

# *Grid-peeling*

Gašper Pust, Mitja Mandić

10. 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:  $P_0 = G_{m,n} = \{0, \dots, m-1\} \times \{0, \dots, n-1\}$ . Naj bo  $C_i = \mathcal{CH}(P_{i-1})$  za  $i = 1, \dots$ .  $V_i$  naj bo množica vozlišč  $C_i$  - kot vozlišče razumemo točko, ki je na vogalu mreže (torej za katero bi zataknilo 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

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.

```
1 class Točka:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6     def kot_med_dvema(self, other):
7         if self.x != other.x:
8             return (self.y - other.y) / (self.x - other.x)
9         else:
10            return 90
11
12    def vektorski_produkt(self, other):
13        return self.x * other.y - self.y * other.x
14
15    def razlika(self, other):
16        return Točka(self.x - other.x, self.y - other.y)
17
18    def razdalja(self, other):
19        return (self.x - other.x) ** 2 + (self.y - other.y) ** 2
20
21    def __str__(self):
22        return '(' + str(self.x) + ', ' + str(self.y) + ')'
23
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 [Točka(i,j) for i in range(m) for j in range(n)]
32
33 def naredi_potencno(m, n):
34     return [Točka(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 [Točka(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 ustvariyo 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 obojnici. 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]) >
14                 seznam[q].razdalja(seznam[l])):
15                 q = i
16             l = q
17             if l == indeks:
18                 break
19         rezultat.append(seznam[q])
20     return rezultat
```

Algoritem najprej poišče najbolj levo točko (2. vrstica kode). Potem ustvari prazen seznam, v katerega postopoma dodaja točke, ki jih obiše. Vanj najprej doda začetno točko, nato izberemo poljubno točko (8. vrstica kode - tu je  $q$  točka izbrana tako, da je zagotovo v seznamu, lahko bi izbrali katerokoli razen začetne), in glede na ti točki poišče največji levi ovinek (torej največji kot) in gre v tisto točko, za katero je tak kot najden (v primeru kolinearnosti, gre v točko, ki je od trenutne točke najdlje). Točko doda v seznam in jo nastavi za trenutno točko. To je del kode od 7. do 15. vrstice. Ko spet pride v začetno točko, se algoritem ustavi.

## 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.
4                             kot_med_dvema(x))
5
6 def graham_scan(seznam):
7     urejene_tocke = uredi_po_kotu(seznam)
8     ovojnica = []
9     for tocka in urejene_tocke:
10         while len(ovojnica) > 1 and smer_razlike(ovojnica[-2],
11             ovojnica[-1], tocka) >= 0:
12             ovojnica.pop()
13         ovojnica.append(tocka)
14     return ovojnica
```

Najprej definiramo funkcijo, ki seznam uredi po naraščajočem kotu med točkama, glede na  $x$ -os. Algoritem najprej točke iz seznama točk uredi po kotu in ustvari prazen seznam (kopico), v katerega bo dodajal točke, ki so v konveksni ovojnici. Če je v ovojnici več kot ena točka, vzame zadnji dve točki v ovojnici in prvo iz urejenega seznama ter preveri kot med njimi. Če zavije v desno (vektorski produkt vektorjev, ki imajo začetek v zadnji točki ovojnice in konec v predzadnji točki ovojnice oz. prvi točki seznama je nenegativen), točko izloči, sicer pa jo doda v ovojnico. Ko pregleda vse urejene točke, se ustavi. V kodi je ta postopek napisan od 8. do 11. vrstice. Algoritem je hitrejši od Jarvisovega obhoda, ker delamo z že urejenimi točkami in nam zato ni treba na vsakem koraku pregledati vseh točk.

## 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
```

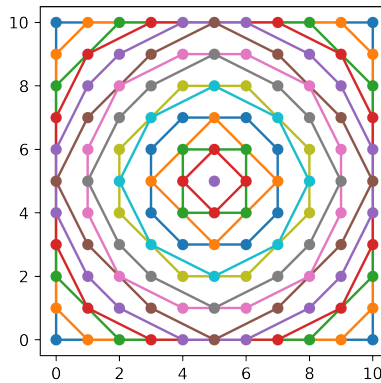
V obeh primerih algoritem najprej označi začetni čas in ustvari prazno množico, v katero bo kasneje shranjeval konveksne ovojnice. Dokler seznam točk ni prazen, algoritem na vsakem koraku poišče konveksno ovojnico (6. vrstica kode,

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.

