resitev

January 28, 2024

0.1 Naloga

Imamo koordinate nekaterih slovenskih krajev.

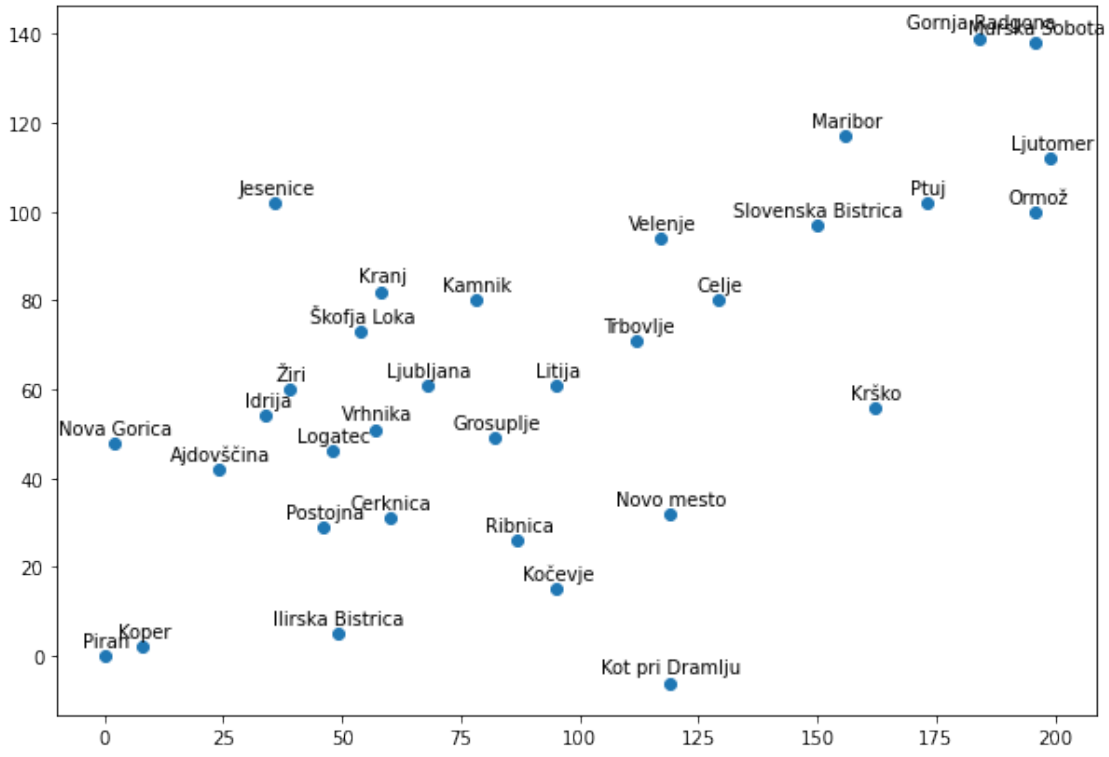
[4]:
```
koordinate = [('Piran', (0, 0)), ('Koper', (8, 2)),
              ('Ilirska Bistrica', (49, 5)), ('Postojna', (46, 29)),
              ('Nova Gorica', (2, 48)), ('Ajdovščina', (24, 42)),
              ('Idrija', (34, 54)), ('Logatec', (48, 46)),
              ('Cerknica', (60, 31)), ('Vrhnika', (57, 51)),
              ('Žiri', (39, 60)), ('Ljubljana', (68, 61)),
              ('Ribnica', (87, 26)), ('Kočevje', (95, 15)),
              ('Grosuplje', (82, 49)), ('Litija', (95, 61)),
              ('Kranj', (58, 82)), ('Kamnik', (78, 80)),
              ('Škofja Loka', (54, 73)), ('Trbovlje', (112, 71)),
              ('Novo mesto', (119, 32)), ('Krško', (162, 56)),
              ('Celje', (129, 80)), ('Maribor', (156, 117)),
              ('Velenje', (117, 94)), ('Slovenska Bistrica', (150, 97)),
              ('Murska Sobota', (196, 138)), ('Ptuj', (173, 102)),
              ('Ormož', (196, 100)), ('Ljutomer', (199, 112)),
              ('Gornja Radgona', (184, 139)),
              ('Jesenice', (36, 102)),
              ("Kot pri Dramlju", (119, -6))]
```

(H krajem, ki so bili sicer v domači nalogi sem tu dodal še dva. Jesenice: Slovenija brez Jesenic je videti čudna, razpotegnjena navzgor. Jesenice potrebujemo, da jo "napnejo" čez celo površino. Kot pri Dramlju: najjužnejši slovenski kraj. V originalnem seznamu je bil Piran hkrati najjužnejši in najzahodnejši, kar olajša nalogo, zato rešitev ni tako nazorna, kot bi bila sicer.)

