

Skupina 6: Wiener inverse interval problem

Projekt v povezavi s predmetom Operacijske raziskave

Končno poročilo

Avtorja:
Tjaša Renko, Darjan Pavšič

Ljubljana, november 2019

Kazalo

1	Predstavitev problema	2
1.1	Problem P1	2
1.2	Problem P2	2
1.3	Izbira programskega jezika in knjižnic	2
2	Problem P1	3
2.1	Reševanje	3
2.2	Rezultati in povzetek	5
3	Problem P2	6
3.1	Reševanje	6
3.2	Rezultati in povzetek	8
4	Zaključek	8

Slike

1	Časovna zahtevnost pri P1 glede na n	4
2	Časovna zahtevnost pri P1 glede na k_{\max}	5
3	Primer točne rešitve za $n = 6$, premer = 6	6
4	Primer točne rešitve za $n = 14$, premer = 12	6
5	Časovna zahtevnost pri P2 glede na n	7
6	Časovna zahtevnost pri P2 glede na število korakov ohlajanja .	8

1 Predstavitev problema

1.1 Problem P1

Za fiksno število vozlišč n in drevo T naj bo \mathcal{T}_{n+1} množica vseh dreves na $n+1$ vozliščih, dobljena iz T z dodajanjem lista enemu iz vozlišč T . $W(\mathcal{T}_{n+1})$ pa množica vrednosti Wienerjevega indeksa za drevesa iz \mathcal{T}_{n+1} . Poiskati želimo tako drevo T na n vozliščih, da bo moč množice $W(\mathcal{T}_{n+1})$ čim manjša (čim večja).

1.2 Problem P2

Za fiksno število vozlišč n iščemo drevo T z največjim možnim premerom, da bo veljalo; obstajata list u in vozlišče w iz T , da je list u , pripet na v in w , tak, da z odstranitvijo povezave uv in dodajanjem uw spremenimo vrednost indeksa za 1.

1.3 Izbira programskega jezika in knjižnic

Za izvedbo naloge sva se odločila za programski jezik *Python*, za lažje delo z grafi pa sva si pomagala s knjižnico `networkx`, tako da sva grafe lahko predstavljala kot objekte, sposodila pa sva si tudi vgrajeni funkciji računanja poti med poljubnima vozliščema grafa ter računanja premera. Uporabljala sva tudi knjižnice `numpy`, `random` in `matplotlib`, s katero so grafi tudi vizualno predstavljeni.

2 Problem P1

2.1 Reševanje

Zaradi velike časovne zahtevnosti sva problem ločila na kodo, ki išče točno rešitev in je uporabna za grafe z majhnim številom vozlišč, ter časovno in prostorsko bolj optimalno, ki si prizadeva najti dober približek točne rešitve za večje grafe.

Glavni del kode za iskanje točnih rešitev se nahaja v datoteki *p1_manjsi_grafi.py*. Tam so definirane funkcije za iskanje wienerjevega indeksa s pomočjo najkrajših poti med posameznima točkama drevesa, generiranje množice \mathcal{T}_{n+1} z računanjem indeksov glede na indeks prvotnega drevesa ter iskanje optimumov. Rešitev pa iščemo tako, da poženemo datoteko *shranjevanje_manjsi_grafi_p1.py* ter v funkciji *zapisi_resitve* nastavimo za argument željeno število vozlišč drevesa, do katerega želimo imeti rezultate. Ne priporočava za n izbrati števila, večjega od 17. Pri 20 že dobimo memory error, ki bi se sicer dal odpraviti, a to nima smisla, saj nam bo algoritem za tako majhne grafe ob pravilni izbiri parametrov zelo verjetno dal točno rešitev mnogo hitreje.

Glavni del kode za iskanje približkov na velikih grafih se nahaja v datoteki *p1_vecji_grafi.py*, koda pa se požene iz datoteke *shranjevanje_vecji_grafi_p1.py*, tako da v funkciji *shrani_resitve* za argumente nastavimo seznam s števili, ki predstavljajo velikosti grafov, za katere želimo rezultate, število korakov izvedbe simuliranega ohlajanja za vsak graf, število dreves za odstranitev iz prvotnega seznama ter število dreves z istim številom vozlišč za prvotni seznam, iz katerega bomo naredili prej omenjeni ožji izbor za ohlajanje.

