Grid-peeling

Gašper Pust, Mitja Mandić

28. november 2020

1 Predstavitev problema

V projektu si bomo podrobneje ogledali konveksne ovojnice $m \times n$ mreže. Konveksna ovojnica množice je najmanjša konveksna množica, ki vsebuje dano množico. Najlažje si jo predstavljamo tako, kot da bi okoli elementov množice napeli elastiko - kar elastika obkroži, je konveksna ovojnica. Lupljenje konveksnih ovojnic mreže, oziroma angleško *grid - peeling* je proces, ko iz mreže iterativno odstranjujemo konveksne ovojnice. S simboli lahko to zapišemo takole: $P_0 = G_{n,m} = \{1, \ldots, n\} \times \{1, \ldots, m\}$. 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 $n \times m$ mrežo v literaturi ni navedenih podatkov, zanimala naju bo morebitna povezava. Simulacije bova izvedla tudi za točke na neenakomerni mreži.

Po izvedenem eksperimentalnem delu, bomo rezultate analizirali in jih primerjali z rezultati iz literature. Zanimalo nas 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 morala najprej definirati razred Točka, s katerim bova konstruirala vsa potrebna orodja in funkcije za uporabo algoritmov za iskanje konveksnih ovojnic.

```
1  class Tocka:
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 Tocka(self.x - other.x, self.y - other.y)
17
18     def smer_razlike(self, other, another):
19         vekt_p = Tocka(self.x - other.x, self.y-other.y).
               vektorski_produkt(Tocka(another.x - other.x, another.y
                - other.y))
20         if vekt_p > 0:
21             return 1
22         elif vekt_p < 0:
23             return -1
24         else:
25             return 0
26
27     def razdalja(self, other):
28         return (self.x - other.x) ** 2 + (self.y - other.y) ** 2
29
30     def __str__(self):
31         return '(' + str(self.x) + ', ' + str(self.y) + ')'
32
33     def uredi_po_kotu(seznam):
34         prvi = seznam[0]
35         return [prvi] + sorted(seznam[1:], key=lambda x: prvi.
               kot_med_dvema(x))
```

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])
                   > 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). Ustvarimo prazen seznam, v katerega bomo postopoma dodajali točke, ki jih bomo obiskali. Vanj najprej dodamo začetno točko, potem pa od trenutne točke poiščemo največji levi ovinek in gremo v tisto točko (v primeru kolinearnosti, gremo v točko, ke je od naše točke najdlje). To točko dodamo v seznam in se postavimo tja (del kode od 7. do 15. vrstice). To počnemo, dokler ne pridemo nazaj v začetno točko in takrat 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(
               ovojnica[-2], tocka) <= 0:
6              ovojnica.pop()
7          ovojnica.append(tocka)
8      return ovojnica
```

Najprej točke iz seznama točk uredimo po kotu in ustvarimo prazen seznam, v katerega bomo dodajali točke, ki so v konveksni ovojnici. Za točko potem pogledamo (naredimo neki k mi ni čist jasno) in dodamo točko v ovojnico. To storimo za vse točke v urejenem seznamu (od 4. do 7. vrstice).

3 Grid-peel algoritem

S pomočjo že omenjenih algoritmov sedaj lahko izpeljemo algoritem za luplenje 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      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
```

4 Rezultati

5 Zaključek