Wieloboki Voronoi – porównanie metod konstrukcji

Algorytmy geometryczne

Zespół projektowy nr. 1

Piotr Rzadkowski

Olgierd Smyka

Grupa nr. 3 (czwartek 11:20 w tygodniu B)

styczeń, 2024

Spis treści

1. Wymagania techniczne

Rozwiązania zostały napisane w języku Python wersji 3.9.18, korzystając z Jupyter Notebook, bibliotek Numpy w wersji 1.26.2 i funkcji przygotowanych przez Koło Naukowe Bit, znajdujących się w module *visualizer*. Testy były wykonywane z użyciem Windows Subsystem for Linux z zainstalowanym Ubuntu 22.04 oraz procesora Intel® Core i7-7700HQ 2.80GHz.

Pełny program zawiera następujące pliki:

- Pakiet **visualizer** (przygotowany przez Koło Naukowe Bit) - szczegółowy opis funkcjonalności można znaleźć pod linkiem: https://github.com/aghbit/Algorytmy-Geometryczne,
- **__init__.py,**
- **dataStructures.py,**
- **myLL.py,**
- **priorityQueue.py,**
- **TInterface.py,**
- **utils.py,**
- **fortune.py,**
- **fortune.ipynb,**
- **delaunay.ipynb,**
- **tests.ipynb.**

2. Dokumentacja

Opisane tutaj zostaną metody i klasy publiczne, które należą do API. Prywatne metody nie są przewidziane do wywoływania przez użytkownika.

2.1 Oznaczenia

- **np** – skrócona nazwa biblioteki **numpy.**
- **pd** – skrócona nazwa biblioteki **pandas.**
- **dataclass** – pythonowy dekorator, dzięki któremu klasa staje się klasą przechowującą dane i między innymi automatycznie generuje metodę hashującą i gettery. Parametr *frozen* oznacza, że parametrów danej instancji nie będzie się dało zmienić.
- **classmethod** – oznacza, że metoda będzie statyczna.

2.2 Plik fortune.py – klasa Voronoi

Zawiera główna strukturę odpowiedzialną za tworzenie diagramu Voronoi. Udostępnia ona następujące metody:

- ***get_voronoi(self, points)*** – funkcja zwraca krawędzie utworzonego diagramu Voronoi w postaci listy krotek krawędzi i parę wierzchołków, które ograniczają pudełko, w którym diagram został zamknięty. (W praktyce wywołuje ona metodę *get_voronoi_visualised*, której opis znajduje się poniżej z tą różnicą, że ignoruje wizualizację.)

- ***get_voronoi_visualised(self, points)*** – funkcja zwraca krawędzie Voronoi, pudełko, jak i również instancje klasy *Visualizer* z modułu *visualizer*, która będzie pomocna do późniejszej wizualizacji. Schemat działania samej funkcji jest następujący: dodanie punktów do kolejki, przetworzenie każdego zdarzenia z kolejki w zależności czy jest on zdarzeniem punktowym, czy okręgowym, a na koniec dokończenie krawędzi, które zostały w strukturze stanu i ucięcie krawędzi, tak aby zmieściły się w pudełku.

2.3 Plik dataStructures.py

- ***Node*** – struktura odpowiedzialna za reprezentację węzłów w strukturze stanu. Pole *arc* i *arc_pair* odnoszą się do przechowywanego w danym węźle łuku (jeśli dany węzeł jest liściem) lub sąsiadujących ze sobą łuków w danej kolejności (jeśli jest to węzeł wewnętrzny). Ponadto klasa posiada metodę *parabolaIntersect(),* która zwraca punkt, w którym parabole przechowywane w *arc_pair* się przecinają.

- ***Point*** – struktura reprezentującą punkt w przestrzeni $\mathbb{R}^2$.

- ***Edge*** – struktura reprezentująca krawędź określoną przez dany punkt przyłożenia *start* i kierunek w którym krawędź idzie *direction*. Będziemy jej potrzebować aby reprezentować pół-krawędzie podczas budowy diagramu Voronoi. Podczas zapisywania krawędzi będziemy wywoływać metodę *close_edge*, która zapisuje koniec krawędzi do zmiennej *end*. Pole *twin* jest wskaźnikiem na inną krawędź która ma wspólny początek.

- **Arc** – struktura reprezentująca łuk. Określa go poprzez ognisko *focus* i aktualne położenie kierownicy *directrix*. Dostarcza ona również następujące metody:

  - ***setDirectrix(cls, directrix, all=True)*** – przesuwa aktualne położenie kierownicy. W zależności od parametru *all* (domyślnie *True*) ustawia lub nie kierownice dla wszystkich instancji klasy *Arc*.

  - ***setLeftEdge(self, side_arc, start=None)*** – znajduje i ustawia wartość lewej krawędzi na taką, której tor pokrywa się z przecięciem parabol *side_arc* i instancji, na której metoda została wywołana (*self*). Jeśli parametr *start* ustawiony jest na *None*, program uznaje, że *side_arc* leży nad *self*, w wyniku czego początek lewej krawędzi rozpocznie się w punkcie na paraboli *side_arc*, znajdującym się na współrzędnej x równej współrzędnej x ogniska paraboli *self.*

- *setRightEdge(self, side_arc, start=None)* – analogicznie jak powyższa funkcja, ale ustawia krawędź pomiędzy *self* a *side_arc*.

- ***value(self, x, directrix=None)*** – zwraca wartości paraboli dla każdego argumentu wektora X.

- ***draw(self, vis, box)*** – generuje wektor argumentów *x,* a następnie pobiera wartości paraboli i filtruje je tak, aby mieściły się one w granicach rysowania diagramu Voronoi - *box*. Na koniec dodaje do wykresu *vis*.

- ***def lookupForIntersectionBetween(self, right_arc)*** – szuka, w którym punkcie parabola *self* przetnie się z parabolą *right_arc* „w przyszłości", to znaczy kiedy kierownica przesunęłaby się niżej. Oblicza taki punkt i zwraca go. Punkt ten będzie kierunkiem dla krawędzi pomiędzy parabolami *self* i *right_arc*.

- ***lookupIntersectionsWithHigher(self, higher)*** – znajduje dwa przecięcia z parabolą *higher,* która znajduje się wyżej niż parabola, na której metoda została wywołana. Wykorzystuje metodę *lookupForIntersectionBetween.*

- ***intersect(self, arc)*** – zwraca przecięcie dwóch paraboli.

- ***Event*** – struktura reprezentująca zdarzenie, będące w strukturze zdarzeń. Przechowuje ona informacje o punkcie zdarzenia *point*, o jego typie *type (*może być albo punktowe *site,* albo okręgowe *circle*) i w przypadku zdarzenia okręgowego wskaźnik na węzeł, którego zdarzenie dotyczy, środek okręgu i informację czy zdarzenie jest fałszywym alarmem.

- ***Pair*** – struktura, która reprezentuje parę łuków z określeniem, który leży po której stronie (pola *left* i *right*). Posiada metodę *parabolaIntersect*, która wywołuje metodę *intesect* między *left,* a *right* i zwraca współrzędną x przecięcia. Współrzędna x jest nam potrzebna do wyszukiwania odpowiedniego węzła w strukturze stanu.

2.4 Plik priorityQueue.py – klasa PriorityQueue

Struktura reprezentująca kolejkę zdarzeń w algorytmie Fortune'a. Udostępnia ona następujące metody:

- ***__init__(self, items=[])*** – konstruktor tworzący instancje kolejki. Możliwe jest podanie mu kolekcji początkowych zdarzeń za pomocą parametru *items.*

- ***add(self, event: Event)*** – dodaje nowe zdarzenie *event* do kolejki. Jeśli jest to zdarzenie okręgowe, sprawdza, czy w kolejce nie znajduje się już zdarzenie okręgowe dla węzła, którego to zdarzenie dotyczy. Jeśli tak jest, nadpisuje je nowym zdarzeniem.

- ***pop(self)*** – zwraca następne zdarzenie w kolejce, które nie jest fałszywym alarmem. Jeśli kolejka się skończy, zwraca *None*.

- ***delete(self, item: Event)*** – usuwa element z kolejki (znajduje element i oznacza go jako fałszywy alarm).

2.5 Plik TInterface.py – klasa T

Struktura będąca interfejsem dla struktury stanu, dla algorytmu Fortune'a. Wszystkie metody będą opisane szczegółowo dla implementacji tego interfejsu.

