Grid-peeling

Gašper Pust, Mitja Mandić

14. december 2020

1 Predstavitev problema

V projektu si bova podrobneje ogledala konveksne ovojnice $m \times n$ mreže. Konveksna ovojnica množice je najmanjša konveksna množica, ki vsebuje dano množico točk. Najlažje si jo predstavljamo tako, kot da bi okoli elementov množice napeli elastiko - kar elastika obkroži, je konveksna ovojnica. Lupljenje konveksnih ovojnic mreže, (angl. *grid-peeling*) je proces, ko iz mreže iterativno odstranjujemo konveksne ovojnice. S simboli lahko to zapišemo takole: $P_0 = G_{m,n} = \{0, \ldots, m-1\} \times \{0, \ldots, n-1\}$. Naj bo $C_i = \mathcal{CH}(P_{i-1})$ za $i = 1, \ldots$. V_i naj bo množica vozlišč C_i - kot vozlišče razumemo točko, ki je na vogalu mreže (torej za katero bi zataknili elastiko). Naj bo sedaj $P_i = P_{i-1} \setminus V_i$. Začnemo torej z $n \times m$ mrežo in iterativno lupimo konveksne ovojnice, dokler ne odstranimo vseh točk.

V projektni nalogi bova s pomočjo simulacij opazovala v literaturi navedene številke za $n \times n$ mrežo - teorija napoveduje $\theta(n^{\frac{4}{3}})$ ovojnic. Za $m \times n$ mrežo v literaturi ni navedenih podatkov, zanimala pa naju bo morebitna povezava. Simulacije bova izvedla tudi za točke na neenakomerni mreži.

Po izvedenem eksperimentalnem delu, bova rezultate analizirala in jih primerjala z rezultati iz literature. Zanimalo naju bo, kako drugačno je število ovojnic na $m \times n$ mreži v primerjavi s simetrično.

2 Orodja in algoritmi

Algoritma sta v datotekah `jarvis_march.py` ter `graham_scan.py`, oba pa za delovanje uvozita datoteko `razredi.py`, kjer je definiran spodaj opisan razred točka in funkcije, ki generirajo mreže. Vizualizacijo sva izvedla v Jupyter Notebook datoteki `vizualizacija.ipynb`.

Za uporabo algoritmov v Pythonu sva najprej definirala razred Točka, v katerem sva skonstruirala vsa potrebna orodja in funkcije za uporabo algoritmov za iskanje konveksnih ovojnic.

```
class Tocka:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def kot_med_dvema(self, other):
        if self.x != other.x:
            return math.atan((self.y - other.y) / (self.x - other.x
    ))
        else:
            return math.pi/2

    def vektorski_produkt(self, other):
        return self.x * other.y - self.y * other.x

    def razlika(self, other):
      return Tocka(self.x - other.x, self.y - other.y)

    def razdalja(self, other):
        return (self.x - other.x) ** 2 + (self.y - other.y) ** 2

    def __str__(self):
        return '(' + str(self.x) + ', ' + str(self.y) + ')'

```

```
24      def __repr__(self):
25          return 'T(' + str(self.x) + ', ' + str(self.y) + ')'
26
27  def smer_razlike(p,q,r):
28      return p.razlika(q).vektorski_produkt(r.razlika(q))
29
30  def enakomerna_mreza(m,n):
31      return [Tocka(i,j) for i in range(m) for j in range(n)]
32
33  def naredi_potencno(m, n):
34      return [Tocka(2**i,2**j) for i in range(m) for j in range(n)]
35
36  def kvazi_cantor_mreza(n):
37      sez = [0,3]
38      for i in range(1,n):
39          nasl_dol = sez[0] - 3 ** i
40          nasl_gor = sez[-1] + 3 ** i
41          sez.append(nasl_gor)
42          sez = [nasl_dol] + sez
43      return [Tocka(i,j) for i in sez for j in sez]
```

Zgornje funkcije bova uporabila v dveh različnih algoritmih za iskanje konveksnih ovojnic, in sicer v Jarvisovem obhodu in Grahamovemu pregledu. Oba algoritma sprejmeta seznam točk in mu priredita konveksno ovojnico, a to storita na drugačen način. Seznam točk ustvarijo zadnje tri funkcije zgoraj, pri čemer prva sestavi enakomerno mrežo točk (i, j), druga ustvari mrežo točk oblike $(2^i, 2^j)$, tretja pa oblikuje mrežo, ki jo začnemo graditi na sredini in ima na i-tem koraku dolžino vsake od stranic 3^i, v množico pa na vsakem koraku dodamo kotne točke te mreže. Ker je vsaka prejšnja stranica ravno srednja tretjina naslednje, sva mrežo poimenovala „kvazi-Cantorjeva mreža."

