**Vilniaus Universitetas**



**Matematikos ir Informatikos Fakultetas**

**III kursas 1 grupė**

**Eimantas Bendorius**

**Užduotis 21:**

**Trumpiausi keliai Euklido erdvėje**

**Uždavinio formuluotė**

**Duota:** Neorientuotas grafas G (turintis n viršūnių ir m briaunų) d-matėje Euklido erdvėje (t.y., duotos viršūnių koordinatės) bei jo viršūnės x ir y. Grafo briaunų svoriai yra euklidiniai atstumai tarp grafo viršūnių.

**Rasti:** Trumpiausią kelią iš x į y.

Realizuoti Deikstros ir Sedgewick-Vitter alogritmus, nenauduojant duomenų struktūros Fibonacci heap. Ištirti šių algoritmų sudėtingumą ir palyginti tarpusavyje:

1. teoriškai,
2. praktiškai kaip priklauso nuo n ir m.

**Realizuotų algoritmų aprašymai**

**Deikstros algoritmas**

**Dijkstros algoritmas** arba **Deikstros algoritmas** – Edgar Dijkstra sukurtas algoritmas randantis trumpiausius kelius nuo vienos viršūnės iki kitų svoriniame grafe su neneigiamais svoriais. Pavyzdžiui, jei grafo viršūnės vaizduoja miestus ir kraštinių svoriai vaizduoja atstumą tarp tų miestų, sujungtą tiesioginiu keliu, Dijkstra algoritmas naudojamas surasti trumpiausius kelius tarp tų miestų. Algoritmas bus įgyvendintas Java programavimo kalba ir bus naudojama prioritetinė eilė. Taip pat bus ieškomas trumpiausias kelias nuo pradinės viršūnės iki pasirinktos grafe viršūnės nesvoriniame grafe.

* Yra neorentuotas grafas **G** su kraštinių svoriais ir šaltinių viršūne **S** grafe **G.**
* **V** visų viršūnių rinkinis grafe **G.**
* Kiekviena grafo kraštinė yra sutvarkyta viršūnių pora **(u,v),** parodanti ryšį tarp viršūnių u ir v.
* Visų kraštinių rinkinys yra **E.**
* Kraštinių svoriai yra užduodami pagal svorių funkciją **w:E;** čia **u(u,v)** yra ne neigiama kaina tiesiogiai einant nuo viršūnės **U** iki viršūnės **V.** Kraštinės kaina yra laikoma atstumu tarp šitų dviejų viršūnių. Kelio kaina tarp dviejų viršūnių yra kraštinių kainų suma tame kelyje.

**Kintamieji:** S - pradžios viršūnė, P – pabaigos viršūnė, F - aplankytų viršūnių sąrašas, G – grafas, E - briauna su viršūnėmis (v1,v2) , V – viršūnė, t(i) - atstumų nuo S iki kiekvienos V masyvas, prev(i) - rodyklė į ankstesnes Vi – indeksas, U - neaplankytų viršūnių sąrašas.

**Pseudo kodas:**

**FOR** i=0 to |V|-1

dist(i)=INFINITY

prev(i)=NULL

**END FOR**

**WHILE**  Q netuščia (imame viršūnę V iš Q su mažiausiu atstumu)

pridedame V į F

IF(V == S)

**BREAK;**

**FOR** V briaunai(v1,v2)

**IF** (dist(v1)+ilgis(v1,v2) < dist(v2)

dist(v2)=dist(v1)+ilgis(v1,v2)

prev(v2)=v1

[galbūt reikia atnaujinti Q]

**END IF**

**END FORE WHILE**

Efektyvumas priklauso nuo prioritetinės eilės realizacijos.

**Pavyzdys:** Tarkime turime 5 miestus ir norime sužinoti trumpiausią kelią tarp miesto A ir B. Kelius tarp miestų pavaizduosime grafu. Virš briaunų nurodyti skaičiai reiškia atstumą tarp miestų, skritulio viduje žymėsime trumpiausią atstumą nuo pradinės viršūnės. Mėlynai žymėsimę viršūnę eilėje su mažiausių atstumų nuo pradinės viršūnės. Nagrinėjama viršūnę žymėsime pilkai, išnagrinėtas žaliai. Pabrauktos briaunos reiškia nagrinėjamus kelius, bei pasiekus reikiamą miestą parodys mums trumpiausią kelią nuo pradinės viršūnės iki jo.

