

Grid-peeling

Gašper Pust, Mitja Mandić

29. november 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: $P_0 = G_{m,n} = \{1, \dots, m\} \times \{1, \dots, n\}$. 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 zataknil 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 ovojníc.

```
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 smer_razlike(self, other, another):
19        vekt_p = Točka(self.x - other.x, self.y - other.y).
20            vektorski_produkt(Točka(another.x - other.x, another.y
21                - other.y))
22        if vekt_p > 0:
23            return 1
24        elif vekt_p < 0:
25            return -1
26        else:
27            return 0
28
29    def razdalja(self, other):
30        return (self.x - other.x) ** 2 + (self.y - other.y) ** 2
31
32    def __str__(self):
33        return '(' + str(self.x) + ', ' + str(self.y) + ')'
34
35    def uredi_po_kotu(seznam):
36        prvi = seznam[0]
37        return [prvi] + sorted(seznam[1:], key=lambda x: prvi.
38            kot_med_dvema(x))
```

Zgornje funkcije bova uporabila v dveh različnih algoritmih za iskanje konveksnih ovojníc, in sicer v Jarvisovem obhodu in Grahamovem pregledu. Oba algoritma sprejmeta seznam točk in mu priredita konveksno ovojnico, a to storita na drugačen način.

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 = seznam[i].smer_razlike(seznam[l], 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 dodajam točke, ki jih obiše. Vanj najprej doda začetno točko, potem pa od trenutne točke poišče največji levi ovinek in gre v tisto točko (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 graham_scan(seznam):
2     urejene_tocke = uredi_po_kotu(seznam)
3     ovojnica = []
4     for tocka in urejene_tocke:
5         while len(ovojnica) > 1 and ovojnica[-1].smer_razlike(
6             ovojnica[-2], tocka) <= 0:
7             ovojnica.pop()
8             ovojnica.append(tocka)
9     return ovojnica

```

Algoritem najprej točke iz seznama točk uredi po kotu in ustvari prazen seznam, v katerega bo dodajal točke, ki so v konveksni ovojnici. Za točko potem pogleda (naredi neki k mi ni čisto jasno) in doda točko v ovojnico. To stori za vse točke v urejenem seznamu. V kodi je ta postopek napisan od 4. do 7. vrstice.

3 Algoritem za lupljenje konveksnih ovojníc

S pomočjo zgornjih algoritmov sedaj lahko izpeljemo algoritem za lupljenje konveksnih ovojníc. Glede na vrsto iskanja konveksnih ovojníc, 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_jarvis(m, n):
2     mreza = [Tocka(i,j) for i in range(m) for j in range(n)]
3     ovojnice = {}
4     i = 0
5     while mreza:
6         ch = jarvis_march(mreza)
7         nova = list(set(mreza) - set(ch))
8         mreza = nova
9         ovojnice[i] = ch
10        i += 1
11    return i, ovojnice

```

```

1 def grid_peel_graham(m, n):
2     mreza = [Tocka(i,j) for i in range(m) for j in range(n)]
3     ovojnice = {}
4     i = 0
5     while mreza:
6         ch = graham_scan(mreza)
7         nova = list(set(mreza) - set(ch))
8         mreza = nova
9         ovojnice[i] = ch
10        i += 1
11    return i, ovojnice

```

V obeh primerih algoritem najprej sestavi seznam točk na mreži velikosti $m \times n$, kjer sta m in n naravni števili, ki jih funkcija sprejme kot argument. Potem ustvari prazno množico, v katero bo potem 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). Kot rezultat algoritma vrnete število korakov do prazne mreže in pa množico konveksnih ovojnic. S pomočjo slednje lahko postopek lupljenja konveksnih ovojnic $m \times n$ mreže tudi narišemo.

3.1 Prikaz rezultatov na primerih

3.2 Časovna zahtevnost algoritma

4 Rezultati

5 Zaključek