Najprej si pripravimo funkcijo, ki nariše zemljevid, da bomo lahko gledali, kaj delamo. Uporabili bomo matplotlib. Zemljevid je razsevni diagram (`plt.scatter`, točke označimo s `plt.annotate`).

[5]:
```
%matplotlib inline

import numpy as np
from matplotlib import pyplot as plt

def zemljevid():
    vid = plt.gcf()
    vid.set_size_inches(10, 7)
```

```
    x = [koord[0] for _, koord in koordinate]
    y = [koord[1] for _, koord in koordinate]
    # Šlo bi tudi v enem zamahu, tako:
    # x, y = zip(*(koord for _, koord in koordinate))

    plt.scatter(x, y)

    for ime, (x, y) in koordinate:
        plt.annotate(ime, (x,y), textcoords="offset points", xytext=(1,5),ha='center')

zemljevid()
```



0.2 Očitnejša rešitev: preudarno dodajanje

Začeli bomo z najzahodnejšim krajem in si želeli iti navzgor. Ker to ne bo šlo, bomo zavili v kraj, ki bo najmanj odstopal od smeri navzgor (to bo Nova Gorica). V naslednjem koraku bomo morali zaviti nekoliko desno; želimo si čim manj desno, torej bomo izbrali tisti kraj, ki bo najbolj v isti smeri kot je bila smer Piran - Nova Gorica. (Nevarnosti, da bi zavili levo, ni, saj je Piran najbolj levi kraj in Nova Gorica najbolj leva možna smer.) Naslednji kraj bodo torej Jesenice. Zdaj nadaljujemo v istem duhu: spet bomo morali zaviti malo desno, vendar bi radi zavili čim manj. Izbrali bomo kraj, ki bo najbolj v isto smer kot je bila smer Nova Gorica - Jesenice; to nas bo

pripeljalo naravno v Gornjo Radgono. Iz nje nadaljujemo v smer, ki je čimbolj podobna smeri Jesenice - Gornja Radgona in se znajdemu v Murski Soboti…

Zdaj, ko smo se domislili splošne strategije, moramo rešiti matematični problem: če imamo, recimo, smer Nova Gorica - Jesenice: kako med vsemi možnimi nadaljevanji izberemo tisto, ki je najbolj podobno tej smeri? Če si povezave med kraji predstavljamo kot vektorje, je "podobnost smeri" kar kot med vektorjema. Spomnimo se kosinus kota je enak normiranemu skalarnemu produktu:

$$cos(\phi) = \frac{\mathbf{ab}}{|\mathbf{a}||\mathbf{b}|}$$

Vsi naši skalarni produkti bodo pozitivni, vsi koti manjši od 180 stopinj … zato se ne bomo zafrkavali s kosinusom, temveč rekli kar, da sta si dve smeri tem bolj podobni, čim večji je normirani skalarni produkt med njima.

Pripravimo si funkcijo `skalp`, ki bo računala skalarni produkt dveh vektorjev, podanih s komponentami. Funkcija bo predpostavila, da je prvi vektor normiran, tako da bo vsoto produktov komponent delila samo z dolžino drugega vektorja. Zgodilo se bo tudi, da bo dolžina drugega vektorja enaka 0; v tem primeru bo delila z `1`, da ne dobimo deljenja z 0.

[6]:
```
from math import sqrt

def skalp(ax, ay, bx, by):
    return (ax * bx + ay * by) / (sqrt(bx ** 2 + by ** 2) or 1)
```

Zdaj pa imamo vse kar potrebujemo. Najprej izberemo prvi kraj. Ker smo se na prejšnjih predavanjih mimogrede naučili nekaj o lambdah, vemo, da lahko pokličemo kar `min(koordinate)`, zraven pa kot ključ podamo funkcijo, ki ji bo `min` podajala elemente seznama `koordinate` (terke) in funkcija bo vrnila element z indeksom 1, torej dejanske koordinate.

V spremenljivkah `kraj`, `tx` in `ty` bomo hranili trenutni kraj in njegove koordinate, `px` in `py` pa bosta želena smer - v začetku je to navzgor. `elastika` bo vsebovala seznam krajev, ki se jih elastika dotakne.