A

B

C

D

E

6

5

1

2

2

5

1

a)

A

B

C

D

E

6

5

2

2

5

1

1

b)

1. Pradžioje pasirenkame pradinę viršunę A taip pat atstumai tarp miestų mums nežinomi jos žymyme begalybėmis. A viršūnės atstumą pažymime 0. Visas viršūnes įdedame į proritetinę eilę.
2. Paimame iš eilės višūnę su mažiausiu atstumu t.y A. Pradedame skaičiuoti atstumus su kaimyninais miestais dist(A, D) = 1 < inf. Taigi, atnaujiname atstumą dist(A, D) := 1 šaliname iš eilės senąjį ir patalpiname naująjį. Taip pat išsaugome trumpiausią kelią t.y iš A į D. Toliau

nagrinėjame kaimynines viršūnes dist(A, B) = 6 < inf atnaujiname atstumą dist(A, B) := 6,

šaliname iš eilės senąjį ir patalpiname naująjį. Išsaugome trumpiausią kelią nuo A iki B. Kadangi, atstumas nuo pradinės viršūnės iki D maženis nei iki B pažymime, kad D sekanti nagrinėjama viršūnė.

A

B

C

D

E

6

5

1

2

2

5

1

c)

A

B

C

D

E

6

5

2

2

5

1

1

d)

1. Paimame iš eilės sekančią viršūnę D. Pradedame skaičiuoti atstumus su dar ne nagrinėtais kaimyniniais miestais dist(A, D) + W(D, B) = 3 < dist(A, B) = 6. Taigi, atnaujiname atstumą dist(A, B) := 3 šaliname iš eilės senąjį ir patalpiname naująjį. Taip pat įsimenam, kad trumpiausias kelias nuo A iki B eina per viršūnę D. Toliau nagrinėjame kaimynines viršūnes dist(A, D) + W(D, E) = 2 < dist(A, E) = inf atnaujiname atstumą dist(A, D) := 2 šaliname iš eilės senąjį ir patalpiname naująjį. Išsaugome trumpiausią kelią nuo A iki E, kuris eina per D. Kadangi, atstumas nuo pradinės viršūnės iki E maženis nei iki B pažymime, kad sekanti nagrinėjama viršūnė yra E.
2. Paimame iš eilės sekančią viršūnę E. Pradedame skaičiuoti atstumus su dar ne nagrinėtais kaimyniniais miestais dist(A, E) + W(E, B) = 4 < dist(A, B) = 3. Taigi, atstumo neatnaujiname. Toliau nagrinejame kaimyninius miestus dist(A, E) + W(E,C) = 7 < dist(A, C) = inf. Taigi, atnaujiname atstumą dist(A, C) := 7 šaliname iš eilės senąjį ir patalpiname naująjį. Kadangi, atstumas nuo pradinės viršūnės iki B maženis nei iki C pažymime, kad sekanti nagrinėjama viršūnė yra B.

1. Paimame iš eilės sekančią viršūnę B su mažiausiu atstumu. Tikriname ar miestas B nėra miestas iki kurio mes ieškojome trumpiausio atstumo nuo miesto A. Kadangi, taip ir yra baigiame paiešką. Trumpiausias kelias: A, D, B. Ilgis: 3.

