Triangulacja Delaunaya - Dokumentacja

Andrzej Podobiński, Aleksandra Marzec

Program prezentuje algorytm Bowyera-Watsona znajdujący triangulację Deloaunay'a zbioru punktów 2D przy użyciu pakietu Jupyter Notebook, napisany został w języku Python 3.7.

Wejście : *chmura punktów 2D*
Wyjście : *podział obszaru wyznaczonego przez zbiór punktów na trójkąty*

1. Triangulacja Delaunay'a chmury punktów

Triangulacja zadanej chmury punktów opiera się na algorytmie Bowyera-Watsona :

- *znalezienie prostokąta lub trójkąta zawierającego wszystkie punkty*
- *początkowa triangulacja T_0 zawierająca jeden lub dwa trójkąty*
- *dla każdego punktu z chmury punktów , dodanie go do aktualnej triangulacji, wyszukanie trójkątów, których koło opisane zawiera ten punkt*
- *usunięcie tych trójkątów*
- *utworzenie nowych elementów poprzez połączenie wierzchołków powstałej wnęki z tym punktem.*
- *po wprowadzeniu wszystkich punktów, usuwamy wszystkie trójkąty których choć jeden wierzchołek należał to początkowej triangulacji T_0*
- *zwrócenie triangulacji Delaunaya*

Istnieją dwa kryteria przeprowadzania triangulacji zbioru punktów : kryterium koła opisanego na trójkącie oraz kryterium kątów wewnętrznych. W naszej implementacji wykorzystujemy pierwsze z nich tzn. dla każdego trójkąta, koło opisane na tym trójkącie zawiera jedynie punkty należące do tego trójkąta.

Podstawowym krokiem algorytmu jest znalezienie trójkąta do którego wpada punkt.

W naszym projekcie przedstawiliśmy trzy rozwiązania tego problemu.

1. *Sposób 1 - Brute Force (plik triangulationBrute.ipynb)*

Klasycznym sposobem lokalizacji trójkąta jest przeszukanie wszystkich istniejących już trójkątów triangulacji . Zaczynamy od ostatnio modyfikowanego elementu. Jeśli okrąg opisany zawiera dodawany punkt, dodajemy go na listę do usunięcia. Po znalezieniu wszystkich usuwamy je, a następnie tworzymy nowe elementy poprzez połączenie wierzchołków powstałej wnęki z dodawanym trójkątem

2. *Sposób 2 - Przeszukiwanie z użyciem listy sąsiadów (plik triangulationOpt.ipynb)*

Zoptymalizowany powyższy algorytm polega na skorzystaniu z grafu sąsiedztwa topologicznego. Dla każdej krawędzi w słowniku przechowujemy jej sąsiadów (trójkąty, maksymalnie 2 dla każdej krawędzi),
Pierwszy naruszony trójkąt znajdujemy metodą Brute Force, natomiast kolejne z nich szukamy za pomocą stworzonego przez nas iteratora przeszukującego trójkąty triangulacji z użyciem algorytmu BFS(opis niżej)

3. *Sposób 3 - Przeszukiwanie z użyciem kd-drzewa (plik triangulationFullOpt.ipynb)*

Korzystając z biblioteki Python i dostępnej struktury kd-drzewa w łatwy sposób można zoptymalizować poprzednie rozwiązania. Każdy trójkąt „mapujemy" poprzez środek okręgu opisanego na nim. Struktura kd-drzewa w czasie (dla średniego przypadku)zwraca nam środek okręgu leżącego najbliżej dodanego punktu. Czynność powtarzamy dopóki punkt nie znajdzie się w jakimś trójkącie Następnie przy użyciu iteratora(BFS po grafie sąsiedztwa topologicznego) znajdujemy naruszonych sąsiadów (podobnie jak wyżej)

2. Struktury danych

Class Point *- klasa reprezentująca punkt, przechowująca funkcję zaokrąglająca, funkcję haszującą, funkcję do wypisywania*
Zmienne :
x , y - współrzędne
r - ilość miejsc po przecinku do jakiej zaokrąglamy

Class Circle *- klasa reprezentująca okrąg przechowująca promień oraz współrzędne środka okręgu*
Zmienne :
r - promień
c - punkt środka okręgu

Class Triangle *- klasa reprezentująca trójkąt przechowująca funkcję sprawdzająca czy dany punkt zawiera się w okręgu opisanym na tym trójkącie, funkcję haszującą, funkcje do wypisywania*
Zmienne :
circle - obiekt klasy Circle, jako okrąg opisany na trójkącie
point_a, point_b, point_c - wierzchołki trójkąta
line_a, line_b, line_c - krawędzie trójkąta

