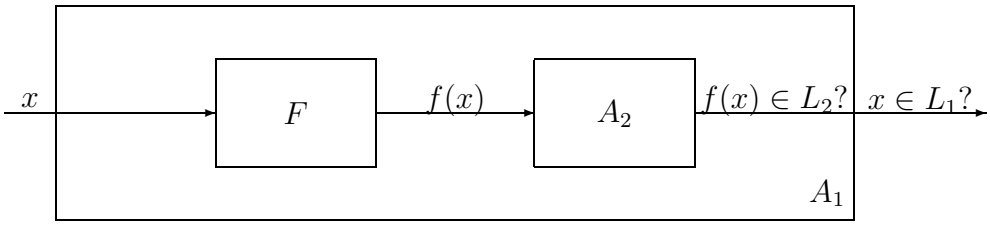
Įrodymas. Tarkime, kad A_2 yra polinominis algoritmas išsprendžiantis kalbą L_2 ir F yra polinominis algoritmas apskaičiuojantis funkciją f tokią, kad $x \in L_1 \iff f(x) \in L_2$. Konstruosime polinominį algoritmą A_1 išsprendžiantį L_1.

Algoritmas A_1 bus paprasčiausia algoritmų F ir A_2 superpozicija (žr.4.2 pav.). Norint nustatyti ar $x \in L_1$ reikia apskaičiuoti $f(x)$ ir gautą rezultatą pateikti kaip duomenis algoritmui A_2. Jo atsakymas rodys ar $x \in L_1$. Kadangi dviejų polinomų suma yra polinomas, tai algoritmas bus polinominis. □

Klasę NP galima apibrėžti kaip klasę kalbų L, kurioms egzistuoja nedeterminuota Turingo mašina M priimanti L per polinominį laiką, t.y., $\forall x \in L$ atsiras skaičiavimo medžio šaka, kurioje M sustoja po ne daugiau kaip po $\text{O}(|x|^k)$ žingsnių ir išduoda atsakymą 1. Pateiksime kitą klasės NP apibrėžimą, kuris leidžia žymiai palengvinti įrodymus, kad nagrinėjamos kalbos priklauso klasei NP. Yra įrodyta, kad šie apibrėžimai yra ekvivalentūs.

Sakysime kad algoritmas A (turintis 2 įėjimus) *patvirtina* žodį $x \in \{0,1\}^*$ jei egzistuoja kitas žodis $y \in \{0,1\}^*$ toks, kad $A(x,y) = 1$. Žodį y vadinsime žodžio x *liudijimu* (arba sertifikatu). Kalba L vadinsime *patvirtinta* algoritmo A, jei

$$L = \{x \in \{0,1\}^*\colon \text{egzistuoja } y \in \{0,1\}^* \text{ toks, kad } A(x,y) = 1\}.$$



4.2 Pav.: Lemos 4.3 įrodymo schema.

Sudėtingumo klase NP vadinsime aibę kalbų, kurios gali būti patvirtintos per *polinominį laiką*, naudojant *polinominio ilgio liudijimus*. Taigi, $L \in \mathrm{NP}$ tada ir tik tada kai egzistuoja polinominis algoritmas A, turintis du įėjimus, ir egzistuoja sveikas skaičius l toks, kad

$$L = \{x \in \{0,1\}^*\colon \begin{array}{l}\text{egzistuoja liudijimas } y \in \{0,1\}^* \text{ ilgio } |y| = \mathrm{O}(|x|^l) \\ \text{toks, kad } A(x,y) = 1\}.\end{array}$$

Pavyzdys 4.5. Panagrinėkime kalbą

$$\text{HAM-CYCLE} = \{\langle G = (E,V)\rangle\colon G \text{ — Hamiltono grafas}\}.$$