2.6 Plik myLL.py – klasa myLL

Implementacja struktury stanu *T* na bazie LinkedListy. Implementuje ona następujące metody:

- ***find_node(self, p)*** – znajduje węzeł, w którym jest łuk pokrywający punkt *p.*

- ***replace(self, arc_node: Arc, new_arc: Arc)*** – metoda ta jest wywoływana w przypadku zdarzenia punktowego. Jej zadaniem jest podział węzła, który poprzednio reprezentował łuk nowym łukiem, który odpowiada aktualnemu zdarzeniu punktowemu oraz odpowiedni podział tych łuków węzłami odpowiadającymi przecięciom łuków. Zwraca dwie części łuku, który został podzielony w celu sprawdzenia wystąpienia zdarzeń okręgowych.

- ***insert(self, p)*** – funkcja odpowiedzialna za wstawienie łuku o ognisku w punkcie *p.* Za pomocna metody *find_node* znajduje ona odpowiedni łuk, następnie wywołuje powyżej opisaną metodę *replace* i na podstawie zwróconych łuków sprawdza czy pojawią się nowe zdarzenia okręgowe, a następnie zwraca stary węzeł i znalezione zdarzenia okręgowe.

- ***checkForCircleEvent(self, node)*** – tworzy i zwraca zdarzenie okręgowe dla węzła *node.* Jeśli takiego zdarzenia nie ma, zwraca *None*.

- ***isCirleEvent(self, arc_node: Node)*** – dla wezła *arc_node* zwraca punkt zdarzenia okręgowego i środek tego okręgu lub *None* jeśli takie zdarzenie nie nastąpi.

- ***handleSquize(self, arc_node)*** – celem funkcji jest usunięcie wezła *arc_node*, czyli takiego, który w zdarzeniu okręgowym zniknie. Aby tego dokonać muszą być również usunięte węzły zawierające przecięcie *arc_node* i jego prawego (*ar)*, i lewego (al) sąsiada. Na koniec następuje sprawdzenie czy pojawi się zdarzenie okręgowe dla tych sąsiadów. Zwraca te zdarzenia.

- ***leftNbour(self, node)*** – zwraca lewy łuk licząc od *node (*w tej strukturze na zmianę są węzły zawierające łuk i węzły zawierające przecięcie łuków; cofamy się „2 razy”*).*

- ***rightNbour(self, node)*** – jak wyżej, ale prawy łuk licząc od *node*.

- ***print(self)*** – wyświetla w linii komend kolejne ogniska łuków w linii brzegowej (dla rosnących współrzędnych x).

2.7 Plik utils.py

- funkcja ***distance(p1: Point, p2: Point)*** – zwraca dystans między dwoma punktami w metryce Euklidesowej.

- funkcja ***mat_det(a, b)*** – zwracająca wyznacznik macierzy 2x2 dla punktów *a* i *b.*

- funkcja ***getIntersect(start1: Point, direction1: Point, start2: Point, direction2: Point)*** – zwraca przecięcie dwóch półprostych o określonych punktach przyłożenia i określonym kierunku. Zwraca *None* jeśli takie przecięcie nie istnieje.

- funkcja ***lineSegmentIntersect(start: Point, end: Point, line_start: Point, line_direction: Point)*** – działa podobnie jak funkcja wyżej, ale zwraca przecięcie między odcinkiem, a półprostą.

2.8 Plik fortune.ipynb

Plik, w którym pokazane są przykłady działania algorytmu.

2.9 Plik delaunay.ipynb

Jest to plik zawierający zarówno całość kodu przeznaczonego do wyznaczania konstrukcji Voronoi poprzez triangulację Delaunay'a, jak i do prezentowania otrzymanych wyników wraz z wizualizacją całego procesu. Notatnik podzielony jest na następujące części:

1. Klasy obiektów – Delaunay

Zawiera klasy wykorzystane do rozwiązania problemu wyznaczenia triangulacji Delaunay'a:

- Klasa punktu - ***Point***
  Struktura reprezentuje punkt w przestrzeni $\mathbb{R}^2$. Obiekty klasy *Point* są hashowalne oraz mogą być ze sobą porównywane. Zaimplementowano także metodę wyznaczania odległości od innego punktu (*distance(self, p)*).

- Klasa trójkąta – ***Triangle***
  Jest to klasa reprezentująca trójkąt. Jej atrybuty to:
  - ***p1, p2, p3*** – wierzchołki trójkąta,
  - ***o*** - środek okręgu opisanego na trójkącie,
  - ***R*** - długość promienia okręgu opisanego na trójkącie.

  Wierzchołki trójkąta i środek okręgu na nim opisanego są obiektami klasy *Point.* Atrybuty *o* oraz *R* inicjalizowane są w metodzie *set_circle(self)*. Obiekty klasy są hashowalne oraz mogą być porównywane z innymi instancjami tej klasy. W oparciu o wyznaczanie położenia punktu względem prostych zawierających krawędzie trójkąta możliwym jest sprawdzenie czy punkt należy do trójkąta (metoda *__contains__(self, p)*). Sprawdzenie położenia punktu względem prostej realizowane jest z wykorzystaniem wyznacznika 3x3 własnej implementacji. Pozostałe metody klasy to:

  - ***in_circle(self, p)*** – sprawdza czy przekazany jako argument punkt należy do koła opisanego na trójkącie (czy odległość przekazanego w argumencie punktu od środka okręgu opisanego na trójkącie *o* jest mniejsza lub równa od promienia okręgu opisanego na tym trójkącie z uwzględnieniem niepewności pomiarowej *EPS*).

  - ***sort_tri_vertexes(self)*** - sortuje wierzchołki trójkąta, tak, że *p1* jest punktem o najmniejszej współrzędnej y i w przypadku braku rozstrzygnięcia punktem o najmniejszej współrzędnej x, a wierzchołki *p1, p2, p3* są w kolejności przeciwnej do ruchu wskazówek zegara.

  - ***get edges(self)*** – zwraca zbiór (*set*) krawędzi trójkąta (krawędzie reprezentowane są jako krotki punktów).

- Klasa triangulacji – ***Triangulation***
  Jej atrybuty to:

  - ***edges*** – reprezentuje zarówno krawędzie triangulacji jak i nie wprost jej trójkąty. Jest to słownik, którego kluczem są krawędzie, których punkty krańcowe są jako wierzchołki trójkąta w kolejności przeciwnej do ruchu wskazówek zegara, a wartością trzeci z wierzchołków trójkąta triangulacji.

  - ***triangles*** – zbiór (set) trójkątów triangulacji.

  - ***data_frame*** – dwuelementowa tablica, której pierwszym elementem jest lewy-dolny, a drugim prawy-górny wierzchołek prostokąta zawierającego wszystkie punkty zbioru wejściowego o najmniejszym polu i równoległych do osi współrzędnych bokach.

- ***map_vertexes*** – tablica wierzchołków prostokąta tworzonego przez początkową triangulację, a w kontekście kontrukcji Voronoi tablica wierzchołków prostokąta ograniczającego krawędzie diagramu.

- ***cenral_point*** – punkt określający swoim zawieraniem się trójkąt triangulacji *central_triangle.*

- **central_triangle** – trójkąt od którego rozpoczynane są poszukiwania trójkąta triangulacji zawierającego nowododawany w każdej iteracji algorytmu Bowyera-Watsona punkt.

Metody klasy:

- ***get_triangulation_start(self, P)*** – metoda wykorzystywana w konstruktorze klasy do inicjalizacji atrybutów. Jako argument przyjmuje chmurę punktów wejściowych.

- ***adjust_map_size(self, map_size=3)*** – wykorzystywana w przetwarzaniu triangulacji Delauna'a na konstrukcję Voronoi. Zmienia atrybut *map_vertexes*, tym samym zmniejszając prostokąt ograniczający chmurę punktów do rozmiaru pozwalającego na dobre przedstawienie wizualne wyników algorytmu.

- ***find_adjacent_tri(self, edge)*** – przyjmując w argumencie krawędź trójkąta o posortowanych wierzchołkach zwraca trójkąt triangulacji dzielący z nim krawędź lub *None* w przypadku, gdy trójkąt o zadanej krawędzi nie dzieli tej krawędzi z innym trójkątem.