Class Line *- klasa reprezentująca odcinki, przechowująca funkcję zaokrąglającą, funkcję haszującą*
Zmienne :
point_a, point_b - współrzędne początku i końca odcinka

Class Graph *- klasa reprezentująca graf sąsiedztwa topologicznego, funkcję dodającą do grafu , funkcję usuwającą z grafu, funkcję która zwraca wszystkich sąsiadów węzła*
Zmienne :
słownik map, lines_map - słowniki przechowujące sąsiadów dla każdego trójkąta triangulacji i jego krawędzi

Class TriangleIterator *- klasa implementująca nasz własny iterator służący do przemieszczania się po elementach ze zbioru w celu znalezienia trójkąta do którego wpada dany punkt, działa na zasadzie algorytmu BFS, zwraca kolejny trójkąt do usunięcia*

Zmienne :
triangle_graph - graf sąsiedztwa topologicznego
triangle - aktualnie przeglądany trójkąt
queue - kolejka z której korzystamy podczas przechodzenia po trójkątach triangulacji BFS'em
visited - tablica przechowująca już odwiedzone trójkąty triangulacji
point - aktualnie dodany punkt

__Class TrianglesSet__ implementuje funkcję TriangleIterator zwracającą iterator do przeszukiwania zbioru trójkątów triangulacji , funkcję update uruchamianą za każdym razem przed dodaniem nowego punktu, funkcję dodająca do grafu, słownika i kd-drzewa, funkcję usuwająca z grafu, kd-drzewa i słownika

Zmienne :

graph - graf sąsiedztwa topologicznego

centerToTriangleMapping - słownik ‚mapujący' trójkąty według środka okręgu opisanego

kdtree - kd drzewo, zwracające środek okręgu opisanego na trójkącie znajdujący się najbliżej dodanego punktu

Funkcje :

def circle_on_three_points(p1,p2,p3) - funkcja wyznaczająca okrąg opisany na trójkącie(środek okręgu, promień), przyjmująca 3 wierzchołki trójkąta

def delunay(points) - funkcja przyjmująca zbiór punktów, wyznaczająca triangulację zbioru

def get_random(range_min,range_max) - funkcja losująca liczbę z zadanego przedziału , przyjmuje przedział

def get_random_from_range(range_min,range_max,n) - funkcja losująca zbiór punktów z danego przedziału , przyjmuje przedział oraz liczbę punktów

def get_random_point(range_min,range_max) - funkcja losująca punkt z zadanego przedziału

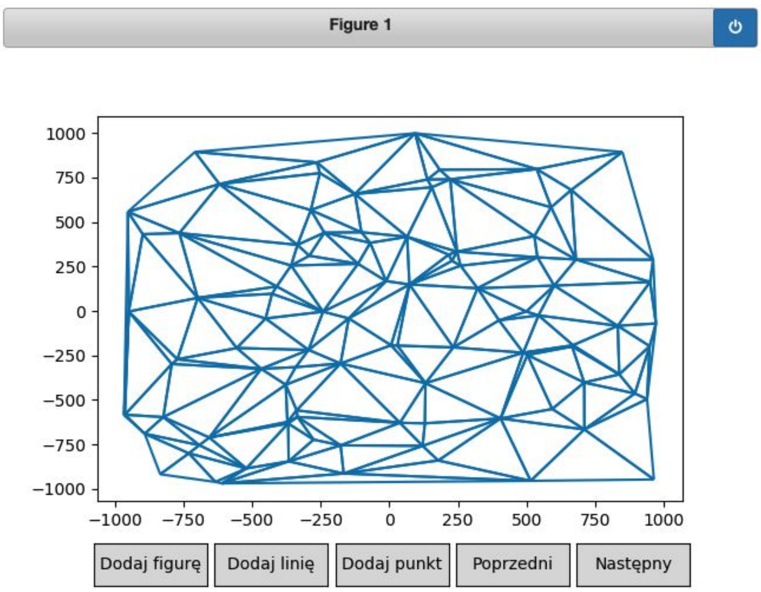
def d(p1, p2) - funkcja wyznaczająca odległość dwóch punktów

