



AKADEMIA GÓRNICZO-HUTNICZA

WYDZIAŁ FIZYKI I INFORMATYKI STOSOWANEJ

EKSPLORACJA DANYCH

---

**Szybka klasteryzacja oparta o maksima gęstości**

---

*Autorzy:*  
Maciej Kubicki  
Tomasz Chronowski

*Prowadzący:*  
dr inż. Szymon Łukasik

20 stycznia 2018

# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>2</b>
1.1	Temat projektu . . . . .	2
1.2	Wykorzystane technologie i oprogramowanie . . . . .	2
1.3	Repozytorium . . . . .	2
<b>2</b>	<b>Implementacja</b>	<b>3</b>
<b>3</b>	<b>Użycie</b>	<b>4</b>
3.1	Działanie algorytmu . . . . .	4
3.1.1	seeds.dataset . . . . .	4
3.1.2	gener . . . . .	6
3.1.3	gener3 . . . . .	7
3.2	Skalowalność algorytmu . . . . .	8
<b>4</b>	<b>Wnioski</b>	<b>9</b>

# 1 Wstęp

## 1.1 Temat projektu

Tematem naszego była implementacja algorytmu szybkiej klasteryzacji opartej o maksima gęstości z usprawnieniem w liczeniu odległości. Algorytm był przedstawiony w artykule autorów Rodriguez'a i Laio'a pod tytułem "Clustering by fast search and find of density peaks" w magazynie Science z 2015.

## 1.2 Wykorzystane technologie i oprogramowanie

Projekt został zrealizowany w Visual Studio 2017. Algorytm został napisany w C++ z wykorzystaniem technologii MPI(z biblioteki boost) i OpenMP. Zgodnie z założeniami użyta została biblioteka FLANN. Pierwotnie chcieliśmy skorzystać z technologii CUDA(to prawdopodobnie przyspieszyłoby algorytm), jednak kompilator z nowego VS nie poradził sobie z budową części biblioteki FLANN odpowiedzialnej za przerzucanie obliczeń na kartę graficzną. W każdym razie w kodzie w komentarzach umieściliśmy odpowiednie przykłady jakby to zostało zrealizowane.

## 1.3 Repozytorium

Adres repozytorium:

<https://github.com/maciekkubicki/DensityPeaksBasedClustering>

## 2 Implementacja

Sama struktura programu jest oparta o MPI. Algorytm działa niezależnie od rozmiaru "świata". W funkcji **main** na początku deklarujemy środowisko MPI. Następnie sprawdzane są parametry z jakimi uruchomiono program - ilość oraz odpowiedniość. Następnie jeśli są odpowiednie to te parametry są przekazane do funkcji dokonującej klasteryzacji - **clustering**. Lista parametrów:

- **fileName** - nazwa plik z danymi do klasteryzacji,
- **radius** - rozmiar sąsiedztwa "punktu",
- **minDistF** - minimalna odległość "punktu" od innego "punktu" o większej gęstości, po to, by "punkt" był środkiem klastra (musi być spełniony warunek niżej),
- **minDensF** - minimalna gęstość "punktu", po to, by "punkt" był środkiem klastra (musi być spełniony warunek wyżej),
- **excludeHalo = false** (opcjonalny) - wartość logiczna czy pominąć tzw. "punkty halo",
- **densCol = R** (opcjonalny)- "punkty halo" będą wykluczane na podstawie gęstości globalnej (gęstość sąsiedztwa bez brania pod uwagi do jakiego klastra należy sąsiad - enum = R = 1), lub też lokalnej(gęstość sąsiedztwa składająca się tylko z sąsiadów tego samego klastra co badany punkt - enum = CR = 4),
- **factor = 1.f** (opcjonalny) - "punkty halo" są identyfikowane jeśli ich gęstość(globalna/lokalna) jest mniejsza, niż największa gęstość w obszarze brzegowym(w sąsiedztwie "punktu" brzegowego są "punkt/y" z innego/innych klastrów). Parametr **factor** dzieli tą największą gęstość, jest użyteczny w przypadku korzystanie z gęstości globalnej.

