Fakulteta za matematiko in fiziko

Operacijske Raziskave

Najcenejše popolno prirejanje v polnem dvodelnem grafu

Avtorji:
Dejan Gašparič
Teja Rupnik
Marko Jordan

Mentorja:
prof. dr. Sergio Cabello
asist. dr. Janoš Vidali

22. februar 2017

1 Ideja projekta ter opis postopka

Ideja našega projekta je poiskati najcenejše popolno prirejanje v uteženem polnem dvodelnem grafu. Polni dvodelni graf $G = (V, E)$ je graf, za katerega velja, da obstajata X in Y taka, da je $V = X \cup Y$, kjer sta X in Y disjunktna, hkrati pa za vsak par vozlišč iz X in Y obstaja povezava med njima ($E \subseteq X \times Y$).

Najcenejše popolno prirejanje v uteženem polnem dvodelnem grafu $G = (V, E)$ je množica povezav $M \subseteq E$, kjer povezave iz M nimajo skupnih krajišč v grafu G, sočasno je tudi vsako vozlišče krajišče neke povezave v množici M, vsota uteži pa je minimalna.

Za reševanje problema iskanja najcenejšega popolnega prirejanja v uteženem polnem dvodelnem grafu bomo uporabili dva pristopa. Prvi pristop bo z uporabo **madžarske metode** (angl. *Hungarian method*), kot drugi pristop pa bomo reševali **celoštevilski linearni program** (angl. *integer linear programming*, v nadaljevanju ILP).

2 Madžarska metoda

2.1 Opis

Vhod: nenegativna matrika uteži nxn, ki predstavlja uteži med posameznimi vozlišči. Element matrike iz i-te vrstice in j-tega stoplca predstavlja utež povezave med x_i in y_i, kjer velja $x_i \in X, y_j \in Y$.

Izhod: najcenejše popolno uteženo prirejanje grafa, ki ga predstavlja zgornja matrika, skupaj z utežmi povezav in vrednostjo prirejanja.

Algoritem, ki smo ga naredili, sicer v času $O(n^3)$ vrne najcenejše popolno uteženo prirejanje v polnem dvodelnem uteženem grafu, toda bistvena v njem je metoda, ki deluje za največje popolno uteženo prirejanje v polnem dvodelnem uteženem grafu. Ker iščemo najcenejše uteženo prirejanje, metoda pa dela za največje, delamo z negativno matriko originalne matrike, vrednost najcenejšega prirejanja pa je tako na koncu enaka nasprotni vrednosti največjega prirejanja (največje prirejanje je namreč v negirani matriki isto kot najcenejše prirejanje v originalni). Tehnično gledano sama metoda na koncu vrne zgolj povezave prirejanja, tako da je vrednosti in uteži (kar je skupaj s povezavami izhod celotnega algoritma) potrebno računati oz. iskati posebej z običajno for zanko.

Metoda v osnovi deluje tako: začnemo s prirejanjem M, ki je prazno, in grafom, za katerega povezave veljajo kasneje omenjene določene lastnosti.

Potem v vsakem koraku povečamo (v splošnem lahko le spremenimo, ne nujno povečamo) prirejanje M za eno povezavo, dokler prirejanje ni popolno. Natančneje, v vsakem koraku gradimo drevo, ki ga v začetku sestavlja prosto vozlišče $u \in X$, v splošnem, vendar ne vedno, nekaj povezav iz prirejanja (lahko tudi vse) in ustrezne povezave med vozlišči, da imamo še vedno drevo. Na koncu postopka gradnje drevesa pa dobimo še prosto vozlišče $y \in Y$ in ustrezno povezavo. Takrat lahko v drevesu najdemo povečujočo pot $u - y$ in tedaj povečamo M. Tekom gradnje drevesa se lahko zgodi, da moramo spremeniti zgoraj omenjeni graf, ki ga sicer ne rišemo, da dobimo nove povezave.

