Vilniaus Universitetas

Matematikos ir Informatikos Fakultetas

Infomatikos Katedra

Kompiuterių mokslas 1

Laimonas Beniušis

Algoritmų analizės projektas #19: Minimalus darbų atlikimo laikas

Turinys

Užduoties aprašymas

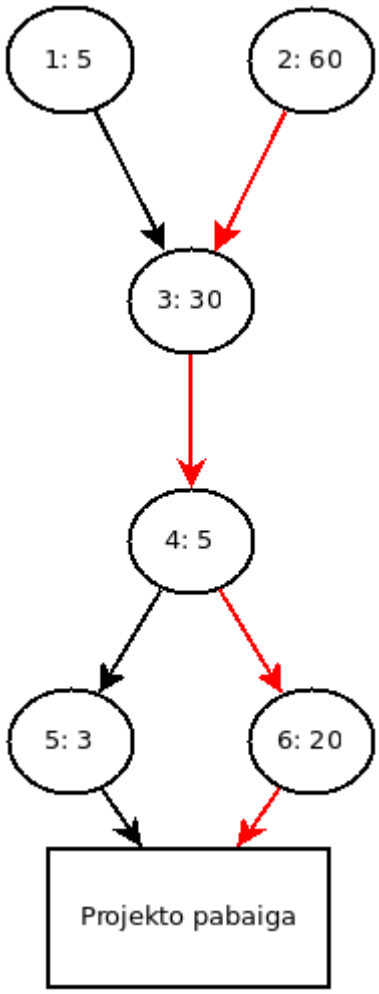
Tarkime turime projekto planą. Planas yra sudarytas iš N darbų T_i (task). Kiekvienas darbas T_i turi savo atlikimo trukmę ir priklausomų darbų sąrašą D_i (dependencies). D_i saraše yra darbai (gali būti tuščias), kurie privalo būti atlikti prieš pradedant nuo jų priklausomą darbą T_j.
$i=1..N, j=1..N, j\neq i$

Tiklas:
Rasti minimalią projekto (visų darbų) trukmę.

Prielaidos:
Nepriklausomi darbai yra atliekami vienu metu.

Pavyzdys:
Projektas “Vakarienė”

Darbai:
1. Padengti stalą
2. Pagaminti maistą
3. Suvalgyti maistą
4. Nurinkti indus
5. Nuvalyti stalą
6. Suplauti indus



Darbų trukmės (min):
1. 5
2. 60
3. 30
4. 5
5. 3
6. 20

Darbų priklausomybės:
1.
2.
3. 1,2
4. 1,2,3
5. 1,2,3,4
6. 1,2,3,4

Darbų priklausomybės (minimizuotos):
1.
2.
3. 1,2
4. 3
5. 4
6. 4

Projekto minimali trukmė yra 60 + 30 + 5 + 20 = 105 min

Užduoties formulavimas grafo pavidalu

Minimalią darbų atlikimo trukmę galima sužinoti suradus ilgiausio (kritinio) kelio ilgį.

Darbų grafas – beciklis svorinis grafas, kurio kiekvienos viršūnės lankai turi vienodą svorį

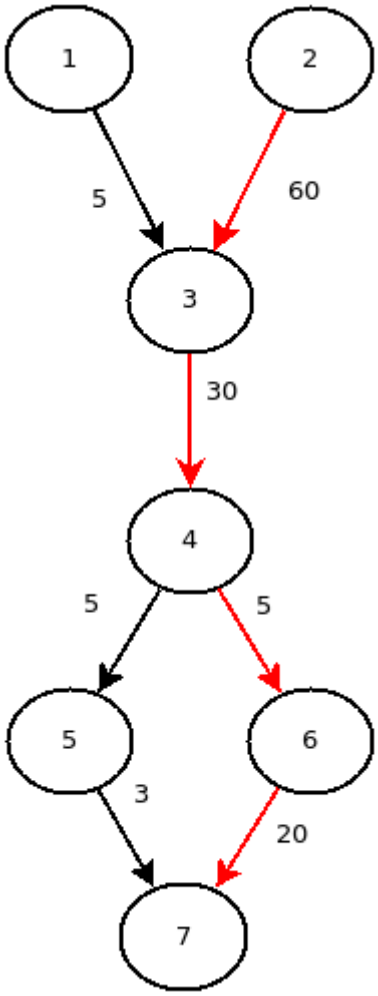
Analizuojami algoritmai:

1. Topologinis rūšiavimas ir perrinkimas (pagrinidinis)
2. Paieška “prieš srovę” (backpropagation / backflow)

Praktiškai, visi atvejai gali būti apibrėžti darbų grafo pavidalu, nes
darbas tūrėtų būti atliekamas vienodą laiką nepriklausomai nuo to, kas bus daroma po jo. Toks grafo pavidalas sutaupytų lankų svorio skaičiavimus.

Toliau yra nagrinėjamas bendriausias atvejis: orientuotas beciklis svorinis grafas.

Užduoties formulavimas orientuoto beciklio svorinio grafo G = (V, E) pavidalu:
V – viršūnių kiekis
E – lankų kiekis



Algoritmo aprašas

Topologinis rūšiavimas:

Kol grafe G yra viršūnių:
Ištraukiame iš G viršūnę, kurios įėjimo laipsnis yra 0 ir ją išsaugojame

Kelio skaičiavimas:

Turime masyvą M kuriame yra saugojamos ilgiausio kelio iki tos viršūnės vertė
Topologine tvarka pasirenkame viršūnę *v0*
Kiekvienai viršūnei *vi* , kuri yra pasiekama iš *v0*, skaičiuojame naują ilgiausią kelią:

$M[vi] = \max(M[vi],\ M[v0] + w(v0,vi))$

funkcija w(x, y) grąžina lanko x → y svorį

Didžiausia masyvo M reikšmė bus ilgiausio kelio ilgis.

Informacija apie implementaciją:

Naudota Python 3 kalba (svarbi indentacija).
Sukurtos 3 pagrindinės klasės:

- *Node*
  - ID : int
  - linksTo : set <int>
  - linkedFrom : set <int>
- *Link*
  - nodeFrom : int
  - nodeTo : int
  - weight : float
- *Orgraph*
  - nodes : Map <int, *Node*>
  - links : Map <int, *Link*>

Taip pat yra algoritmų klasė *Algorithms*, kurioje ir yra toliau aprašyti algoritmai.

Topologinis rūšiavimas

Rekursyvus algoritmas

```
def topologicalSort(graph:Orgraph) -> list:
    nodesToVisit = list()
    order = list()
    # calculate in-degree for each Node
    inDegreeMap = dict()
    for n in graph.nodes.values():
        inDegree = len(n.linkedFrom)
        inDegreeMap[n.ID] = inDegree
        if inDegree == 0:
            nodesToVisit.append(n.ID)
    def visit(nodeID:int):
        order.append(nodeID)
        node = graph.getNode(nodeID)
        for nextNodeID in node.linksTo:
            inDegreeMap[nextNodeID] -= 1
            if inDegreeMap[nextNodeID] == 0:
                visit(nextNodeID)
    for n in nodesToVisit:
        visit(n)
    return order
```

Kahn’o algoritmas

```
def topologicalSortKahn(graph:Orgraph) -> list:
    workList = deque()
    order = list()
    # calculate in-degree for each Node
    inDegreeMap = dict()
    for node in graph.nodes.values():
        inDegree = len(node.linkedFrom)
        inDegreeMap[node.ID] = inDegree
        if inDegree == 0:
            workList.append(node)
    while len(workList) > 0:
        node = workList.popleft()
        order.append(node.ID)
        for nextNodeID in node.linksTo:
            inDegreeMap[nextNodeID] -= 1
            if inDegreeMap[nextNodeID] == 0:
                workList.append(graph.getNode(nextNodeID))
    return order
```

Kada nenaudoti rekursijos:

Jeigu grafo ilgis yra didelis, nes galima lengvai viršyti rekursijos iškvietimų limitą.

Kada naudoti rekursiją:

Jeigu grafo ilgis nėra didelis (rekursijos iškvietimų ribose) ir nėra galimybės naudoti dinamiško sąrašo su $O(1)$ sudėtingumo įstumimo/išėmimo operacijomis.