A

B

C

D

E

6

5

2

2

5

1

1

e)

Algoritmo realizacija:

**public** List<Character> gautiTrumpiausiaKelia(Character pradzia, Character pabaiga) **throws** Exception {

gautiVirs(pabaiga);

PriorityQueue<Virsune> eile = **new** PriorityQueue<Virsune>();

**final** Map<Character, Double> atstumai = **new** HashMap<Character, Double>();

//aprašo trumpiausia kelia, ji reikia isvyniuoti nuo galo

**final** Map<Character, Virsune> trumpKelias = **new** HashMap<Character, Virsune>();

**final** List<Character> aplankytos = **new** ArrayList<Character>();

//i eile patalpiname virsune ir priskiriame pradinei virsune 0

//likusiom virsunem begalybe (musu atveju Double.MAX\_VALUE)

**for**(Character virsune : grafas.getVirsunes().keySet()) {

**if** (virsune == pradzia) {

atstumai.put(virsune, 0d);

eile.add(gautiVirs(pradzia));

} **else** {

atstumai.put(virsune, Double.***MAX\_VALUE***);

eile.add(**new** Virsune(virsune, Double.***MAX\_VALUE***));

}

trumpKelias.put(virsune, **null**);

}

//nagrinejame virsune kol eile nebus tuscia

**while** (!eile.isEmpty()) {

Virsune v = eile.poll(); //is eile paimame virsune su maziausiu atstumu

aplankytos.add(v.getVirsunePavadinimas());

//patikriname ar tai ne musu ieskoma pabaigos virsune, jei taip gauname kelia

**if** (v.getVirsunePavadinimas() == pabaiga) {

**final** List<Character> kelias = **new** ArrayList<Character>();

setAtstumasXY(atstumai.get(v.getVirsunePavadinimas()));

**while** (trumpKelias.get(v.getVirsunePavadinimas()) != **null**) {

kelias.add(v.getVirsunePavadinimas());

v = trumpKelias.get(v.getVirsunePavadinimas());

}

kelias.add(v.getVirsunePavadinimas());

Collections.*reverse*(kelias);

**return** kelias;

}

//pabaigiame darba, tai reiskia, kad pries tai buvusios virsunes neturejo kaimyniu

**if** (atstumai.get(v.getVirsunePavadinimas()) == Double.***MAX\_VALUE***) {

**break**;

}

//einame per visas virsunes kaimynes, kurios nebuvo isimtos is eiles ir skaiciuojame trump ats

**for** (Virsune kaimyne : grafas.getVirsunes().get(v.getVirsunePavadinimas())) {

//patikriname ar tai ne virsune, kuri jau buvo isimta is eiles

**if**(!arAplankyta(kaimyne.getVirsunePavadinimas(), aplankytos)){

**double** atstumasIkiKaimyno = v.paskaiciuotiAtstuma(kaimyne);

**double** naujasAtstumas = atstumai.get(v.getVirsunePavadinimas()) + atstumasIkiKaimyno;

**if** (naujasAtstumas < atstumai.get(kaimyne.getVirsunePavadinimas())) {

atstumai.put(kaimyne.getVirsunePavadinimas(), naujasAtstumas);

trumpKelias.put(kaimyne.getVirsunePavadinimas(), v){

//eileje ieskome kaimynes ir atnaujiname atstuma

forloop:

**for**(Virsune vv : eile) {

**if** (vv.getVirsunePavadinimas() == kaimyne.getVirsunePavadinimas()) {

eile.remove(vv);

kaimyne.setAtstumas(naujasAtstumas);

eile.add(kaimyne);

**break** forloop;

}

}

}

}

}

}

**throw** **new** Exception("Neimano pasiekti virsunes" + pabaiga);

**Sudetingumas:** O(v^2)

**Realizuotų algoritmų aprašymai**

**Sedgewick-Vitter**

**Sedgewick-Vitter** algorimtas, tai yra šiek tiek pakeistas Deikstros algoritmas. Deikstros algoritme prioritetinėje eilėje talpinami trumpiausi atstumai nuo pradinės viršūnės iki visų likusių viršūnių. Sedgewick-Vitter algoritme eilėje talpinamini atstumai nuo pradinės viršūnės iki kaimynės ir tiesioginis atstumas iki pabaigos viršūnės.

