Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Oliver Leontiev

Zadanie 4b

Klastrovanie

Predmet: Umelá inteligencia
Cvičiaci: Ing. Ivan Kapustík

ZS 2020/2021

Zadanie

Našou úlohou je implementovať 4 klastrovacie algoritmy na zhlukovanie 20020 bodov v 2D priestore. Tieto algoritmy máme porovnať a výsledky zhlukovania graficky znázorniť. Za úspešný zhlukovač považujeme taký, v ktorom žiadny klaster nemá priemernú vzdialenosť bodov od stredu klastra väčšiu ako 500.

Algoritmy, ktoré máme implementovať sú:

1. K-means (centroid)
2. K-means (medoid)
3. Divízne zhlukovanie
4. Aglomeratívne zhlukovanie

Implementácia

Program je konzolová aplikácia, ktorá bola vytvorená v programovacom jazyku Python 3. Spúšťa sa súbor *Main.py* a to štandardne:

$ python *Main.py*

Na grafické zobrazenie bola použitá knižnica/nástroj **bokeh**: https://bokeh.org/

Štruktúra

Program je rozdelený do dvoch súborov: *Main.py* a *Cluster.py*

Main.py obsahuje všetky funkcie s algoritmami na samotné zhlukovanie, zatiaľ čo *Cluster.py* obsahuje triedu Cluster:

```
class Cluster:
    def __init__(self, ref_point, xs, ys, changed=False):
        self.ref_point = ref_point
        self.xs = xs
        self.ys = ys
        self.changed = changed
```

Cluster si drží informáciu o svojom aktuálnom **referenčnom bode** (medoid alebo centroid) a v poliach **xs** a **ys** o bodoch patriacich do klastra. Atribút **changed** je pomôcka - referenčný bod treba prepočítať iba ak sa v klastri niečo zmenilo.

Súradnice boda *i* sú xs[*i*], ys[*i*]. Podobne sú body reprezentované aj vo zvyšku programu (dve polia, jedno pre každú os)

V *Main.py* sú body v dvojrozmernom poli *points*, v ktorom na indexe 0 sú všetky x-ové súradnice a na indexe 1 všetky y-ové súradnice. Bod *i* má potom súradnice points[0][*i*], points[1][*i*].

K-means algoritmus

Tento algoritmus vytvára klastre na základe prideľovania bodov k najbližším referenčným bodom. Tieto referenčné body sa neustále prepočítavajú podľa bodov pridelených klastru a algoritmus končí ak sa všetky body ustália. Podľa spôsobu výpočtu referenčného bodu máme variantu s centroidmi a medoidmi.

Postup:

1. Inicializácia

Náhodne sa vyberie *k* referenčných bodov a všetky body určené na zhlukovanie sa rozdelia medzi tieto ref. body podľa blízkosti. Tým sa vytvorí prvých *k* klastrov.

2. Prepočítavanie a vymieňanie bodov

Následne sa pre každý klaster vypočíta nový referenčný bod na základe bodov, ktoré do klastra patria. Potom všetky body, ktoré sú bližšie k referenčnému bodu iného klastra ako toho, do ktorého aktuálne patria, sa priradia do bližšieho klastra.

Toto sa opakuje, kým sa aj po prepočítaní ref. bodov žiadne zmeny neudejú - žiadny bod sa nepremiestni.

3. Koniec

Takýmto spôsobom získame zaručene k-klastrov (alebo menej).

Centroid

Centroid pre klaster získame ako bod, ktorého súradnice sú priemerom súradníc bodov v klastri.

```
def calculate_centroid(self):
    if not self.xs:
        return self.ref_point[0], self.ref_point[1]
    amount = len(self.xs)
    x_total = 0
    y_total = 0
    for i in range(0, amount):
        x_total += self.xs[i]
        y_total += self.ys[i]
    return x_total / amount, y_total / amount
```

Centroid je teda bod v priestore a nie jeden z bodov zo vzorky. Z toho vyplýva, že niektoré centroidy môžu byť na začiatku tak nešťastne vygenerované, že k nim žiadny bod nie je bližšie ako k inému centroidu. Takto nám môžu vzniknúť prázdne klastre a ultimátne počet klastrov menší ako *k*.

