Dokumentacja

Metoda doskonalenia triangulacji

Algorytm Kirkpatrick’a

Czwartek 13:00
grupa nr 6

Mateusz Bartnicki

Jan Kalęba

Spis treści

Część I - część techniczna

1. Wymagania techniczne

Program został napisany przy użyciu języka Python. Wizualizacja algorytmu wykonana została przy pomocy narzędzia wizualizacyjnego dostarczonego na zajęcia przez koło naukowe BIT.
W implementacji oraz testach wykorzystane zostały biblioteki: numpy, random, time, math, functools, matplotlib, scipy.spatial, mapbox_earcut oraz Visualizer dostarczony przez KN BIT.
Dane techniczne komputera, na których był pisany program oraz przeprowadzane testy:

Procesor	AMD Ryzen 7 4800H
System operacyjny	Windows 11
Interpreter	Python 3.10

Tabela 1 – dane techniczne komputera

2. Opis programu

Program służy do lokalizacji zadanego punktu w podziale poligonowym. Jako wejście przyjmuje chmurę punktów, która ulegnie triangulacji Delaunay'a oraz punkt do zlokalizowania. Wyjściem programu jest obiekt klasy Face (opisany poniżej) oznaczający trójkąt, w którym został znaleziony punkt

3. Klasy

W tym dziale znajduje się opis wszystkich klas wykorzystanych przy implementacji algorytmu Kirkpatrick'a.

Point

Klasa reprezentująca punkt na płaszczyźnie.
Atrybuty:

- x - współrzędna x punktu
- y - współrzędna y punktu
- neighbors – zbiór elementów klasy Point, z którymi punkt jest połączony krawędzią
- faces - zbiór elementów klasy Face, w których zawiera się dany punkt

Funkcje:

- position() – zwraca krotkę w postaci (self.x, self.y)
- add_neighbors(v) – dodaje punkt v do zbioru neighbors
- assign_face(face) - dodaje element do zbioru faces
- __str__/__repr__ - zmienia sposób wypisywania punktu jako napis: „Point: (self.x, self.y)"

- __add__ - nadpisuje metodę dodawania, w naszym przypadku do punktu można dodać wektor, co poskutkuje otrzymaniem nowego punktu z przesuniętymi współrzędnymi
- __eq__/__hash__ - nadpisuje metodę porównywania oraz zmienia hashowanie, punkty są porównywane po obu współrzędnych

Face

Face odpowiada figurom tworzonym podczas procesu retriangulacji
Atrybuty:

- coordinates - zbiór zawierający wierzchołki figury (obiekty klasy Point)

Funkcje:

- triangle() - zwraca wartość prawda/fałsz, czy dany Face jest trójkątem
- contains(point) - zwraca informację, czy podany punkt leży wewnątrz Face
- inside_triangle(point) - funkcja pomocnicza do contains. Oblicza wyznacznik i zwraca wartość prawda/fałsz, czy punkt znajduje się wewnątrz trójkąta
- __str__/__repr__ - zmienia sposób wypisywania figury jako napis: „Face: self.coordinates”
- __eq__/__hash__ - nadpisuje metodę porównywania oraz zmienia hashowanie - figury są takie same, jeśli mają takie same zbiory coordinates

Node

Węzły w drzewie. Głównym jego atrybutem jest face, który oznacza trójkąt, jaki reprezentuje dany węzeł.
Atrybuty:

- face – obiekt klasy Face
- children – lista węzłów – dzieci, które zostały utworzone w wyniku retriangulacji Face

Funkcje:

- __str__/__repr__ - zmienia sposób wypisywania figury jako napis: „Node: self.face”

4. Funkcje

Spis wszystkich funkcji użytych w programie

funkcja do zadawania wierzchołków z myszki

W programie zadeklarowany jest osobny moduł dla tej funkcji. Zostało to zrobione w ten sposób, ponieważ samo skompilowanie go już ją włącza. Wyświetla się wtedy na ekranie okienko, na którym wyświetlony jest obszar 2D. Jeśli zostanie on kliknięty, w tym miejscu ukaże nam się punkt, który zostaje równocześnie dodany do tablicy polygon, którą można później używać. Po zadaniu wszystkich punktów, należy zamknąć okienko.

generate_points(n, max_x, max_y)

Wejście: liczba punktów do wygenerowania, maksymalna współrzędna x, maksymalna współrzędna y

Korzystając z funkcji uniform z biblioteki random, generuje losowe n punktów jako krotki (x, y), gdzie 0 < x < max_x oraz 0 < y < max_y.

Wyjście: tablica n losowo wygenerowanych punktów

determinant(a, b, c)

Wejście: Punkty a, b i c będące obiektami klasy Point

Funkcja oblicza wyznacznik ze wzoru: $(b_x - a_x) * (c_y - a_y) - (c_x - a_x) * (b_y - a_y)$.