Natančnejši opis:
Definiramo $l : V \to R$, da za vsak par x, y velja : $l(x) + l(y) \geqslant (x, y)$**(*)**. Tedaj rečemo, da je l dopusten. Sedaj si oglejmo graf, ki ga sestavljajo povezave, za katere v zgornji neenakosti velja enakost.
Velja: če najdemo v tem grafu popolno prirejanje M, potem je to prirejanje tudi največje uteženo prirejanje originalnega grafa. Res, naj bo M' neko popolno prirejanje originalnega grafa in naj $w(M')$ predstavlja vrednost prirejanja M'.
Potem velja:
$w(M') = \sum\limits_{x,y \in M'} w(x, y) \leqslant \sum\limits_{x,y \in M'} (l(x) + l(y)) = \sum\limits_{v \in V} l(v) = w(M)$ **(**)**
Torej je M optimalna rešitev.

Naj bo: $\forall x \in X$, $l(x) = \max_{y \in Y} \{w(x, y)\}$ in $l(y) = 0, \forall y \in Y$**(***)**
Potem očitno velja $(*)$.
Definirajmo še $N_l(x)$ kot sosede elementa x v grafu, ki ga določajo enakosti v $(*)$, in $N_l(S) = \bigcup\limits_{x \in S} N_l(x)$, $S \subseteq X$.
Z E_l pa označimo graf, v katerem so zgolj povezave, za katere vozlišča v $(*)$ velja enakost.

2.2 Psevdokoda metode

1. Definirajmo prazno prirejanje M in nek dopusten l.
2. While M ni popoln:
   - Najdi prosto vozliščee $u \in X$.
   - Naj bo $S = [u]$, $T = [\ ]$
3. If $N_l(S) = T$:
   - $a_l = \min_{s \in S, y \notin T} \{l(x) + l(y) - w(x, y)\}$

- $l'(v) = \begin{cases} l(v) - a_l & \text{, if } v \in S \\ l(v) + a_l & \text{, if } v \in T \\ l(v) & \text{, sicer} \end{cases}$

4. if $N_l(S) \neq T$: izberi $y \in N_l(S) - T$

   4.1. Če y ni v prirejanju M, je $u - y$ povečujoča pot. Povečaj M in pojdi v 2.

   4.2. Če je y v prirejanju, torej obstaja nek z iz X, da je povezava $z - y$ v prirejanju, povečaj drevo in dodaj z v S in y v T. Pojdi v 3.

Očitna lastnost 3. faze je še:

- Če je $(x, y) \in E_l$ za $x \in S$, $y \in T$, potem je $(x, y) \in E_{l'}$
- Če je $(x, y) \in E_l$ za $x \notin S$, $y \notin T$, potem je $(x, y) \in E_{l'}$
- Obstaja tudi $(x, y) \in E_{l'}$ za neke $x \in S$, $y \notin T$

2.3 Komentar psevdokode

V 1. fazi definirajmo l kot v $(***)$.

2. faza je glavni del algoritma. Če M ni popoln, mora obstajati prosto vozlišče u iz X. Dodamo ga v množico S, ki bo naša množica vozlišč iz X, ki smo jih že obiskali. T naj bo prazna množica, zanjo pa sicer velja, da bo naša množica vozlišč iz Y, ki smo jih že obiskali.

Preskočimo zaenkrat 3. fazo in si oglejmo 4. fazo.

V 4. fazi najprej najdemo nek y, ki je povezan z nekim elementom iz S v grafu E_l, pa ga še nismo obiskali, torej ni v T. Potem se vprašamo, če je ta y morda že v prirejanju. Če je (faza 4.2.), dodamo y v T, kajti y smo ravnokar obiskali, element z iz X, s katerim je bil y povezan v prirejanju M, pa dodamo v S. Ker y ni bil prost in nismo mogli najti povečujoče poti, postopek ponovimo, zato gremo v fazo 3.

Če pa je y prost (faza 4.1.), imamo povečujočo pot $u - y$ v drevesu. Povečamo M in gremo v fazo 2.

