UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Finančna matematika – 1. stopnja

Eva Ozebek
Jan Kolenc

Minimum vertex cover

Projekt v povezavi z OR

Ljubljana, 2019

Kazalo

1. Navodilo

Define the Minimum vertex cover problem as an ILP and solve it for some examples. Also, solve the LP relaxation of this problem for the same cases. Note that its LP relaxation gives a solution that is at most twice bigger than the optimal one. Compare the sizes of both solutions on various graphs to verify this and determine experimentally by how much, in average, the LP relaxation solution is larger than the optimal one. Finally, present and implement a greedy algorithm and the one using the maximal matching described in the book below. Test the sizes of these three solutions. Try to determine for how large graphs each of these algorithms is tractable.

2. Uvod

Najina naloga je, da problem najmanjšega vozliščnega pokritja predstaviva kot problem ILP, LP relaksacije ter požrešnega algoritma. Pri tem bova algoritme preverila za 1000 različnih primerov grafov, ki imajo naključno med 5 in 100 vozlišč in naključnih povezav. Primerjala bova rezultate različnih algoritmov, kjer naju bo še posebej zanimala velikost rešitve oz. v najinem primeru moč vrnjene množice. Najine algoritme bova testirala na podatkih, ki jih bova generirala sama.

Problem *najmanjšega vozliščnega pokritja* je eden izmed osnovnih problemov pokrivanja. Kot pove že ime samo, je problem definiran s pomočjo teorije grafov. Gre za pokrivanje povezav grafa z vozlišči, t.j. za iskanje najmanjše podmnožice vozlišč grafa, ki vsebuje vsaj eno krajišče vsake povezave grafa.

Problem najmanjšega vozliščnega pokritja, ki je eden izmed zahtevnejših optimizacijskih problemov iz teorije grafov, spada v kategorijo NP-težkih problemov. Verjetnost, da je mogoče optimalno rešitev zanj najti v polinomskem času, je zelo majhna, obstaja pa več načinov za iskanje približka optimalne rešitve.

Pri programiranju in pisanju algoritma sva se odločila za program Sage, saj je le-ta za najino nalogo najbolj primeren.

3.3 Minimum Vertex Cover

The Internet had been expanding rapidly in the Free Republic of West Mordor, and the government issued a regulation, purely in the interest of improved security of the citizens, that every data link connecting two computers must be equipped with a special device for gathering statistical data about the traffic. An operator of a part of the network has to attach the government's monitoring boxes to some of his computers so that each link has a monitored computer on at least one end. Which computers should get boxes so that the total price is minimum? Let us assume that there is a flat rate per box.

It is again convenient to use graph-theoretic terminology. The computers in the network are vertices and the links are edges. So we have a graph $G = (V, E)$ and we want to find a subset $S \subseteq V$ of the vertices such that each edge has at least one end-vertex in S (such an S is called a **vertex cover**), and S is as small as possible.

This problem can be written as an integer program:

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for every edge } \{u, v\} \in E \\ & x_v \in \{0, 1\} & \text{for all } v \in V. \end{array} \tag{3.2}$$

For every vertex v we have a variable x_v, which can attain values 0 or 1. The meaning of $x_v = 1$ is $v \in S$, and $x_v = 0$ means $v \notin S$. The constraint $x_u + x_v \geq 1$ guarantees that the edge $\{u, v\}$ has at least one vertex in S. The objective function is the size of S.

It is known that finding a minimum vertex cover is a computationally difficult (NP-hard) problem. We will describe an approximation algorithm based on linear programming that always finds a vertex cover with at most twice as many vertices as in the smallest possible vertex cover.

An LP relaxation of the above integer program is

$$\begin{array}{lll} \text{minimize} & \sum_{v \in V} x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for every edge } \{u, v\} \in E \\ & 0 \leq x_v \leq 1 & \text{for all } v \in V. \end{array} \tag{3.3}$$

The first step of the approximation algorithm for vertex cover consists in computing an optimal solution $\mathbf{x}^*$ of this LP relaxation (by some standard algorithm for linear programming). The components of $\mathbf{x}^*$ are real numbers in the interval $[0,1]$. In the second step we define the set

$$S_{\mathrm{LP}} = \{v \in V : x_v^* \geq \tfrac{1}{2}\}.$$

This is a vertex cover, since for every edge $\{u,v\}$ we have $x_u^* + x_v^* \geq 1$, and so $x_u^* \geq \frac{1}{2}$ or $x_v^* \geq \frac{1}{2}$.