2.1 Jarvisov obhod

Jarvisov obhod (angl. *Jarvis March*) ali algoritem zavijanja darila je postopek, ki dani množici točk poišče konveksno ovojnico v eni ali več dimenzijah (osredotočili se bomo na dve dimenziji). Algoritem se imenuje po R.A. Jarvisu, ki ga je objavil leta 1973. Časovna zahtevnost algoritma je $O(nh)$, kjer n predstavlja število vseh točk, h pa število točk, ki ležijo na konveksni ovojnici. V najslabšem primeru, ko so vse podane točke tudi elementi konveksne ovojnice, torej v primeru $h = n$, je njegova časovna zahtevnost $O(n^2)$. Jarvisov obhod se največkrat uporablja za majhne n ali pa v primeru, ko pričakujemo, da bo h zelo majhen glede na n.

```
1   def jarvis_march(seznam):
2       zacetna_tocka =  min(seznam, key = lambda tocka: tocka.x)
3       indeks = seznam.index(zacetna_tocka)
4       l = indeks
5       rezultat = []
6       rezultat.append(zacetna_tocka)
7       while (True):
8           q = (l + 1) % len(seznam)
9           for i in range(len(seznam)):
10              if i == l:
11                  continue
12              d = smer_razlike(seznam[l], seznam[i], seznam[q])
13              if d > 0 or (d == 0 and seznam[i].razdalja(seznam[l]) >
     seznam[q].razdalja(seznam[l])):
14                  q = i
```

```
15          l = q
16          if l == indeks:
17              break
18          rezultat.append(seznam[q])
19      return rezultat
```

Algoritem najprej poišče najbolj levo točko (2. vrstica kode). Potem ustvari prazen seznam, v katerega postopoma dodaja točke, ki jih obišče.

Vanj najprej doda začetno točko, označimo jo z l, nato izberemo poljubno točko, ki jo označimo s q (8. vrstica kode - tu je q točka izbrana tako, da je zagotovo v seznamu, lahko bi izbrali katerokoli razen začetne). Potem se začne iteracija, točke v iteraciji označimo z i. Če kot (l, i, q) predstavlja ovinek v desno, q in i zamenjamo, če je ta kot ovinek v levo nadaljujemo brez sprememb, če pa so točke p, q ter i kolinearne, izmed q in i izberemo tisto, ki ima večjo x koordinato (je dlje od začetne točke). Ko tako pridemo do konca in pregledamo vse točke, smo konveksno ovojnico našli in jo algoritem vrne kot seznam točk.

Opisan postopek se v zgornji kodi odvije od 7. do 15. vrstice. Funkcija `smer_razlike` izračuna kot med vektorji (oziroma natančneje, izračuna vektorski produkt vektorjev $(l - i) \times (q - i)$). Če je ta vrednost pozitivna (imamo ovinek v desno) oziroma 0 in je i dlje od začetne točke kot q, zamenjamo q in i.

Na koncu preverimo, če smo prišli nazaj v začetno točko l. V tem primeru algoritem ustavimo, ali pa postopek ponovimo z novo začetno točko. To je tista, ki smo jo v prejšnji ponovitvi zanke dodali v ovojnico.

2.2 Grahamov pregled

Alternativa prejšnjemu algoritmu je tako imenovani Grahamov pregled (angl. *Graham's scan*). Algoritem se imenuje po Ronaldu Grahamu, ki ga je objavil leta 1972. V primerjavi z Jarvisovim obhodom je Grahamov pregled hitrejši, saj ima časovno zahtevnost $O(n \log n)$.

```
1  def uredi_po_kotu(seznam):
2      prvi = min(seznam, key = lambda tocka: (tocka.x, tocka.y))
3      return [prvi] + sorted(seznam[1:], key=lambda x: prvi.
       kot_med_dvema(x))
4
5  def graham_scan(seznam):
6      urejene_tocke = uredi_po_kotu(seznam)
7      ovojnica = []
8      for tocka in urejene_tocke:
9          while len(ovojnica) > 1 and smer_razlike(ovojnica[-2],
       ovojnica[-1], tocka) >= 0:
10             ovojnica.pop()
11         ovojnica.append(tocka)
12     return ovojnica
```

Najprej definiramo funkcijo, ki seznam uredi po naraščajočem kotu med točkama, glede na x-os. Za urejanje uporabimo kar vgrajeno Pythonovo funkcijo `sorted`, ki bazira na *timsort* algoritmu, ki ima časovno zahtevnost $O(n \log n)$. Dodali smo le še funkcijo, ki računa kot med dvema točkama, kar pa se zgodi v linearnem času.