Pred zanko damo v elastiko prvi kraj. Nato vrtimo zanko, dokler se ne vrnemo v izhodiščni kraj, torej dokler je `elastika[0]` različna od `elastika[-1]`, pri čemer seveda popazimo na posebni primer v začetku, ko `elastika` vsebuje en sam kraj. (Morda je kdo opazil: če bi Python imel zanko `do - while`, bi lahko prestavili pogoj na konec, pa ne bi potrebovali `len(elastika) == -1`. Kot sem rekel na drugih predavanjih: Python takšne zanke nima, vendar nas to ne boli preveč, saj je možno tudi v situacijah, ko bi jo res potrebovali, brez večjih muk shajati brez nje.)

Znotraj zanke poiščemo tisti kraj in njegove koordinate, za katerega velja, da je smer do njega čimbolj poravnana z zadnjo prepotovano smerjo. Zadnja prepotovana smer je `px`, `py`, novo smer pa dobimo tako, da od koordinate posamičnega kraja odštejemo trenutne koordinate `tx`, `ty`. Zdaj bomo uporabili `max(koordinate)`, ključ pa je `lambda`, ki prejema argumente `k`, ki so terke `(kraj, (x, y))`. Do koordinat torej pridemo s `k[1][0]` in `k[1][1]`, kar ni najbolj pregledno, je pa žal neizogibno, če hočemo tole rešiti z lambdo.

Ko imamo naslednji kraj (`kraj`) in njegove koordinate `(nx, ny)`, izračunamo novo trenutno smer (`px`, `py`), shranimo nove koordinate kot trenutne koordinate in dodamo kraj v elastiko.

To je vse:

[7]:
```
kraj, (tx, ty) = min(koordinate, key=lambda t: t[1])
px, py = 0, 1

elastika = [kraj]
while len(elastika) == 1 or elastika[0] != elastika[-1]:
    kraj, (nx, ny) = max(koordinate, key=lambda k: skalp(px, py, k[1][0] - tx,␣
 ↪k[1][1] - ty))
    px, py = nx - tx, ny - ty
    tx, ty = nx, ny
    elastika.append(kraj)
```

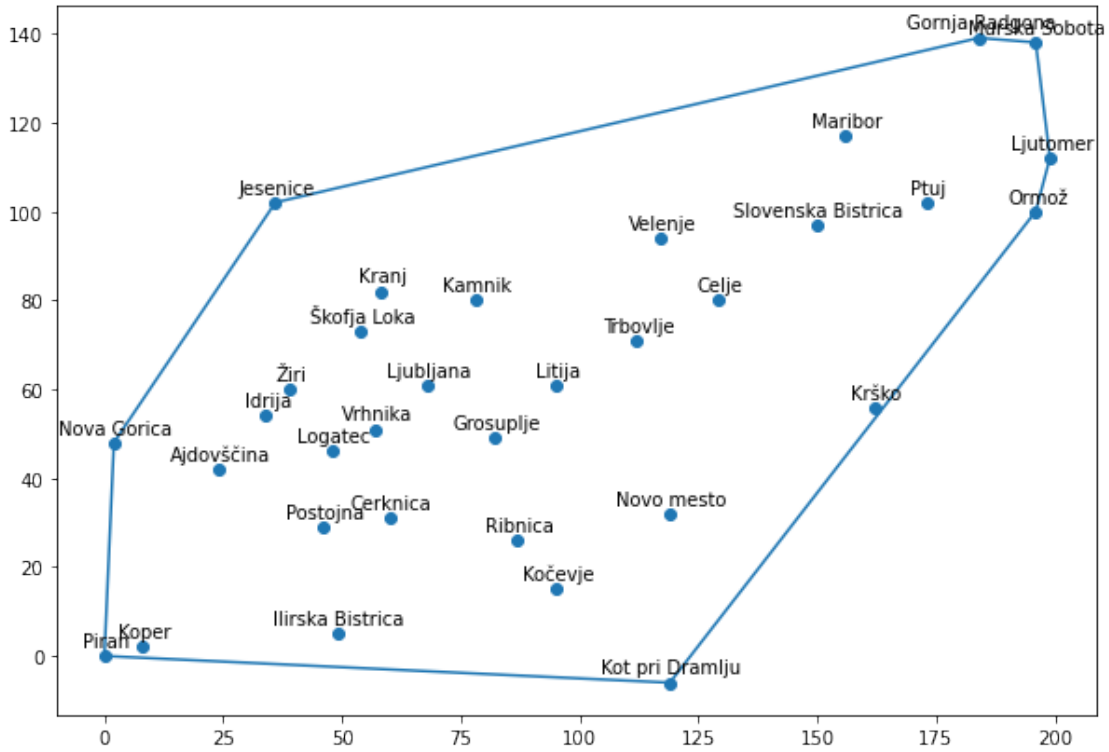
(Je kdo opazil, da vektor `(px, py)` v resnici ni normiran? Skalarni produkt pa misli, da bo? Nič hudega. Vsi skalarni produkti so zato preveliki, vendar so preveliki za enak faktor. Ker jih uporabljamo za urejanje kandidatov, zaradi tega ne bo nič narobe.)

