Grid-peeling

Gašper Pust, Mitja Mandić

2. 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: $P_0 = G_{m,n} = \{1, \ldots, m\} \times \{1, \ldots, n\}$. 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

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 (self.y - other.y) / (self.x - other.x)
        else:
            return 90

    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) + ')'

    def __repr__(self):
        return 'T(' + str(self.x) + ', ' + str(self.y) + ')'

def smer_razlike(p,q,r):
    return p.razlika(q).vektorski_produkt(r.razlika(q))

def naredi_neenakomerno(st_tock, zgornja_meja): #naredi
    neenakomerno mrezo, ki je predstavljena kasneje
    seznam = sorted(random.sample(range(zgornja_meja),st_tock))

def naredi_potencno(m, n): #naredi potencno mrezo, ki je
    predstavljena kasneje
    return [Tocka(2**i,2**j) for i in range(m) for j in range(n)]
```

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.

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 dodajam točke, ki jih obišče. 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, ke 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 smer_razlike(ovojnica[-2],
       ovojnica[-1], tocka) >= 0:
6              ovojnica.pop()
7          ovojnica.append(tocka)
8      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 čist 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 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_jarvis(m, n):
2      start = time.time()
3      mreza = [Tocka(i,j) for i in range(m) for j in range(n)]
4      ovojnice = {}
5      i = 0
6      while mreza or len(mreza)>1:
7          ch = jarvis_march(mreza)
8          nova = [x for x in mreza if x not in ch]
9          mreza = nova
10         ovojnice[i] = ch
11         i += 1
12     casovna_zahtevnost = time.time() - start
13     return i, ovojnice, casovna_zahtevnost
```

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

V obeh primerih algoritem najprej označi začetni čas in 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 kasneje shranjeval konveksne ovojnice. Dokler seznam točk ni prazen, algoritem na vsakem koraku poišče konveksno ovojnico (7. 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 6. do 11. vrstice (v obeh algoritmih). Preden vrne rezultat, algoritem še enkrat izmeri čas in odšteje začetnega, da dobimo časovno zahtevnost. 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.

Za primer, ko mreža ni enakomerna, v algoritmih samo prilagodimo množico točk, ki jo algoritem ustvari:

- Za potenčno mrežo, torej za točke oblike $(2^i, 2^j), 0 \leq i \leq m, 0 \leq j \leq n$, 3. vrstico algoritma spremenimo v `mreza = naredi_potencno(m, n)`.

3.1 Prikaz rezultatov na primerih

3.2 Časovna zahtevnost algoritma

4 Rezultati

5 Zaključek