Medoid

Medoid je vždy jeden z bodov z našej vzorky. Vypočítame ho ako bod, ktorého súčet vzdialeností k ostatným bodom v klastri je najmenší.

```
def calculate_medoid(self):
    amount = len(self.xs)
    min_distance = math.inf
    best = None
    for i in range(0, amount):
        curr_dist = 0
        for j in range(0, amount):
            curr_dist += distance(self.xs[i], self.ys[i],
                                  self.xs[j], self.ys[j])
        if curr_dist < min_distance:
            min_distance = curr_dist
            best = (self.xs[i], self.ys[i])
    return best
```

Vypočítať centroid je časovo menej náročné ako vypočítať medoid - **O(n)**. Medoid nám zaručuje *k* klastrov (keďže do klastra vždy bude patriť minimálne samotný medoid) a zároveň je menej citlivý na body, ktoré sú priveľmi vzdialené od svojho klastra. Jeho výpočet je však oveľa zložitejší - **O(n²)**.

Optimalizačné prvky

Pre k-means algoritmy sme implementovali **dve optimalizácie**.

1. **Minimálnu vzdialenosť medzi prvými náhodnými referenčnými bodmi (1500).** Vďaka tomu zamedzíme klastrom rozdeleným na dve polovice (vygenerujú sa dva referenčné body do jedného zhluku). Táto optimalizácia má väčší význam pre medoidy, keďže centroidy sa môžu vygenerovať dostatočne ďaleko od seba, ale stále v blízkosti toho istého zhluku a nakoniec byť "pritiahnuté" k sebe.

2. **Prepočet referenčného bodu iba ak sa klaster zmenil.** Pokiaľ si nejaké klastre vymenia body tak sa oba označia za "zmenené". Pri prepočte referenčných bodov sa potom prepočítavajú iba zmenené klastre.

```
def new_ref(self, type):
    if self.changed:
        if type == "centroid":
            return self.calculate_centroid()
        if type == "medoid":
            return self.calculate_medoid()
    else: return self.ref_point[0], self.ref_point[1]
```

Divízne zhlukovanie

Pri implementovaní divízneho zhlukovania sa opierame o **k-means zhlukovanie okolo centroidov**.

Začíname s dvoma veľkými klastrami a postupne každý klaster delíme na dva menšie. Pokiaľ klaster spĺňa podmienku úspešnosti (priemerná vzdialenosť bodov od stredu je menšia ako 500), tak ho nedelíme.

Pokiaľ všetky klastre spĺňajú podmienku úspešnosti alebo máme *k* klastrov, tak končíme.

```
def divisive(k, points):
    clusters = k_means(2, points, "centroid", True)
    is_success = False
    while not is_success:
        clust_am = len(clusters)
        if clust_am >= k:
            break
        new_clusters = []
        is_success = True
        for cluster in clusters:
            if cluster.average_dist() > 500 and clust_am < k:
                is_success = False
                new_points = [cluster.xs, cluster.ys]
                tmp = k_means(2, new_points, "centroid", True)
                clust_am += len(tmp) - 1
                for tmp_cluster in tmp:
                    new_clusters.append(tmp_cluster)
            else:
                new_clusters.append(cluster)
        clusters = new_clusters
    return clusters
```

Pri zadaní dostatočne veľkého *k* je divízne klastrovanie vždy 100% úspešné a taktiež je veľmi rýchle. Problém je, že klastre nie sú dobre odizolované, keďže o sebe všetky nevedia (delíme vždy iba podmnožinu bodov), a teda výsledok nie je ideálne pozhlukovaný - napriek tomu, že podmienka je splnená.

Aglomeratívne zhlukovanie

Implementovali sme variantu aglomeratívneho zhlukovania s názvom **SLINK**:
https://sites.cs.ucsb.edu/~veronika/MAE/summary_SLINK_Sibson72.pdf

Aglomeratívne zhlukovanie inicializuje **všetky body ako klastre**, obsahujúce iba samé seba. Následne sa opakovane zlučujú najbližšie klastre a počet klastrov nám postupne klesá (pri ostatných algoritmoch nám stúpal alebo bol rovnaký).