Niekam dar nepavyko surasti polinominio algoritmo išsprendžiančio šią kalbą. Bandant perrinkti visus galimus Hamiltono ciklus, gausime $\mathrm{O}(n!)$ sudėtingumo algoritmą. Tačiau, tarkime, pas jus ateina draugas ir sako: "Žinai, aš tavo grafe G radau Hamiltono ciklą" bei pateikia jums viršūnių seką H. Aišku, kad tokiu atveju nesunku rasti polinominį algoritmą, kuris patikrina ar ta seka H tikrai bus Hamiltono ciklas. Tereikia patikrinti ar H yra viršūnių aibės V kėlinys ir ar tikrai aibėje E egzistuoja briaunos jungiančios gretimas (o taip pat paskutinę ir pirmąją) sekos H viršūnes. Tam pakanka $\mathrm{O}(n^2)$ operacijų ($n = |V|$). Taigi, sekos H kodas $y = \langle H\rangle$ bus liudijimas, kad jūsų grafas G yra Hamiltono grafas, ir kalba HAM-CYCLE priklausys klasei NP.

Lema 4.4. $\mathrm{P} \subseteq \mathrm{NP}$.

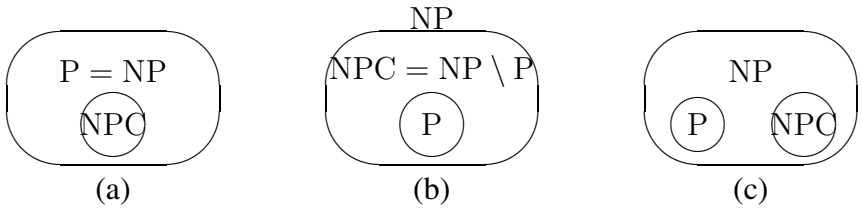
Įrodymas. Bet kuriam žodžiui $x \in \{0,1\}^*$ liudijimu y paėmę tuščią žodį, gauname, kad tas pats algoritmas kuris polinomiškai išsprendžia kalbą L, kartu ir patvirtina tą kalbą per polinominį laiką. Taigi, $L \in \mathrm{P} \Rightarrow L \in \mathrm{NP}$. □

Yra manoma, kad sudėtingumo klasė P yra tikras poaibis klasės NP, t.y. $\mathrm{P} \subset \mathrm{NP}$. Tačiau kol kas šios hipotezės nepavyksta nei įrodyti, nei paneigti. Tai viena iš centrinių informatikos mokslo problemų!

4.3 Sudėtingumo klasė NPC

4.3.1 Klasės NPC apibrėžimas

Kalbą L vadiname NP-*sunkia*, jei $L' \leq_{\mathrm{p}} L$ kiekvienai $L' \in \mathrm{NP}$. Kalbą L vadiname NP-*pilna*, jei:



(a) (b) (c)

4.3 Pav.: Sudėtingumo klasės P, NP ir NPC.

(1) $L \in \mathrm{NP}$ ir

(2) L yra NP-sunki.

NP-pilnų kalbų klasę žymėsime NPC. Dabar parodysime, kad NP-sunkios kalbos betarpiškai siejasi su problemos "$\mathrm{P} = \mathrm{NP}$?" sprendimu.

Teorema 4.1. (1) *Jei bent viena* NP*-pilna kalba yra polinomiškai išsprendžiama* (*priklauso* P)*, tai* $\mathrm{P} = \mathrm{NP}$.

(2) *Jei bent viena* NP*-pilna kalba nėra polinomiškai išsprendžiama* (*nepriklauso* P)*, tai ir visos kitos* NP*-pilnos kalbos nėra polinomiškai išsprendžiamos* ($\mathrm{P} \cap \mathrm{NPC} = \emptyset$).

Įrodymas. Tarkime, kad $L \in \mathrm{NPC}$ ir $L \in \mathrm{P}$. Jei $L' \in \mathrm{NP}$, tai pagal NP-pilnumo apibrėžimą $L' \leq_{\mathrm{p}} L$, taigi pagal 4.3 lemą ir $L' \in \mathrm{P}$. Taigi, $\mathrm{NP} \subseteq \mathrm{P}$, o tai reiškia, kad $\mathrm{P} = \mathrm{NP}$.

