Wieloboki Voronoi – porównanie metod konstrukcji - dokumentacja.

Piotr Rzadkowski

Olgierd Smyka

Czwartek 11:20, styczeń 2023

Algorytmy geometryczne

# 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*.

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**
* **util.py**
* **voronoi.py**
* **voronoiNotes.ipynb**

# Dokumentacja

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

1. **Oznaczenia**

* - **np** – skrócona nazwa biblioteki **numpy**
* - **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.

1. **Plik voronoi.py – klasa Voronoi**

Główna struktura odpowiedzialna za tworzenie diagramu Voronoi. Udostępnia ona następujące metody:

* \_\_init\_\_(self, points) – konstruktor, którzy przyjmuje punktu, których diagram Voronoi ma dotyczyć
* get\_voronoi(self) – funkcja zwraca krawędzie utworzonego 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 poniżej, jedynie ignoruje wizualizacje)
* get\_voronoi\_visualised(self) – funkcja zwraca krawędzie Voronoi, pudełko, jak i również instancje klasy Visualizer z modułu visualizer, która będzie pomocna do późniejszej wizualizacji. Schemat działania samej funkcji to: 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.

1. **Plik dataStructures.py**
   1. **Node**

Struktura odpowiedzialna za reprezentację węzłów w strukturze stanu. Pole arc i arc\_pair odnoszą się do przechowywanym w danym węźle łuku (jeśli dany węzeł jest liściem) lub sąsiadujących ze sobą łukach 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.

* 1. **Point**

Struktura reprezentującą punkt w przestrzeni .

* 1. **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.

* 1. **Arc**

Struktura reprezentująca łuk. Określa go 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 ustaje że *side\_arc* leży nad *self*, przez to początek lewej krawędzi rozpocznie się w punkcie na paraboli *side\_arc* znajdującym się na współrzędnej x równej wpółrzędnej 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 parabola higher która znajduje się wyżej niż parabola, na której metoda została wywołana, wykorzystując metodę *lookupForIntersectionBetween*
* intersect(self, arc) – zwraca przecięcie dwóch paraboli.
  1. **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 informacja czy zdarzenie jest fałszywym alarmem.

* 1. **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.

1. **Plik priorityQueue.py – klasa PriorityQueue**

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

* \_\_init\_\_(self, items=[])-konstruktortworzący 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 następny 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.)

1. **Plik TInterface.py – klasa T**

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

1. **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, który pokrywa punkt *p.*
* replace(self, arc\_node: Arc, new\_arc: Arc) – metoda ta jest wywoływana w przypadku zdarzenia punktowego. Zadaniem jej jest podział węzła który poprzednio reprezentował łuk nowym łukiem, który odpowiada aktualnemu zdarzeniu punktowemu i odpowiednio podzielić te łuki węzłami odpowiadającym 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 nie ma, zwraca *None*.
* isCirleEvent(self, arc\_node: Node) – dla wezła *arc\_node* zwraca punkt zdarzenia okręgowego i środek tego okręgu, lub *None* jeśli takiego punktu nie będzie.
* 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 sprawdza, czy pojawi się zdarzenie okręgowe dla tych sąsiadów. Zwraca te zdarzenia.
* leftNbour(self, node) – zwraca lewy łuk licząc od *node (*ponieważ 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)

1. **Plik util.py**
   1. funkcja distance(p1: Point, p2: Point)– zwraca dystans między dwoma punktami w metryce Euklidesowej
   2. funkcja mat\_det(a, b) – zwracająca wyznacznik macierzy 2x2 dla punktów *a* i *b*
   3. funkcja getIntersect(start1: Point, direction1: Point, start2: Point, direction2: Point) – zwracająca przecięcie dwóch półprostych o określonych punktach przyłożenia i kierunku. Zwraca None jeśli takie przecięcie nie istnieje.
   4. 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. **Plik voronoiNotes.ipynb**

Plik, w którym pokazane są przykłady działania algorytmu.