Wyjście: wartość obliczonego wyznacznika

distance(a, b)

Wejście: Punkty a i b będące obiektami klasy Point

Oblicza odległość między punktami a i b korzystając z metryki Euklidesowej.

Wyjście: odległość pomiędzy dwoma punktami

lower_left(polygon)

Wejście: lista punktów zadanych jako obiekty klasy Point

Funkcja pomocnicza wykorzystywana w algorytmie Grahama. Służy do wyznaczenia lewego dolnego punktu w zadanej liście.

Wyjście: znaleziony punkt

graham_algorithm(polygon)

Wejście: lista punktów zadanych jako obiekty klasy Point

Wyznacza otoczkę wypukłą na danej liście punktów za pomocą algorytmu Grahama.

Wyjście: lista punktów (obiektów klasy Point) należących do otoczki wypukłej

generate_outer_triangle(points)

Wejście: lista punktów zadanych jako obiekty klasy Point

Wyznacza trójkąt zawierający w sobie wszystkie punkty z listy.

Wyjście: lista trzech obiektów klasy Point będących wierzchołkami powstałego trójkąta

triangulate_points(points)

Wejście: lista punktów jako obiekty klasy Point

Przeprowadza triangulację pośród punktów za pomocą funkcji Delaunay z biblioteki scipy.spatial.

Wyjście: zbiór obiektów klasy Face, odpowiada trójkątom powstałym w wyniku działania algorytmu

connect_outer_triangle(convex_hull, outer_triangle)

Wejście: listy punktów zadanych jako obiekty klasy Point: punkty wyznaczające otoczkę wypukłą oraz trójkąt wyznaczony w funkcji powyżej

Łączy trójkąt zewnętrzny z punktami otoczki wypukłej.

Wyjście: zbiór elementów klasy Face oznaczających powstałe w wyniku triangulacji trójkąty

prepare_polygon(points, point_to_remove)

Wejście: zbiór punktów jako obiekty klasy Point będących sąsiadami punktu do usunięcia oraz punkt, który będzie usuwany podczas retriangulacji

Korzystając z funkcji alpha (link do opisu w bibliografii), sortuje zbiór points tak, że punkty w kolejności przeciwnej do ruchu wskazówek zegara względem punktu do usunięcia. Jest to przygotowanie zadanego obszaru do triangulacji.

Wyjście: zbiór obiektów klasy Face, odpowiada trójkątom powstałym w wyniku działania algorytmu

ear_triangulation(points)

Wejście: lista punktów jako obiekty klasy Point, które mają ulec triangulacji (punkty muszą być zadane w kolejności przeciwnej do ruchu wskazówek zegara)

Przeprowadza retriangulację wewnątrz figury o zadanych w wejściu punktach metodą „ear clipping” wykorzystując do tego gotową funkcję triangulate_float64 z biblioteki mapbox_earcut.

Wyjście: zbiór obiektów klasy Face, odpowiada trójkątom powstałym po zretriangulowaniu obszaru

triangulate_region(face, point_to_remove)

Wejście: obiekt klasy Face, który ma ulec retriangulacji oraz punkt do usunięcia

Funkcja pomocnicza do retriangulacji face. Jeśli figura wymaga retriangulacji, to wywołuje funkcję ear_triangulation. W przeciwnym przypadku odpowiednio przypisuje sąsiadów do punktów.

Wyjście: zbiór obiektów klasy Face, odpowiada trójkątom powstałym po zretriangulowaniu obszaru

remove_independent_set(points, faces, outer_triangle)

Wejście: zbiór punktów jako obiekty klasy Point, z których mają zostać usunięte wierzchołki, lista trójkątów w figurze jako obiekty klasy Face oraz punkty należące do trójkąta zewnętrznego – punkty, których nie należy usuwać

Wyszukuje największy zbiór niezależnych wierzchołków wewnątrz figury oraz je usuwa, a następnie wywołuje funkcję triangulate_region opisaną powyżej.

preprocess_regions(points, vis)

Wejście: lista punktów jako obiekty klasy Point, visualizer (od KN BIT)

Przygotowuje wejście pod preprocessing, a następnie go wykonuje – używając odpowiednich funkcji, zmniejsza ilość trójkątów w figurze aż pozostanie ostatni. Buduje jednocześnie drzewo, w którym poszukiwana będzie odpowiedź. Dodatkowo rysuje obszar po triangulacji Delaunay'a.