Norėdami įrodyti antrą teoremos dalį tarkime, kad $L \in \mathrm{NPC}$ ir $L \notin \mathrm{P}$. Jei atsirastų kalba L' tokia, kad $L' \in \mathrm{NPC}$ ir $L' \in \mathrm{P}$, tai kadangi $L \leq_{\mathrm{p}} L'$, pagal 4.2.2 lemą gautume $L \in \mathrm{P}$, o tai prieštarautų pradinei prielaidai. □

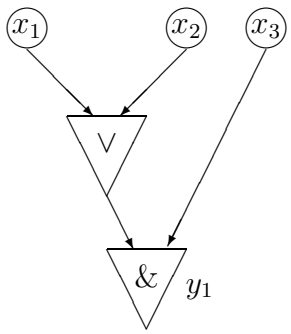
Pav. 4.3 vaizduoja tris galimus klasių P, NP ir NPC tarpusavio sąryšių variantus. Dauguma tyrinėtojų linkę manyti, kad teisingas yra variantas (c):

$$\mathrm{P} \subset \mathrm{NP}, \quad \mathrm{NPC} \subset \mathrm{NP} \quad \text{ir} \quad \mathrm{P} \cap \mathrm{NPC} = \emptyset.$$

Kadangi NP-pilni uždaviniai yra tokie svarbūs algoritmų sudėtingumo teorijoje, tai kiekvieno naujo uždavinio įtraukimas į šią klasę vertinamas kaip naujas mokslinis rezultatas. Šiuo metu yra žinoma keli šimtai, o kartu su įvairiomis tų pačių uždavinių versijomis gal ir keli tūkstančiai NP-pilnų uždavinių. Norint įrodyti, kad kalba L yra NP-sunki, reikia įrodyti, kad į kalbą L galima polinomiškai redukuoti bet kurią kitą klasės NP kalbą. Tiesiogiai tai įrodyti būna gana sunku. Tačiau jei mes tai tiesiogiai įrodytume bent vienai kalbai, tai visoms kitoms kalboms galima būtų taikyti tokią lemą:

Lema 4.5. *Jei* $L' \leq_{\mathrm{p}} L$ *ir* $L' \in \mathrm{NPC}$*, tai* L *yra* NP*-sunki. Be to, jei* $L \in \mathrm{NP}$*, tai tada* $L \in \mathrm{NPC}$.

Įrodymas. Kadangi L' yra NP-sunki, tai kiekvienai $L'' \in \mathrm{NP}$ turime $L'' \leq_{\mathrm{p}} L'$. Polinominė redukcija tenkina tranzityvumo dėsnį (dviejų polinomų suma vėl bus polinomas),



4.4 Pav.: Išpildoma Būlio schema.

todėl gauname $L'' \leq_{\mathrm{p}} L$, taigi L yra NP-sunki. Jei dar žinoma, kad $L \in \mathrm{NP}$, tai pagal apibrėžimą kalba L bus NP-pilna. $\square$

Ši lema mums duoda paprastą būdą kaip įrodyti, kad kuri nors kalba L yra NP-sunki:

(1) Reikia įrodyti, kad $L \in \mathrm{NP}$.

(2) Reikia pasirinkti jau žinomą NP-pilną kalbą L'.

(3) Reikia pateikti algoritmą F realizuojantį funkciją $f : \{0,1\}^* \to \{0,1\}^*$, kuri kalbos L' žodžius atvaizduoja į kalbos L žodžius.

(4) Reikia įrodyti, kad bet kokiam $x \in \{0,1\}^*$ bus teisinga $x \in L'$ tada ir tik tada kai $f(x) \in L$.

(5) Reikia įrodyti, kad algoritmas F yra polinominio sudėtingumo.

Pirmasis uždavinys, kuris mums leis praverti klasės NPC duris ir įkišti į tarpdurį koją (t.y., šį uždavinį) bus Būlio schemos išpildomumo uždavinys CIRCUIT-SAT.