Funkcja **clustering** jest przystosowana do działania równoległe. Proces o indeksie 0 wczytuje dane do macierzy i liczy maksymalną możliwą odległość między dwoma "punktami". Proces 0 dzieli dane między wszystkie procesy w "świecie" - użycie **broadcast** i **scatter**. W pierwszym kroku każdy proces znajduje gęstość sąsiedztwa przydzielonych mu "punktów" - **radiusSearch** z biblioteki FLANN. Następnie obliczone gęstości są porozysyłane do wszystkich procesów - **all\_gather**. W kolejnym kroku posiadając gęstości procesy znajdują odległości do punktów o większej gęstości, przechowujemy również indeks najbliższego sąsiada z większą gęstością - to ułatwi klasteryzację. Tu używamy **knnSearch** z FLANN(zgodnie z dokumentacją projekt jest kompilowany z OpenMP i do przekazujemy do tej funkcji parametr z atrybutem **cores** równym 0, więc wyszukiwanie jest realizowane przy największej liczbie dostępnych rdzeni). W przypadku, gdy nie ma "punktu" o większej gęstości przypisana zostanie wartość maksymalnej możliwej odległości między punktami(znaleziona na początku). Dodatkowo może się zdarzyć, że w sąsiedztwie będzie kilka "punktów" o maksymalnej gęstości - jeśli to się stanie to naprawiamy to funkcjom **fix2**, której działanie przydałoby się usprawnić. Następnie znalezione odległości są normalizowane, by znajdowały się w przedziale  $< 0, 1 >$  i ustalane są środki klastrów na podstawie parametrów **minDistF** i **minDensF**. Sama klasteryzacja realizowana jest w miejscu przy pomocy funkcji rekurencyjnej - "punkt" jest przypisany do tego samego klastra co jego najbliższy sąsiad o większej gęstości. Następnie, jeśli wartość parametru **excludeHalo** jest prawdą to realizujemy pozbywanie się tych "punktów" z wykorzystaniem zadanego rodzaju gęstości i wielkościom parametru **factor**. Tu również używana jest funkcja **radiusSearch** z FLANN. Następnie dane są zebrane przy pomocy **all\_gather** i proces 0 zapisuje dane do pliku **outputp.dat**.

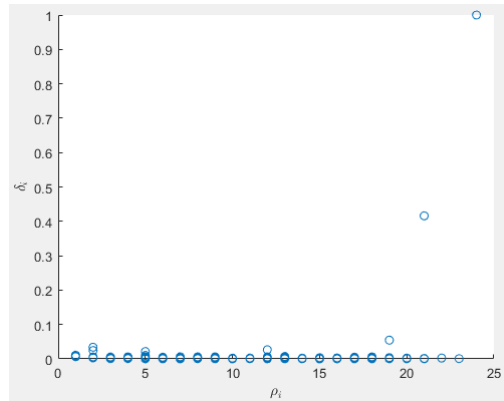
### 3 Użycie

Do przedstawienie algorytmu przygotowaliśmy kilka przykładowych zbiorów danych, część jest wygenerowanych przez nas, a część z Internetu. Przykładowe komendy uruchomienia algorytmu wraz z parametrami znajdują się w pliku **readmeRunExample.txt**. Dodatkowo przygotowaliśmy dwa skrypty w Matlabie przydatne do tworzenia wykresów - **densdisttest.m**(wykres stosunku **min-DensF** do **minDistF** i **clustertest.m**(porównywanie wyników naszej klasteryzacji z kmeans).

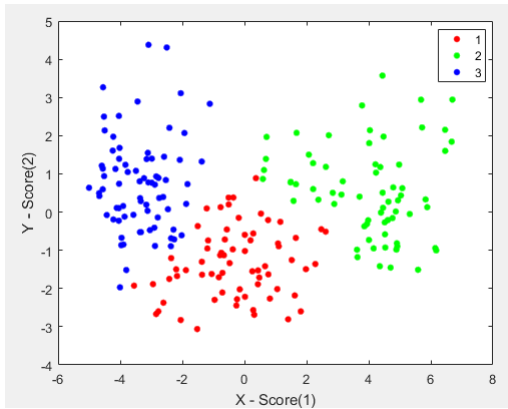
#### 3.1 Działanie algorytmu

##### 3.1.1 seeds\_dataset

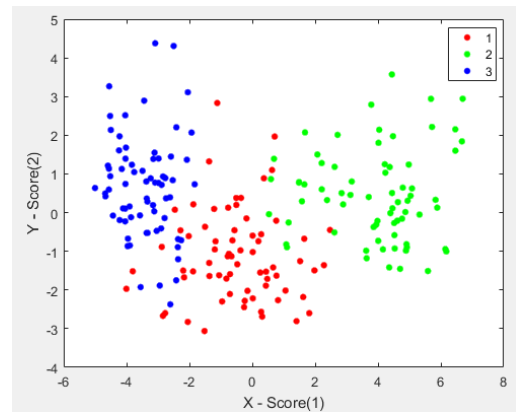
```
1 mpiexec -np 4 .\ConsoleApplication1.exe seeds.txt 0.9 0.05 15
2 mpiexec -np 4 .\ConsoleApplication1.exe seeds.txt 0.9 0.05 15 true 4 1
```



