Haladó algorimtusok, beadandó dokumentáció

Seres Richárd Sándor (DOOHWJ)

Tartalomjegyzék

1. Feladatok

A három feladat, amelyet elkészítettem az alábbiak voltak:

1. Képszegmentálás – Kmeans algoritmussal
2. Gócpontkeresés (GPS adatok szűrése) –DBScan algoritmussal
3. Poligonközelítés – Hillclimber módszerrel

2. Képszegmentálás

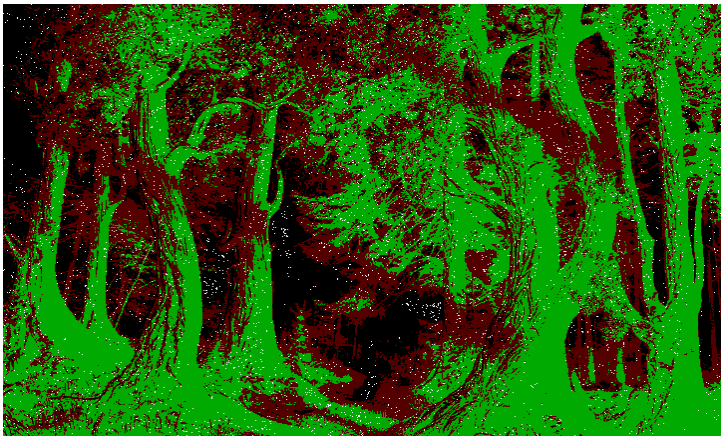
A projekt helye: /kepszegmentalas-kmeans/

A program beolvas egy tetszőleges JPG képet grayscaleli azt. Ezután a kódban megadott számú clusterrel elindul a Kmeans algoritmus.

A Logics.cs-ben található:

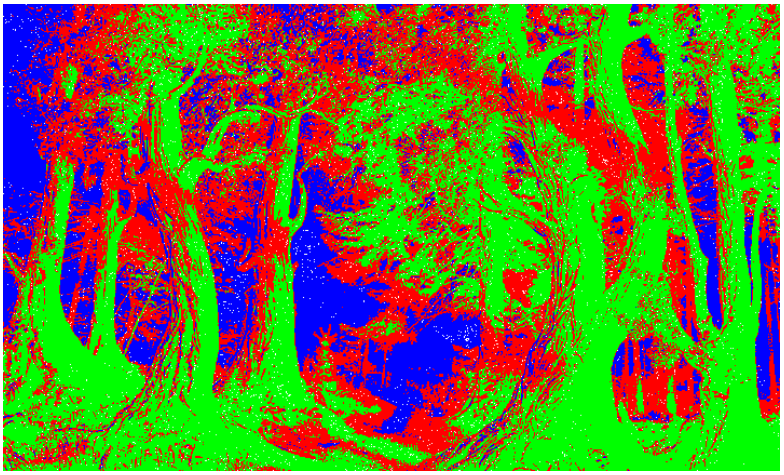
- **int[] hisztogram2** :[256] méretű tömb, itt tárolom az grayscalelt értékeket
- **void GrayScale(Bitmap img)**: grayscaleli a kapott képet, és elhelyezi a hisztogramba a kapott értékeket

- **int CalculateDistance(int a, int b):** visszatérési értékként a két pont értéke közti értéket adja meg
- **void KMeans(Bitmap img, int n_cluster, Bitmap output):** metódus, ahol megtörténik a clusterezés, centroidok random kijelölése, minden elem kijelölése egy random clusterbe, centroidok újraszámítása, oszcilálásra figyelni kell, klaszterek kiírása az output Bitmapbe
  (továbbbi kommentek a kódban)



Baloldalt az eredeti kép látható, jobboldalt a kimenet 3 cluster esetén.

A kimeneti színek közül sokfélét kipróbáltam, hogy a clusterek számának arányában látható legyen a különbség a különböző clusterek között, és egyben rugalmasan lehessen a clusterek darabszámát állítani. Három clusternél az alábbi eredmény látható, itt a kiíratásnál az RGB értékeket ennek megfelelően állítattam be.



3. Gócpontkeresés – GPS adatok szűrésére

A projekt helye: /dbscan_gocpontok/

A DBScan algoritmus alkalmazására kisebb gondolkodás után arra juttottam, hogy jól lehetne szemléltetni az algoritmus hasznosságát, GPS adatok szűrésére. A GPS adatok általában tartalmaznak zajokat, olyan jeleket, amelyek egyértelműen nem validak a többi jel mellett. Ezeknek a zajoknak a szűrésére alkalmaztam a DBScan algoritmust.