Topologinio rūšiavimo algoritmo sudėtingumas: $O(E)$
Jeigu pradinių duomenų užpildymą laikome algortimo dalimi: $O(V + E)$
Atlikus topologinį rūšiavimą lieka perrinkti viršūnes pagal gautą tvarką.
Algoritmas yra pateiktas toliau.

Perrinkimas pagal topologinę tvarką

```
def criticalPath(graph:Orgraph,topologicalOrder:list) -> (dict, dict, int):
     """Must be Directed Acyclic Graph"""
     distanceMap = dict()  # node : longest distance to that node
     pathMap = dict()      # node : reached from
     negativeInf = float('-inf')
     for k in graph.nodes:
          distanceMap[k] = negativeInf
          pathMap[k] = None
     for nodeID in topologicalOrder:
          node = graph.getNode(nodeID)
          currentDist = distanceMap.get(nodeID)
          for neighbour in node.linksTo:
               if currentDist == negativeInf:
                    currentDist = 0
               neighbourDist = distanceMap.get(neighbour)
               tryNewDist = currentDist + graph.weight(nodeID, neighbour)
               if neighbourDist < tryNewDist:
                    distanceMap[neighbour] = tryNewDist
                    pathMap[neighbour] = nodeID
     startAt = Algorithms.getMapExtremum(distanceMap, maximise=True)
     return pathMap, distanceMap, startAt
```

Kaip matome, ilgiausio kelio radimas yra paskirtytas į 2 etapus, tačiau tai nėra būtina. T.y. galima daryti viršūnių perrinkimą topologinio rūšiavimo metu. Toliau yra pateiktas toks algoritmas, kuris būtent tai ir atlieka.

Paieška “prieš srovę” (backpropagation / backflow)

```
def criticalPathBackpropagation(graph: Orgraph) -> (dict, dict, int):
    """Must be Directed Acyclic Graph"""
    distanceMap = dict()  # node : longest distance from that node
    pathMap = dict()      # node : next node to go to
    outDegreeMap = dict()
    nodesToUpdate = list()
    for potentialEnd in graph.nodes.values():
        outDegree = len(potentialEnd.linksTo)
        outDegreeMap[potentialEnd.ID] = outDegree
        if outDegree == 0:
            nodesToUpdate.append(potentialEnd)
            distanceMap[potentialEnd.ID] = 0
        else:
            distanceMap[potentialEnd.ID] = float('-inf')
        pathMap[potentialEnd.ID] = None
    while len(nodesToUpdate) > 0:
        node = nodesToUpdate.pop()
        for seenBy in node.linkedFrom:
            outDegreeMap[seenBy] -= 1
            if outDegreeMap[seenBy] == 0:
                nodesToUpdate.append(graph.getNode(seenBy))
            tryNewDist = distanceMap.get(node.ID) + graph.weight(seenBy, node.ID)
            if distanceMap[seenBy] < tryNewDist:
                distanceMap[seenBy] = tryNewDist
                pathMap[seenBy] = node.ID
    startAt = Algorithms.getMapExtremum(distanceMap, maximise=True)
    return pathMap, distanceMap, startAt
```

Algoritmo sudėtingumas: $O(E)$
Jeigu pradinių duomenų užpildymą laikome algortimo dalimi: $O(V + E)$

Beciklių grafų generavimas

Semi-atsitiktinis grafas

Papraščiausiai generuojamas grafas ir jau esamas grafas gali būti lengvai plečiamas.
Algoritmas:

1. Pasirinkti 2 skirtingas viršūnes A,B
2. Jeigu A > B
    1. Padaryti lanką B → A
3. Jeigu A < B
    1. Padaryti lanką A → B
4. Jeigu tokio lanko nėra, jį įterpti
5. Kartoti kol gaunamas tenkinamas kiekis lankų

Sluoksniuotas grafas

Paprastai generuojamas, tačiau jau esamas grafas nėra lengvai plečiamas, nes reikia žinoti paskutinio sluoksnio viršūnes, norint pratęsti generaciją.