- ***find_all_adjacent_tri(self, triangle)*** – przyjmując w argumencie trójkąt zwraca zbiór zawierający wszystkie trójkąty dzielące z nim krawędź.

- ***remove(self, triangle)*** – usuwa trójkąt z triangulacji.

- ***add(self, triangle)*** – dodaje trójkąt do triangulacji.

- ***adjust_triangulation(self, tri_to_remove, p)*** – aktualizuje triangulację o dodawany punkt *p*, gdy wyznaczono już obszar (listę trójkątów *tri_to_remove*) do aktualizacji.

- ***remove_map_vertexes(self)*** – usuwa z triangulacji wszystkie te trójkąty triangulacji, których przynajmniej jeden z wierzchołków należy do wierzchołków początkowej triangulacji *map_vertexes.*

2. Funkcje pomocnicze funkcji Delaunay

Zawiera funkcje pomocnicze głównej funkcji wyznaczającej triangulację Delaunay'a algorytmem Bowyera-Watsona. Są to:

- ***get_Points(Points)*** – zamienia daną, początkową listę punktów *Points* reprezentowanych jako krotki współrzędnych na listę punktów reprezentowanych jako obiekty klasy Points.

- ***find_containing(T, p)*** – rozpoczynając w *T.central_triangle* szuka w sposób uporządkowany wśród trójkątów triangulacji *T,* trójkąta zawierającego punkt *p*. Określa w tym celu położenie punktu *p* względem prostych zawierających krawędzie trójkąta, aby określić optymalny do analizy następny trójkąt.

3. Główna funkcja Delaunay

Zawiera implementację głównej funkcji algorytmu Bowyera-Watsona zwracającej obiekt triangulacji Delaunay'a - *Triangulation.*

4. Dane testowe

Umieszczono w niej funkcje pomocnicze do generowania chmury punktów i wyświetlania wyznaczonej triangulacji. W tej części przygotowano także przykładowe zbiory punktów, na których wykonano algorytm wyznaczania triangulacji Delaunay'a.

5. Testy - Delaunay

Przedstawiono w niej na wykresach wyniki funkcji triangulującej chmurę punktów dla przykładowych zbiorów punktów.

6. Diagram Voronoi

Zawiera kod wykorzystywany do uzyskiwania konstrukcji Voronoi z triangulacji Delaunay'a. Na potrzebę przechowywania konstrukcji utworzono nową klasę *Voronoi.* Posiada ona jeden atrybut – zbiór (*set*) krawędzi diagramu Voronoi - *edges*. Klasa ta nie posiada żadnych metod.

Stworzono liczne funkcje pomocnicze:

- ***get_triangles_o(triangle, triangle_edge, T)*** – zwraca listę *triangles_o*, której pierwszym argumentem jest środek trójkąta *triangle*, a drugim środek trójkąta przyległego do *triangle*, dzielącego z nim wspólną krawędź *triangle_edge* (jeśli istnieje).

- ***is_in_map(o, T)*** – sprawdza czy punkt *o* należy do prostokąta o wierzchołkach *T.map_vertexes.*

- ***is_added(edge, V)*** – sprawdza czy krawędź *edge* została już dodana do krawędzi konstrukcji Voronoi (do *V.edges*).

- ***check_intersection(e1, e2)*** – przyjmuje za argumenty dwa odcinki i zwraca ich punkt przecięcia. Jeśli odcinki nie przecinają się, funkcja zwraca *None*. Wykorzystana została postać parametryczna odcinka.

- ***found_adjacent(T, V, triangles_o)*** – funkcja służy wyznaczeniu odcinka łączącego punkty środków okręgów opisanych na trójkątach, znajdujące się w *triangles_o*. Odcinek ten jest krawędzią konstrukcji Voronoi. W celu poprawnego ograniczenia konstrukcji krawędziami prostokąta o wierzchołkach *T.map_edges* rozpatrywane są przypadki, że: *triangles_o[0]* oraz *triangles_o[1]* leżą wewnątrz prostokąta ograniczającego, *triangles_o[0]* leży wewnątrz prostokąta ograniczającego, a *triangles_o[1] nie, triangles_o[0]* nie leży wewnątrz prostokąta ograniczającego, a *triangles_o[1] tak, triangles_o[0]* i *triangles_o[1]* nie leżą wewnątrz prostokąta ograniczającego.

- ***get_parallel_line(triangle_edge, point)*** – zwraca odcinek imitujący półprostą zaczynającą się w *point* i przechodzącą przez środek krawędzi trójkąta (odcinka) *triangle_edge. W przypadku, gdy point* jest środkiem okręgu opisanego na trójkącie, którego jedną z krawędzi jest *triangle_edge*, to otrzymana półprosta zawiera symetralną *triangle_edge.*

- ***get_opposite_parallel_line(triangle_edge, point)*** – zwraca odcinek imitujący półprostą zaczynającą się w *point*, której zwrot jest przeciwny do zwrotu wektora zaczepionego w *point,* zwróconego w kierunku środka krawędzi trójkąta (odcinka) *triangle_edge.* Gdy *point* to środek okręgu opisanego na trójkącie, którego krawędzią jest *triangle_edge*, to zwracana półprosta zawiera symetralną *triangle_edge.*

- ***not_found_adjacent(T, V, triangle_edge, triangle)*** – funkcja wyznacza krawędź konstrukcji Voronoi będącą ograniczoną przez prostokąt o wierzchołkach *T.map_vertexes* półprostą o początku w środku okręgu opisanego na trójkącie *triangle.o* oraz zawierającą się w symetralnej krawędzi trójkąta *triangle* – *triangle_edge* . W celu poprawnego wyznaczenia odcinka analizowane jest położenie punktu *triangle.o*. Sprawdzane jest czy leży on wewnątrz trójkąta *triangle*, a jeśli nie to po której stronie *triangle_edge* się znajduje.

- ***add_edge_no_triangles(V, edge, map_edges)*** – funkcja pomocnicza funkcji *no_triangles()* dodająca do krawędzi diagramu Voronoi symetralną odcinka *edge* ograniczoną krawędziami prostokąta zawartymi w *map_edges.*

- ***no_triangles(T, V, Points)*** – funkcja wyznaczająca krawędzie diagramu Voronoi w przypadku zdegenerowanym, gdy triangulacja Delaunay'a nie zawiera żadnego trójkąta.

- ***add_edge_V(T, V, triangle_edge, triangle)*** – funkcja wywołująca odpowiednią z funkcji pomocniczych *found_adjacent()* lub *not_found_adjacent()* w zależności od tego czy trójkąt *triangle* dzieli krawędź *triangle_edge* z innym trójkątem triangulacji.

Główną funkcją algorytmu jest:

- ***voronoi(T, Points)*** – funkcja przyjmująca jako argumenty instancję klasy *Traingulation* wyznaczonej triangulacji Delaunay'a *T* oraz rozpatrywany zbiór punktów *Points.* Analizuje kolejno wszystkie trójkąty triangulacji generując zbiór krawędzi konstrukcji Voronoi. Funkcja zwraca krawędzie diagramu Voronoi jako krotki punktów końcowych odcinka.

7. Dane testowe - Voronoi

- Umieszczono w niej funkcje pomocnicze do prezentowania wyników konstrukcji diagramu Voronoi oraz wyników triangulacji zbioru punktów wraz z opartą o nią konstrukcją diagramu Voronoi.

8. Testy - Voronoi

- Przedstawiono w niej na wykresach wyniki funkcji *voronoi()* konstruującej diagram Voronoi oraz wyniki funkcji *delaunay()* zestawione z wyznaczonym diagramem Voronoi dla przykładowych zbiorów punktów.

9. Kod do wizualizacji

- Zawiera przerobione pod kątem możliwości stworzenia wizualizacji działania algorytmu funkcje i metody używane do wyznaczenia konstrukcji Voronoi, opisane powyżej.

10. Przykłady wizualizacji

- Zawiera wizualizacje działania algorytmu dla przykładowych, zadanych chmur punktów.

2.10 Plik tests.ipynb

- funkcja ***read_test_data(new=False)*** – funkcja odpowiedzialna za odczytanie (lub wygenerowanie) danych, na których algorytmy będą testowane. Dane są generowane na nowo, jeśli algorytm nie znajdzie pliku z danymi, lub jeśli parametr *new* jest ustawiony na *True.* W takim przypadku wygenerowane dane są zapisywane do pliku. Rozwiązanie to wspiera wielokrotne wykonywanie testów na tych samych danych.