Adatok, amelyeket használtam: GeoLife Trajectories 1.3, ingyenesen elérhető

Adatok felépítése:

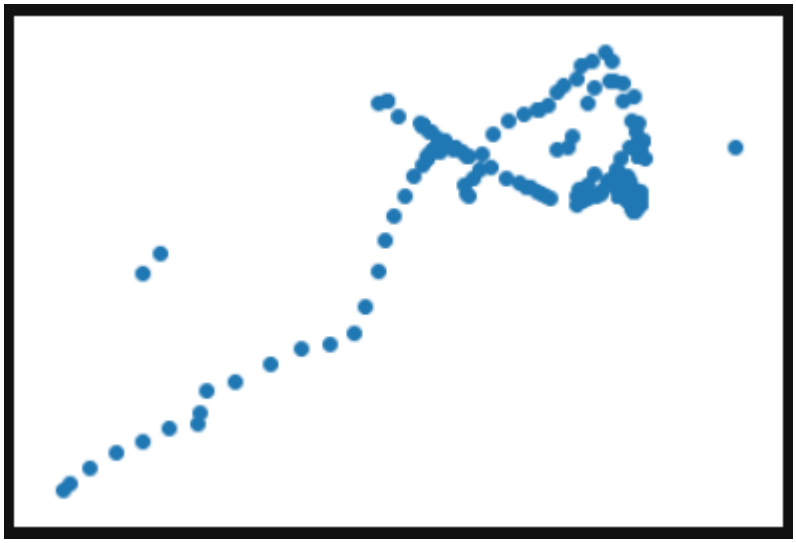
```
1  Geolife trajectory
2  WGS 84
3  Altitude is in Feet
4  Reserved 3
5  0,2,255,My Track,0,0,2,8421376
6  0
7  40.008304,116.319876,0,492,39745.0902662037,2008-10-24,02:09:59
8  40.008413,116.319962,0,491,39745.0903240741,2008-10-24,02:10:04
9  40.007171,116.319458,0,-46,39745.0903819444,2008-10-24,02:10:09
```

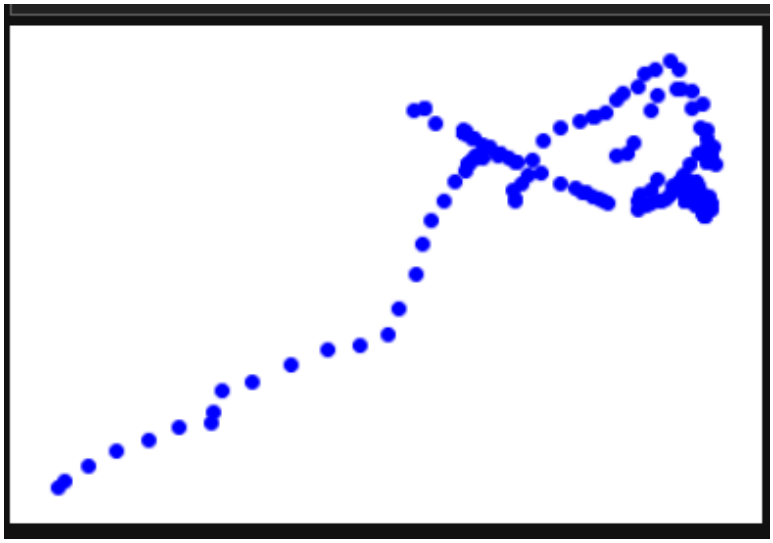
Az első hat sor irreleváns.
A további sorokban pedig az első kettő adattagot használom fel. Az első a hosszuság a második pedig a szélesség koordináták.

Kódbázis felépítése:

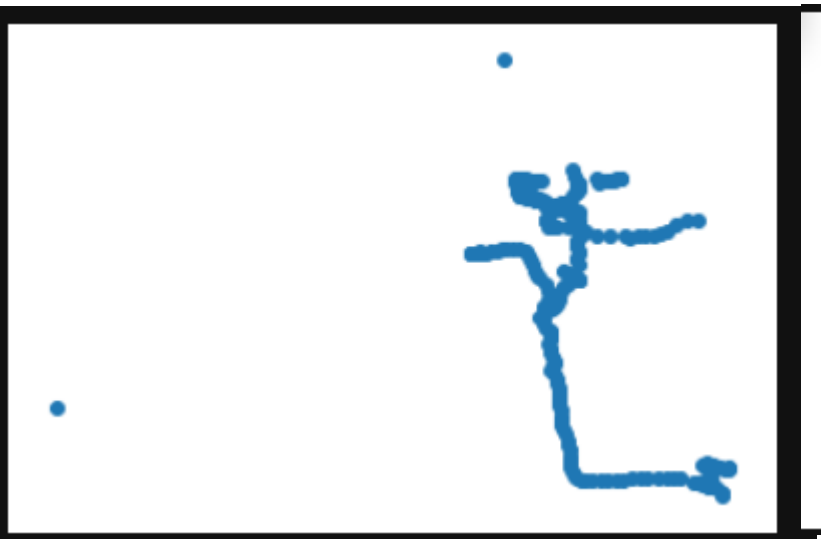
- Program.cs: meghívja a file beolvasást, elindítja a DBScant.
- Coordinate.cs: tartalmazza egy konstruktort, és a decimal x,y propertyket, illetve int key propertyt, amit ellenőrzéshez használtam, a kódban az x koordináta maradt az hosszúság és y a szélesség, de ez nem számít amíg jól jelenítjük meg az értékeket később
- Logic.cs: tartalmazza a DBScan metódushoz szükséges metódusokat
    - **void DBScan(List<Coordinate> points,int minPts, decimal e):** futtatja a DBScan metódust
        - points tartalmazza a feldolgozandó koordinátákat
        - List<List<Coordinate>> p_cluster: a clusterek listáját tartalmazó lista
        - List<Coordinate> processed_points: feldolgozott koordináták listája
        - A metódus feldolgozza a pontokat folyamatosan növeli a p_cluster listát és a végén kiírja ezeket különböző output[i].plt fileokba
    - **bool LegitCheck(List<List<Coordinate>> p_cluster):** tesztelés során duplikáció checkre használtam
    - **decimal CalculateDistance(Coordinate a, Coordinate b):** kiszámolja a két koordináta közti távolságot, visszaadja decimális értékben
- FileHandler.cs: Tartalmazza az adatok kiírásához szükséges függvényeket
    - **void ReadData(List<Coordinate> points,string path):** beolvas egy file-t és feltölti a megadott Coordinates típusú listát

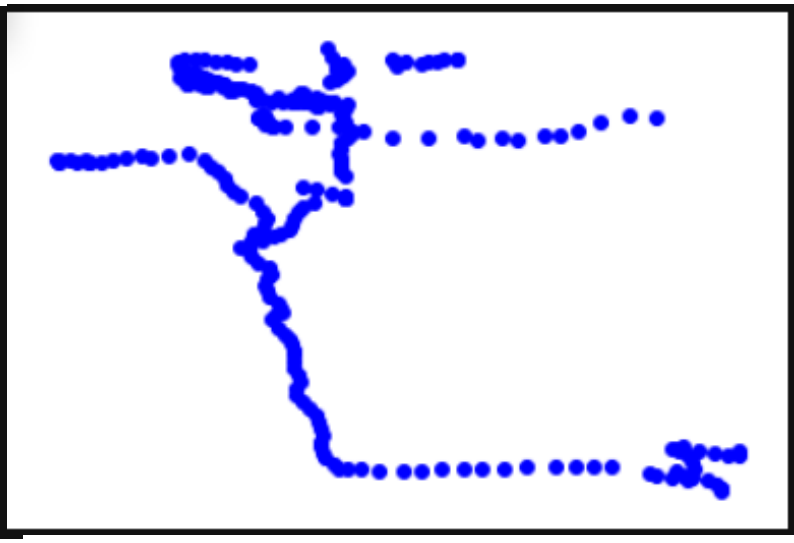
- **void WriteData(List<Coordinate> points,string path):** kiírja egy Coordinate lista tartalmát a megadott .plt fileba, ugyanazzal a szintaktussal mint ahogy a forrás fileban volt, az első 6 sor nélkül





Egy bemeneti file koordinátái a baloldali formát adják. Itt látható zaj a három látványosan nem a többivel lévő pont. Ennek a bemenetek a szűrése után az a baloldali koordinátákat kapjuk eredményül. Itt látható, hogy kiszűri a program a nem gócpontként megjelölt elemeket. A futás előtt megkell adni a”minPts” illetve az „e „ értéket.





Itt a baloldali képen ugyancsak látható két nem oda illő pont. A baloldali képen látható a szűrés utáni eredmény. Az arány itt meg is változott, hisz a kimenetben nem volt benne a maradék 2 pont, így azt nem is ábrázolta a matplot.

3.Poligonközelítés

A projekt helye: /polygon_kozelites/polygon

Harmadikként poligon közelítést választottam. Itt a stohastic hillclimber módszert használom a poligon közelítésére. A program beolvasssa az input.log file-t. Az input.log file tartalmazza a pontokat, amelyeket közelíteni szeretnénk. A program az output.log fileba irja ki a közelítés eredményét.