Wyjście: ostatni trójkąt utworzony w wyniku preprocessingu jako obiekt klasy Node

locate_point(points, searched_point, vis)

Wejście: punkty zadane w wejściu przetworzone na obiekty klasy Point, punkt do odszukania jego lokalizacji, visualizer (od KN BIT)

Celem funkcji jest zlokalizowanie, w którym trójkącie leży podany punkt. Wykonuje przeszukiwanie drzewa i zwraca odpowiedź. Jeśli punkt nie leżał w zadanym regionie, zwracana jest wartość None.

Wyjście: obiekt klasy Face, w którym zlokalizowany jest punkt

kirkpatrick(triangles, searched_point, vis)

Wejście: lista krotek zawierających współrzędne trzech punktów zadanych jako krotki współrzędnych (x, y), punkt, który chcemy odszukać (również jako krotka współrzędnych), visualizer (od KN BIT)

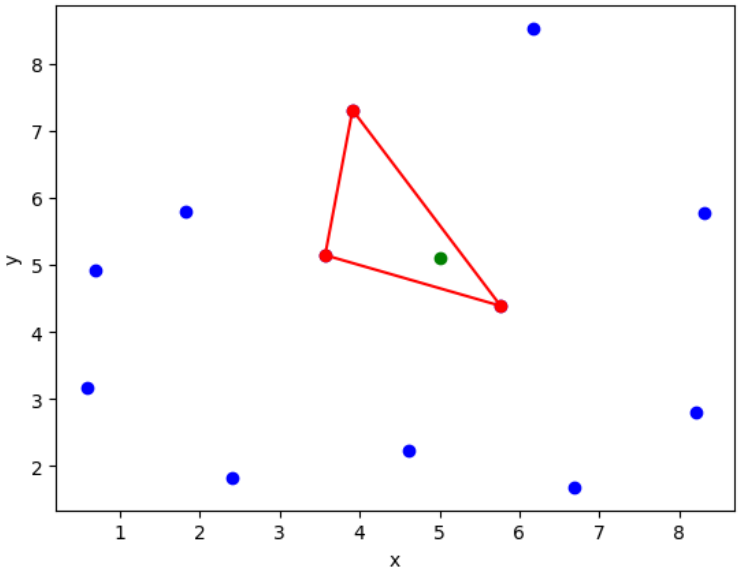
Główna funkcja całego programu. Przerabia wejście na odpowiednie klasy oraz wywołuje funkcję odpowiedzialną za zlokalizowanie punktu.

Wyjście: wynik wykonanej funkcji locate_point

show_result(points, final_face, searched_point)

Wejście: lista krotek zawierających współrzędne trzech punktów zadanych jako krotki współrzędnych (x, y), obiekt klasy Face, w którym zlokalizowany jest punkt oraz punkt, który był poszukiwany (krotka współrzędnych (x, y))

Jest to funkcja rysująca zbiór wszystkich punktów (jako kolory niebieskie) oraz trójkąt (kolor czerwony), w którym został znaleziony szukany punkt (kolor zielony).



Rysunek 1 – przykładowe wyjście

Część II - użytkowanie

1. Sposób uruchamiania

Aby uruchomić program Kirkpatrick, należy najpierw skompilować wszystkie moduły. (UWAGA! Podczas kompilacji modułu, w którym zawarta jest funkcja służąca do zadawania wierzchołków z myszki, automatycznie zostanie ona włączona!) Gdy już to będzie zrobione, można przekazać do funkcji kirkpatrick(points, searched_point, vis) wierzchołki figury, która ma ulec triangulacji Delaunay'a, szukany punkt (patrz: Opis programu) oraz visualizer, który jest zadeklarowany w mainie, gdzie rysuje początkowe punkty. Jeśli podane dane były poprawne, funkcja zwraca wynik obiekt klasy Face, w którym zawarty był punkt. Jeśli punkt leżał dokładnie na jednym z punktów chmury, wynik również zostanie zwrócony - jako jeden z trójkątów, którego był wierzchołkiem. Wizualizowany także jest początkowo zadany zbiór, zbiór po triangulacji Delaunay'a oraz odpowiedź (opisana w ostatniej funkcji w poprzedniej części). W osobnym programie KirkpatrickWithVis wizualizowane są wszystkie kolejne kroki, jakie wykonuje algorytm.