- funkcja ***run_tests(saveOutput=True, tested=['fortune', 'delaunay'])*** – główna funkcja wykonująca testy. Wczytuje ona dane testowe za pomocą *read_test_data,* a następnie dla każdych danych mierzy czas wykonania, dla każdej z funkcji opisanej w parametrze *tested.* Jeśli parametr *saveOutput* jest ustawiony na *True,* funkcja zapisuje wyniki triangulacji do pliku wynikowego.

- funkcja ***save_and_visualize(new=True)*** – funkcja, której celem jest wizualizacja wyników testów. Jeśli parametr *new* jest ustawiony na *True*, to funkcja wywołuje funkcję *run_test* na nowo, aby dostać nowe wyniki. W przeciwnym przypadku czyta wyniki testów z pliku. Wyniki przedstawiane są w tabeli za pomocą biblioteki *Pandas* i rysowane na wykresach.

3. Poradnik do wykorzystania

3.1 Generowanie diagramu Voronoi algorytmem Fortune

- **bez wizualizacji**

Tworzymy zbiór punktów:

```
points = [(0, 0), (1, 1), (2, 3), (-1, 2.5)]
```

Tworzymy instancje klasy *Voronoi* z listą punktów jako parametrem, a następnie wywołujemy metodę *get_voronoi:*

```
vor = Voronoi(points)
vor_edges, box = vor.get_voronoi()
```

W zmiennej *vor_edges* zapisana zostanie lista krawędzi wygenerowanych przez klasę:

```
print(vor_edges)
✓ 0.0s
[((0.5454545454545445, 2.4772727272727284), (0.2916666666666563, 4.0)), ((1, 2.25), (0.545454
```

Natomiast w zmiennej *box* znajduje się lewy dolny i prawy górny wierzchołek pudełka.

```
left_bottom, right_top = box
print(left_bottom, right_top)
✓ 0.0s
Point(-2.000, -1.000) Point(3.000, 4.000)
```

- **z wizualizacją**

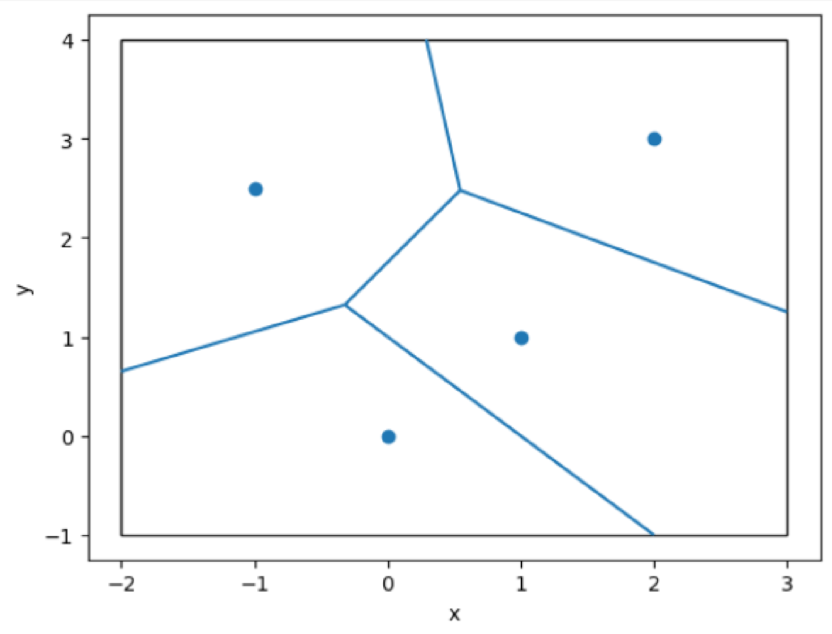
W przypadku metody *get_voronoi_visualised* dostajemy również 3 obiekt, czyli Visualizer.

```
vor = Voronoi(points)
vor_edges, box, vis = vor.get_voronoi_visualised()
```

Zgodnie z opisem, na Github, narzędzia graficznego od Koła Naukowego Bit, Visualizer wspiera metody *show()* i *show_gif()*, które odpowiednio zwracają wizualizacje gotowego diagramu i proces jego tworzenia.

```
vis.show()  # przykład dla punktów points=[(0, 0), (1, 1), (2, 3), (-1, 2.5)]
```

✓ 0.2s



Rysunek 1 – prezentacja wyniku programu dla wywołania metody show()

3.2 Generowanie diagramu Voronoi poprzez triangulację Delaunay'a

W celu wygenerowania diagramu Voronoi na zaproponowanych w notatniku chmurach punktów, zarówno z wizualizacją jak i bez, zaleca się wykonanie całej zawartości pliku *delaunay.ipynb*. Wykonywanie poszczególnych bloków kodu może skutkować niepoprawnym oraz nieprzewidywalnym zachowaniem programu, w szczególności z powodu zmian definicji metod obiektów na potrzeby wizualizacji w sekcji *Kod do wizualizacji.* Aby wygenerować diagram Voronoi własnej chmury punktów wraz z wykresem służącym wizualizacji zaleca się w sekcji *Testy – Voronoi* dodanie bloku kodu postaci:

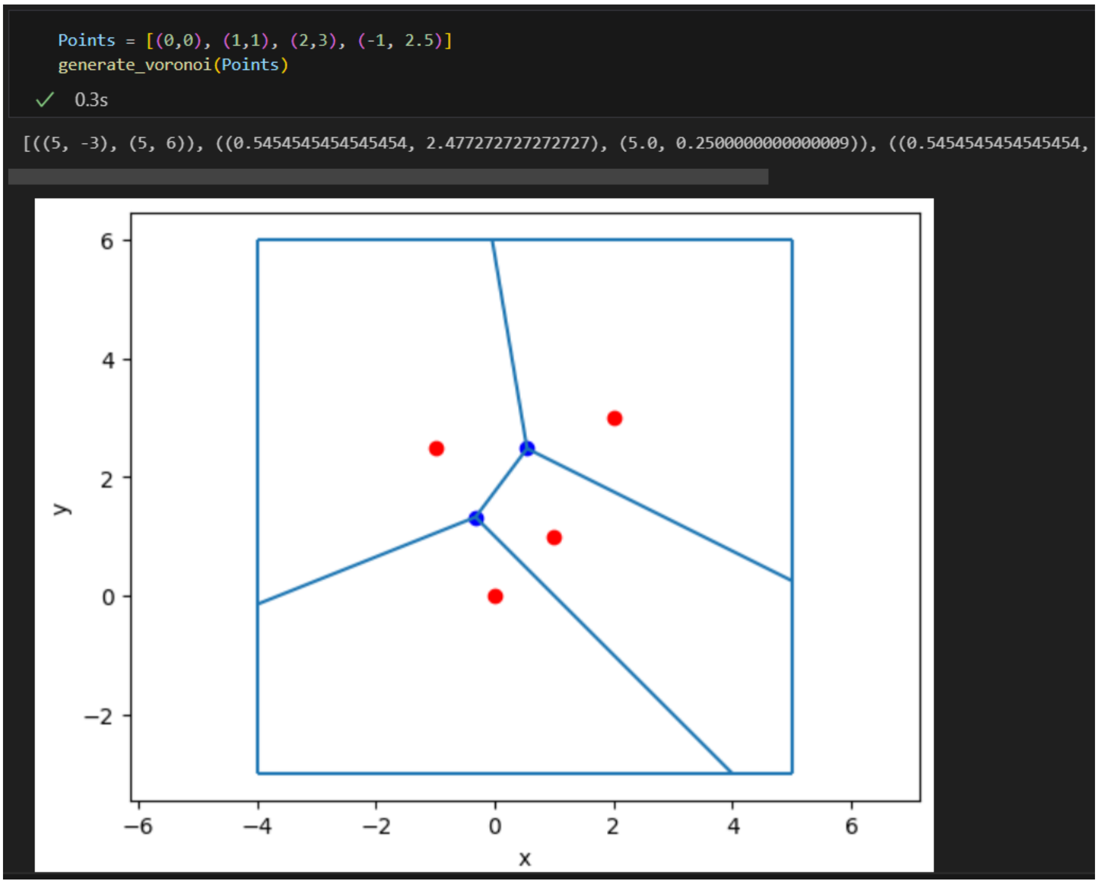
```
Points = [(0,0), (1,1), (2,3), (-1, 2.5)]
generate_voronoi(Points)
```

Gdzie:

- *Points* to punkty, z których chcemy wyznaczyć diagram Voronoi.
- *Generate_voronoi()* to funkcja wypisująca zwrócone przez *voronoi()* krawędzie diagramu Voronoi zadanego zbioru punktów *Points* oraz rysująca wykres konstrukcji.