Sedaj si oglejmo še fazo 3: če je množica sosedov $N_l(S)$ enaka T, v fazi 4. ne bomo mogli najti novega vozlišča. Zato moramo spremeniti graf, da dobimo nove povezave in posledično nova vozlišča v $N_l(S)$, ki jih nato lahko izberemo v naslednji fazi.

Česar pa psevdokoda ne opisuje, je natančen opis gradnje drevesa. Gradimo tako: v 2. fazi najprej dodamo vozlišče u. Ko v 4. fazi dodamo vozlišče y, vedno dodamo v drevo povezavo, preko katere smo iz S prišli do y. Izberemo eno povezavo (lahko bi jih bilo več), da imamo spet drevo. Če je y prost, v grafu obstaja povečujoča pot $u - y$, na podlagi katere povečamo M. Če pa y ni prost, pa je potrebno v graf dodati še povezavo $z - y$, kjer je $z - y$ povezava v prirejanju M.

Omenimo še, da ko spreminjamo l v 3. fazi, nato vedno delamo s tem istim l, dokler ga ponovno ne spremenimo. Ob vsaki spremembi l in torej ob vsaki spremembi grafa pa vedno velja, da se vse povezave v drevesu ohranijo, saj so tudi v spremenjenem grafu. Ravno tako se ohranijo povezave prirejanja. To je ključno, saj to pomeni, da se vsaka sprememba prirejanja, ki se lahko zgodi le glede na trenutni l, še vedno nanaša na graf, za katerega veljajo enakosti v $(*)$, torej še vedno velja $(**)$.

Za vsak y, ki ni v T, definirajmo še: $slack_y = \min_{x \in S} \left\{ l(x) + l(y) - w(x, y) \right\}$

$Slack_y$ določimo oz. spremenimo v fazi 4.2. Zanj porabimo $O(n)$ časa. V 3. fazi pa nato zgolj poiščemo najmanjše število in pripadajoče y, kar vzame spet $O(n)$ časa. V 3. fazi torej ne iščemo celotnega minimuma povsem od začetka (to je namreč $O(n^2)$).

Celotna zahtevnost metode: 2. faza se izvede n-krat (izbrati prosto vozlišče, določiti S, T ter dodati u v drevo pa je $O(n)$), v 4. fazi, ki se izvaja glede na 2. fazo, bomo največ $O(n)$-krat potrebovali, da najdemo prosto vozlišče y. Tedaj spremenimo $slack_y$, S, $N_l(S)$, T in drevo, kar vzame $O(n)$. Skupaj torej $O(n) \times O(n) \times O(n) = O(n^3)$. Če je potrebno še spremeniti graf v 3. fazi, najdemo minimum in vozlišča, ki ta minimum dajo, v $O(n)$, ravno tako pa spremenimo l in $slack_y$ v $O(n)$. Skupaj torej spet $O(n^3)$ (3. faza se najprej zgodi v odvisnosti od 2. faze, potem pa še glede na fazo 4.2). Vse skupaj pa potem znaša $O(n^3) + O(n^3) = O(n^3)$.

Celoten algoritem sicer definira in uporablja še nekatere funckije, kot npr. bfs in funkcijo, s katero povečamo prirejanje. Ravno tako na koncu povezavam prirejanja dodamo še uteži in izračunamo vrednost prirejanja. Temu navkljub se časovna zahtevnost ne spremeni. Bfs je v tem primeru $O(n)$ (za skoraj vsaki dodani vozlišči dodamo le dve povezavi), funkcija, s katero povečamo pot, pa je ravno tako $O(n)$. Ker izvajamo ti dve funkcij le ob vsaki spremembi prirejanja, torej n -krat, dobimo skupaj še dodatnih $O(n^2)$. Računanje vrednosti prirejanja in dodajanje uteži je $O(n)$. Določitev negativne matrike originalne matrike, definiranje praznega slovarja M in računanje začetnega dopustnega l kot v $(***)$, kar se vse zgodi povsem na začetku, pa je $O(n^2)$. Celoten algoritm je torej $O(n^3)$.