Narišimo, da vidimo, da je rešitev pravilna.

Mimogrede strpamo koordinate v slovar, da bomo elastiko narisali tako, da bomo nabrali vse koordinate `x` in `y` za vsak kraj v `elastika`.

[8]:
```
zemljevid()

koord = dict(koordinate)
plt.plot([koord[ime][0] for ime in elastika], [koord[ime][1] for ime in␣
 ↪elastika])
```

[8]:
```
[<matplotlib.lines.Line2D at 0x7f9800097040>]
```



0.2.1 Razmislek o rešitvi

Rešitev je kar preprosta in očitna, ni pa hitra. Recimo, da elastika vsebuje k krajev, vseh krajev pa je n. Rešitev tedaj izračuna kn skalarnih produktov. V najslabšem primeru so v elastiki vsi kraji; v tem primeru rešitev izračuna n^2 skalarnih produktov. Če vzamemo desetkrat več krajev, bo trajalo stokrat dlje. To ni najhujša katastrofa, ki se lahko zgodi, obstajajo tudi hujši problemi s počasnejšimi rešitvami. Vseeno pa raje naredimo hitreje, če gre. In gre.

0.3 Hitrejša rešitev: postopno odstranjevanje

Spet izberimo najvzhodnejši kraj; če bi bila takšna kraja dva, bomo vzeli najjužnejšega. Vse kraj uredimo po kotu, ki ga oklepajo z navpično. Z drugimi besedami, razvrstimo jih v smeri urinega kazalca, ki bi bil pripet v najvzhodnejši kraj. Na začetek in konec prilepimo še ta kraj.

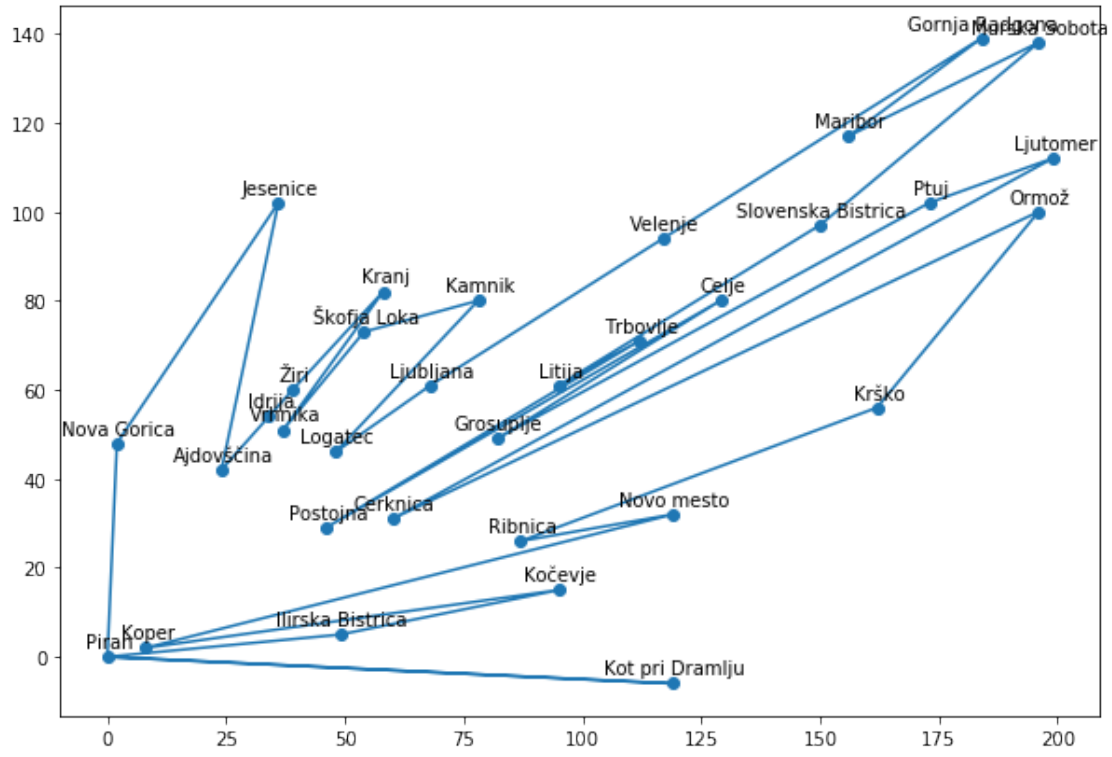
[6]:
```
kraj0, (x0, y0) = el0 = min(koordinate, key=lambda t: t[1])

elastika = [el0] \
           + sorted(koordinate, key=lambda k: (k[1][1] - y0) / sqrt((k[1][0] -␣
 ↪x0)** 2 + (k[1][1] - y0) ** 2 or 1)) \
           + [el0]
```