Za začetek reševanja tega problema sva potrebovala ustrezne grafe. Ker naključna drevesa navadno ne bodo dovolj dobra, sva generirala nekaj boljših s pomočjo modificiranega Kruskalovega algoritma tako, da sva vzela pot velikosti nekje med 80 in 95% števila vseh vozlišč in nato naključno dodajala povezave na pot tako, da je nastalo povezano drevo, kar se je izkazalo za zelo učinkovito.

Za časovno optimalno računanje Wienerjevih indeksov sva najprej shranila dolžine najkrajših poti med vsakima točkama grafa. Tako sva potem lahko izračunala indeks vsakega posameznega novega drevesa v linearnem času; potrebovala sva namreč le vsoto poti iz vozlišča, ki sva mu dodala list, vsoto vseh poti ter število vozlišč; vse te podatke pa sva že imela.

Ko sva dobila podatke za na začetku generirana drevesa, sva med temi izbrala boljša in na njih izvedla algoritem za iskanje lokalnih ekstremov v okolici grafa, in sicer vrsto metahevrstike z imenom simulirano ohlajanje. Ta na vsakem koraku preveri stanje sosednjega grafa in se s padajočo verjetnostjo odloča, ali se bo vanj preselila, hkrati pa shranjuje najboljši rezultat do sedaj. Sosednji graf za algoritem sva ustvarila tako, da sva od prejšnjega odstranila povezavo z naključnega lista in naredila naključno novo iz njega. Po končanem algoritmu simuliranega ohlajanja je bilo treba samo še poiskati najboljšega izmed vseh rezultatov.

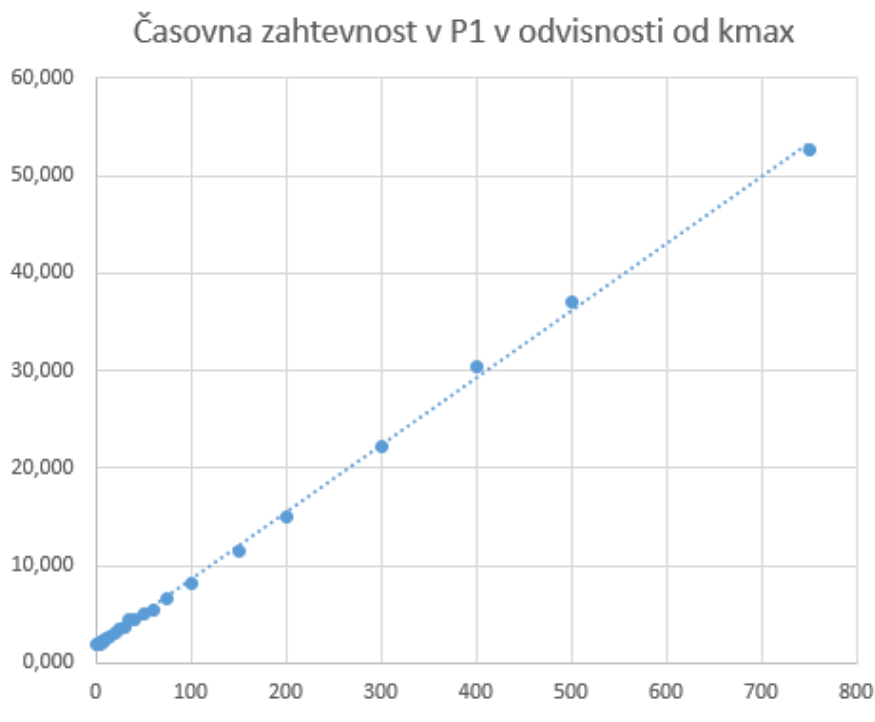
Časovna zahtevnost - izračunana

Časovno zahtevnost kode s simuliranim ohlajanjem sva preverila tudi eksperimentalno za nekaj različnih vrednosti n (pri parametrih št. grafov = 1, kmax = 20, št. dreves za odstanitev = 48, št. dreves za izbiro = 50):