Algoritmas:

```
nodes = 1
for i = 0 to height:
        nodes += width
        for j = (nodes – width) to nodes:
                for k = nodes to (nodes + width):
                        pridėti briauną j → k
```

Atsitiktinis grafas

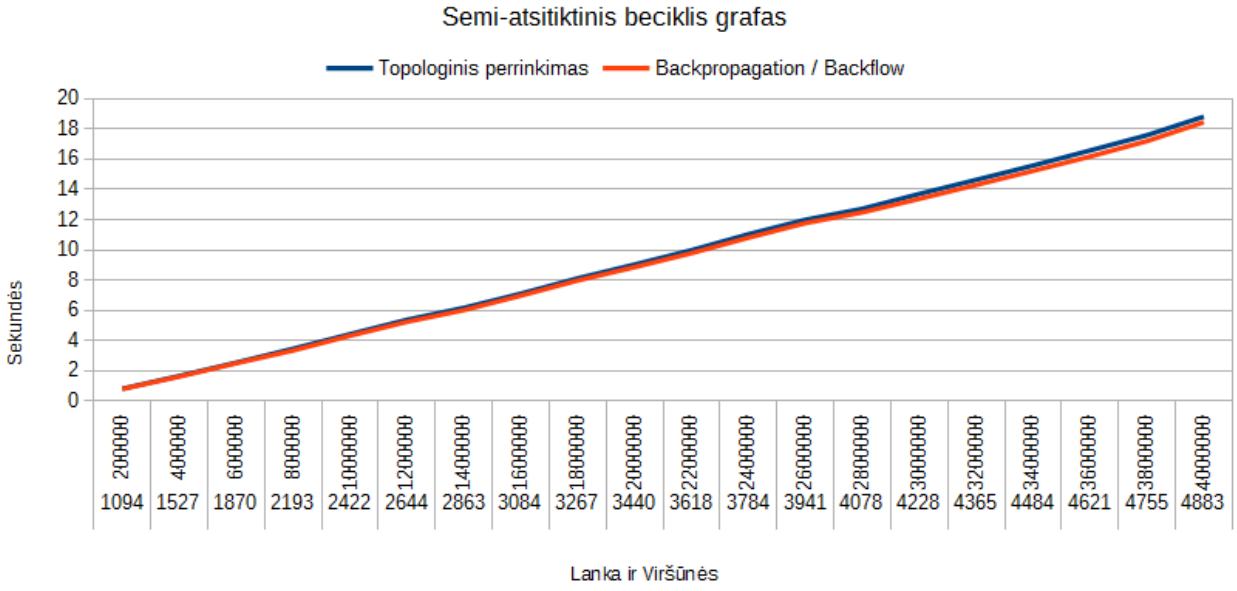
Naivus algoritmas

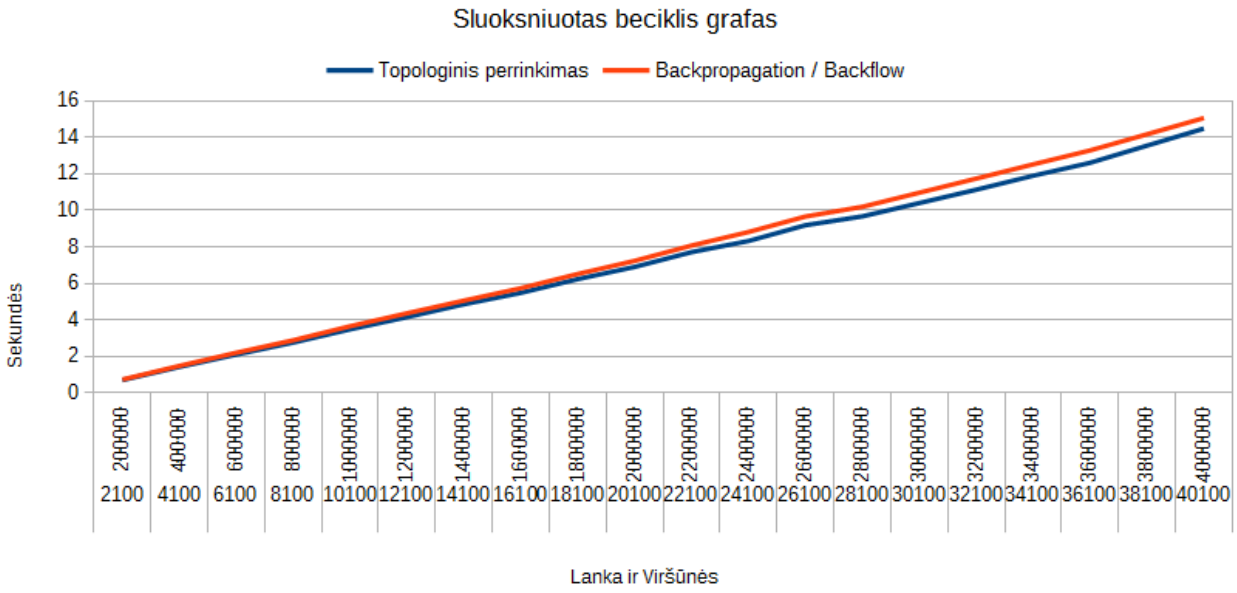
1. Pasirinkti 2 skirtingas viršūnes A,B
2. Įterpti lanką A → B
3. Jeigu susidaro ciklas, lanką išimti
4. Kartoti kol gaunamas norimas lankų kiekis