Ključ za funkcijo `sorted` je skalarni produkt z navpičnico.

Zakaj to počnemo, bomo videli, če pogledamo, kaj smo dobili.

[7]:
```
zemljevid()

plt.plot([koord[ime][0] for ime, _ in elastika], [koord[ime][1] for ime, _ in␣
 ↪elastika])
```

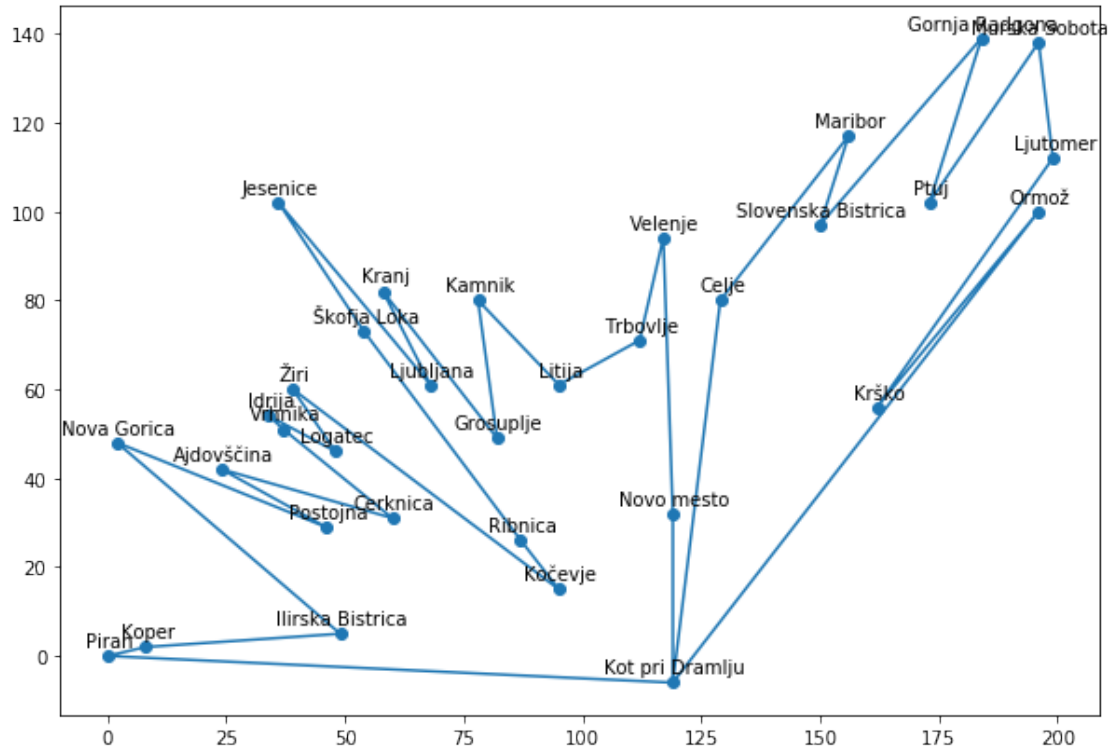
[7]: [<matplotlib.lines.Line2D at 0x7fb0020ef910>]



Namesto najvzhodnejšega bi lahko vzeli tudi najjužnejši kraj in kraje razvrščali glede na kot z vodoravnico.