Let S_{OPT} be some vertex cover of the minimum possible size (we don't have it but we can theorize about it). We claim that

$$|S_{\mathrm{LP}}| \leq 2 \cdot |S_{\mathrm{OPT}}|.$$

To see this, let $\tilde{\mathbf{x}}$ be an optimal solution of the integer program (3.2), which corresponds to the set S_{OPT}, i.e., $\tilde{x}_v = 1$ for $v \in S_{\mathrm{OPT}}$ and $\tilde{x}_v = 0$ otherwise. This $\tilde{\mathbf{x}}$ is definitely a feasible solution of the LP relaxation (3.3), and so it cannot have a smaller value of the objective function than an optimal solution $\mathbf{x}^*$ of (3.3):

$$\sum_{v \in V} x_v^* \leq \sum_{v \in V} \tilde{x}_v.$$

On the other hand, $|S_{\mathrm{LP}}| = \sum_{v \in S_{\mathrm{LP}}} 1 \leq \sum_{v \in V} 2x_v^*$, since $x_v^* \geq \frac{1}{2}$ for each $v \in S_{\mathrm{LP}}$. Therefore

$$|S_{\mathrm{LP}}| \leq 2 \cdot \sum_{v \in V} x_v^* \leq 2 \cdot \sum_{v \in V} \tilde{x}_v = 2 \cdot |S_{\mathrm{OPT}}|.$$

This proof illustrates an important aspect of approximation algorithms: In order to assess the quality of the computed solution, we always need a bound on the quality of the optimal solution, although we don't know it. The LP relaxation provides such a bound, which can sometimes be useful, as in the example of this section. In other problems it may be useless, though, as we will see in the next section.

Remarks. A natural attempt at an approximate solution of the considered problem is again a *greedy algorithm*: Select vertices one by one and always take a vertex that covers the maximum possible number of yet uncovered edges. Although this algorithm may not be bad in most cases, examples can be constructed in which it yields a solution at least ten times worse, say, than an optimal solution (and 10 can be replaced by any other constant). Discovering such a construction is a lovely exercise.

There is another, combinatorial, approximation algorithm for the minimum vertex cover: First we find a maximal matching M, that is, a matching that cannot be extended by adding any other edge (we note that such a matching need not have the maximum possible *number* of edges). Then we use the vertices covered by M as a vertex cover. This always gives a vertex cover at most twice as big as the optimum, similar to the algorithm explained above.

The algorithm based on linear programming has the advantage of being easy to generalize for a **weighted vertex cover** (the government boxes may have different prices for different computers). In the same way as we did for unit prices one can show that the cost of the computed solution is never larger than twice the optimum cost. As in the unweighted case, this result can also be achieved with combinatorial algorithms, but these are more difficult to understand than the linear programming approach.

3.1. Generiranje grafov.

Najprej sva definirala funkcijo testiranje, ki vzame argumente a (število različnih velikosti grafov), b (število grafov iste velikosti), c (največje možno število vozlišč), kar bomo uporabili pri generiranju grafov. Funkcija si na začetku shrani tudi 3 števce in sicer: koliko grafov smo že pregledali, seštevek koeficientov razlik - razlika velikosti rešitev algoritmov ILP in LP ter razlika velikosti rešitev algoritmov greedy in max matching.

```
def testiranje (a,b,c): #a-število različnih velikosti grafov,b-število grafov iste velikosti,c-na
jvečje možno število vozlišč
    count = 0 #Števec pregledanih grafov
    koeficient_razlike=0 #Koeficient ki meri za koliko je v povprečju rešitev ILP večja od rešitve
LP
    koeficient_razlike_greedymax=0 #Koeficient ki meri za koliko je v povprečju rešitev "greedy" m
anjša od rešitve max matching
```

Ko izžrebamo število vozlišč moramo še 10x žrebati število povezav. Generirala sva naključno število vozlišč in povezav, kjer sva število povezav morala omejiti (da ne presegajo števila povezav polnega grafa na n vozliščih (*st_vozlisc*)), od tu izraz *st_vozlisc*(st_vozlisc - 1)/2*. Ko se vozlišča in povezave naključno izberejo, na vsakem izmed 1000 korakov dobimo graf na katerem uporabljamo algoritme.

```
for i in range(1,a+1): #Generiranje nakljičnega števila vozlišč in povezav
    try:
        st_vozlisc = randint(5,c)
    except: pass
    try:
        st_povezav = randint(5,(st_vozlisc*(st_vozlisc-1)/2))
    except: pass

    for j in range(1,b+1):
        graf = graphs.RandomGNM(st_vozlisc, st_povezav) #Generiranje grafa za izžrebano število vozlišč in povezav
        graf_copy = graf #Kopija grafa za nadaljno uporabo
```

3.2. **Požrešni algoritem in algoritem maximum matching.**

Definirala sva požrešni algoritem, ki sledi reševanju problemov lokalno optimalne izbire na vsaki stopnji, z namenom poiskati globalni optimum. Le-ta vsakič vzame vozlišče z največ povezavami in ta vozlišča shranjuje v seznam.