4.3.2 Uždavinys CIRCUIT-SAT

Šiame skyrelyje nagrinėsime Būlio schemas su neribotu įėjimų skaičiumi ("unbounded fan-in circuits", angl.), t.y., sudarytas iš funkcinių elementų $\&, \vee, \neg$, kur elementai $\&$ realizuoja bet kokio ilgio konjunkcijas $z_1 \,\&\, \cdots \,\&\, z_k$, o elementai $\vee$ realizuoja bet kokio ilgio disjunkcijas $z_1 \vee \cdots \vee z_l$. Būlio schemą S su n įėjimų $x_1, \ldots, x_n$ ir 1 išėjimu vadiname *išpildoma*, jei egzistuoja kintamųjų $x_1, \ldots, x_n$ reikšmių rinkinys $(\alpha_1, \ldots, \alpha_n) \in \{0,1\}^n$ toks, kad įstačius į schemos S įėjimus reikšmes $x_1 = \alpha_1, \ldots, x_n = \alpha_n$, šios schemos išėjime gauname reikšmę 1. Pav. 4.4 vaizduojama schema S yra išpildoma, nes parinkę įėjimų reikšmes $x_1 = x_2 = x_3 = 1$, schemos išėjime gauname 1.

Kadangi Būlio schemos yra orientuotieji žymėtieji grafai, jas galima koduoti nuliukais ir vienetukais bet kuriuo būdu, naudojamu koduoti grafams. Taigi, kiekvienai schemai S galime priskirti jos kodą $\langle S \rangle$. Jei schema S turi n įėjimų ir m funkcinių elementų, tai

akivaizdu, kad jų skaičius yra ne didesnis už schemos kodo ilgį ($n, m \leq |\langle S \rangle|$), nes kiekvienai grafo viršūnei koduoti prireiks bent vieno bito. Pažymėkime

$$\text{CIRCUIT-SAT} = \{\langle S \rangle \colon S \text{ — išpildoma Būlio schema}\}.$$

Akivaizdu, kad jei Būlio schema turi n įėjimų, tai perrinkus visas galimas jų reikšmes $(\alpha_1, \ldots, \alpha_n) \in \{0,1\}^n$ galima nustatyti, ar duotoji schema yra išpildoma (brutalios jėgos algoritmas). Kai schemos dydis polinomiškai priklauso nuo įėjimų skaičiaus, gauname, kad brutalios jėgos algoritmas yra eksponentinio sudėtingumo schemos kodo ilgio atžvilgiu. Deja, nieko geriau uždaviniui CIRCUIT-SAT nėra žinoma.

Teorema 4.2. CIRCUIT-SAT $\in$ NPC.

Įrodymas. Pagal NP pilnų kalbų apibrėžimą reikia įrodyti, kad:

(1) CIRCUIT-SAT $\in$ NP ir

(2) uždavinys CIRCUIT-SAT yra NP-sunkus.

(1) Liudijimu bus schemos S įėjimų reikšmių rinkinys $y = (\alpha_1, \ldots, \alpha_n) \in \{0,1\}^n$, kuriam schemos S išėjimo reikšmė bus lygi 1. Liudijimo ilgis yra ne didesnis už schemos kodo ilgį $|\langle S \rangle|$. Kadangi kiekvienas funkcinis schemos elementas turi ne daugiau kaip $|\langle S \rangle|$ įėjimų, o pačių elementų taip pat yra ne daugiau kaip $|\langle S \rangle|$, tai per polinominį žingsnių skaičių galime apskaičiuoti kiekvieno funkcinio elemento bei schemos išėjimo reikšmę. Taigi, egzistuoja polinominio sudėtingumo algoritmas $A(x, y)$, toks, kad $\langle S \rangle \in$ CIRCUIT-SAT $\leftrightarrow \exists y \colon A(\langle S \rangle, y) = 1$ ir liudijimo y ilgis yra polinominis.