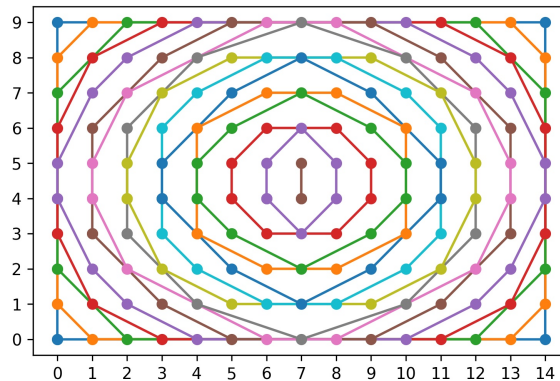
### 3.1 Prikaz rezultatov na primerih

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  $11 \times 11$ , drugi, prikazan na sliki 2, pa je postopek lupljenja navadne mreže velikosti  $15 \times 10$ .

Slika 1: Postopek lupljenja ovojnic enakomerne  $11 \times 11$  mreže

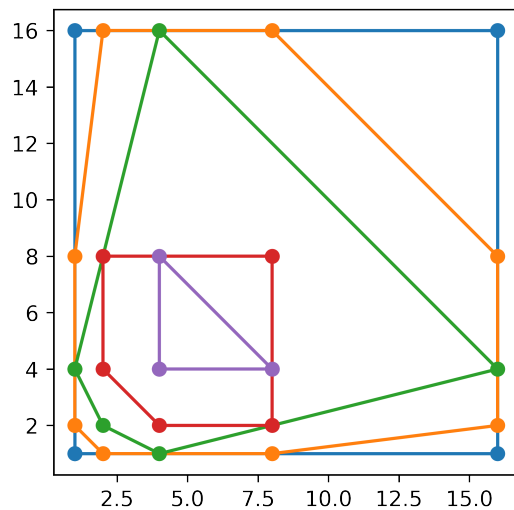


Slika 2: Postopek lupljenja ovojnic enakomerne  $15 \times 10$  mreže

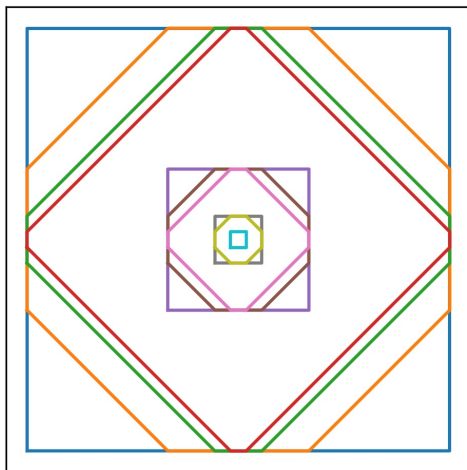


Na sliki 3 je prikazan postopek na potenčni mreži velikosti  $5 \times 5$ , na sliki 4 pa postopek na Cantorjevi mreži velikosti  $4 \times 4$ .

Slika 3: Postopek lupljenja ovojnic potenčne  $5 \times 5$  mreže



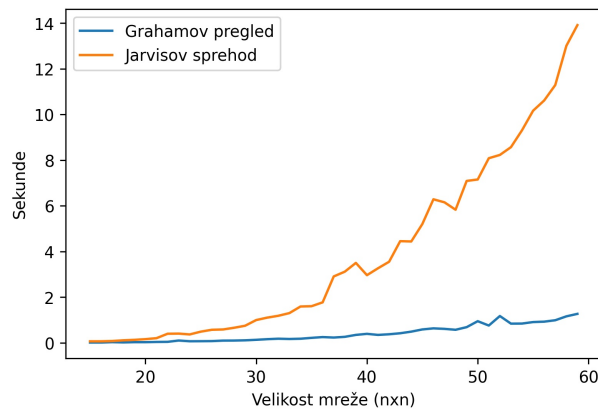
Slika 4: Postopek lupljenja ovojnic Cantorjeve  $8 \times 8$  mreže