Legenda kolorów w programie KirkpatrickWithVis:

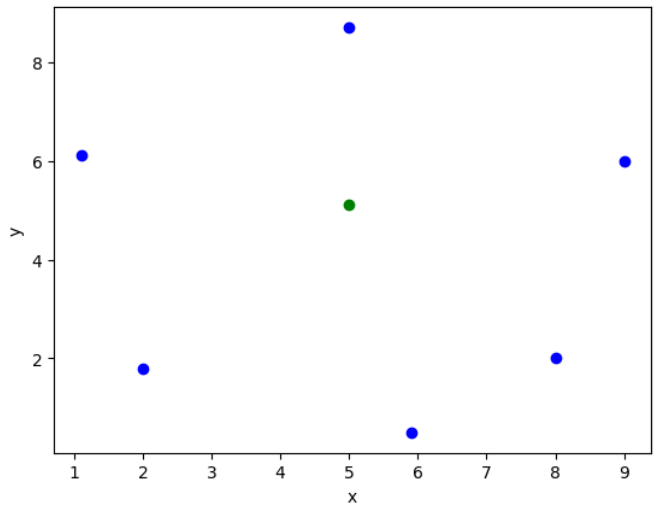
- Niebieski – zbiór aktualnie istniejących punktów oraz krawędzi
- Czerwony – wierzchołek, który jest usuwany
- Pomarańczowy – sąsiedzi wierzchołka usuwanego, czyli punkty wyznaczające obszar do retriangulacji
- Żółty – krawędzie utworzone po retriangulacji

2. Przykładowe użycie funkcji

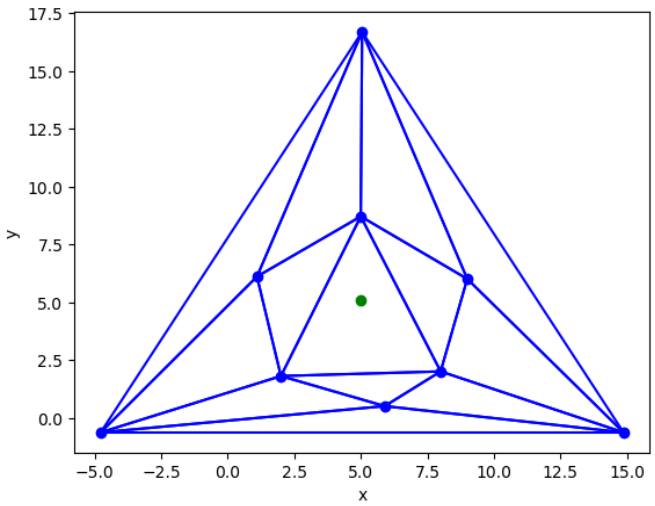
Główną funkcją całego programu jest funkcja o nazwie kirkpatrick i to ona odpowiada za wykonanie całego algorytmu. Wywołanie funkcji może wyglądać następująco:

```
points = (tutaj wprowadź swoją chmurę punktów, np. [(5, 8.7), (1.1, 6.1), (2, 1.8), (5.9, 0.5), (8, 2), (9, 6)] bądź wpisz polygon, jeśli chcesz, żeby były to punkty zadane z myszki)
searched_point = (tutaj wpisz punkt, który ma zlokalizować program, np. (5, 5.1))
vis = Visualizer()
kirkpatrick(points, searched_point, vis)
```

Wizualizacja wejścia przed i po triangulacji: (kolor niebieski – wierzchołki i krawędzie, kolor zielony – szukany punkt)

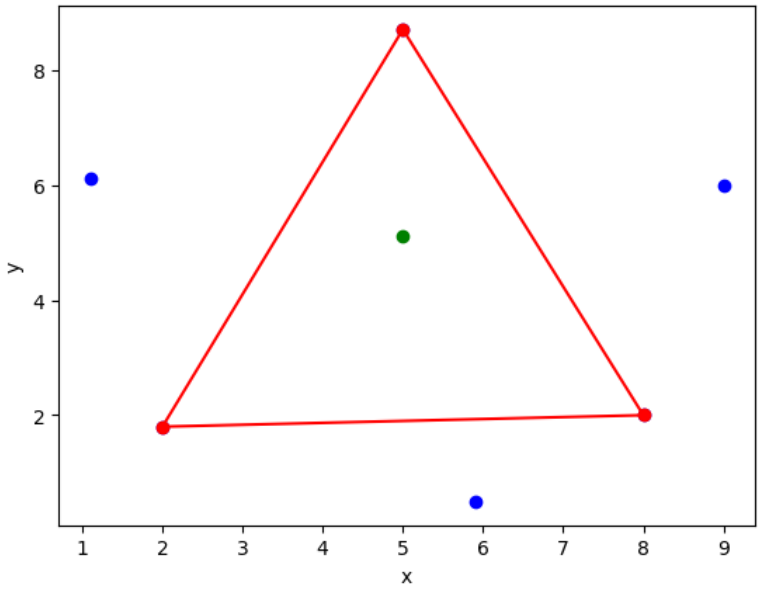


Rysunek 2 – wizualizacja zbioru punktów i punktu szukanego



Rysunek 3 – wizualizacja po triangulacji

W ten sposób wywołana funkcja zwraca wynik: Face: {Point: (8, 2), Point: (2, 1.8), Point: (5, 8.7)}. Jest to obiekt klasy Face, w którym zawierał się szukany punkt.
Wizualizacja wyniku: (kolor czerwony – trójkąt, w którym został znaleziony wierzchołek, kolor niebieski – pozostałe wierzchołki zbioru wejściowego, kolor zielony – szukany punkt)