(2) Tegu L — bet kuri kalba iš klasės NP. Įrodysime, kad $L \leq_{\text{p}}$ CIRCUIT-SAT. Konstruosime polinominio sudėtingumo algoritmą F, kuris bet kuriam žodžiui $x \in \{0,1\}^*$ priskirs schemą $S_x = F(x)$ tokią, kad $x \in L$ tada ir tik tada, kai S_x yra išpildoma, t.y., $\langle S_x \rangle \in$ CIRCUIT-SAT.

Kadangi $L \in$ NP, tai egzistuoja polinominis algoritmas $A(x, y)$, patikrinantis kalbą L per polinominį laiką. Galime laikyti, kad algoritmas A yra programa, parašyta žemo lygio kalba, naudojančia atminties adresus. Tada kompiuterio atmintyje programa A yra saugoma kaip komandų sąrašas, kur kiekvieną komandą sudaro operacijos kodas, operandų adresai kompiuterio atmintyje ir adresas, kur reikia įrašyti operacijos rezultatą. Specialioje atminties vietoje laikomas *komandų skaitiklis* KS, saugantis adresą komandos, kurią reikės vykdyti. Vykdant programą, šis adresas nuolat keičiasi, nurodydamas arba sekančią komandą, stovinčią po jau įvykdytos, arba kurią nors kitą komandą, jei buvo vykdoma sąlyginė ar ciklo komanda.

Laikysime, kad be vykdomos programos ir komandų skaitiklio kompiuterio atmintį dar sudaro procesoriaus registrai, pradiniai programos duomenys ir darbinė atmintis (žr. 4.5 pav.). Fiksuotą atminties būseną (t.y., atminties ląstelių (bitų) reikšmes tam tikru laiko momentu) vadinsime *konfigūracija*. Vienos komandos vykdymą atitinka ankstesnės konfigūracijos atvaizdavimas į naują konfigūraciją, kurį atlieka kompiuterio procesorius. Kadangi bet kuri konfigūracija yra nuliukų ir vienetukų rinkinys, tai šį atvaizdavimą galima realizuoti Būlio schema M. Kadangi pagal algoritmų savybes kiekviena komanda

turi būti elementari, tai visada galima išsirinkti tokį komandų rinkinį, kad vienai komandai įvykdyti pakaks polinominio dydžio Būlio schemos (priklausomai nuo konfigūracijos ilgio).

Pažymime $T(n)$ algoritmo A žingsnių skaičių blogiausiu atveju duomenims ilgio n. Tegu $k \geq 1$ yra tokia konstanta, kad $T(n) = \mathrm{O}(n^k)$ ir liudijimo y ilgis taip pat yra $\mathrm{O}(n^k)$. Taip visada galima pasirinkti, nes algoritmo A sudėtingumas yra polinominis jo įėjimo ilgio atžvilgiu, o jo įėjimo ilgis yra $n + |y|$, kur $|y|$ savo ruožtu yra polinomas n atžvilgiu.

Dabar algoritmo A darbą galime vaizduoti $T(n)$ konfigūracijų seka $c_0, c_1, \ldots, c_{T(n)}$, kur pradinę konfigūraciją c_0 sudaro programa A, komandų skaitiklis KS, procesoriaus registrai, pradiniai duomenys x ir y bei darbinė atmintis. Konfigūraciją c_i padavus į schemos M įėjimus, jos išėjimuose gauname konfigūraciją c_{i+1} ($i = 0, 1, \ldots, T(n) - 1$).