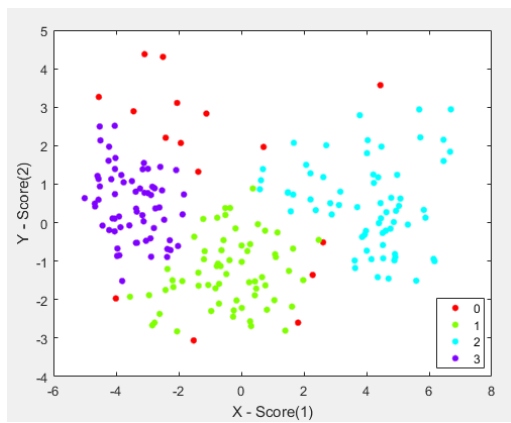
Rysunek 1: Stosunek  $\rho_i$  do  $\delta_i$



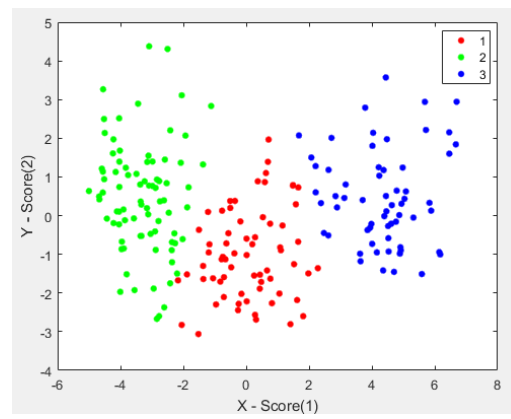
Rysunek 2: Algorytm bez wykluczania "punktów halo"



Rysunek 3: Oryginalne klastry



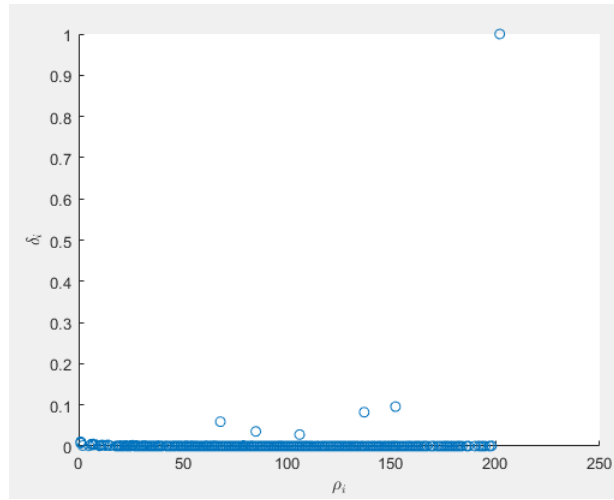
Rysunek 4: Algorytm z wykluczeniem "punktów halo"



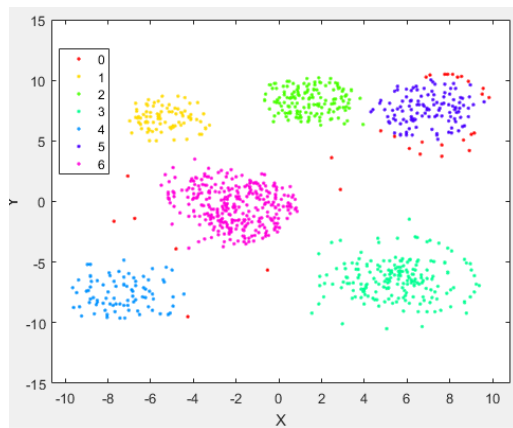
Rysunek 5: Klastry z kmeans

### 3.1.2 gener

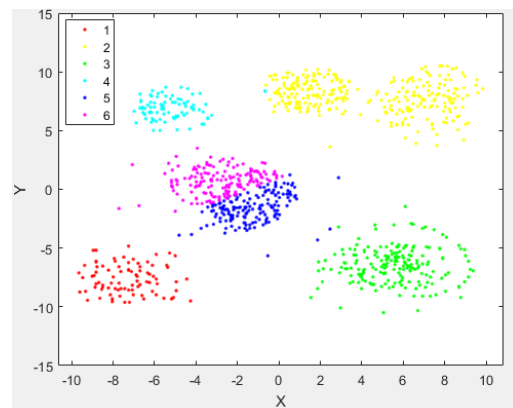
```
1 mpiexec -np 4 .\ConsoleApplication1.exe gener.txt 5 0.02 50 true 4 1
```