## 3.2 Časovna zahtevnost algoritma

Najprej bova prikazala časovno primerjavo med Jarvisovim obhodom in Grahamovim pregledom. Glede na izbrani algoritem za iskanje konveksnih ovojníc 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 ovojníc  $n \times n$  mreže v odvisnosti od  $n$

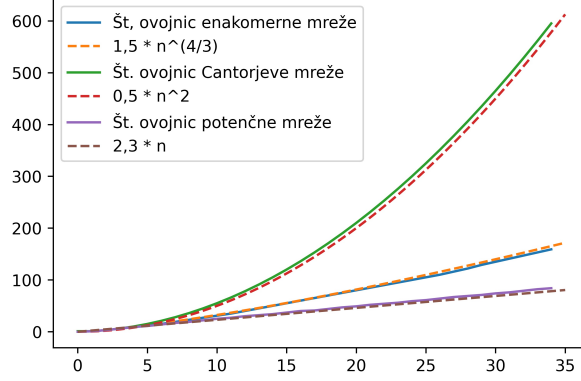


Kot že omenjeno, je v literaturi omenjena časovna zahtevnost  $\mathcal{O}(n^{\frac{4}{3}})$ , kar pomeni, da je število konveksnih ovojníc na  $n \times n$  mreži  $\Theta(n^{\frac{4}{3}})$ . Na sliki 6 je predstavljeno število ovojníc izvedenega algoritma v odvisnosti od  $n$ , kjer za  $n$  vzamemo naravna števila od 1 do 35 in mreže različnih vrst. Opazimo, da se razmerju med številom ovojníc 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 ovojníc in velikostjo mreže,  $2,3 \cdot n$ , torej je časovna zahtevnost algoritma v tem primeru  $\mathcal{O}(n)$ . Na enak način za Cantorjevo  $n \times n$  mrežo dobimo časovno zahtevnost  $\mathcal{O}(n^2)$ , vendar pa to velja samo za  $n$  manjši od 35, za višje  $n$  pa se število ovojníc Cantorjeve mreže obnaša precej nenavadno, česar pa nisva znala pojasniti. Ugibava, da gre za težave pri računanju zelo velikih števil (velikosti ranga  $3^{40} \approx 10^{20}$ ).

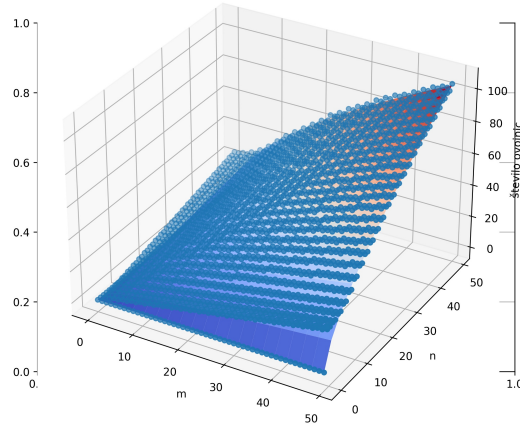
Za  $m \times n$  mreže v literaturi ni omenjene časovne zahtevnosti, zato sva jo poiskovala določiti sama. Število ovojníc je tu odvisno od dveh parametrov,  $m$  in  $n$ , zato je ta graf tridimenzionalen. Rezultate lahko vidimo na sliki 7.



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



Slika 7: Število ovojnic  $m \times n$  mreže v odvisnosti od  $m$  in  $n$



## 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 na je v tem primeru torej  $\mathcal{O}(n^{\frac{4}{3}})$ . 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  $\mathcal{O}(n)$ , za Cantorjeve mreže pa  $\mathcal{O}(n^2)$  (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  $\mathcal{O}((mn)^{\frac{2}{3}})$ .