Aprašysime algoritmą F, kuris pagal duotą x konstruoja schemą S_x tokią, kad S_x būtų išpildoma tada ir tik tada, kai egzistuoja liudijimas y toks, kad $A(x, y) = 1$. Algoritmas F pirmiausia apskaičiuoja $n = |x|$ bei $T(n) = \mathrm{O}(n^k)$ ir konstruoja Būlio schemą S', sudarytą iš $T(n)$ schemos M kopijų. Šios schemos įėjimas bus pradinė skaičiavimo $A(x, y)$ konfigūracija, o išėjimas bus konfigūracija $c_{T(n)}$. Dabar schemą S_x nesunku gauti iš schemos S'. Pirmiausia schemos S' įėjimus, atitinkančius programą A, komandų skaitiklį, procesoriaus registrus, pradinę darbinės atminties konfigūraciją ir įėjimą x, fiksuojame, t.y., prijungiame tiesiai prie žinomų jų reikšmių iš aibės $\{0, 1\}$. Taigi, laisvi lieka tik schemos įėjimai, atitinkantys liudijimą y. Antra, visus schemos S' išėjimus ignoruojame, išskyrus vieną, kuriame gaunamas algoritmo $A(x, y)$ darbo rezultatas. Šis išėjimas ir bus konstruojamos schemos S_x išėjimas.

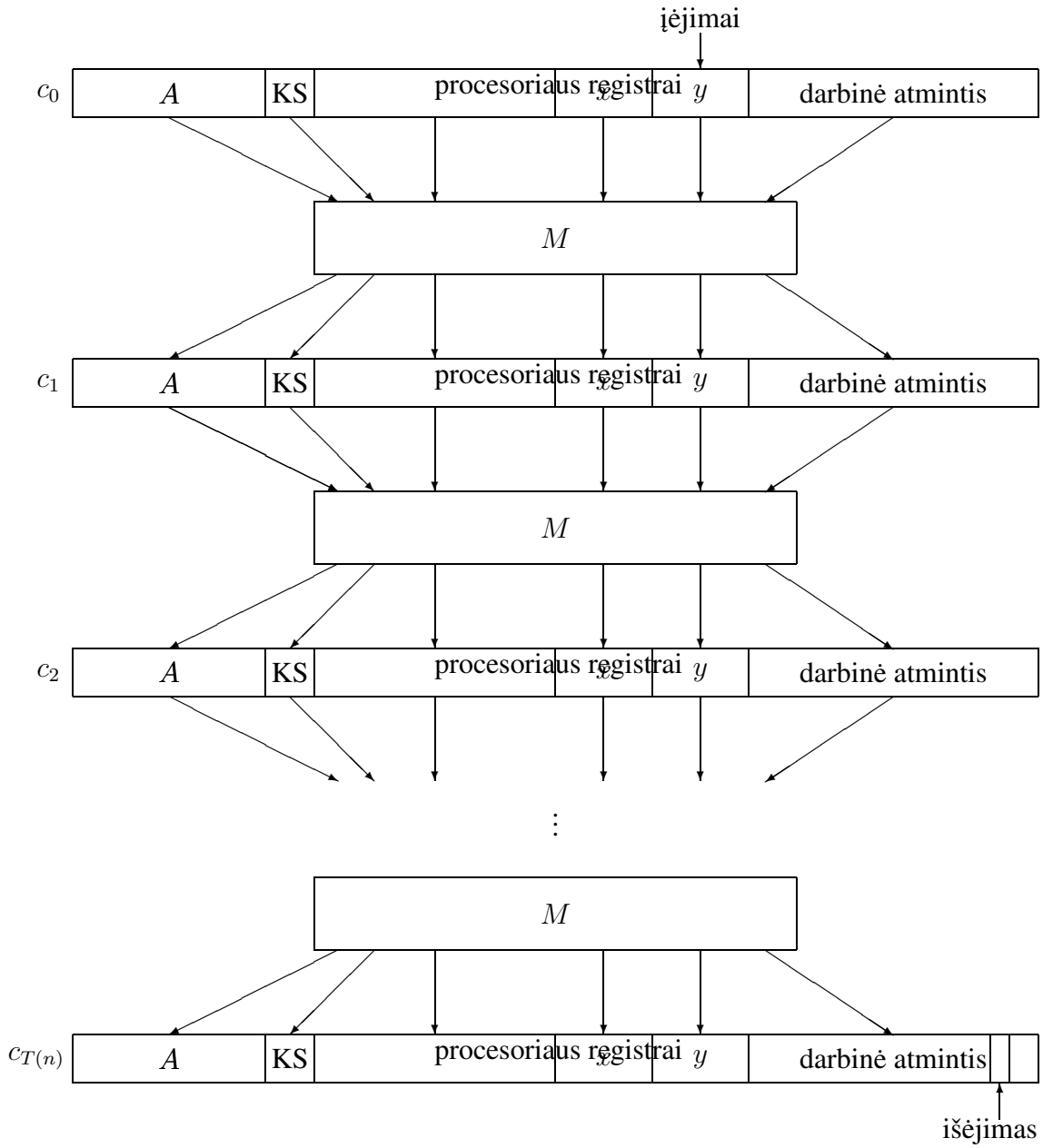
Lieka įrodyti, kad:

(a) S_x yra išpildoma tada ir tik tada, kai egzistuoja polinominio ilgio liudijimas y toks, kad $A(x, y) = 1$ ir

(b) algoritmas F yra polinominio sudėtingumo $n = |x|$ atžvilgiu.

(a) Tarkime, kad egzistuoja polinominio ilgio liudijimas y toks, kad $A(x, y) = 1$. Pateikę šias y reikšmes schemos S_x įėjimuose, gausime, kad jos išėjimas yra lygus 1 (nes sutampa su $A(x, y)$ išėjimu), taigi schema S išpildoma. Ir atvirkščiai, jei S išpildoma, tai atsiras įėjimas y toks, kad $S(y) = 1$, o tai reikš, kad ir $A(x, y) = 1$. Kadangi schemos S įėjimams išskirta lygiai $\mathrm{O}(n^k)$ bitų, tai liudijimo ilgis bus polinominis $|x|$ atžvilgiu.

(b) Įrodysime, kad schema S yra polinominio dydžio ir algoritmas F yra polinominio sudėtingumo $n = |x|$ atžvilgiu. Pirmiausia įrodysime, kad kiekviena konfigūracija c_i yra polinominio dydžio $n = |x|$ atžvilgiu. Programos dydis nepriklauso nuo x ir yra konstanta, kaip ir komandų skaitikliui bei procesoriaus registrams skirta atmintis. Įėjimo x dydis yra n, o liudijimo y dydis yra $\mathrm{O}(n^k)$. Kadangi algoritmas A daro ne daugiau, kaip $\mathrm{O}(n^k)$ žingsnių, tai jam reikalingos darbinės atminties dydis taip pat yra polinominis n atžvilgiu. Taigi, kiekviena konfigūracija yra polinominio dydžio n atžvilgiu, o viena schema M taip pat yra polinominio dydžio konfigūracijos dydžio atžvilgiu. Kadangi tokių schemų M skaičius $T(n)$ taip pat yra polinominis n atžvilgiu, tai ir schema S bus polinominio dydžio n atžvilgiu. Akivaizdu, kad ir ją konstruojantis algoritmas F bus polinominio sudėtingumo. $\square$



4.5 Pav.: Kompiuterio konfigūracijų seka, atitinkanti algoritmo $A(x, y)$ darbą. Konfigūracija c_i vaizduoja kompiuterio būseną po i-ojo algoritmo A žingsnio. Pradinėje konfigūracijoje c_0 visų bitų, išskyrus liudijimo lauką y, reikšmės yra fiksuotos. Kiekvieną konfigūraciją į sekančią perdirba Būlio schema M. Jungtinės schemos išėjimas yra vienas iš darbinės atminties bitų.

4.3.3 Kiti NP-pilni uždaviniai

Nagrinėsime Būlio formules virš bazės $B = \{\&, \vee, \neg, \to, \leftrightarrow\}$. Formulė $F(x_1, \ldots, x_n)$ *išpildoma*, jei egzistuoja kintamųjų reikšmių rinkinys $(\alpha_1, \ldots, \alpha_n) \in \{0,1\}^n$, paverčiantis formulę teisinga: $F(\alpha_1, \ldots, \alpha_n) = 1$. Pavyzdžiui, Būlio formulė $F(x_1, x_2, x_3) = \neg(x_1 \to x_2)\&\neg x_3$ yra išpildoma, nes $F(1,0,0) = 1$. Pažymėkime

$$\text{SAT} = \{\langle F\rangle\text{: } F \text{ — išpildoma Būlio formulė}\}.$$

Teorema 4.3. $\text{SAT} \in \text{NPC}$.