Następnie należy wykonać kod umieszczony w notatniku powyżej dodanego fragmentu oraz sam blok (od początku do sekcji *Testy - Voronoi* włącznie).

Dla przykładu w rezultacie wywołania powyższego bloku kodu otrzymano:



Rysunek 2 – prezentacja wyniku programu dla wywołania funkcji generate_voronoi()

4. Sprawozdanie

4.1 Opis ćwiczenia

Zadaniem do wykonania było zaimplementowanie dwóch algorytmów tworzenia diagramu Voronoi, czyli podziału płaszczyzny z zadanymi punktami P na taki, aby w każdej komórce był tylko jeden punkt P, i aby odległość każdego innego punktu znajdującego się w komórce K_i od punktu P_i w danej komórce była mniejsza niż odległość od każdego innego punktu P_j ($i \neq j$).

4.2 Realizacja ćwiczenia

Oba zaimplementowane algorytmy przyjmują wejściowy zbiór punktów jako tablicę krotek współrzędnych. Algorytm Fortune'a zwraca konstrukcję Voronoi jako listę krawędzi (krotek) wraz z lewym-dolnym oraz prawym-górnym wierzchołkiem ograniczającego diagram prostokąta. Algorytm wyznaczania diagramu Voronoi z triangulacji Delaunay'a, z kolei, zwraca zbiór krawędzi (reprezentowanych jako krotki punktów) wyznaczonej konstrukcji, do których zaliczone zostały również krawędzie prostokąta ograniczającego.

4.3 Algorytm Fortune

Idea

Pierwszym algorytmem, który zaimplementowaliśmy był algorytm wymyślony przez Stevena Fortune'a. Algorytm ten opiera się na idei algorytmów zamiatania. Zawiera on strukturę zdarzeń, przechowującą dwa rodzaje zdarzeń – punktowe i okręgowe oraz strukturę stanu przechowującą informacje o łukach tworzących linię brzegową. Podczas przesuwania miotły, łuki parabol mających ognisko w punktach, dla których chcemy znaleźć diagram Voronoi, rozszerzają się i przecinają ze sobą nawzajem. Okazuje się, że gdyby na przecięciach tych parabol umieścić krawędzie, to utworzona siatka byłaby diagramem Voronoi.

Struktura zdarzeń Q

Struktura zdarzeń w tym algorytmie musi przechowywać informacje o kolejnych zdarzeniach, które mają być uporządkowane względem współrzędnej y malejąco, ale również wspierać możliwość usunięcia zdarzenia z kolejki. Aby spełnić oba te wymagania zaimplementowaliśmy własną kolejkę, która używa funkcji wbudowanej *heapq*, tworzącą ze zwykłej listy kopiec. Jako mechanizm usuwania wykorzystaliśmy zapisywanie do osobnego słownika każdego wstawionego zdarzenia, wraz z wskaźnikiem na niego w kolejce. Kiedy dane zdarzenie trzeba było usunąć mogliśmy go łatwo znaleźć i oznaczyć jako usunięty. Wtedy kiedy wyciągaliśmy z kolejki element oznaczony jako usunięty, ignorowaliśmy go i wyciągaliśmy kolejny.

Struktura stanu T

W strukturze stanu będziemy przechowywać informacje o łukach i przecięciach między tymi łukami. Łuki będą uporządkowane rosnąco względem współrzędnej x, więc optymalnym rozwiązaniem byłoby użycie zbalansowanego drzewa wyszukiwań binarnych, w którym, w liściach przechowujemy łuki, a w węzłach przecięcia łuków. Wtedy znajdywanie odpowiedniego łuku działa w czasie logarytmicznym, a ta operacja jest najdroższa w naszym algorytmie. Niestety jednak, nasza implementacja wykorzystuje LinkedListę zamiast drzewa, ponieważ implementacja w oparciu o strukturę drzewa wyszukiwań binarnych okazała się być zbyt skomplikowana. Złożoność znalezienia danego łuku wydłuża się więc do liniowej. Niemniej algorytm działa prawidłowo.

Schemat działania algorytmu

Algorytm składa się z następujących kroków:

1. Dodanie punktów do kolejki jako zdarzenia punktowe.
2. Dla każdego zdarzenia z kolejki odpowiednie go przetworzenie.
3. Zdjęcie ze struktury stanu pozostałych krawędzi i dodanie ich do diagramu Voronoi, odpowiednio obcinając je tak, aby nie były nieskończonej długości – z wykorzystaniem pudełka.

Przetworzenie zdarzenia punktowego

Przetworzenie zdarzenia punktowego dla punktu *p* składa się następujących kroków. Najpierw należy znaleźć łuk pod którym leży punkt. Nazwijmy go l. Następnie rozdzielamy go na dwa (nowo stworzone łuki nazwiemy odpowiednio l_l, l_p) i wstawiamy między nie łuk l_n o ognisku w p. Tak utworzoną sekwencję $<l_l, l_p, l_n>$ należy rozdzielić jeszcze przecięciami sąsiadujących łuków, które będziemy reprezentować jako półproste (nazywane potem krawędziami), których początek jest w punkcje na łuku l dla x równego współrzędnej x punktu *p*, a następnie wstawić do struktury T zamiast łuku l. Następnym krokiem jest sprawdzenie czy nie pojawią się zdarzenia okręgowe. Musimy zrobić to dla l_l i l_p. Sprawdzenie to polega na pobraniu sąsiadujących do danego łuku krawędzi i określeniu czy się przetną. Przecięcie tych krawędzi będzie oznaczało, że dwa sąsiadujące łuki wyprą w linii brzegowej sprawdzany łuk, czyli pojawienie się zdarzenia okręgowego.

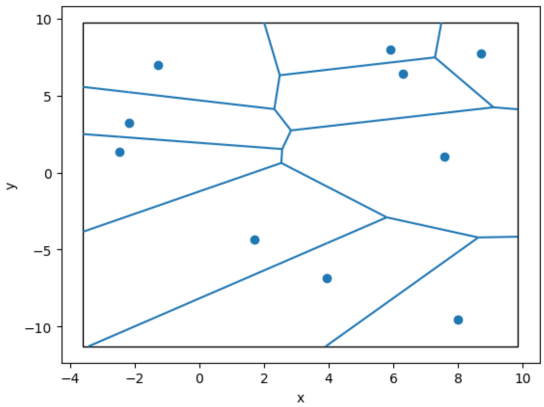
Przetworzenie zdarzenia okręgowego

W zdarzeniu okręgowym, które dotyczy łuku *l,* pobieramy krawędzie sąsiadujące z łukiem *l* (oznaczmy je przez e_l i e_r). Zdarzenie to oznacza, że opisane krawędzie będzie można zakończyć i dodać do finalnego diagramu. Następnie usuwamy z *T* łuk *l* i krawędzie e_l, e_r, a zamiast nich wstawiamy węzeł reprezentujący przecięcie łuków będących prawym i lewym sąsiadem *l*. Na koniec sprawdzamy pojawienie się zdarzenia okręgowego dla opisanych sąsiadów.

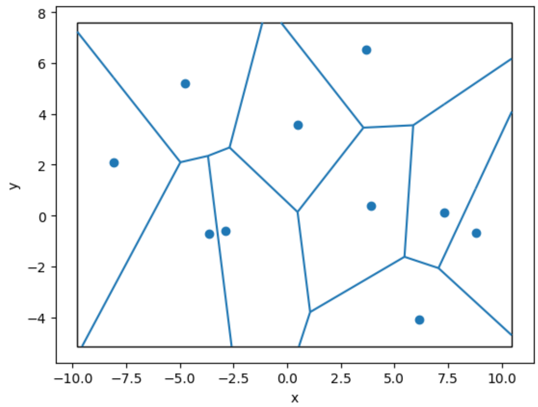
Zakończenie algorytmu

Kiedy w kolejce skończą się zdarzenia, należy zdjąć z *T* pozostałe krawędzie i dodać je do diagramu. Ponieważ krawędzie te są nieskończone, musimy je ograniczyć, a robimy to przez pudełko, będące prostokątem zawierającym wszystkie punkty, zwiększonym o dany margines.