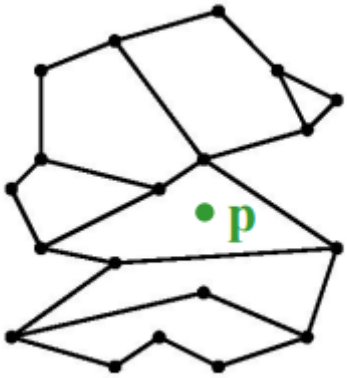
Rysunek 4 – wizualizacja wyjścia

Można również korzystać z każdej innej funkcji opisanych w części pierwszej. Trzeba jednak zadbać o to, aby wejście zgadzało się z wymaganiami właśnie tam opisanymi. W przeciwnym wypadku, funkcje mogą nie zadziałać lub zwrócić błędne wyniki.

Część III - sprawozdanie

1. Opis problemu

Algorytm Kirkpatrick'a wykorzystywany jest w celu zlokalizowania punktu w figurze będącej triangulacją. Celem projektu było zaimplementowanie tego algorytmu.



Rysunek 5 – wizualizacja problemu

2. Analiza złożoności

n – liczba wierzchołków

Preprocessing:

Przetwarzanie wstępne polega na budowaniu hierarchii trójkątów. Hierarchia ma (log n) poziomów. Na każdym poziomie konstrukcja trójkątów może być wykonana w czasie O(n). Ogólna złożoność przetwarzania wstępnego wynosi O(n * log n).

Odszukiwanie wyniku:

Na każdym poziomie sprawdzenie, czy punkt zapytania znajduje się w trójkącie, może być wykonane w czasie O(1), a poziomów jest (log n). Odszukiwanie wyniku jest zatem w czasie O(log n).

Całkowita złożoność algorytmu wynosi O(n * log n).

3. Pseudokod

Kirkpatrick(T, point):

T – zbiór trójkątów, point – szukany punkt
jeśli otoczka wypukła figury nie jest trójkątem:
    utwórz trójkąt obejmujący wszystkie punkty
    połącz trójkąt z otoczką wypukłą
S – zbiór wszystkich punktów
utwórz drzewo poprzez preprocessing
odszukaj trójkąt zawierający punkt w drzewie

Preprocess(T, S):

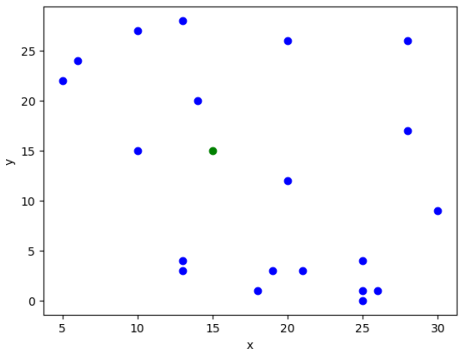
T – zbiór trójkątów w danym momencie, S – zbiór nieprzerobionych punktów
dopóki |T| > 1:
znajdź zbiór niezależnych wewnętrznych wierzchołków
usuń trójkąty zawierające te wierzchołki
usuń wierzchołki z S wraz z krawędziami
ponownie strianguluj S i dodaj nowe trójkąty do T
do drzewa dodaj nowe trójkąty jako dzieci obszaru, z którego powstały

4. Testy

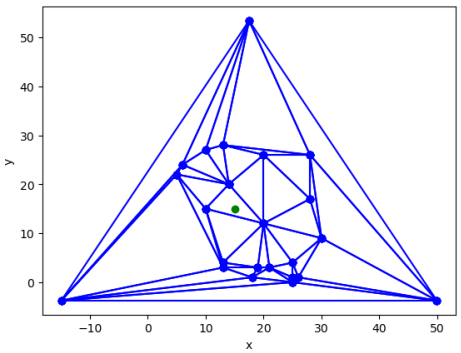
W celu przetestowania napisanego przez nas algorytmu, zdecydowaliśmy się na wybór czterech testów. Pierwszy test jest wygenerowany przez funkcje generate_points dla n = 20, max_x = 30, max_y = 30. Drugi test składa się z figury, która nie zawiera żadnych punktów w środku, a jej otoczka wypukła nie składa się z całej zadanej chmury. W ten sposób sprawdzane jest, czy łączenie trójkąta zewnętrznego z otoczką listy punktów odbywa się poprawnie. Trzeci test natomiast jest szczególnym przypadkiem wykonywania algorytmu, kiedy nie jest wymagane generowanie trójkąta zewnętrznego, gdyż obwód figury już sam w sobie jest trójkątem. Jako ostatnią, czwartą chmurę punktów, wybraliśmy figurę utworzoną po triangulacji, w której podczas preprocessingu powstają wielokąty wklęsłe. Sprawdzimy dzięki temu, czy poprawnie przebiega proces retriangulacji.