**Pavyzdys:** Turime keturis miestus A, B, C, D punktyrine linija pažymėtas skaičiuojamas tiesioginis atstumas, norime sužinoti trumpiausią kelią tarp miesto A ir D. Kelius tarp miestų pavaizduosime grafu. Virš briaunų nurodyti skaičiai reiškia atstumą tarp miestų, skritulio viduje žymėsime trumpiausią atstumą nuo pradinės viršūnės + tiesioginis atstumas iki pabaigos viršūnės. Mėlynai žymėsimę viršūnę eilėje su mažiausių atstumų nuo pradinės viršūnės iki kaimynės ir pabaigos viršūnės. Nagrinėjama viršūnę žymėsime pilkai. B(g) – atstumas iki kaimynės, h – tiesioginis atstumas iki pabaigos viršūnės.

4

A

C

B

1,1

1,9

5

D

6,1

6,9

|  |  |  |  |
| --- | --- | --- | --- |
| Viršūnės | B(g) | h | f = g + h |
| A | 1,9 | 5 | 6,9 |
| C | 1,1 | 4 | 6,1 |

Kadangi, algoritmas vykdomas panašiai kaip Deikstros pavyzdys detaliai nebus išnagrinėtas. Iš pavyzdžio matome, kad eileje su mažiausiu atstumu yra viršūnė A, todėl ją ir pasirenkame sekančia, nagrinėjant kaimynes pasiekiame tikslo viršūnę darbą pabaigiame. Ir gauname, trumpiausią kelią B, A, D atstumas 6,1.

Algoritmo realizacija:

**public** List<Character> gautiTrumpiausiaKelia2(Character pradzia, Character pabaiga) **throws** Exception {

Virsune pabVirs = gautiVirs(pabaiga);

PriorityQueue<Virsune> eile = **new** PriorityQueue<Virsune>();

**final** Map<Character, Double> atstumai = **new** HashMap<Character, Double>();

//aprašo trumpiausio kelia, ji reikia isvyniuoti nuo galo

**final** Map<Character, Virsune> trumpKelias = **new** HashMap<Character, Virsune>();

**final** List<Character> aplankytos = **new** ArrayList<Character>();

//i eile patalpiname virsune ir priskiriame pradinei virsune 0

//likusiom virsunem begalybe (musu atveju Double.MAX\_VALUE)

**for**(Character virsune : grafas.getVirsunes().keySet()) {

**if** (virsune == pradzia) {

atstumai.put(virsune, 0d);

eile.add(gautiVirs(pradzia));

} **else** {

atstumai.put(virsune, Double.***MAX\_VALUE***);

eile.add(**new** Virsune(virsune, Double.***MAX\_VALUE***));

}

trumpKelias.put(virsune, **null**);

}

//nagrinejame virsune kol eile nebus tuscia

**while** (!eile.isEmpty()) {

Virsune v = eile.poll(); //is eile paimame virsune su maziausiu numatomu atstumu

aplankytos.add(v.getVirsunePavadinimas());

//jei atejome iki reikiamos virsunes baigiame darba ir graziname kelia

**if** (v.getVirsunePavadinimas() == pabaiga) {

**final** List<Character> kelias = **new** ArrayList<Character>();

setAtstumasXY(atstumai.get(v.getVirsunePavadinimas()));

**while** (trumpKelias.get(v.getVirsunePavadinimas()) != **null**) {

kelias.add(v.getVirsunePavadinimas());

v = trumpKelias.get(v.getVirsunePavadinimas());

}

kelias.add(v.getVirsunePavadinimas());

Collections.*reverse*(kelias);

**return** kelias;

}

//pabaigiame darba, tai reiskia, kad pries tai buvusios virsunes neturejo kaimyniu

**if** (atstumai.get(v.getVirsunePavadinimas()) == Double.***MAX\_VALUE***) {

**break**;

}

//einame per visas virsunes kaimynes ir skaiciuojame trumpiausia atstuma

//jei randame atnaujiname eile ismeta su senu atstumu virsune ir idedame su nauju