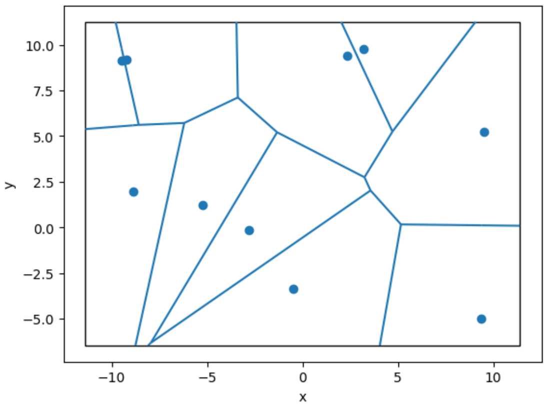
Przykłady działania algorytmu



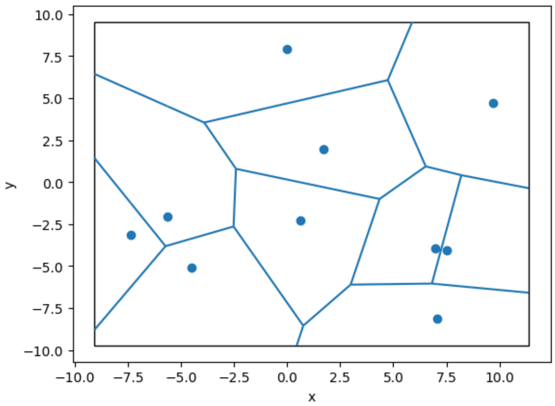
Wykres 1 – przykład działania algorytmu dla 10 punktów



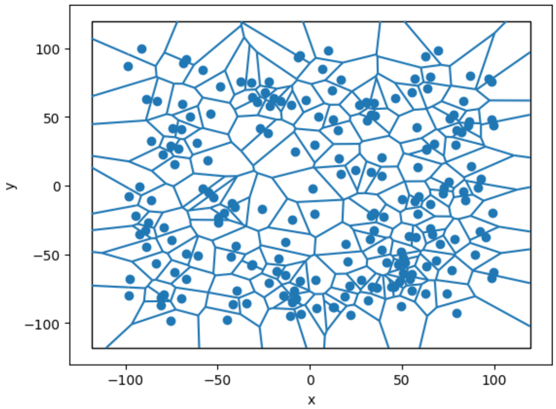
Wykres 2 – przykład działania algorytmu dla 10 punktów



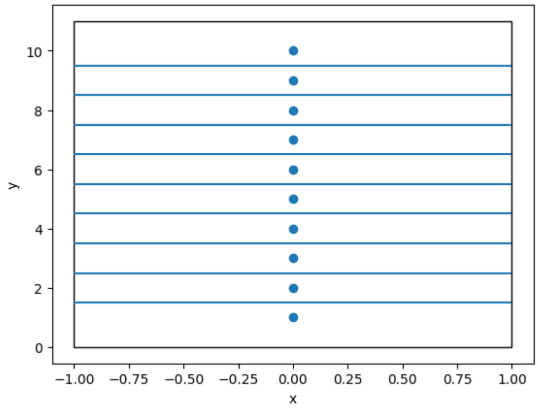
Wykres 3 – przykład działania algorytmu dla 10 punktów



Wykres 4 – przykład działania algorytmu dla 10 punktów



Wykres 5 – przykład działania algorytmu dla 200 punktów



Wykres 6 – przykład działania algorytmu dla 10 punktów ustawionych w pionowej linii

4.4 Algorytm wyznaczania konstrukcji Voronoi przez triangulację Delaunay'a

Idea

Drugim wybranym przez nas algorytmem jest algorytm wyznaczający diagram Voronoi z triangulacji Delaunay'a, to znaczy takiej triangulacji, że wewnątrz koła opisanego na dowolnym trójkącie triangulacji nie znajduje się żaden pozostały punkt zbioru wejściowego. Cechą charakteryzującą tę konstrukcję jest jej dualność. U podstaw programu leży prosta obserwacja, że wierzchołki diagramu Voronoi to środki okręgów opisanych na trójkątach triangulacji Delaunay'a, tak więc wyznaczenie jednej z konstrukcji pozwala na określenie drugiej. Do wyznaczenia triangulacji posłużyliśmy się inkrementacyjnym algorytmem Bowyera-Watsona, którego ideą jest inkrementacja po wszystkich punktach zbioru wejściowego i aktualizowanie wyznaczonej triangulacji o nowododany punkt.

Przyjęta dokładność

Algorytm triangulacji narażony jest na liczne błędy numeryczne wynikające z zaawansowanych obliczeń na współrzędnych punktów, co mogłoby odbić się na poprawności analizy położenia punktu względem prostej oraz klasyfikowania punktu jako należącego do koła opisanego na trójkącie, bądź też nie. W związku z powyższym przyjęto niepewność pomiarową równą $EPS = 10^{-8}$.

Wyznaczanie triangulacji Delaunay'a

Triangulację Delaunay'a wyznaczamy korzystając z algorytmu Bowyera-Watsona przyjmując za triangulację początkową dwa takie trójkąty prostokątne, że mają wspólną krawędź przeciwprostokątną i tym samym tworzą prostokąt zawierający wszystkie punkty zbioru wejściowego. Prostokąt ten jest znacznie większy od otoczki wypukłej wejściowej chmury punktów. Po stworzeniu początkowej triangulacji, przygotowaniu danych i zainicjowaniu obiektu klasy triangulacji *Triangulation* wykonywana jest główna pętla programu. W każdej iteracji do triangulacji dodawany jest nowy punkt spośród punktów zbioru wejściowych, a sama triangulacja jest aktualizowana. Proces aktualizacji rozpoczyna się znalezieniem trójkąta aktualnej triangulacji, zawierającego nowododany punkt. W naszym programie zadanie to realizuje funkcja *find_containing()*, która rozpoczynając poszukiwania od trójkąta centralnego triangulacji (atrybut klasy *Triangulation*) rozpatruje położenie punktu względem krawędzi trójkąta, w każdym kroku zbliżając się do odpowiedniego trójkąta. Następnie inicjalizowane są odpowiednie tablice: tablica trójkątów, które należy zaktualizować, tj. należących do tzw. wnęki (*tri_to_remove*), tablica odwiedzonych trójkątów (*tri_visted*), tablica reprezentująca stos. Z użyciem tychże tablic poprzez sąsiedztwo topologiczne wyznaczamy trójkąty wnęki; warunkiem należenia do wnęki dla trójkąta jest posiadanie wewnątrz okręgu opisanego na jego wierzchołkach dodawanego punktu. Gdy aktualizowany obszar jest już wyznaczony, dla danej iteracji ostatnim krokiem (realizowanym w metodzie *adjust_triangulation()*) jest usunięcie wszystkich krawędzi obszaru nie należących do brzegu wnęki (tj. takich, które nie należą do obwodu wnęki – u nas wszystkich tych nie będących w *outer_edges*) oraz połączenie krawędzią dodawanego punktu, z każdym z wierzchołków leżących na obwodzie aktualizowanego obszaru. Takie połączenie jest oczywiście możliwe, ponieważ na mocy twierdzenia dodawany punkt jest widoczny z wszystkich punktów brzegu. Gdy wykonane zostaną wszystkie iteracje pętli z końcowej triangulacji usuwane są wszystkie te trójkąty, których minimum jednym z wierzchołków jest wierzchołek początkowej triangulacji.