Torej, ko imamo vozlišče z največ povezavami, te povezave odstranimo in to ponavljamo tako dolgo, dokler v grafu ni več nobenih povezav. Ta izbrana vozlišča so vertex cover. Kot lahko opazimo je naša rešitev seznam, dolžina seznama pa predstavlja velikost rešitve.

Hkrati sva uporabila implementiran algoritem za maximum matching, ki nam vrne število povezav, zato na koncu rešitev pomnožimo z 2, saj nas zanima število vozlišč.

```
def greedy_vs_maxmatch (graf):
    sez = [] #Seznam kamor shranjujemo vozlišča z največ povezavami
    maxmatch = graf.matching(value_only=true) #Vgrajena funkcija ki išče max matching
    while graf.size() != 0: #Pregledujem dokler graf nima več povezav
        u = graf.degree().index(max(graf.degree())) #U je vozlišče z največ povezavami
        sez.append(u) #Na vsakem koraku dodam U v zgornji seznam
        for i in graf.edges(labels=False,sort=True): #S to zanko uničim vse povezave iz vozlišča u
na tem koraku
            if u == i[0] or u == i[1]:
                graf.delete_edge(i)
    return [len(sez), maxmatch*2]
```

3.3. **Celoštevilski linearni program.**

Ko imamo generiran graf, na njem rešimo ILP in LP ter uporabimo funkcijo *greedy_vs_maxmatch.*

ILP (*integer linear program*) uporablja samo cela števila na zaprtem intervalu od 0 do 1, zato nastavimo *binary = True.* Z *set_objective* določimo kaj želimo maksimizirati pri linearnih programih, v najinem primeru je to seštevek vrednosti prirejenih vozliščem.

Nato omejitve (*add_constraints*) nastavimo tako, da je vsaka povezava v grafu šteta vsaj enkrat. Nato pa na vozlišča zapišemo vrednosti, ki jih jim je priredil ILP. Končno rešitev - torej vozlišča vsebovana v pokritju in velikost rešitve, bova prikazala kasneje.

```
            #ILP===================================================================================
===============
            ILP = MixedIntegerLinearProgram(maximization = False) #Definiram ILP za minimum
            vozlisca = ILP.new_variable(binary = True) #Vozliscem priredi 0 ali pa 1
            ILP.set_objective( sum([vozlisca[v] for v in graf]) ) #Zapisem da minimizira vsoto ste
vil prirejenih vozliscem

            for u,v in graf.edges(labels = False):
                ILP.add_constraint( vozlisca[u] + vozlisca[v] >= 1 )#Pogoj

            ILP.solve()
            vozlisca = ILP.get_values(vozlisca)
```

3.4. **Relaksacija celoštevilskega na linearni program.**

Tukaj se ne ukvarjamo več s celimi števili, zato nastavimo *real* = *True*. Realna števila sva omejila na interval od 0 od 1, nastavila sva iste pogoje in enak objective.

Ker uporabimo pogoj da mora biti vsota vozlišč večja ali enaka ena, zato vemo da mora biti posamezno vozlišče večje ali enako eni polovici.

Tudi tukaj bova rezultate in ugotovitve prikazala v naslednjem razdelku.

```
            #Relaxacija na LP=================================================================
================
            LP = MixedIntegerLinearProgram(maximization = False)#Definiram LP za minimum
            vozlisca_LP = LP.new_variable(real = True)#Vozliscu priredim realno število med 0 in 1
kar sta spodnji omejitvi
            LP.set_max(vozlisca_LP,1)
            LP.set_min(vozlisca_LP,0)
            LP.set_objective( sum([vozlisca_LP[v] for v in graf.vertices()]) )

            for u,v in graf.edges(labels = False):
                LP.add_constraint( vozlisca_LP[u] + vozlisca_LP[v] >= 1 ) #Pogoj


            LP.solve()
            vozlisca_LP = LP.get_values(vozlisca_LP)
```

3.5. **Rezultati funkcij.**

Na tej točki upoštevama funkcijo *greedy_vs_maxmatch*, ki sva jo definirala zgoraj. Print vrne velikosti rešitev (greedy in maximum matching) hkrati, ter velikosti rešitev LP in ILP.