Algoritem najprej točke uredi po kotu in ustvari prazen seznam (kopico), v katerega bo dodajal točke, ki so v konveksni ovojnici.

Najprej v kopico dodamo prvi dve točki iz urejenega seznama, ti bosta zagotovo tudi v konveksni ovojnici. Če je v kopici več kot ena točka, vzame zadnji

dve točki v ovojnici in prvo iz urejenega seznama ter preveri kot med njimi. Če torej označimo zadnji dve točki iz kopice z x in y ter točko, ki je prva v seznamu z i, izračunamo vektorski produkt $(x-y)\times(i-y)$. Če je pozitiven (kot z vrhom v y in krakoma skozi x ter i predstavlja ovinek v desno), točko izloči, sicer pa jo doda na vrh kopice (v našem primeru na konec seznama). Ko pregleda vse urejene točke, se ustavi. V zanki moramo le enkrat pregledati vse točke, in ker se vse zgoraj opisano računanje izvaja v linearnem času, je zahtevnost tega dela $O(n)$ (kar pa je manj od zahtevnosti urejanja, torej je to zgornja meja za zahtevnost algoritma).

3 Algoritem za lupljenje konveksnih ovojnic

S pomočjo zgornjih algoritmov sedaj lahko izpeljemo algoritem za lupljenje konveksnih ovojnic. Glede na vrsto iskanja konveksnih ovojnic, torej po Jarvisovem ali Grahamovem načinu, ločimo dva algoritma, ki pa se razlikujeta le v koraku, v katerem iščemo konveksno ovojnico.

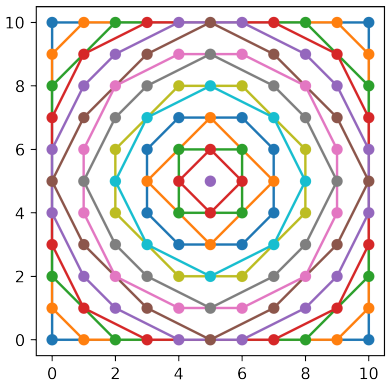
```
1  def grid_peel_graham(mreza): #oz. grid_peel_jarvis(mreza)
2      start = time.time()
3      ovojnice = {}
4      i = 0
5      while mreza or len(mreza) > 1:
6          ch = graham_scan(mreza) #oz. ch = jarvis_march(mreza)
7          nova = [x for x in mreza if x not in ch]
8          mreza = nova
9          ovojnice[i] = ch
10         i += 1
11     casovna_zahtevnost = time.time() - start
12     return i, ovojnice, casovna_zahtevnost
```

V obeh primerih algoritem najprej označi začetni čas in ustvari prazno množico, v katero bo kasneje shranjeval konveksne ovojnice. Dokler seznam točk ni prazen, algoritem na vsakem koraku poišče konveksno ovojnico (6. vrstica kode, tu se algoritma razlikujeta) in iz seznama mrežnih točk odstrani vse točke, ki so v konveksni ovojnici. Točke v konveksni ovojnici, dobljeni na i-tem koraku, doda v množico konveksnih ovojnic kot i-ti element in števec poveča za 1. To je del kode od 5. do 10. vrstice (v obeh algoritmih). Preden vrne rezultat, algoritem še enkrat izmeri čas, ki mu odšteje začetnega in razlika predstavlja čas potreben za izvedbo algoritma. Kot rezultat algoritma vrneta število korakov do prazne mreže, množico konveksnih ovojnic in pa potreben čas za izvedbo algoritma. S pomočjo množice konveksnih ovojnic lahko postopek lupljenja $m\times n$ mreže tudi narišemo.

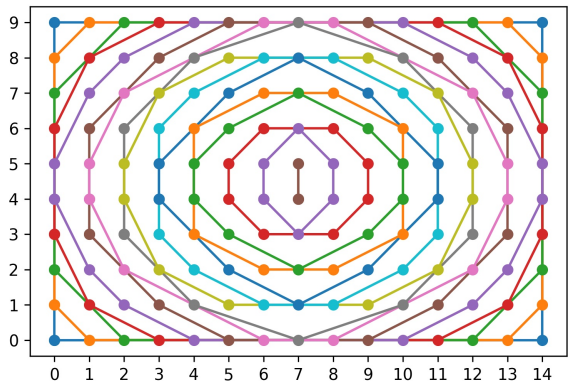
3.1 Prikaz rezultatov na primerih

Za boljšo predstavo, kako algoritem deluje, sva postopek lupljenja konveksnih ovojnic prikazala na nekaj primerih. Prvi, ki je prikazan na sliki 1, je postopek lupljenja navadne mreže velikosti 11×11, drugi, prikazan na sliki 2, pa je postopek lupljenja navadne mreže velikosti 15×10.