Wszystkie testowane chmury są zapisane w mainie w pliku KirkpatrickNoVis.ipynb jako komentarze. Po włączeniu programu z tymi danymi, ukaże się pełna wizualizacja programu, krok po kroku.

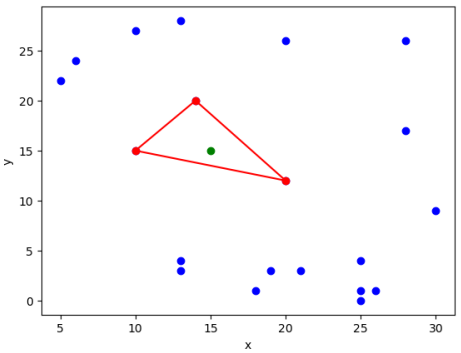
Test nr 1:
searched_point = (15, 15)



Rysunek 6 – wizualizacja zbioru testowego nr 1



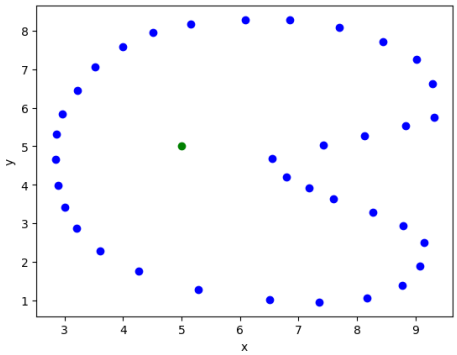
Rysunek 7 – zbiór testowy nr 1 po triangulacji



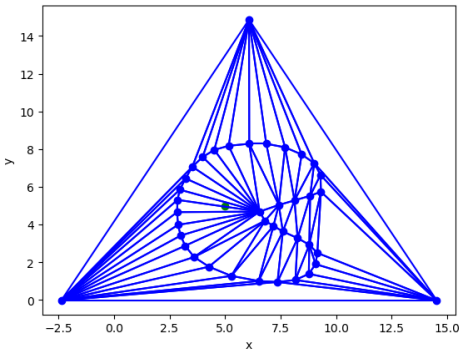
Rysunek 8 – wizualizacja wyjścia dla zbioru testowego nr 1

Wynik: Face: {Point: (10, 15), Point: (20, 12), Point: (14, 20)}

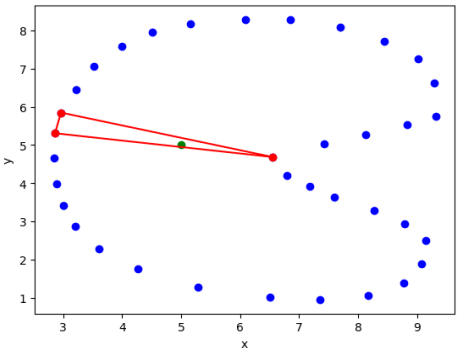
Test nr 2:



Rysunek 9 – wizualizacja zbioru testowego nr 2



Rysunek 10 – zbiór testowy nr 2 po triangulacji

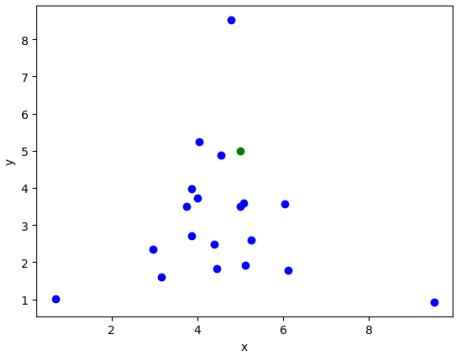


Rysunek 11 – wizualizacja wyjścia dla zbioru testowego nr 2

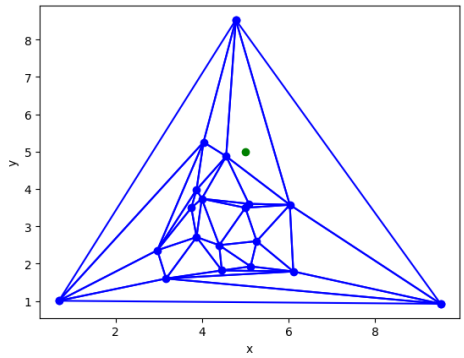
Wynik: Face: {Point: (2.9637096774193545, 5.8495670995671), Point: (2.8629032258064515, 5.30844155844156), Point: (6.55241935483871, 4.686147186147187)}

Jak widać, żaden punkt nie należący do otoczki wypukłej nie został połączony z wierzchołkami trójkąta zewnętrznego, a więc ta część algorytmu wykonywana jest poprawnie.