Potem primerjamo velikosti rešitev greedy in maximum matching, da vidimo za koliko krat je rešitev greedy manjša od velikosti rešitve maximum matching ter koeficient prištejemo k števcu za ta dva algoritma. Postopek ponovimo še za ILP in LP.

Na koncu pa nama funkcija vrne povprečne koeficiente za oba para algoritmov.

```
            greed = greedy_vs_maxmatch(graf_copy)
            #Analiza==============================================================================
=================
            koeficient_razlike += (RR(len([v for v,i in vozlisca_LP.items() if i>=1/2]))/RR(len([v
for v,i in vozlisca.items() if i])))
            koeficient_razlike_greedymax += RR(greed[0])/RR(greed[1])
            print[len([v for v,i in vozlisca.items() if i]),len([v for v,i in vozlisca_LP.items()
if i>=1/2])] #Prvi element je št vozlišč ki jih pridela ILP drugi pa LP
            count += 1
            print(greed) #Prvi element je št vozlišč ki jih pridela Greedy drugi pa Max matchng
            print('Do konca je še {} grafov'.format(((a*b) - count)))
    return[koeficient_razlike/((a)*(b)),koeficient_razlike_greedymax/((a)*(b))]
```

4. Zaključek

V najini analizi sva se, kot omenjeno že prej, osredotočila na 1000 primerov, ki sva jih generirala sama. Najprej sva definiranim grafom naključno izbrala med 5 in 100 vozlišč, nato pa dodala še naključne povezave.

Tekom analize sva uporabljala sledeče algoritme:

- ILP
- LP
- Požrešni algoritem
- Maximal matching

```
testiranje(100,10,100)
```

```
[92, 97]
[93, 96]
Do konca je še 999 grafov
[92, 97]
[93, 96]
```

```
Do konca je še 1 grafov
[21, 22]
[21, 22]
Do konca je še 0 grafov
Out[3]: [1.09907523261151, 0.930618743581213]
```

Kot vidimo zgoraj, sva opravila testiranje, kjer sva ugotovila, da je velikost rešitve LP v povprečju za 9,908% večja od velikosti rešitve ILP. Prav tako pa sva ugotovila tudi, da je velikost rešitve maximum matching za 7,458% večja od velikosti rešitve greedy.

```
testiranje(1,1,300)
graf_copy.degree()
graf.degree()
```

```
CPU times: user 9min 12s, sys: 27 s, total: 9min 39s
Wall time: 9min 41s
CPU times: user 297 ms, sys: 0 ns, total: 297 ms
Wall time: 294 ms
CPU times: user 9.44 s, sys: 172 ms, total: 9.61 s
Wall time: 9.6 s
CPU times: user 11 s, sys: 265 ms, total: 11.3 s
Wall time: 11.3 s
[215]
[223]
[218, 222]
Do konca je še 0 grafov

[1.03720930232558, 0.981981981981982]
```

Zanimalo naju je, koliko časa za velike grafe porabijo zgoraj opisani algoritmi. Za grafe do velikosti 50 vozlišč so bile razlike le v sekundah, kot pa vidimo na zgornjem primeru, kjer je bilo več kot 220 vozlišč, pa sta se časa reševanja ILP in LP razlikovala že za skoraj 10 minut.

Pri algoritmih greedy in maximum matching ni bilo velikih razlik. Greedy je porabil 1,7 sekunde, maximum matching pa 9,6 sekunde.

Kljub temu, da ILP problem rešuje največ časa, nam da najbolj točno rešitev, torej se lahko na podlagi velikosti grafa in zahtevane natančnosti odločimo za uporabo najbolj ustreznega algoritma za dan primer.

Za grafe, ki imajo do 200 vozlišč bi priporočala ILP, za večje grafe pa gotovo uporabo LP algoritma.

Literatura

[1] J. Matoušek in B. Gartner, *Understanding and Using Linear Programming*

[2] J. Mihelič, *Algoritem za problem najmanjšega vozliščnega pokritja*, [ogled 18. 11. 2019], dostopno na https://www.researchgate.net/publication/308929074_Algoritmi_za_problem_najmanjsega_vozliscnega_pokritja?fbclid=IwAR0n6CzQluY3ZFM6g2-yQ_fPLnUiT8zrb78S_o3VUwjmf7F-_DNQzxHVqb0