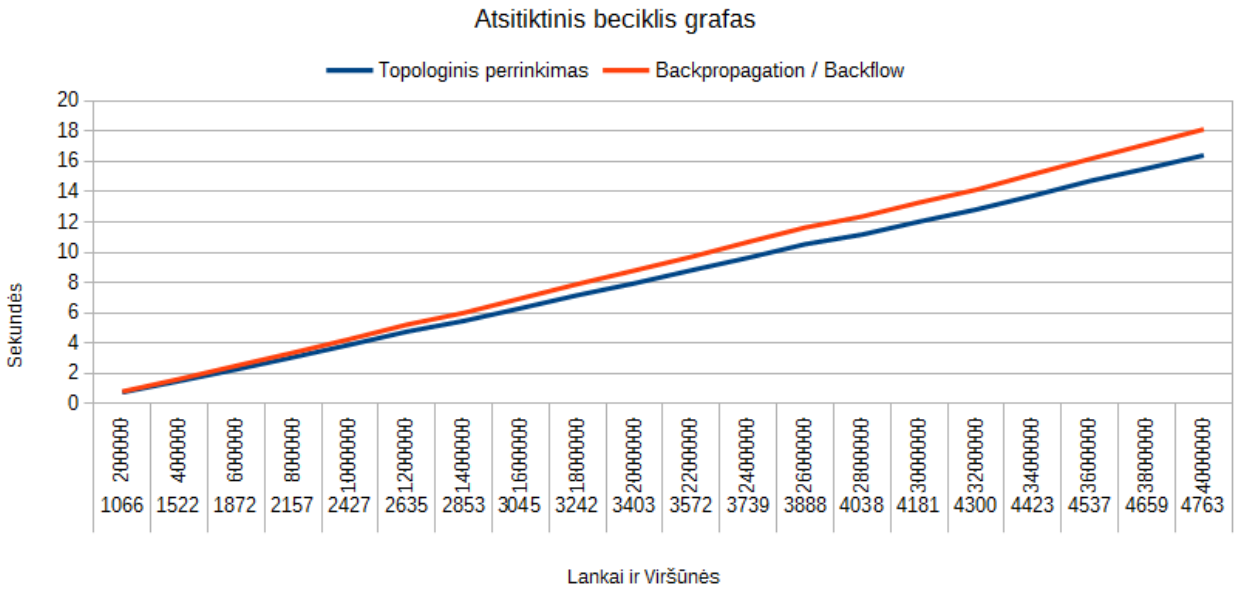
Efektyvus algoritmas

1. Sudaryti esamų viršūnių sąrašą S ir įterpti 1 naują viršūnę
2. Išimti iš S atsitiktinę viršūnę A
3. Išimti iš S visas viršūnes tiesiogiai išeinančias iš A
4. Iš A viršūnės pereiti per priešingas jungtis ir išimti visas aplankytas viršūnes iš S. (Išimti visus A viršūnės „protėvius“). Taip išimamos visos viršūnės, iš kurių egzistuoja kelias į A
5. S sąraše lieka visos viršūnės su kuriomis galima kurti lankus. $S = \{X_1, X_2, ..., X_n\}$
6. Sudarome 1 arba daugiau lankų $A \rightarrow X_i$ $i = 1..n$, kuriuos įterpiame į grafą
7. Kartoti kol gaunamas norimas lankų kiekis