Aglomeratívne zhlukovanie je časovo veľmi problematické. **SLINK zvyšuje časovú efektívnosť** oproti naivnej implementácii z **O(n³)** na **O(n²)**. Používame na to myšlienku, že síce na zapamätanie vzdialeností medzi bodmi potrebujeme n x n maticu, tak najbližší bod pre každý bod si vieme zapamätať v jednorozmernom poli. Na indexe *i* bude číslo *j* – to znamená že bod *i* má najbližšie k bodu *j*. Túto najkratšiu vzdialenosť si môžeme udržiavať v ďalšom jednorozmernom poli, kde na indexe *i* je vzdialenosť medzi *i* a *j*, a teda najkratšia vzdialenosť od bodu i k ďalšiemu bodu.

Pole indexov bodov označíme **Av** a pole najkratších vzdialeností označíme **Ad**. Po každom zlúčení vymažeme odstránení klaster z matice a aktualizujeme hodnoty v týchto dvoch poliach. Zostrojiť maticu nám stále trvá O(n²), ale pri každom opakovaní už sledujeme iba riadky a stĺpce matice, ktoré sa týkajú práve zlúčených klastrov a naše dve polia, v ktorých vieme nájsť ďalšiu najmenšiu vzdialenosť na zlúčenie v O(n).

```
def aglomerative(k, points):
    clusters, matrix, Av, Ad, a_min = make_matrix(points)
    is_success = False
    while not is_success:
        v1 = min(a_min, Av[a_min])
        v2 = max(a_min, Av[a_min])
        clusters[v1].merge(clusters[v2])
        clusters.pop(v2)
        matrix, min_new_d = recalc_matrix(matrix, v1, v2)
        Av[v1] = min_new_d[0]
        Ad[v1] = min_new_d[1]
        Av = np.delete(Av, obj=v2)
        Ad = np.delete(Ad, obj=v2)
        min_dist = 500000000
        a_min = -1
        for i in range(0, len(Av)):
            if Av[i] == v2:
                Av[i] = v1
            if Av[i] > v2:
                Av[i] -= 1
            if Ad[i] < min_dist:
                a_min = i
                min_dist = Ad[i]

        if len(clusters) <= k or clusters[v1].average_dist() >= 500:
            is_success = True

    return clusters
```

Končíme, keď dosiahneme *k* klastrov alebo nejaký klaster prestane spĺňať podmienku úspešnosti (na začiatku ju spĺňajú všetky).

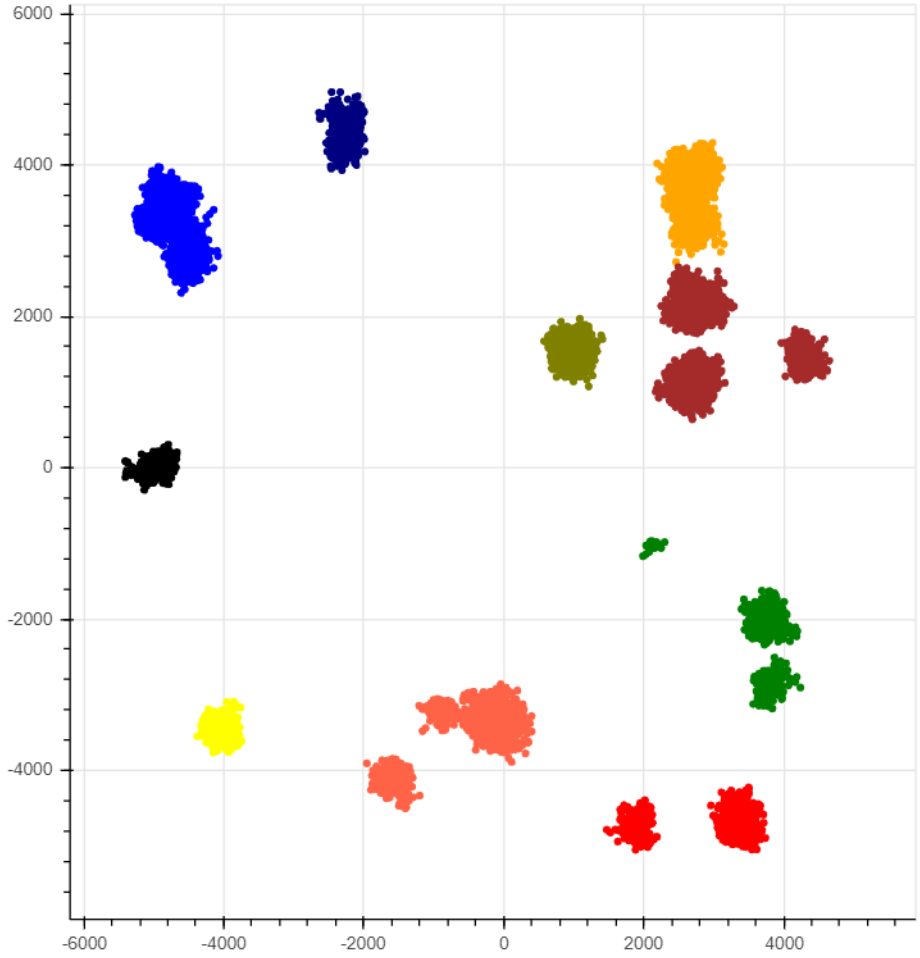
Podrobné vysvetlenie SLINK algoritmu je vo vyššie priloženom dokumente, z ktorého sa vychádzalo pri jeho implementácii.