Slika 1: Postopek lupljenja ovojnic enakomerne 11×11 mreže

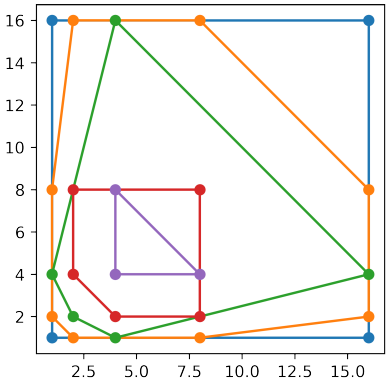


Slika 2: Postopek lupljenja ovojnic enakomerne 15×10 mreže

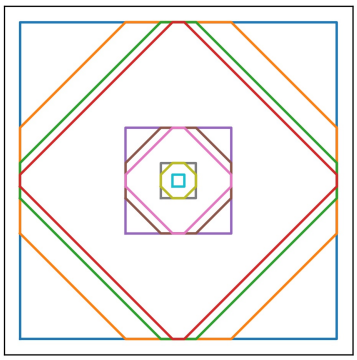


Na sliki 3 je prikazan postopek na potenčni mreži velikosti 5×5, na sliki 4 pa postopek na Cantorjevi mreži velikosti 4×4.

Slika 3: Postopek lupljenja ovojnic potenčne 5×5 mreže



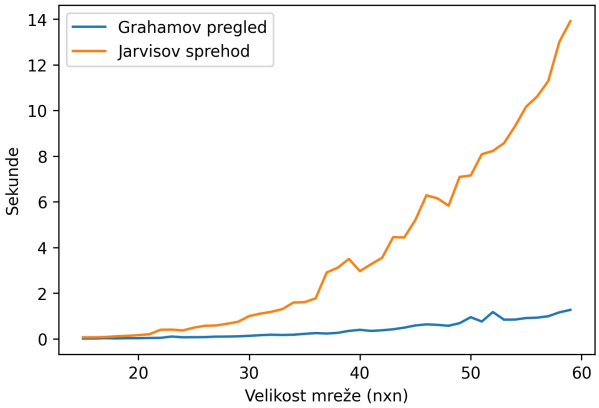
Slika 4: Postopek lupljenja ovojnic Cantorjeve 8×8 mreže



3.2 Časovna zahtevnost algoritma

Najprej bova prikazala časovno primerjavo med Jarvisovim obhodom in Grahamovim pregledom. Glede na izbrani algoritem za iskanje konveksnih ovojnic se čas izvajanja algoritma za lupljenje lahko zelo razlikuje, kar je jasno razvidno iz slike 5. Tu sva prikazala samo primer $n \times n$ mreže, vendar pa je za $m \times n$ mreže rezultat podoben. Absolutni čas izvajanja algoritma je seveda odvisen od številnih dejavnikov (zmogljivost računalnika, koliko procesov se izvaja v ozadju,...), vendar se jasno vidi večjo učinkovitost Grahamovega pregleda.

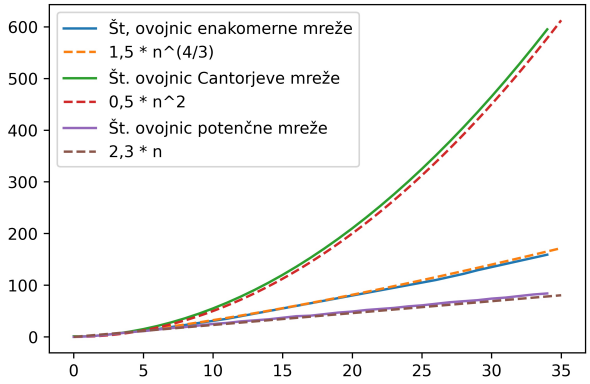
Slika 5: Čas izvedbe algoritma lupljenja konveksnih ovojnic $n \times n$ mreže v odvisnosti od n