[8]:
```
kraj0, (x0, y0) = el0 = min(koordinate, key=lambda t: t[1][::-1])

elastika = [el0] \
           + sorted(koordinate, key=lambda k: (k[1][0] - x0) / sqrt((k[1][0] -␣
 ↪x0)** 2 + (k[1][1] - y0) ** 2 or 1)) \
           + [el0]

zemljevid()

plt.plot([koord[ime][0] for ime, _ in elastika], [koord[ime][1] for ime, _ in␣
 ↪elastika])
```

[8]: [<matplotlib.lines.Line2D at 0x7fb002287e50>]



Pravzaprav bi lahko vzeli poljuben kraj ali, recimo, težišče krajev ali kako drugo točko in razvrščali glede na kote. Vendar ostanimo pri najbolj južnem, ker je to najbolj praktično, pa tudi slika je kar lepa.

Kar smo dobili, je neka še kar pametno napeta elastika, ki pa vsebuje veliko odvečnih krajev.

Če gremo po vrsti po teh krajih - Piran, Koper, Ilirska Bistrica, Nova Gorica … vemo, da bomo vedno zavijali samo desno, nikoli levo. To bo recept za odstranjevanje krajev.

Začnemo s Piranom.

- Dodamo Koper. Vse je v redu.
- Dodamo Ilirsko Bistrico. V Kopru je desni ovinek in to je OK. Dodamo Novo Gorico. Ozremo se nazaj in vidimo, da smo v Ilirski Bistrici zato naredili levi ovinek. To je prepovedano: Ilirsko Bistrico odstranimo. Zdaj se ozremo iz Nove Gorice v Koper in vidimo, da smo tudi tam naredili levi ovinek. Odstranimo še Koper. Pirana ne moremo odstraniti.
- Dodamo Postojno; imamo torej Piran - Nova Gorica - Postojna. Iz Postojne se ozremo v Novo Gorico. Tam je desni ovinek, vse je OK.
- Dodamo Ajdovščino. Ozremo se v Postojno; tam je levi ovinek: Postojno odstranino. Ozremo se iz Ajdovščine v Novo Gorico; tam je desni ovinek in to je OK. Nazaj nam ni potrebno preverjati.
- Dodamo Cerknico. Iz Cerknice se ozremo v Ajdovščino (za katero je Nova Gorica). Ajdovščina je ravno toliko izmaknjena, da jo pustimo pri miru.
- Dodamo Vrhniko. Iz nje se ozremo v Cerknico (in najprej v Ajdovščino). V Cerknici je levi ovinek; ven z njo. Zdja pogledamo iz Vrhnike če Ajdovščino v Novo Gorico: spet levi ovinek,

Ajdovščina gre ven.

In tako naprej.

Kako vemo, ali je ovinek levi ali desni? Prej smo imeli skalarni produkt, za določanje orientacije potrebujemo vektorskega. Vedno bomo opazovali zadnje tri točke; njihove koordinate naj bodo `(ax, ay)`, `(bx, by)` in `(cx, cy)`. Vektor iz prve v drugo je `(bx - ax, by - ay)` in vektor iz druge v tretjo je `(cx - bx, cy - by)`. Zanima nas, v kakšno smer je potrebno zavrteti prvi vektor, da dobimo drugega. Če je predznak vektorskega produkta pozitiven, gre za vrtenje nasprotno smeri urinega kazalca (*counter clockwise*, *ccw*) - to je levi ovinek in to je tisto, česar ne maramo.

Sprogramirajmo funkcijo, ki vrne `True`, če so točke a, b, c razporejene tako, da je ovinek, ki ga naredimo v b, levi.

[9]:
```
def ccw(ax, ay, bx, by, cx, cy):
    return (bx - ax) * (cy - by) - (cx - bx) * (by - ay) >= 0
```

Vse je pripravljeno. Sprogramirajmo. Dodali bomo še nekaj `print`-ov, da bo lažje slediti.

`i` bo indeks v `elastiko`. Vse točke do `i`-te so "pogojno sprejete*. V vsakem krogu zanke povečamo `i`, s čimer sprejmemo novo točko. Potem kličemo `ccw` s koordinatami točk z indeksi `i`, `i - 1` in `i - 2`. Če tvorijo levi ovinek, odstranimo točko `i - 1`; zaradi praktičnosti to storimo tako, da zmanjšamo `i` in pobrišemo `i`-to. Tudi odstranjevanje je v zanki, saj lahko v istem koraku odstranimo več točk.