Je na mieste spomenúť, že síce vykonanie tohto algoritmu pre **20020 bodov trvalo 2,5 hodiny**, tak má bezkonkurenčne **najlepšie výsledky**, čo sa týka vizuálne správneho zhlukovania.

Ukážky výsledkov všetkých algoritmov

K-means centroid

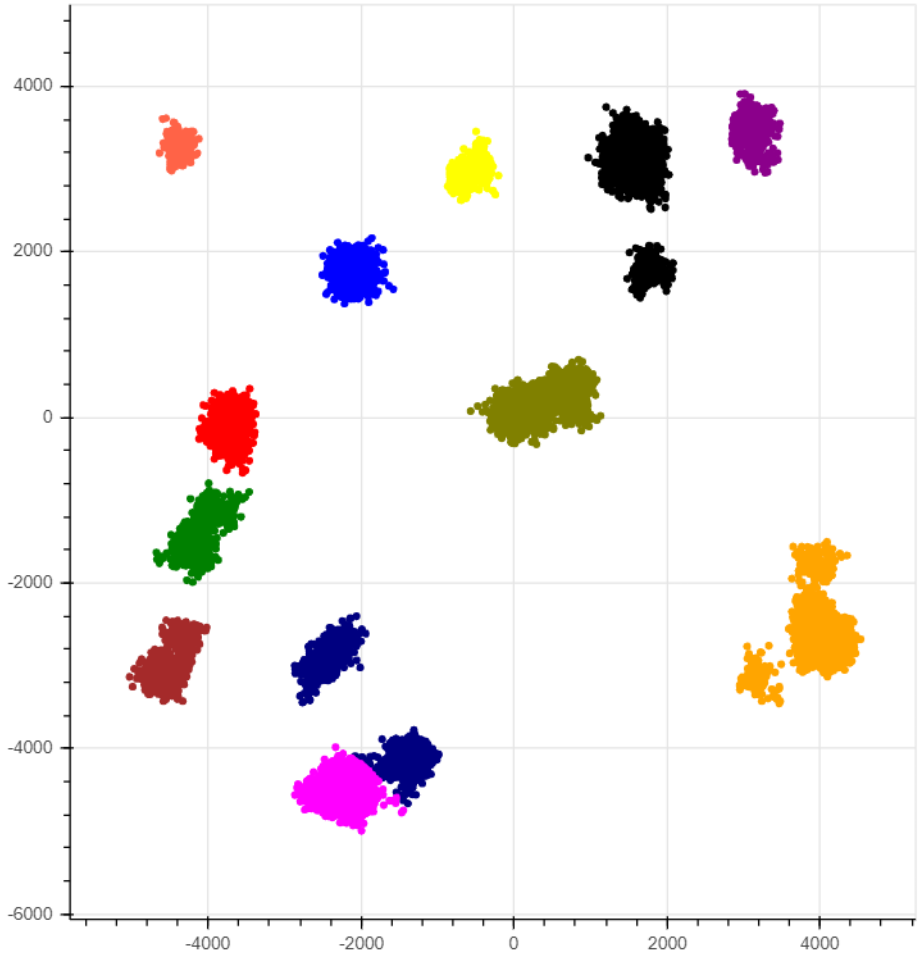
k = 12




```
10 clusters made
0.6500487327575684 seconds
90.0% success rate
```