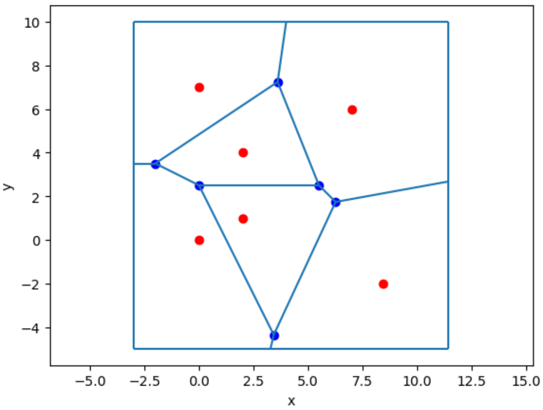
Otrzymanie konstrukcji Voronoi korzystając z triangulacji Delaunay'a

Kluczową obserwacją jest zauważenie, że wszystkie punkty stanowiące wierzchołki konstrukcji Delaunay'a są środkami okręgów opisanych na trójkątach wyznaczonej triangulacji. Aby uzyskać konstrukcję Voronoi wykonywana jest funkcja *voronoi()*, wewnątrz której w pętli analizowane są krawędzie każdego trójkąta triangulacji (służy temu funkcja *add_edgeV()*). W zależności od tego czy rozpatrywany trójkąt dzieli krawędź z innym trójkątem triangulacji wywoływana jest odpowiednia funkcja. Gdy krawędź należy do więcej niż jednego trójkąta triangulacji, krawędzią diagramu Voronoi jest odcinek łączący środki okręgów opisanych na sąsiadujących trójkątach. W przeciwnym przypadku krawędzią diagramu jest półprosta o początku w środku okręgu opisanym na rozpatrywanym trójkącie. Na potrzeby zwizualizowania wyników oraz w celu ułatwionego przechowywania krawędzi diagramu, konstrukcja ograniczana jest przez prostokąt o ustalonych wierzchołkach, zawierający wszystkie punkty zbioru wejściowego.

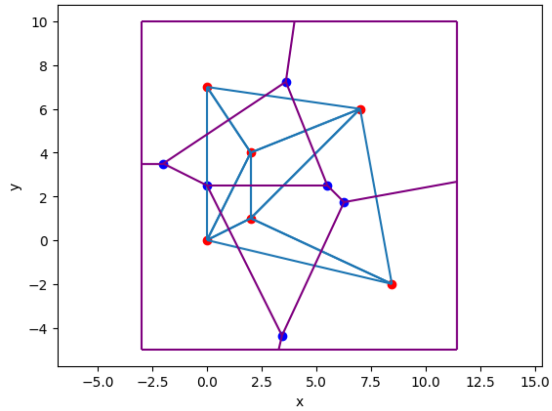
Przypadki zdegenerowane

Łatwo zauważyć, że istnieją przypadki takiej konfiguracji punktów dla których triangulacja nie istnieje. Konfiguracja ta, to ułożenie współliniowe wszystkich punktów zbioru wejściowego (uwzględniając przyjętą niepewność pomiarową). Przypadek ten również został przez nas wzięty pod uwagę. Gdy liczba trójkątów wyznaczonej triangulacji Delauna'a jest równa 0 wywoływana jest funkcja *no_triangles()*. Funkcja ta sortuje punkty po współrzędnej y, a w przypadku równości po współrzędnej x, przy pomocy funkcji wbudowanej *sorted()*, aby następnie dodawać do krawędzi konstrukcji Voronoi symetralne odcinków pomiędzy każdymi dwoma sąsiadującymi w posortowanej liście punktami.

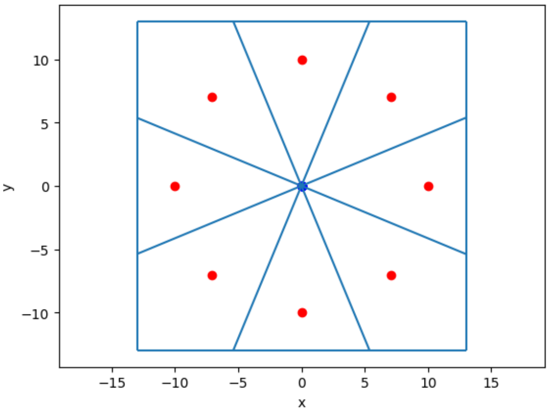
Przykłady działania algorytmu



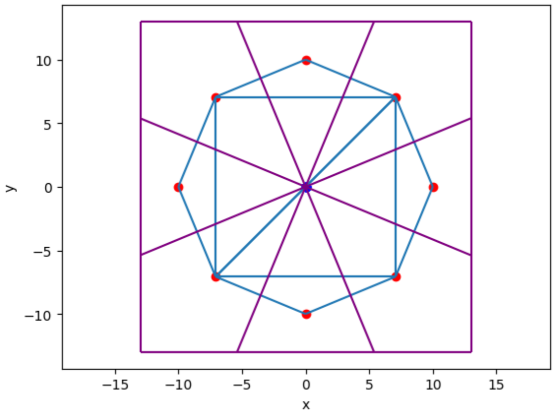
Wykres 7 – wyznaczona dla 6 punktów przez algorytm konstrukcja Voronoi



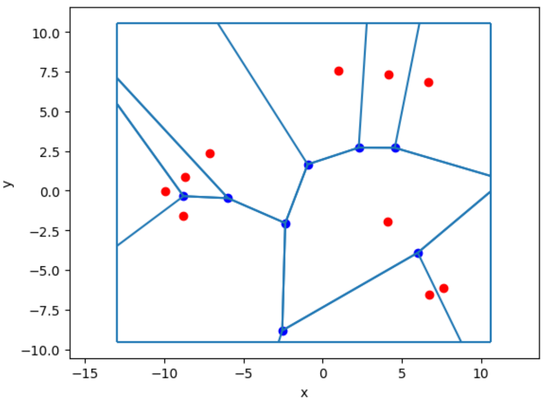
Wykres 8 – wyznaczona dla 6 punktów przez algorytm konstrukcja Voronoi wraz z triangulacją Delaunay'a



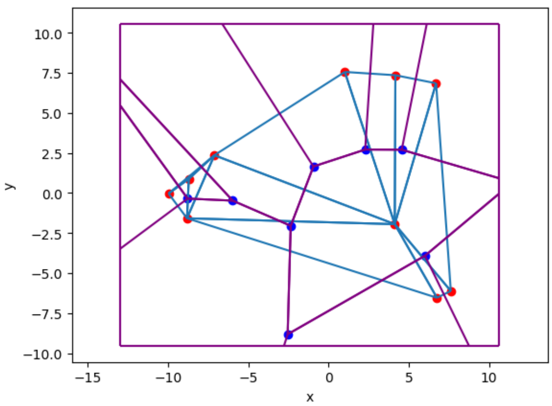
Wykres 9 – wyznaczona dla 8 punktów zbliżonych do współokręgowych przez algorytm konstrukcja Voronoi



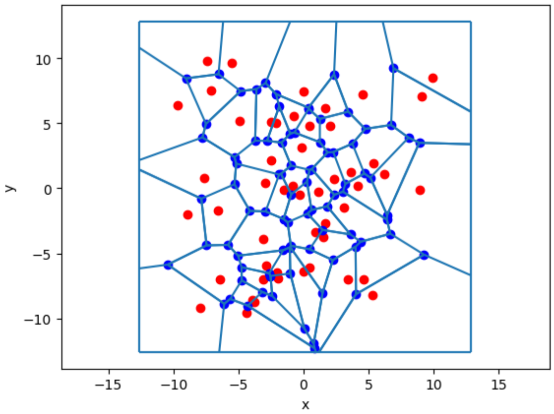
Wykres 10 – wyznaczona dla 8 punktów zbliżonych do współokręgowych przez algorytm konstrukcja Voronoi wraz z triangulacją Delaunay'a



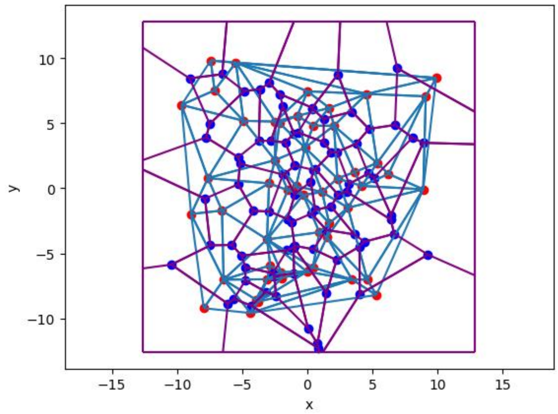
Wykres 11 – wyznaczona dla 10 punktów przez algorytm konstrukcja Voronoi