Slika 1: Časovna zahtevnost pri P1 glede na n

Pri fiksnem številu vozlišč na 100 pa je časovna odvisnost glede na k_{\max} linearna:



Slika 2: Časovna zahtevnost pri P1 glede na k_{\max}

2.2 Rezultati in povzetek

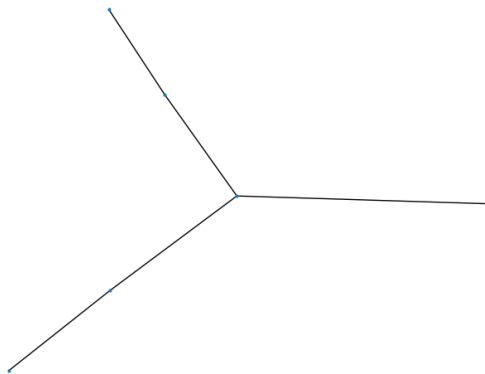
Meniva, da sva našla globalni minimum za poljubno število vozlišč, da je enak 2 in je dosežen pri zvezdah. Našla pa sva še en minimum za soda števila vozlišč n , sestavljen iz dveh zvezd velikosti $n/2$, povezanih s središčema. Kar se tiče maksimumov, jih je sodeč po točnih rešitvah na manjših grafih in uspešnostjo algoritmov na večjih precej veliko in so običajno po velikosti malo pod številom vozlišč ter največ enaki. Njihovo obliko si lahko ogledamo v shranjenih slikah, vidimo pa, da imajo relativno velik premer.

3 Problem P2

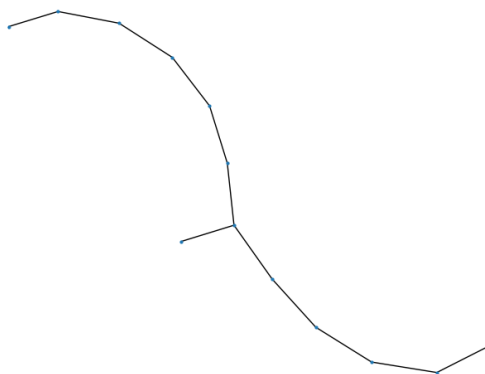
3.1 Reševanje

Za manjše grafe, kjer je možno eksaktno računanje, sva napisala funkcijo za generiranje vseh izomorfnih dreves na n vozliščih ter funkcije, ki najprej iz vseh poberejo tista z iskano lastnostjo, to je spremembo Wienerjevega indeksa za 1 ob odstranitvi ene in dodajanju nove povezave kot v opisu problema, nato pa izmed teh vrnejo tisto drevo z največjim premerom, če tako sploh obstaja. To je točna rešitev problema, a je časovno zahtevna ne le generacija grafov, temveč tudi računanje premera. Zato sva za večje n tudi tukaj napisala novo kodo.

Primeri dveh dreves, ki sva jih dobila z eksaktnim računanjem.



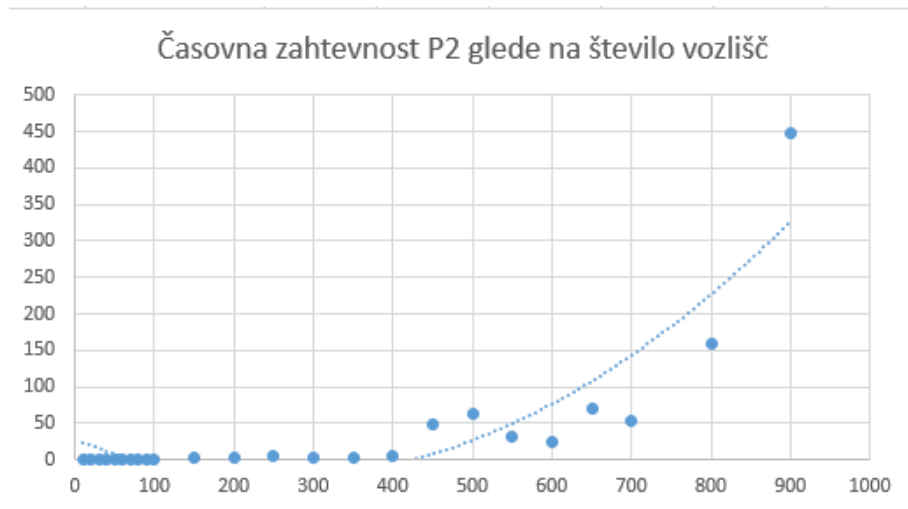
Slika 3: Primer točne rešitve za $n = 6$, premer = 6