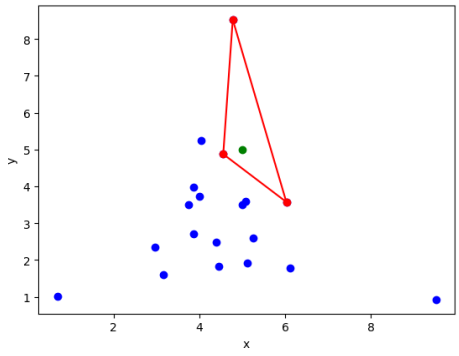
Test nr 3:



Rysunek 12 – wizualizacja zbioru testowego nr 3



Rysunek 13 – zbiór testowy nr 3 po triangulacji

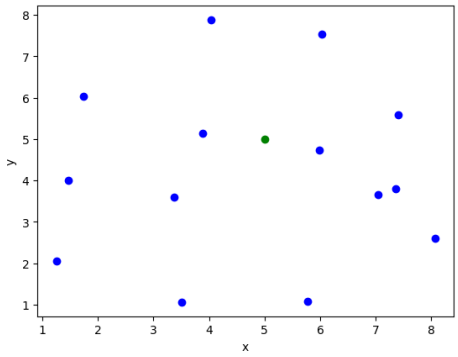


Rysunek 14 – wizualizacja wyjścia dla zbioru testowego nr 3

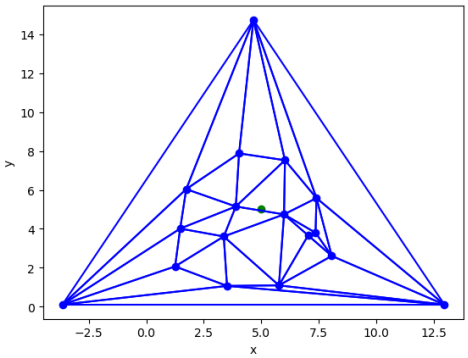
Wynik: Face: {Point: (4.556451612903226, 4.875541125541126), Point: (4.778225806451612, 8.528138528138529), Point: (6.028225806451612, 3.576839826839828)}

Trójkąt zewnętrzny składa się z wierzchołków zadanych w chmurze. Nie został utworzony dodatkowy, większy trójkąt.

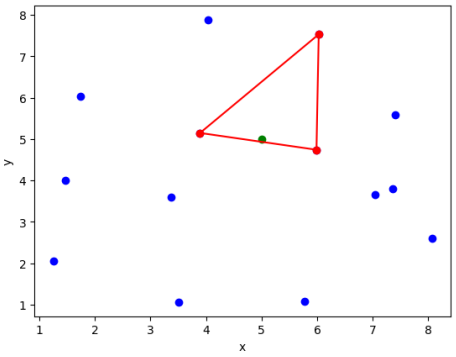
Test nr 4:



Rysunek 15 – wizualizacja zbioru testowego nr 4



Rysunek 16 – zbiór testowy nr 4 po triangulacji



Rysunek 17 – wizualizacja wyjścia dla zbioru testowego nr 4

Wynik: Face: {Point: (3.8911290322580645, 5.146103896103897), Point: (5.987903225806452, 4.740259740259741), Point: (6.028225806451612, 7.527056277056278)}

Poszczególne kroki triangulacji można zobaczyć po wywołaniu funkcji z danymi podpisanymi jako test 4.