Kódbázis felépítése:

- Program.cs: az algoritmus futtatása, input, output fileok nevének megadása, a közelítő poligon csúcsainak a számának a megadása is itt történik.
- FileManagement.cs: tartalmazza a kiíratást és a beolvasást elvégző metódusokat
- Hillclimber.cs:
    - **float CheckDistance(Point p1, Point p2,Point p):**
        - kiszámolja a két pont közti távolságot, visszaadja floatként
    - **float OuterDistanceToBoundary(List<Point> solution,List<Point> points):**
        - minden egyes pontra kiszámolja, hogy milyen messze van a poligontol, ahogy a videóban volt, hasonlóan implementáltam
    - **float LengthOfBoundary(List<Point> solution):**
        - poligon oldalainak a hosszának a meghatározása itt történik
    - **static List<Point> HillClimber(List<Point> points,int p_count):**
        - maga a sztohasztikus hillclimber method itt fut
        - paraméterként megadjuk a poligon csúcsszámot és a pontokat, amiket közelítünk
        - először legenerálódik egy random megoldás, ez a közelítendő pontok közé egy nagy körben történik
        - penalty változót beállítjuk valamekkora értékre, erre azért van szükség hogy a fitness függvény büntesse a rossz értékeket
        - a program fut amíg a stop feltétel meg nincs
        - miután megoldást talált, megvizsgáljuk, hogy jó megoldás-e, azaz nincs e kilógó pont, elég magas penalty érték megadása esetében nem szokott rossz értéket adni, bent hagytam, hogy a heurisztikus szemlélet látszódjon, hogy nem mindig ad jó megoldást
        -
    - **static bool StopCondition(int stop_count_max,ref int stop_count, ref float p_fitness,ref float last_fitness, float stop_fitness,float stop_fitness_diff):**
        - vizsgáljuk azt, hogy ténylegesen történik e változás, nem csak beragad egy értékre
        - ha túl sokat nem változik akkor leállítjuk, ha viszont a „türelmi időben” talál jobb megoldást akkor reseteljük a stop countert
    - **static List<Point> GenerateRandomChange(List<Point> solution,float e):**
        - random változást generál a pontokban
    - **static float fitness(List<Point> solution,List<Point> points,int penalty):**
        - fitness érték kiszámítása, penalty segítségével

- **static List<Point> GenerateRandomPoints(List<Point> t_points,int p):**
  - random pontokat generál a közelítendő pontok köré egy nagy körben

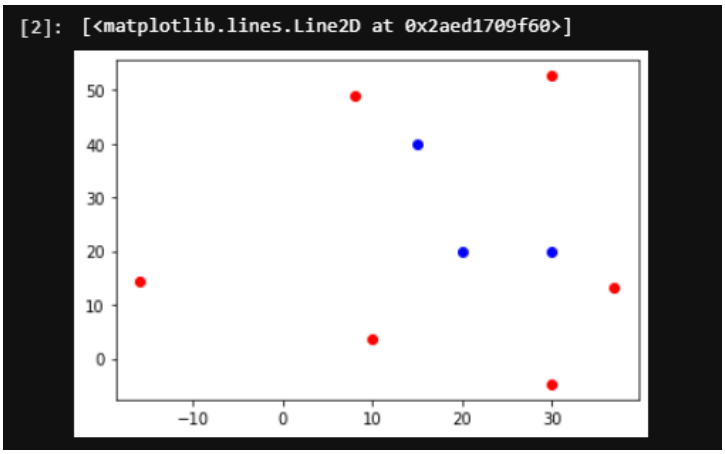
- Point.cs
  - x és y koordináták propertykként

Az alábbiakban látható ahogy a program operál.

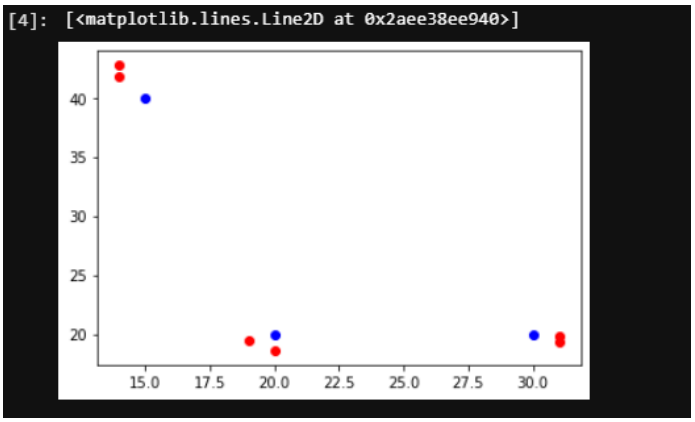
```
Fitness 470,0037      127,5214      99982
Fitness 447,7486      127,5214      99984
Fitness 477,3087      127,5214      99986
Fitness 411,4878      127,5214      99988
Fitness 187,4026      127,5214      99990
Fitness 581,3389      127,5214      99992
Fitness 969,8936      127,5214      99994
Fitness 236,0089      127,5214      99996
Fitness 450,0134      127,5214      99998
Fitness 650,9126      127,5214      100000
DONE SEARCHING
0:-> Jo megoldas
```