Slika 4: Primer točne rešitve za $n = 14$, premer = 12

Ideja algoritma za iskanje drevesa na večjem številu vozlišč pri P2 je, da začneva na poti z željenim številom vozlišč in za njo posebej izračunava wien-nerjev indeks na časovno preprost način. Nato na vsakem koraku odstraniva povezavo iz lista in dodava novo ter glede na prejšnji graf izračunava nov indeks ter ju primerjava, nato pa star graf nadomestiva z novim. To omogoča preverjanje velikega števila grafov, kar je za problem P2 ključno.

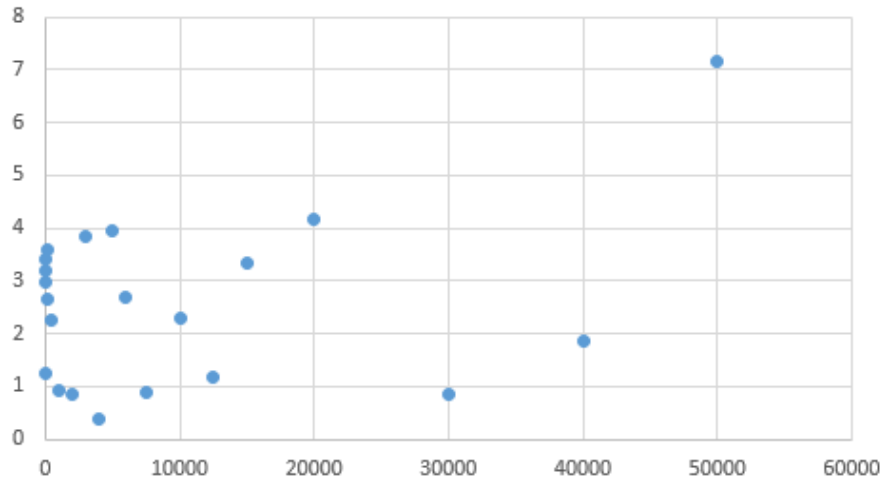
Časovna zahtevnost pri spreminjajočem se številu vozlišč n pri 1.000.000 korakih:



Slika 5: Časovna zahtevnost pri P2 glede na n

Še časovna zahtevnost pri fiksnem $n = 200$ in različnem številu korakov ohlajanja:

Časovna zahtevnost v odvisnosti od št. korakov (v tisoč) v funkciji iskanje(k,200)



Slika 6: Časovna zahtevnost pri P2 glede na število korakov ohlajanja

3.2 Rezultati in povzetek

Očitno je dobrih dreves precej malo, vsekakor dosti manj kot pri P1, saj je treba izvesti ogromno število korakov spreminjanja, da najdemo vsaj eno; to pa še narašča s številom vozlišč; pri 2000 namreč morda celo 1 000 000 ni dovolj. Videti je, da najin algoritem dobiva maksimume v velikosti okoli treh četrtin števila vozlišč. Težko je oceniti kvaliteto rešitev s pomočjo točnih rešitev manjših grafov, saj je koda tako časovno zahtevna, da imava rezultate le to dreves s številom vozlišč 14. Opazila sva tudi, da ne moreva dobiti iskanega drevesa za liho število vozlišč, in to ne le s poskušanjem, ampak tudi na malih grafih. Predpostavljava, da ne obstaja.

4 Zaključek