K-means medoid

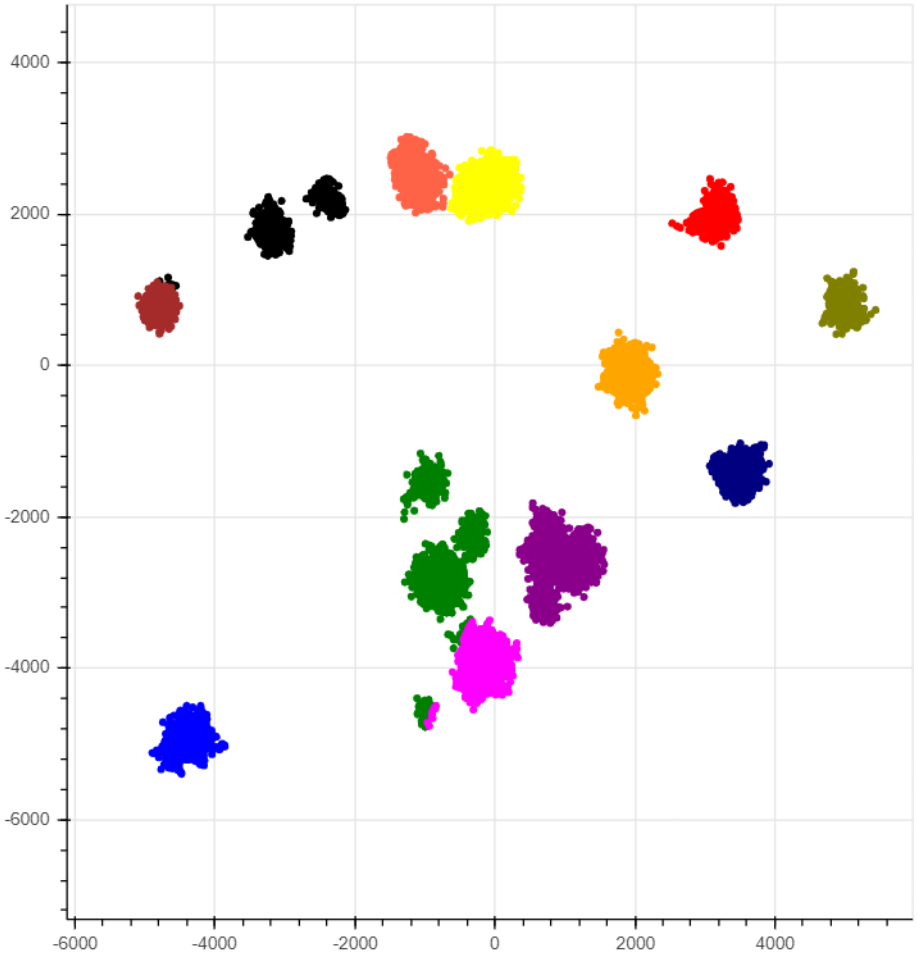
k = 12




```
12 clusters made
69.87345767021179 seconds
91.66666666666666% success rate
```

Divízne

Algoritmus bežal, kým všetky klastre nesplnili podmienku úspešnosti.