Wizualizacja, porównanie sposobów wyszukiwania trójkątów

1. *100 punktów z zakresu (-1000,1000)*

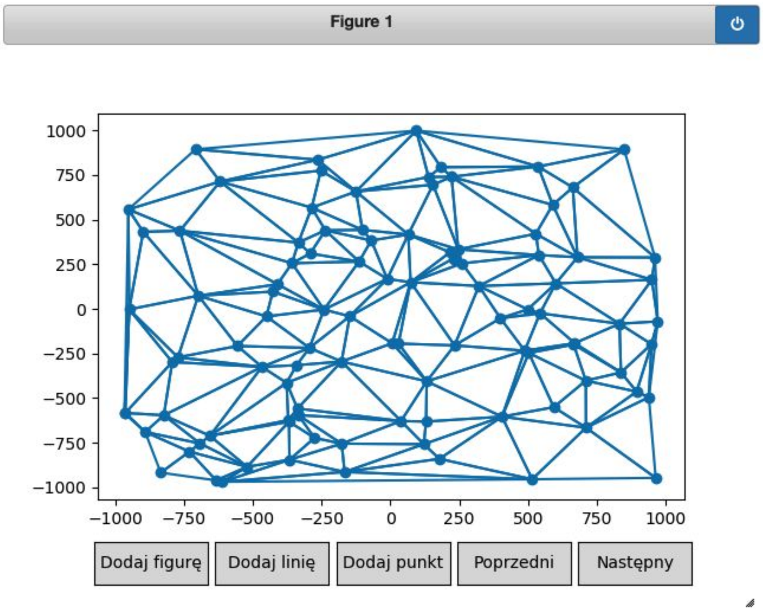
Sposób 1

```
BruteTime : 0.3362569808959961 sec
```



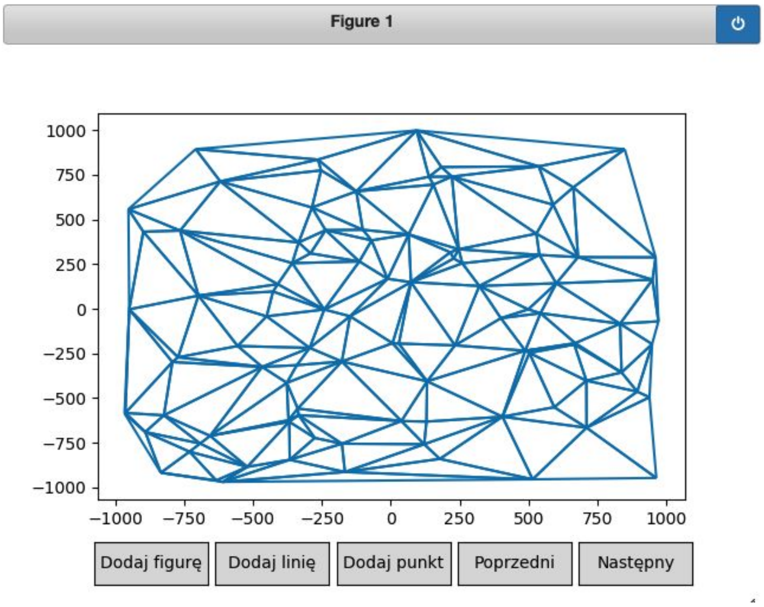
Sposób 2

```
FastTime : 0.24508428573608398 sec
```



Sposób 3

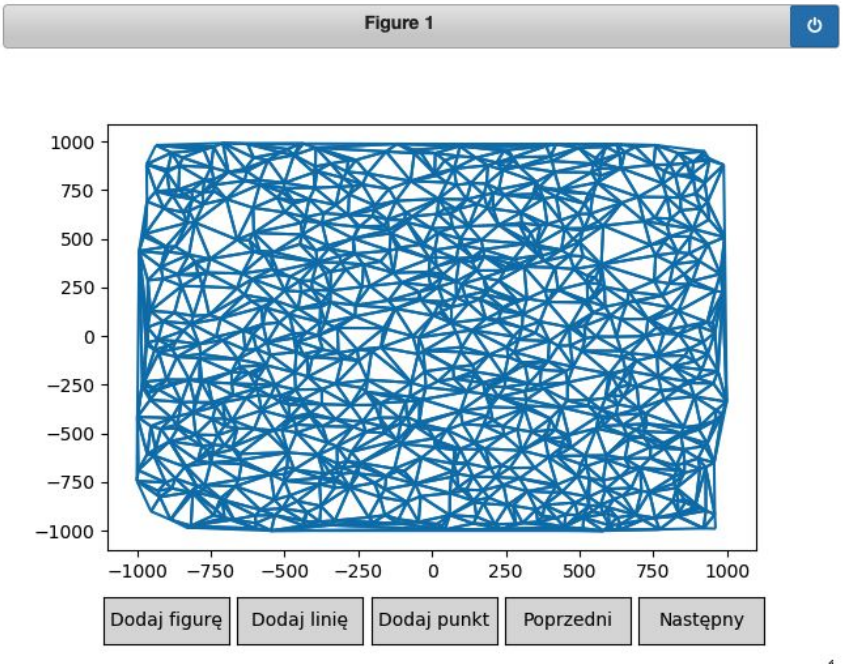
```
FastTime : 0.23281216621398926 sec
```