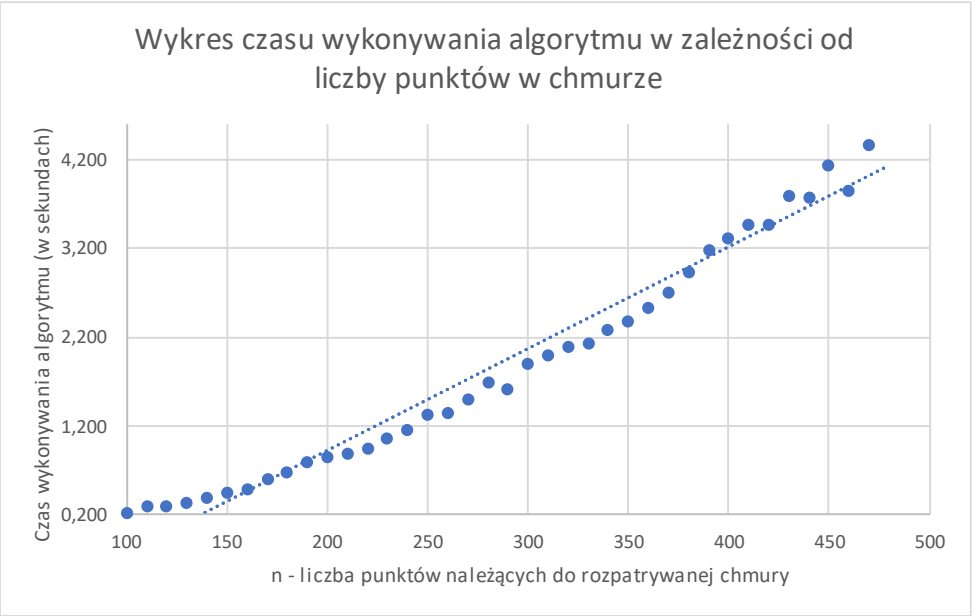
5. Analiza czasu działania funkcji

Poniżej znajduje się tabela z czasami działania funkcji dla przykładowych n punktów generowanych za pomocą funkcji generate_points. Podczas mierzenia czasów, wykomentowane zostały wszelkie fragmenty odpowiedzialne za wizualizację, aby go nie zawyżać. W każdym przypadku szukanym punktem był punkt na samym środku przedziału max_x, max_y, które z kolei były generowane losowo.

Tabela 2 – Tabela czasów działania algorytmu dla różnych n punktów

Liczba punktów	Czas wykonania algorytmu [s]	Liczba punktów	Czas wykonania algorytmu [s]	Liczba punktów	Czas wykonania algorytmu [s]
100	0,235	230	1,073	360	2,534
110	0,308	240	1,168	370	2,708
120	0,298	250	1,331	380	2,947
130	0,347	260	1,364	390	3,190
140	0,391	270	1,514	400	3,319
150	0,457	280	1,701	410	3,468
160	0,499	290	1,617	420	3,477
170	0,618	300	1,914	430	3,797
180	0,688	310	2,008	440	3,789
190	0,796	320	2,098	450	4,146
200	0,865	330	2,145	460	3,861
210	0,890	340	2,298	470	4,367
220	0,960	350	2,381	480	4,625

Po przeniesieniu otrzymanych wartości do Excela oraz utworzeniu wykresu funkcji t(n) (zależności czasu od ilości punktów), otrzymaliśmy następujący wynik



Rysunek 18 – wykres zależności czasu wykonania funkcji od ilości punktów w chmurze

Po analizie złożoności wnioskiem było, że funkcja działa w czasie O(n * log n). Kształt linii trendu, zaznaczony na wykresie przerywaną linią, jest bardzo zbliżony wyglądem do tej właśnie funkcji. Potwierdza to wniosek o oszacowaniu złożoności oraz jest to również informacja o poprawnej (pod względem złożoności) implementacji algorytmu.

6. Wnioski

Algorytm był sporym wyzwaniem do zaimplementowania i pochłonął bardzo dużo czasu. Jednym z problemów podczas była retriangulacja wielokątów. Dopiero po wnikliwej analizie i przeszukaniu wielu stron internetowych, udało się natrafić na metodę „ear clippingu" z gotową jej implementacją w bibliotece. Jej złożoność wynosi $O(n^2)$, jednak ze względu na ograniczone n, czas wykonania jest stały, lecz niestety z tego powodu stała czasowa wykonania całego algorytmu znacznie rośnie. Następnym problemem napotkanym podczas implementacji, było wyznaczanie punktów według ruchu przeciwnego do wskazówek zegara. Ostatecznie posłużyliśmy się sortowaniem punktów przy użyciu funkcji alpha, której opis znaleźliśmy na internecie. Link do jej opisu zamieściliśmy w bibliografii. Ostatecznie implementacja się udała i wszystko działa poprawnie, czego dowodzą wizualizacje utworzone przy użyciu visualizera. Cały algorytm, jak wcześniej było wspomniane, pomimo asymptotycznie optymalnego czasu wykonania, ma bardzo dużą stałą, przez co nie jest on tak efektywny.

Bibliografia

- https://sites.cs.ucsb.edu/~suri/cs235/Location.pdf
- https://ics.uci.edu/~goodrich/teach/geom/notes/Kirkpatrick.pdf
- Prezentacja z wykładu
- https://github.com/charliermarsh/point-location (zaczerpnięta inspiracja o możliwości wykorzystania funkcji Delaunay z biblioteki scipy.spatial)
- stackoverflow.com
- http://www.algorytm.org/geometria-obliczeniowa/porzadkowanie-wierzcholkow-wg-rosnacych-katow-nachylenia-ich-wektorow-wodzacych.html - funkcja alpha
- https://ealter.github.io/point-location/about.html - cenne wskazówki na temat implementacji