Funkciji `ccw` moramo poslati šest argumentov, ki jih dobimo v treh terkah. Klic poenostavimo tako, da damo pred terko zvezdico in Python bo vse elemente terke uporabil kot ločene argumente.

[10]:
```
kraj0, (x0, y0) = el0 = min(koordinate, key=lambda t: t[1][::-1])

elastika = [el0] \
           + sorted(koordinate, key=lambda k: (k[1][0] - x0) / sqrt((k[1][0] -␣
 ↪x0)** 2 + (k[1][1] - y0) ** 2 or 1)) \
           + [el0]

print("Imam", elastika[0][0])
i = 1
while i < len(elastika):
    print("Dodal sem", elastika[i][0])
    while i > 1 and ccw(*elastika[i - 2][1], *elastika[i - 1][1],␣
 ↪*elastika[i][1]):
        i -= 1
        print("Odstranim", elastika[i][0])
        del elastika[i]
    i += 1
```

```
Imam Kot pri Dramlju
Dodal sem Piran
Dodal sem Koper
Dodal sem Ilirska Bistrica
Dodal sem Nova Gorica
```

```
Odstranim Ilirska Bistrica
Odstranim Koper
Dodal sem Postojna
Dodal sem Ajdovščina
Odstranim Postojna
Dodal sem Cerknica
Dodal sem Vrhnika
Odstranim Cerknica
Odstranim Ajdovščina
Dodal sem Idrija
Odstranim Vrhnika
Dodal sem Logatec
Dodal sem Žiri
Odstranim Logatec
Odstranim Idrija
Dodal sem Kočevje
Dodal sem Ribnica
Odstranim Kočevje
Dodal sem Škofja Loka
Odstranim Ribnica
Odstranim Žiri
Dodal sem Jesenice
Odstranim Škofja Loka
Dodal sem Ljubljana
Dodal sem Kranj
Odstranim Ljubljana
Dodal sem Grosuplje
Dodal sem Kamnik
Odstranim Grosuplje
Odstranim Kranj
Dodal sem Litija
Dodal sem Trbovlje
Odstranim Litija
Odstranim Kamnik
Dodal sem Velenje
Odstranim Trbovlje
Dodal sem Novo mesto
Dodal sem Kot pri Dramlju
Dodal sem Celje
Odstranim Kot pri Dramlju
Odstranim Novo mesto
Dodal sem Maribor
Odstranim Celje
Odstranim Velenje
Dodal sem Slovenska Bistrica
Dodal sem Gornja Radgona
Odstranim Slovenska Bistrica
Odstranim Maribor
```

```
Dodal sem Ptuj
Dodal sem Murska Sobota
Odstranim Ptuj
Dodal sem Ljutomer
Dodal sem Krško
Dodal sem Ormož
Odstranim Krško
Dodal sem Kot pri Dramlju
```

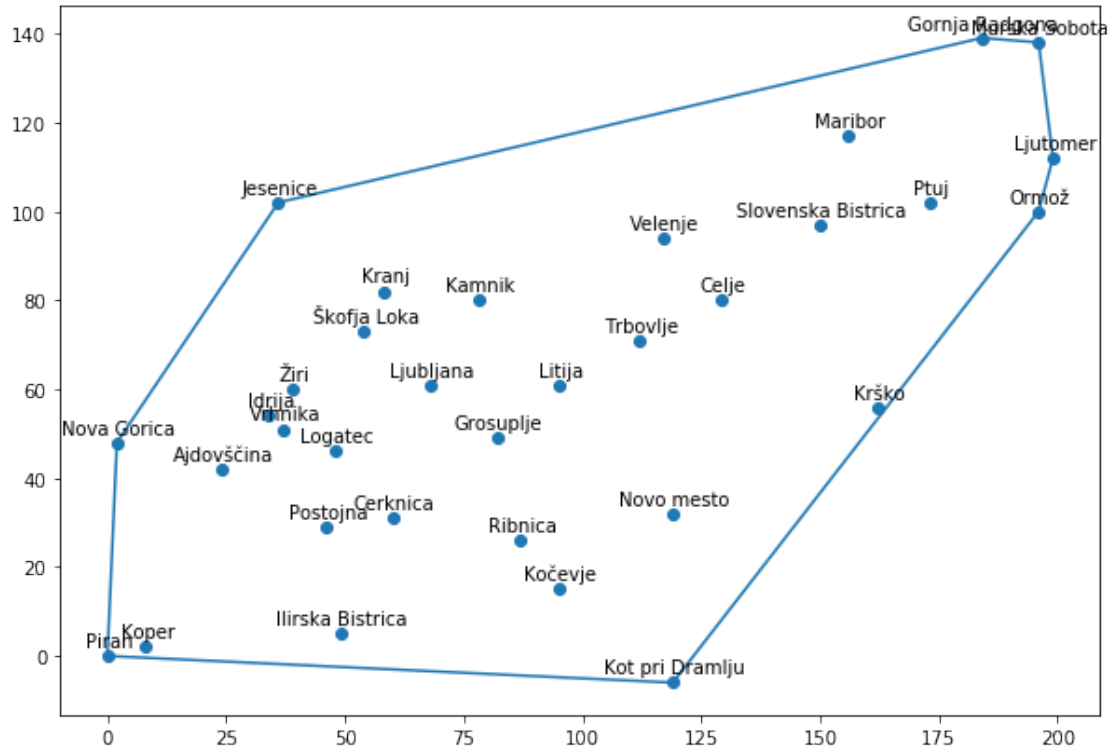
Pa še brez `print`-ov, da bo očitneje, kako preprosto je vse skupaj (ko enkrat veš, kaj bi rad sprogramiral).