Dabar nagrinėsime specialaus pavidalo Būlio formules virš bazės $B_0 = \{\&, \vee, \neg\}$. *Litera* vadinsime kintamąjį arba jo neiginį, t.y., $p, \neg p$ yra literos. *Elementariąja disjunkcija* vadiname skirtingų literų disjunkciją. *Normaliąja konjunkcine forma* (CNF, angl.) vadiname bet kokią skirtingų elementarių disjunkcijų konjunkciją:

$$F = D_1 \,\&\, D_2 \,\&\, \cdots \,\&\, D_m, \quad \text{kur } D_i = x_{i_1}^{\sigma_1} \vee \cdots \vee x_{i_k}^{\sigma_k}\ (x_{i_j} \neq x_{i_l}) \text{ ir } D_p \neq D_r.$$

Pagaliau normaliąją konjunkcinę formą vadinsime *3-normaliąja konjunkcine forma* (3-CNF, angl.), jei kiekviena jos elementarioji disjunkcija yra sudaryta iš lygiai 3 skirtingų literų. Pažymėkime

$$\text{3-CNF-SAT} = \{\langle F\rangle\text{: } F \text{ yra išpildoma 3-normalioji konjunkcinė forma}\}.$$

Pavyzdžiui, jei $F(p,q,r) = (p \vee q \vee r)\,\&\,(\neg p \vee \neg q \vee \neg r)\,\&\,(p \vee \neg q \vee \neg r)\,\&\,(\neg p \vee q \vee \neg r)$, tai $\langle F\rangle \in$ 3-CNF-SAT, nes $F(1,1,0) = 1$.

Teorema 4.4. $\text{3-CNF-SAT} \in \text{NPC}$.

Grafo *klika* vadiname bet kokį pilną jo pografį. Klikos dydžiu vadiname jos viršūnių skaičių. Pavyzdžiui, bet kuris grafas, turintis n viršūnių ir m briaunų turi lygiai n klikų dydžio 1 ir m klikų dydžio 2. Optimizavimo uždavinyje MAX-CLIQUE reikia duotame grafe surasti maksimalaus dydžio kliką. Jį atitinkantis egzistavimo uždavinys CLIQUE klausia, ar duotame grafe egzistuoja bent viena klika dydžio $k \in \mathbb{N}$:

$$\text{CLIQUE} = \{\langle G, k\rangle\text{: grafe } G \text{ egzistuoja klika dydžio } k\}.$$

Teorema 4.5. $\text{CLIQUE} \in \text{NPC}$.

Grafo $G = (V, E)$ *viršūniniu denginiu* vadiname jo viršūnių poaibį $V' \subseteq V$ tokį, kad kiekvienai briaunai $(u, v) \in E$ turime $u \in V'$ arba $v \in V'$ (arba abu atvejai teisingi). Taigi, viršūnės iš aibės V' "uždengia" bent vieną kiekvienos briaunos galą. Viršūninio denginio dydžiu vadiname jo viršūnių skaičių. Optimizavimo uždavinyje MIN-VERTEX-COVER reikia rasti mažiausią duotojo grafo viršūninį denginį. Jį atitinkantis egzistavimo uždavinys VERTEX-COVER klausia, ar egzistuoja duotojo grafo viršūninis denginys dydžio $k \in \mathbb{N}$:

$$\text{VERTEX-COVER} = \{\langle G, k\rangle\text{: grafas } G \text{ turi viršūninį denginį dydžio } k\}.$$

Teorema 4.6. $\text{VERTEX-COVER} \in \text{NPC}$.

4.4 Apytiksliai algoritmai