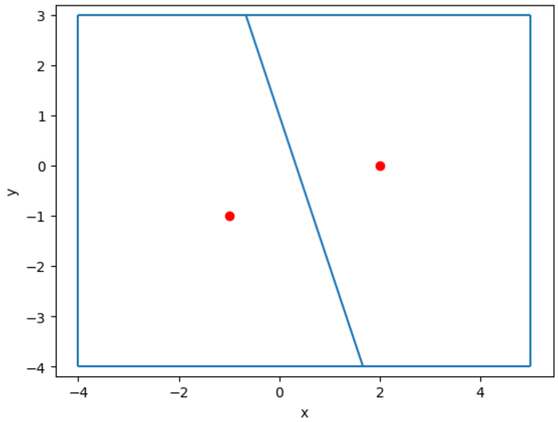
Wykres 12 – wyznaczona dla 10 punktów przez algorytm konstrukcja Voronoi wraz z triangulacją Delaunay'a



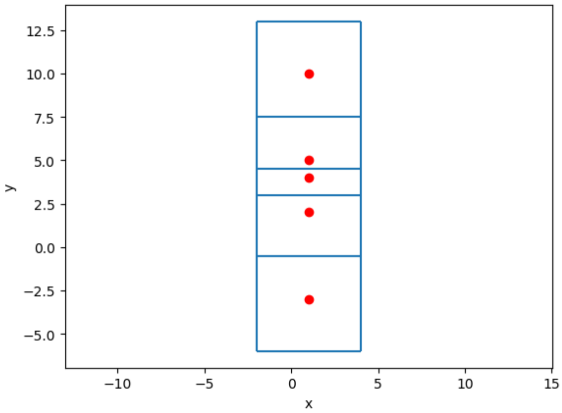
Wykres 13 – wyznaczona dla 50 punktów przez algorytm konstrukcja Voronoi



Wykres 14 – wyznaczona dla 50 punktów przez algorytm konstrukcja Voronoi wraz z triangulacją Delaunay'a



Wykres 13 – wyznaczona dla 2 punktów przez algorytm konstrukcja Voronoi (brak triangulacji Delaunay'a)



Wykres 14 – wyznaczona dla 5 punktów współliniowych przez algorytm konstrukcja Voronoi (brak triangulacji Delaunay'a)

5. Porównanie czasów działania obu algorytmów

5.1 Przygotowanie danych

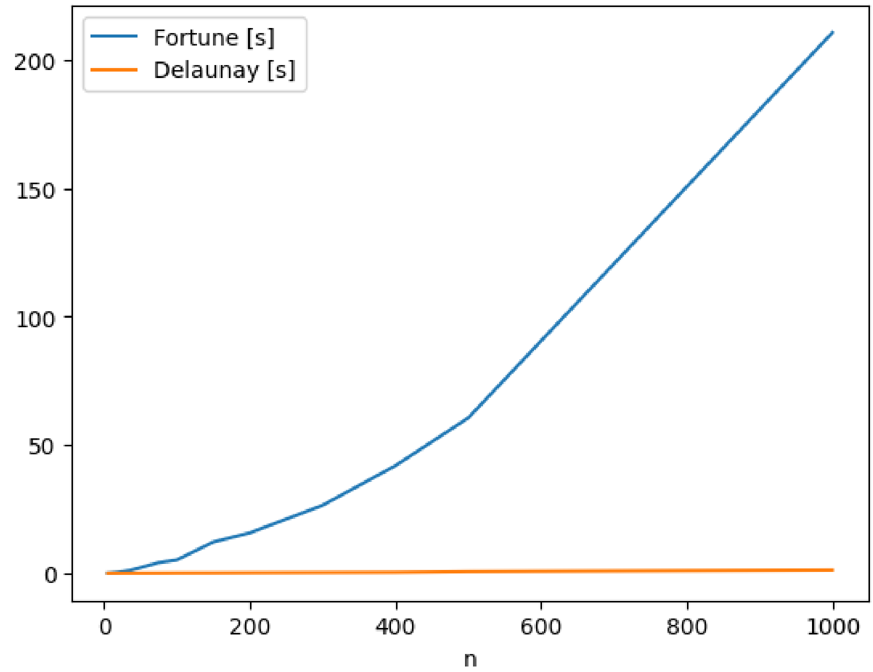
Aby porównać czas działania obu algorytmów, wybraliśmy funkcje, które zwracają diagramy Voronoi, bez wizualizacji (wizualizacja zakłamałaby czas działania algorytmu). W przypadku implementacji algorytmu Fortune jest to funkcja *get_voronoi(points)*, a dla algorytmu wykorzystującego triangulację Delaunay'a są to: *delaynay(points)* (aby uzyskać triangulacje) i *voronoi(triangulation, points)* (do uzyskania diagramu). Następnie za pomocą funkcji zaimplementowanej na pierwszym laboratorium *generate_uniform_points* wygenerowaliśmy równomiernie rozmieszczone punkty w liczbach: 5, 10, 20,35, 50, 75, 100, 150, 200, 300, 400, 500, 1000. Diagramy generowaliśmy używając jednej maszyny, zapisując czas wykonania algorytmu.

5.2 Wyniki

W poniższej tabeli (Tabela 1) znajduje się porównanie czasowe działania algorytmów. Kolumna o nazwie *n* pokazuje liczbę punktów dla danego testu. Obok (na wykresie 15) znajduje się wizualizacja zmierzonych czasów.

n	Fortune [s]	Delaunay [s]
5	0.0369065	0.002403021
10	0.1323271	0.005480766
20	0.4149733	0.011526108
35	1.0874844	0.026529312
50	2.1889706	0.035391092
75	4.1368194	0.052121162
100	5.2197244	0.086582422
150	12.1797957	0.118786812
200	15.7140830	0.17547369
300	26.4622085	0.270816565
400	41.9027805	0.365414143
500	60.4842865	0.645820618
1000	210.5087557	1.244166613

Tabela 1 – porównanie czasów działania algorytmów Fortune i Delanuay



Wykres 15 – porównanie czasów działania algorytmów Fortune i Delanuay

Z wyników badania można zaobserwować wielką dysproporcję w szybkościach obu algorytmów. Wynika ona prawdopodobnie z dobranej podwójnie łączonej listy jako struktury stanu w implementacji algorytmu Fortune.

6. Wnioski

Na podstawie wizualizacji przedstawiających diagramy Voronoi, stwierdzamy, że oba algorytmy zostały zaimplementowane poprawnie. Nasuwającym się wnioskiem wynikającym z porównania szybkości algorytmów jest stwierdzenie, że rozwiązanie produkcyjne wykorzystujące algorytm Fortune należałoby zaimplementować z wykorzystaniem struktury drzewa wyszukiwań binarnych jako struktury stanu lub zamiast tego skorzystać z algorytmu opartego na dualności konstrukcji triangulacji Delanuay'a. Wadą naszych implementacji algorytmów jest również struktura danych, w której zwracana jest konstrukcja Voronoi.

7. Źródła

Źródła wykorzystywane przy implementacji algorytmu Fortune'a:

[1] Mark de Berg, Marc van Kreveld, Mark Overmars, Otfried Schwarzkopf, *"Geometria Obliczeniowa - Algorytmy i zastosowanie"* (polskie tłumaczenie *"Computational Geometry - Algorithms and Applications"* - z angielskiego przełożył Mirosław Kowaluk)

[2] Dr inż. Barbara Głut, Wykład przedmiotu *Algorytmy Geometryczne* na Akademii Górniczo-Hutniczej w Krakowie im. Stanisława Staszica w Krakowie (2023/2024)

[3] https://jacquesheunis.com/post/fortunes-algorithm/

[4] https://pvigier.github.io/2018/11/18/fortune-algorithm-details.html

Źródła wykorzystywane przy implementacji algorytmu wyznaczającego konstrukcję Voronoi poprzez triangulację Delaunaya'a:

[5] Monika Wiech, *"Triangulacja Delaunaya w geometrii obliczeniowej"*, Uniwersytet Jagielloński, Kraków, obrona 2020-09-22 (*"Delaunay triangulation in computational geometry"*)

[6] Mark de Berg, Marc van Kreveld, Mark Overmars, Otfried Schwarzkopf, *"Geometria Obliczeniowa - Algorytmy i zastosowanie"* (polskie tłumaczenie *"Computational Geometry - Algorithms and Applications"* - z angielskiego przełożył Mirosław Kowaluk)

[7] Dr inż. Barbara Głut, Wykład przedmiotu *Algorytmy Geometryczne* na Akademii Górniczo-Hutniczej w Krakowie im. Stanisława Staszica w Krakowie (2023/2024)