2. *1000 punktów z przedziału (-1000,1000)*

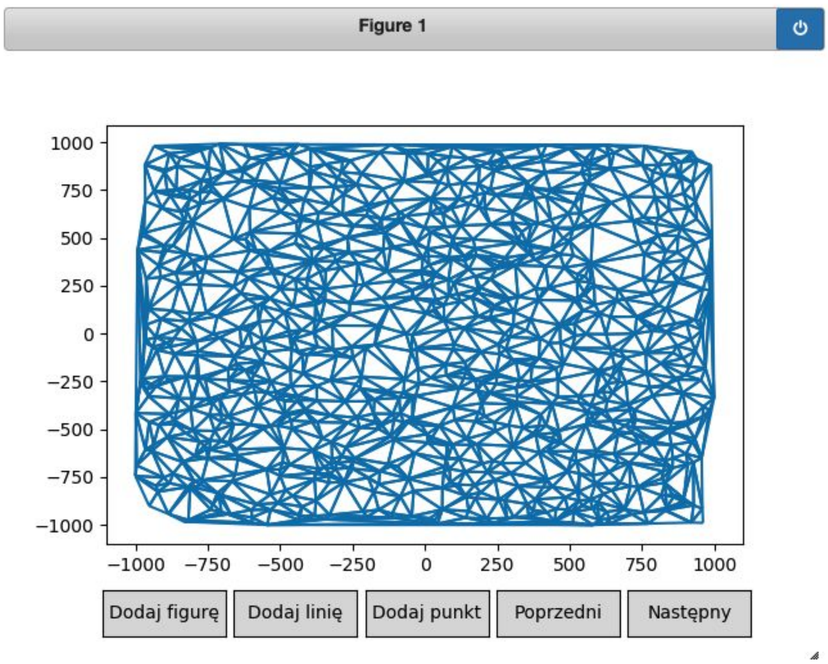
Sposób 1

```
BruteTime : 24.888833045959473 sec
```



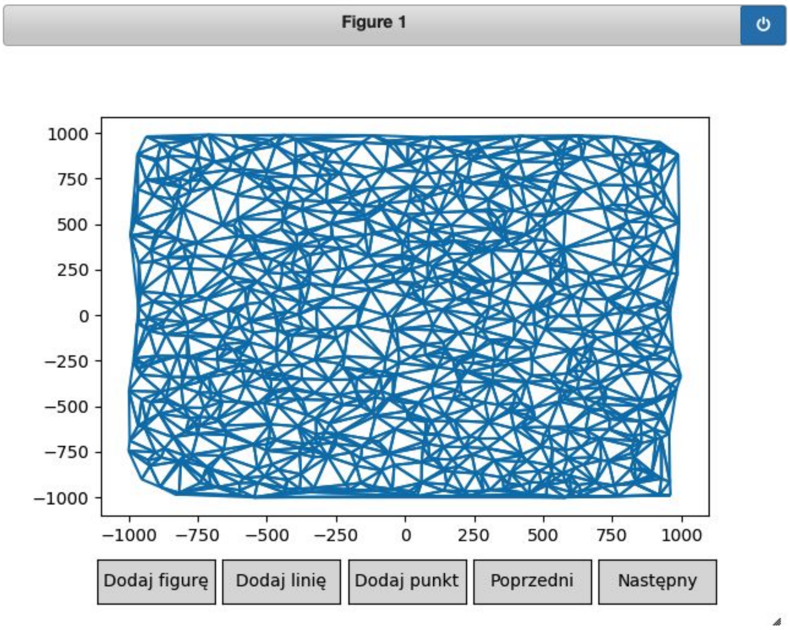
Sposób 2

```
FastTime : 10.798218011856079 sec
```



Sposób 3

```
FastTime : 2.991905927658081 sec
```



Złożoność :

*Jak można zauważyć, najszybszą metodą jest skorzystanie z kd-drzewa. Co daje nam w średnim przypadku O(m*log(n)), gdzie m to jest liczba trójkątów do których wpada punkt, a n to liczba trójkątów triangulacji.*
Dwie pozostałe metody pesymistyczną złożoność mają taką samą O(n^2) , jednak z użyciem listy sąsiadów wypada lepiej ponieważ, ponieważ trójkąt do którego wpada punkt znajdujemy statystycznie w połowie przeszukiwania

Bibliografia

1. http://home.agh.edu.pl/~jurczyk/papers/diss.pdf

2. prezentacja z wykładu Triangulacja Delaunay

3. https://en.wikipedia.org/wiki/Delaunay_triangulation

4. https://python-kdtree.readthedocs.io/en/latest/