3 ILP

3.1 Opis metode

Naj nenegativna nxn matrika W predstavlja uteži povezav med posameznimi vozlišči. Element matrike iz i-te vrstice in j-tega stoplca, to je $w[i,j]$, predstavlja utež povezave med x_i in y_i, kjer velja $x_i \in X, y_j \in Y$.

Definiramo:

$$x[i,j] = \begin{cases} 1 & \text{, če je povezava } x_i - y_j \text{ v prirejanju} \\ 0 & \text{, sicer} \end{cases}$$

Zapišemo:

(1) min $\sum_{i,j=1}^{n} w[i,j]x[i,j]$

(2) p.p. $\sum_{i=1}^{n} x[i,j] = 1 \quad \forall j \in [n]$

(3) $x[i,j] \in \{0,1\} \quad \forall (i,j) \in nxn$

Razlaga:

1. Minimiziramo torej skupno ceno prirejanja.
2. Prvi pogoj nam zagotavlja, da ima vsako vozlišče natanko eno povezavo v prirejanju.
3. Drugi pogoj nam zagotavlja, da je x ali 0 ali 1

3.2 Algoritem

Zapišemo algoritem za ILP.

Vhod: zgoraj omenjena matrika W.

Izhod: najcenejše popolno uteženo prirejanje grafa, ki ga predstavlja zgornja matrika, skupaj z utežmi povezav in vrednostjo prirejanja.

```
def MinWeightedMatching(A):
    n = A.ncols()
    p = MixedIntegerLinearProgram(maximization=False)
```

```
    x = p.new_variable(binary=True)

    p.set_objective(sum(sum(A[i, j]*x[i, j]
    for j in range(n)) for i in range(n)))

    for i in range(n):
        p.add_constraint(sum(x[i, j] for j in range(n)) == 1)
        p.add_constraint(sum(x[j, i] for j in range(n)) == 1)

    teza = p.solve()
    prirejanje = p.get_values(x)

    resitev = [i for i, j in prirejanje.items() if j == 1]

    return(resitev, teza)
```

4 Primerjava metod

Obe metodi smo med seboj tudi primerjali na naključno generiranih matrikah različnih velikosti, kjer nas je predvsem zanimala časovna zahtevnost v odvisnosti od vhodnih podatkov.
Generirali smo matrike s pomočjo naslednje kode ter si zapisovali čas izvajanja za določeno velikost (n predstavlja velikost matrike ($n \times n$), elementi matrike so pa celoštevilski in iz intervala $[h, k]$, zadnja vrstica nam poda čas izvajanja v sekundah, kjer ponovi izvajanje 10 krat ter vzame povprečen čas izvajanja):

```
for n in range(10,201,10):
    h = 0
    k = 100
    A = Matrix(
    [[(int(random()*(k-h+1))+h) for j in range(n)] for i in range(n)])
    %timeit(repeat=10, seconds=True) madzarska_metoda(A)
```

Potem smo v programu R-Studio vstavili podatke izračunanih časov, naredili tabelo ter na enem grafu prikazali čase izvajanja algoritmov v odvisnosti od vhodnih podatkov (matrike velikosti $n \times n$), kot prikazuje naslednja slika:

Kot je videti iz slike sta pri manjših razsežnostih (do 20×20) časa madžarske metode in ILP-ja precej podobna, ko pa velikost matrike večamo, se pa čas računanja s pomočjo ILP-ja močno poveča v primerjavi z madžarsko metodo. Taki rezultat smo lahko tudi pričakovali, saj vemo, da ima madžarska metoda časovno zahtevnost $O(n^3)$, medtem ko je ILP v splošnem NP-težek. Torej je za matrike velikih razsežnosti vsekakor pametno uporabljati algoritem za madžarsko metodo, čeprav je pisanje algoritma kar dolgotrajno v primerjavi z ILP formulacijo, vendar je izvedba bistveno hitrejša.

Literatura

[1] http://www.cse.ust.hk/ golin/COMP572/Notes/Matching.pdf

[2] https://www.topcoder.com/community/data-science/data-science-tutorials/minimum-cost-flow-part-three-applications/