**for** (Virsune kaimyne : grafas.getVirsunes().get(v.getVirsunePavadinimas())) {

//jei atejome iki reikiamos virsunes baigiame darba ir graziname kelia

**if** (kaimyne.getVirsunePavadinimas() == pabaiga) {

**final** List<Character> kelias = **new** ArrayList<Character>();

//setAtstumasXY(atstumai.get(v.getVirsunePavadinimas()));

kelias.add(kaimyne.getVirsunePavadinimas());

**while** (trumpKelias.get(v.getVirsunePavadinimas()) != **null**) {

kelias.add(v.getVirsunePavadinimas());

v = trumpKelias.get(v.getVirsunePavadinimas());

}

kelias.add(v.getVirsunePavadinimas());

Collections.*reverse*(kelias);

**return** kelias;

}

**if**(!arAplankyta(kaimyne.getVirsunePavadinimas(), aplankytos)){

//atsumas kaimynes iki pabaigos virsunes, imamas tiesus atstumas

**double** numanomasAtstumas = kaimyne.paskaiciuotiAtstuma(pabVirs);

**double** atstumasIkiKaimyno = v.paskaiciuotiAtstuma(kaimyne);

**double** naujasAtstumas = atstumai.get(v.getVirsunePavadinimas()) +

atstumasIkiKaimyno + numanomasAtstumas;

**if** (naujasAtstumas < atstumai.get(kaimyne.getVirsunePavadinimas())) {

atstumai.put(kaimyne.getVirsunePavadinimas(), naujasAtstumas);

trumpKelias.put(kaimyne.getVirsunePavadinimas(), v);

//eileje ieskome kaimynes ir atnaujiname atstuma

forloop:

**for**(Virsune vv : eile) {

**if** (vv.getVirsunePavadinimas() == kaimyne.getVirsunePavadinimas()) {

eile.remove(vv);

System.***out***.println("Naujas atstumas" + naujasAtstumas);

kaimyne.setAtstumas(naujasAtstumas);

eile.add(kaimyne);

**break** forloop;

}

}

}

}

}

}

**throw** **new** Exception("Neiimano pasiekti virsunes" + pabaiga);

}

**Sudetingumas:** O(v^2)

Algoritmų analizė

Sudėtingumo analizė buvo atliekama dvimatėje Euklido erdvėje parinkus 100 viršūnių koordinates.

Skaičiuojama nuo pradinės viršūnės x = 1 trumpiausią atstumą į visas likusias.

**Išvada:** analizuojant bandymo rezultatus, galima padaryti išvada, kad esant mažam briaunų sk. algoritmų vykdymo laikas yra gana panašus, bet didinant briaunų sk. Sedgewick-Vitter vykdymo laikas yra žymiai mažesnis. Taip yra todėl, nes Deikstros algoritmas neskaičiuoja tiesioginio atstumo iki ieškomos viršūnės.

**Programos naudojimo instrukcija**

Programa neturi grafinės vartotojo sąsajos. Dirbama su konsoliniu rėžimu.

Paleisti programą: java Main

Grafas pateikiamas faile: failas.txt

1. Be parametrų paleidus programa vykdomi abu algoritmai su atsitiktiniais duomenimias, įvedus n ir m.
2. Su parametru 1 – vykdomas Deikstros algoritmas, grafas nuskaitomas iš failo. Įvedamas x ir y viršūnės paskaičiuojamas trumpiausias kelias.
3. Su parametru 2 – vykdomas Sedgewick-Vitter algoritmas, grafas nuskaitomas iš failo. Įvedamas x ir y viršūnės paskaičiuojamas trumpiausias kelias.
4. Su parametrias 1 2 - vykdomas Sedgewick-Vitter ir Deikstros algoritmai, grafas nuskaitomas iš failo. Įvedamas x ir y viršūnės paskaičiuojamas trumpiausias kelias.

**Literatūra**:

1. J.A. McHugh, Algorithmic Graph Theory, chapter 3, pp. 1—5, 17--19 (žr. chapter3.pdf).

2. T.H. Cormen,C.E. Leiserson and R.L. Rivest, Introduction to Algorithms, 2nd edition, MIT Press,Cambridge, MA,pp. 580—586, 595—599 (žr. Cormen580-640.pdf)