Rysunek 6: Stosunek  $\rho_i$  do  $\delta_i$



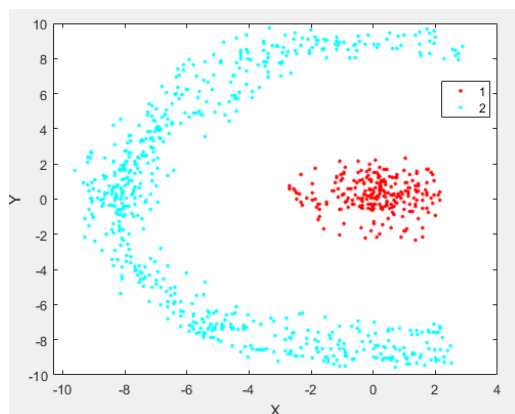
Rysunek 7: Algorytm z wykluczaniem "punktów halo"



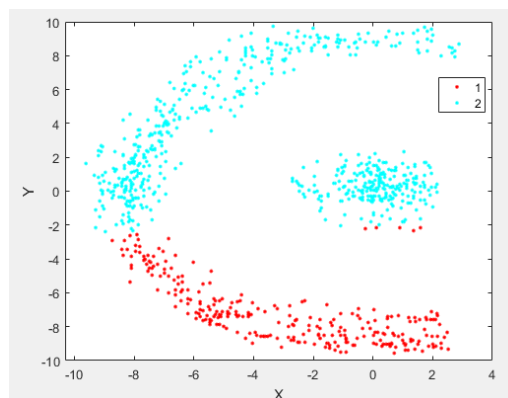
Rysunek 8: Klastry z kmeans

### 3.1.3 gener3

```
1 mpiexec -np 4 .\ConsoleApplication1.exe gener3.txt 1.1 0.1 60
```



Rysunek 9: Algorytm bez wykluczania "punktów halo"



Rysunek 10: Klastry z kmeans



### 3.2 Skalowalność algorytmu

Czasy wykonania algorytmu zostały zmierzone na laptopie z procesorem dwurdzeniowym. Każdy rdzeń jest dwuwątkowy.

Dataset	Wiersze	Kolumny	excludeHalo	Liczba procesów NP	1	2	4	8
seeds_dataset	210	7	FALSE	Czas [s]	0.030	0.020	0.018	0.072
seeds_dataset	210	7	TRUE		0.036	0.025	0.020	0.080
gener	1000	2	TRUE		0.714	0.342	0.227	0.320
gener2	2500	2	TRUE		4.687	2.027	1.469	1.470
gener3	900	2	FALSE		0.418	0.225	0.162	0.224
htru	5555	8	FALSE		29.126	12.058	8.027	8.134
htru2	11111	8	FALSE		110.713	56.718	37.275	38.180

Rysunek 11: Czas wykonywania algorytmu w zależności od rozmiaru danych i parametru **np** MPI

## 4 Wnioski

Powyżej przedstawione przykłady dają dobry obraz działania algorytmu, które są bardzo dobre. Można zaobserwować, że wyniki w niektórych przypadkach są nawet lepsze od klasteryzacji kmeans. Dodatkową zaletą naszego algorytmu jest możliwość wykluczenia "punktów halo". W implementacji stworzyliśmy dwie interpretacje takich "punktów". Pierwsza w oparciu o gęstość globalną (nie zależy od numeru klastra) w obszarze granicznym, a druga o lokalną (tylko punkty z tego samego klastra). Dodatkowo możliwe jest użycie współczynnika, którym można manipulować liczbą punktów wykluczonych.

Nasz algorytm w odróżnieniu na przykład od kmeans nie wymaga podania liczby klastrów tylko sam ją ustala na podstawie dwóch parametrów - **minDistF** i **minDensF**, które dużo łatwiej ustalić patrząc na wykres stosunku  $\rho_i$  do  $\delta_i$ . Wadą naszego algorytmu jest to, że w przypadku znaczących odległości między punktami wartość **minDistF** znacząco maleje.

Jeśli chodzi o wyniki czasowe to są również dobre. Samo zrównoleglenie wynikające ze stosowania MPI daje dobre efekty. Narzuty komunikacyjne nie są duże. Dodatkowo przy poszukiwaniu sąsiadów wykorzystujemy OpenMP. Prawdopodobnie, gdybyśmy skorzystali z CUDA wyniki byłyby sporo lepsze, biorąc pod uwagę platformę sprzętową, na której testowaliśmy algorytm. Obserwując powyższe wyniki czasowe, wydają nam się, że większy narzut czasowy powoduje wzrost wierszy, niż kolumn.

Podsumowując otrzymane wyniki są zadowalające, chociaż na pewno znalazłoby się parę miejsc do udoskonalenia.