Lehet látni a jelenlegi fitness baloldalt, a középső oszlopban a legjobb elért fitness értéket lehet látni, jobb oldalon pedig a stop_counter értékét ahogy növekszik.

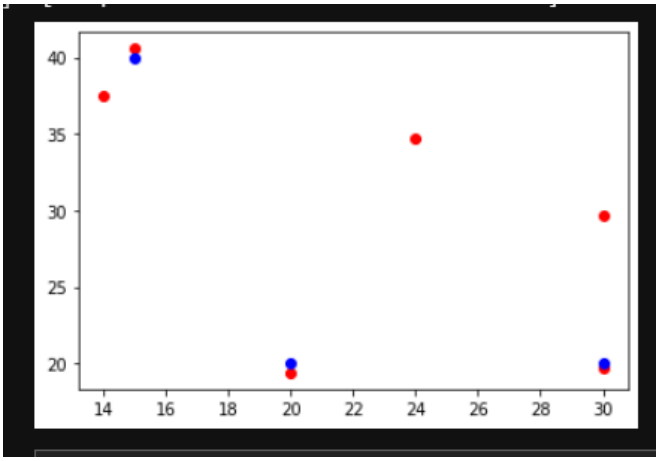
Itt 3 pontot közelítettunk egy 6 csúcsú poligonnal. A megoldás alább látható. Kékkel a közelített pontok, pirossal az eredmény.



Lentebb pedig több futás is látható, ugyanerre a pontokra közelítve.

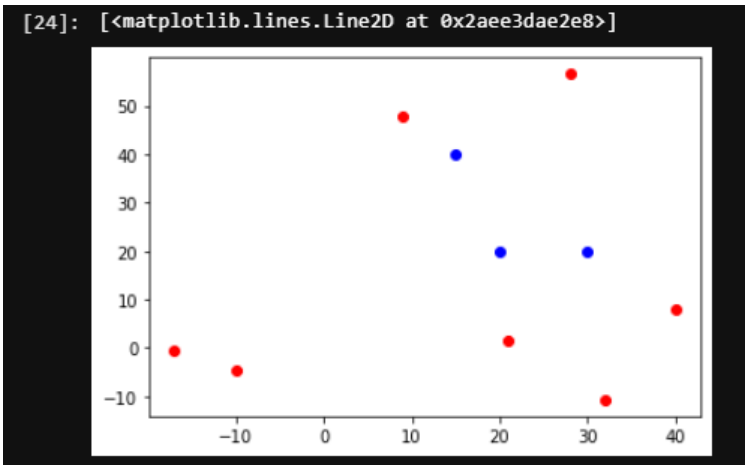



```
Fitness 11569,6       67,57899      99982
Fitness 16283,81       67,57899      99984
Fitness 11482,25       67,57899      99986
Fitness 1140,154       67,57899      99988
Fitness 14207,87       67,57899      99990
Fitness 10137,4       67,57899      99992
Fitness 380,0763       67,57899      99994
Fitness 5502,586       67,57899      99996
Fitness 10024,36       67,57899      99998
Fitness 16784,19       67,57899      100000
DONE SEARCHING
```




```
Fitness 397,0705        60,84604        99990
Fitness 864,1888        60,84604        99992
Fitness 546,5239        60,84604        99994
Fitness 551,7852        60,84604        99996
Fitness 64,03067        60,84604        99998
Fitness 246,5582        60,84604        100000
DONE SEARCHING
0:-> Jo megoldas
```

7 csúccsal közelítése a fentebbi pontoknak.




```
Fitness 490,7408        313,2982        99990
Fitness 478,6832        313,2982        99992
Fitness 493,0009        313,2982        99994
Fitness 1018,931        313,2982        99996
Fitness 576,5446        313,2982        99998
Fitness 642,3146        313,2982        100000
DONE SEARCHING
0:-> Jo megoldas
```