[11]:
```
kraj0, (x0, y0) = el0 = min(koordinate, key=lambda t: t[1][::-1])

elastika = [el0] \
           + sorted(koordinate, key=lambda k: (k[1][0] - x0) / sqrt((k[1][0] -␣
 ↪x0)** 2 + (k[1][1] - y0) ** 2 or 1)) \
           + [el0]

i = 1
while i < len(elastika):
    while i > 1 and ccw(*elastika[i - 2][1], *elastika[i - 1][1],␣
 ↪*elastika[i][1]):
        i -= 1
        del elastika[i]
    i += 1
```

Narišimo, da vidimo, da res deluje.

[12]:
```
zemljevid()

plt.plot([koord[ime][0] for ime, _ in elastika], [koord[ime][1] for ime, _ in␣
 ↪elastika])
```

[12]:
```
[<matplotlib.lines.Line2D at 0x7fb0022733d0>]
```



0.3.1 Razmislek o rešitvi

Tale rešitev je bistveno hitrejša. Najpočasnejši del je urejanje: n točk je mogoče urediti v času sorazmernem $n \log n$, kar je bistveno manj od n^2.

Sledita jim dvojni zanki, kar bi na prvi pogled lahko imelo zahtevnost n^2, vendar ni tako: vsaka točka je lahko odstranjena le enkrat. Torej se bo notranja zanka skupno, v celotnem poteku algoritma, zavrtela največ tolikokrat, kolikor je točk.

Rešitve, ki so jih pošiljali študenti, so bili pogosto neka variacija te rešitve. Nekateri so napisali praktično enako rešitev, le na kak daljši način in z drugo začetno točko; tipično je bila to neka točka v sredini, na primer težišče. Drug pogost pristop je bilo, da so vzeli najbolj levi in najbolj desni kraj, ostale pa razdelili na tiste, ki so nad in pod povezavo med njima. Kraje so uredili po koordinati x in nato potovali po zgornji strani in izločali kraje, ki so bili med dvema višjima sosedoma, nato pa po spodnji strani in izločali kraje med dvema nižjima. Še druge variacije rešitve so razdelile zemljevid v štiri kvadrante.

Vse to je dobro in je na poti do te, optimalne rešitve.

0.4 Razmislek o nalogi - kaj smo počeli?

Problem, ki smo ga reševali, se imenuje iskanje *konveksna ogrinjača* (convex hull). Le-ta je, skupaj z nekaj sorodnimi problemi, pomembna reč v matematiki in računski geometriji. Konveksno ovojnice lahko iščemo tudi v tridimenzionalnem ali v kakem višje- dimenzionalnem prostoru, kjer so stvari

seveda bolj zapletene. Takšnole preprosto urejanje po kotu si lahko privoščimo le na ravnini. Algoritem, ki smo ga sestavili, je Grahamovo preiskovanje.

Funkcijo, ki smo jo poimenovali `ccw`, običajno imenujemo `ccw` in je vlečni konj računske geometrije. Za primer: zanima nas, ali se daljici AB in CD sekata. Najzanesljivejši način, da to preverimo je tale: orientaciji (ccw) trojke ABC in ABD morata biti različni, prav tako morata biti različni orientaciji trojk CDA in CDB. Z drugimi besedami, C in D morata biti na različnih straneh AB, A in B pa na različnih straneh CD.