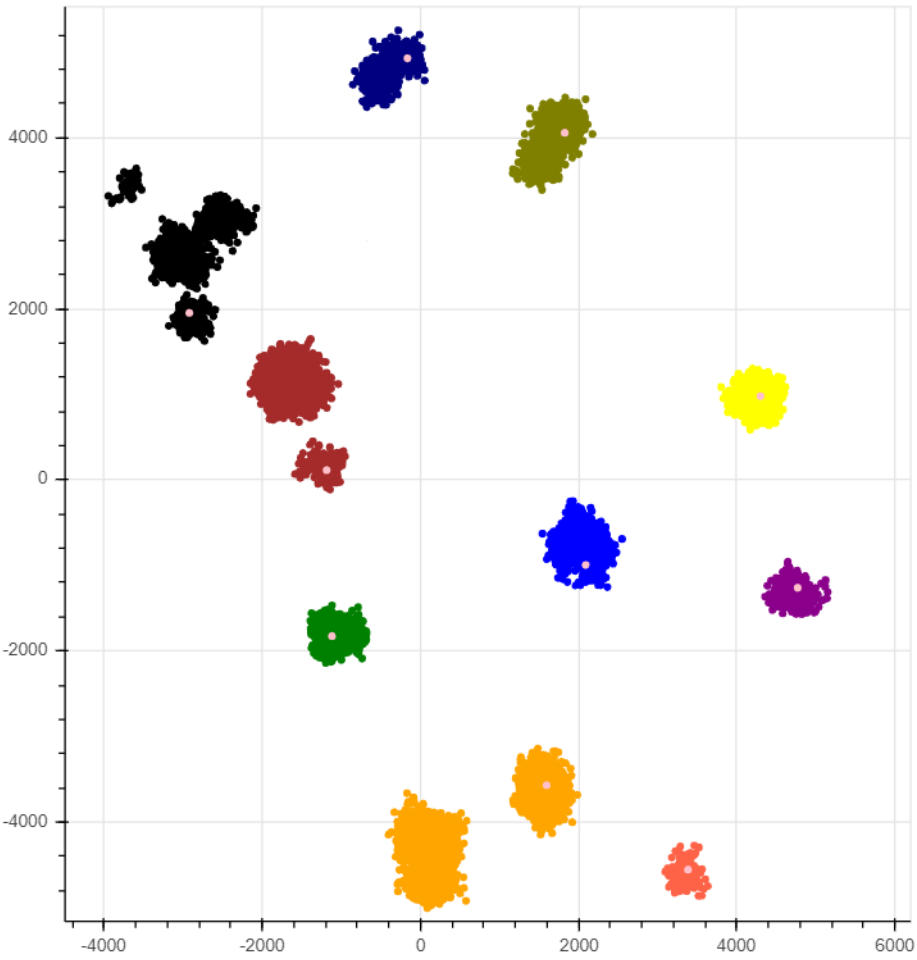



```
12 clusters made
1.2744357585906982 seconds
100.0% success rate
```

Aglomeratívne

Výsledok pre 20020 bodov a algoritmus bežal, kým nejaký klaster neprekročil podmienku úspešnosti.

* ružovou sú prvé inicializované body v rámci klastra




```
10 clusters made
9587.7647399902
90.0% success rate
```

Testovanie a zhodnotenie

Z testovania máme odpozorované, že optimálny počet klastrov pre 20020 bodov sa pohybuje okolo 12 – 15.

Tabuľka úspešnosti a času vykonania

Aglomeratívne bolo testované na menšej vzorke - iba 2020 bodov.

k algoritmus	8	10	12	15	20	25
k-means (centroid)	**50 % 2,3 sek**	**80 % 1,5 sek**	**72 % 1 sek**	**86 % 4,8 sek**	**100 % 4,6 sek**	**94 % 3,9 sek**
k-means (medoid)	**50 % 132 sek**	**60 % 118 sek**	**100 % 126 sek**	**87 % 50 sek**	**100 % 141 sek**	**92 % 110 sek**
divízne	**37,5 % 0,7 sek**	**60 % 1,2 sek**	**91 % 1,2 sek**	**100 % 1,3 sek**	**100 % 1,2 sek**	**100 % 1,3**
aglomeratívne	**92,9 % 19 sek**	**91.7 % 19 sek**	**92,8 % 19,5 sek**	**100 % 20,2 sek**	**100 % 21.4 sek**	**100 % 19,8 sek**

Z nášho testovania sme odpozorovali, že všetky štyri algoritmy sa správajú veľmi odlišne. **Najvyššiu úspešnosť vykazuje aglomeratívne** a (pri vyšších *k*) divízne. To je spôsobené najmä postupným menením počtu a veľkosti klastrov, čo nám umožňuje ustrážiť si podmienku úspešnosti v priebehu algoritmu. **Divízne je** taktiež **časovo najefektívnejšie**. K-means algoritmy sú menej konzistentné, čo sa týka úspešnosti, keďže záležia veľmi od zadaného *k*. S narastajúcim *k* sú veľmi úspešné oba. **Vizuálne podáva lepšie výsledky k-medoid** a zdá sa byť o niečo úspešnejší aj podľa testov. **K-medoid je rádovo pomalší ako k-centroid** a jeho úspešnosť nie je dostatočne lepšia, aby to stálo za ten časový rozdiel. **Aglomeratívne je časovo najmenej efektívne** - pre 20 020 bodov algoritmus bežal 2,5 hodiny.