Kot že omenjeno, je v literaturi omenjena časovna zahtevnost $O(n^{\frac{4}{3}})$, kar pomeni, da je število konveksnih ovojnic na $n \times n$ mreži $\theta(n^{\frac{4}{3}})$. Na sliki 6 je predstavljeno število ovojnic izvedenega algoritma v odvisnosti od n, kjer za n vzamemo naravna števila od 1 do 35 in mreže različnih vrst. Opazimo, da se razmerju med številom ovojnic in velikostjo navadne mreže najbolj prilega funkcija $1,5 \cdot n^{\frac{4}{3}}$. Eksperimentalni rezultat se torej ujema s tistim, navedenim v literaturi. Za potenčne $n \times n$ mreže je najboljša funkcija, ki prikaže razmerje

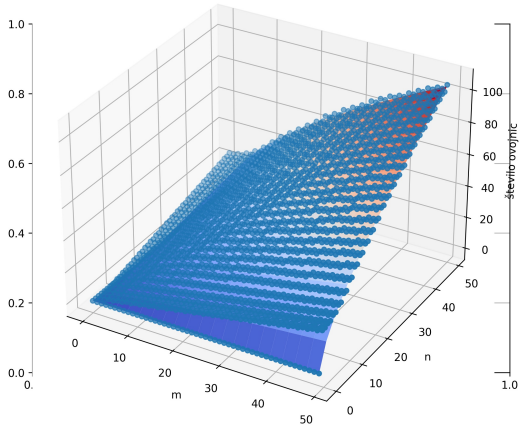
med številom konveksnih ovojnic in velikostjo mreže, $2,3 \cdot n$, torej je časovna zahtevnost algoritma v tem primeru $O(n)$. Na enak način za Cantorjevo $n \times n$ mrežo dobimo časovno zahtevnost $O(n^2)$, vendar pa to velja samo za n manjši od 35, za višje n pa se število ovojnic Cantorjeve mreže obnaša precej nenavadno, česar pa nisva znala pojasniti. Ugibava, da gre za težave pri računanju zelo velikih števil (velikosti ranga $3^{40} \approx 10^{20}$).

Slika 6: Število ovojnic različnih $n \times n$ mrež v odvisnosti od n



Za $m \times n$ mreže v literaturi ni omenjene časovne zahtevnosti, zato sva jo poiskušala določiti sama. Število ovojnic je tu odvisno od dveh parametrov, m in n, zato je ta graf tridimenzionalen. Rezultate lahko vidimo na sliki 7.

Slika 7: Število ovojnic 50×50 mreže v odvisnosti od m in n



Na zgornji sliki vidimo graf števila ovojnic mreže velikosti 50×50 v odvisnosti od m in n. Rezultatom sva s pomočjo aproksimacijskega algoritma narisala ploskev, ki se najbolje prilega rezultatom. Ta algoritem je sprejel podatke o številu ovojnic pri posameznih parih (m, n) in narisal te točke ter ploskev oblike $a(m^b \cdot n^c)$, ki se jim najbolje prilega. Poleg tega je vrnil še pripadajoče parametre

a, b in c te ploskve, iz katerih sva ocenila, da je časovna zahtevnost lupljenja konveksnih ovojnic na $m \times n$ mreži $O((mn)^{\frac{2}{3}})$. Aproksimacijski algoritem, ki sva ga uporabila, je podrobneje predstavljen na spletni strani [3].

4 Rezultati

Kot že rečeno, se za enakomerne $n \times n$ mreže, za katere je bila v literaturi podana časovna zahtevnost, teoretični in eksperimentalni podatki ujemajo. Časovna zahtevnost izvedbe algoritma je v tem primeru torej $O(n^{\frac{4}{3}})$. Za potenčne mreže velikosti $n \times n$, tj. množice točk oblike $(2^i, 2^i), 0 \leq i \leq n-1$, je časovna zahtevnost enaka $O(n)$, za Cantorjeve mreže pa $O(n^2)$ (za $n \leq 35$), kar se ravno tako ujema z literaturo. Za mreže velikosti $m \times n$ v literaturi ni bilo podatka o časovni zahtevnosti, zato sva jo določila eksperimentalno. Profesor Cabello je predvideval, da se bo mreža obnašala podobno kot $\min\{m, n\} \times \min\{m, n\}$, eksperimentalno pa sva ocenila, da je za take mreže časovna zahtevnost algoritma $O((mn)^{\frac{2}{3}})$. Več podatkov o vizualizaciji in risanju slik, pa je v datoteki `vizualizacija.ipynb`.

Literatura

[1] David Eppstein, Sariel Har-Peled, Gabriel Nivasch. *Grid Peeling and the Affine Curve-Shortening Flow.*

[2] Sariel Har-Peled, Bernard Lidický. *Peeling the Grid.* 2012.

[3] Fitting 3d Data,
`https://stackoverflow.com/questions/53506257/fitting-3d-data?noredirect=1&lq=1`