Testavimas




Sluoksniuotas beciklis grafas
Topologinis perrinkimas
Backpropagation / Backflow
Sekundės
0
2
4
6
8
10
12
14
16
2000000 2100
4000000 4100
6000000 6100
8000000 8100
10000000 10100
12000000 12100
14000000 14100
16000000 16100
18000000 18100
20000000 20100
22000000 22100
24000000 24100
26000000 26100
28000000 28100
30000000 30100
32000000 32100
34000000 34100
36000000 36100
38000000 38100
40000000 40100
Lanka ir Viršūnės


Atsitiktinis beciklis grafas
Topologinis perrinkimas
Backpropagation / Backflow
Sekundės
0
2
4
6
8
10
12
14
16
18
20
2000000 1066
4000000 1522
6000000 1872
8000000 2157
10000000 2427
12000000 2635
14000000 2853
16000000 3045
18000000 3242
20000000 3403
22000000 3572
24000000 3739
26000000 3888
28000000 4038
30000000 4181
32000000 4300
34000000 4423
36000000 4537
38000000 4659
40000000 4763
Lankai ir Viršūnės

Išvados

- Sugaištas laikas nuo grafo dydžio auga tiesiškai.
- Savo eigoje “Backpropagation / Backflow“ daro panašų darbą kaip ir topologinis rūšiavimas t. y. mažina viršūnių laipsnį. Taigi, nebūtina ilgiausio kelio paiešką atskirti į 2 etapus.
- Pagal empirinius tyrimus, algoritmai veikia tokiu pat greičiu (skaičiavimo paklaidos ribose).

Literatūra

Dr. Larry Bowen, Scheduling Algorithms
http://www.ctl.ua.edu/math103/scheduling/scheduling_algorithms.htm

James A. Mc Hugh, Algorithmic Graph Theory, Chapter 4,
New Jersey Institute of Technology
http://www.mathe2.uni-bayreuth.de/axel/papers/mchugh:algorithmic_graph_theory.pdf

Programos naudojimas

Kadangi yra sukurta orientuoto grafo klasė ir statinė algoritmų klasė, žinant minimaliai Python kalbos sintaksės, galima sukurti nedidelę tekstinę bylą, kuri naudojasi sukurtomis klasėmis ir gauti norimą rezultatą.

Pvz:

```
from Graphs import *
#grafo sudarymas
gr = DependencyDAG()
gr.addLink(Link(0,1,5))
gr.addLink(Link(1,2,5))
gr.addLink(Link(1,3,5))
gr.addLink(Link(3,2,9))
gr.addLink(Link(0,4,5))
gr.addLink(Link(4,1,1))
gr.addLink(Link(4,5,1))
gr.addLink(Link(5,3,1))
gr.addLink(Link(6,1,7))
gr.addLink(Link(3,7,9))
# topologinis rūšiavimas
order = Algorithms.topologicalSortKahn(gr)
res = Algorithms.criticalPath(gr, order)
size = res[1][res[2]]
path = Algorithms.iterateMapKeys(res[0], res[2], None)
path.reverse()
print(size)
print(path)
# backpropagation
res = Algorithms.criticalPathBackpropagation(gr)
size = res[1][res[2]]
path = Algorithms.iterateMapKeys(res[0], res[2], None)
print(size)
print(path)
```

Taip pat galima suvesti tekstinę bylą, kurios kiekviena eilutė yra tokios formos:
int int float
int – sveikas skačius
float – realusis skaičius

Pirmi 2 skaičiai reiškia lanką (iš pirmo į antra), o paskutinis skaičius reiškia jo svorį.

Paleisti programą reikia turėti Python 3. Jeigu turite Python interpretatorių sisteminiame kelyje:

python main.py ”kelias iki tekstinės bylos”

Priešingu atveju:

”kelias iki Python 3 iterpretatoriaus” main.py ”kelias iki tekstinės bylos”

Programa išveda ilgiausią kelią ir jo svorį.