# Poradnik do wykorzystania

1. Generowanie diagramu Voronoi bez wizualizacji

Tworzymy zbiór punktów:

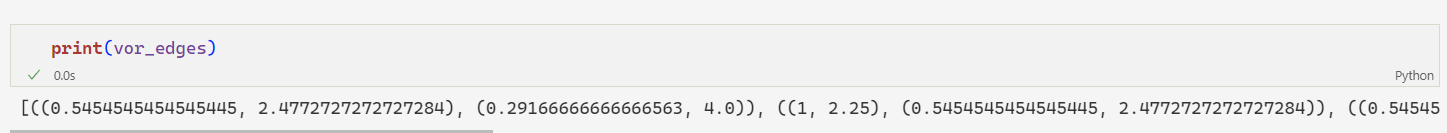
points = [(0, 0), (1, 1), (2, 3), (-1, 2.5)]

Tworzymy instancje klasy *Voronoi* z listą punktów jako parametr, a następnie wywołujemy metodę *get\_voronoi:*

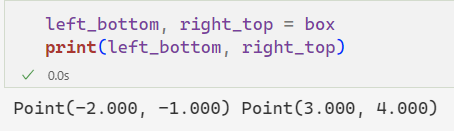
vor = **Voronoi**(points)

vor\_edges, box = vor.**get\_voronoi**()

W zmiennej *vor\_edges* zapisana zostanie lista krawędzi wygenerowanych przez klasę:



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



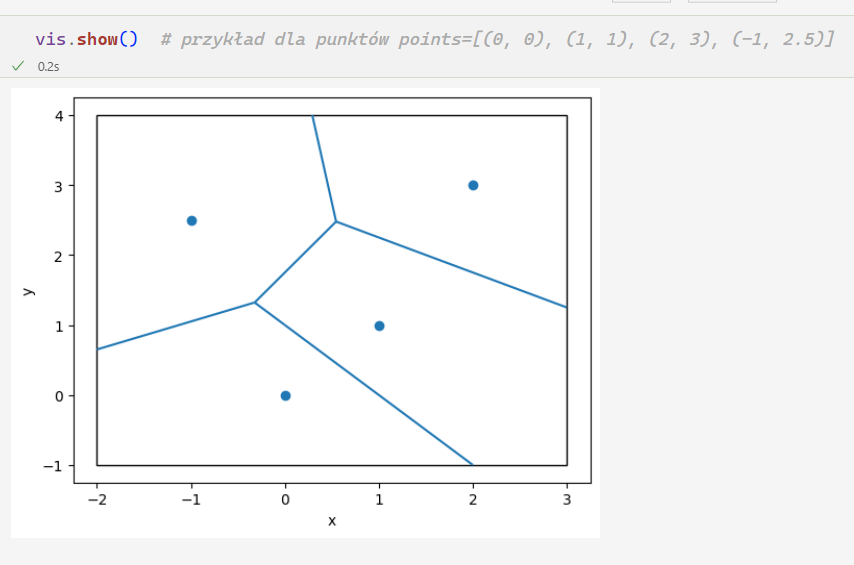
1. Generowanie diagramu Voronoi z wizualizacją

Natomiast w metodzie *get\_voronoi\_visualised*, dostajemy również 3 obiekt, czyli Visualizer.

vor = **Voronoi**(points)

vor\_edges, box, vis = vor.**get\_voronoi\_visualised**()

Tak jak jest opisane 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.



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

# 

# Sprawozdanie

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 Ki od punktu Pi w danej komórce była mniejsza niż odległość od każdego innego punktu Pj (i j).

1. Plan i sposób wykonania ćwiczenia
   1. 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, z przechowującą dwa rodzaje zdarzeń – punktowe i okręgowe, i strukturę stanu przechowującą informację o łukach tworzących *linię brzegową*. Podczas przesuwania miotły łuki paraboli mających ognisko w punktach, dla których chcemy znaleźć diagram Voronoi, rozszerzają się i przecinają ze sobą nawzajem. Okazuje się, że gdyby na przecięciach tych paraboli postawić 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*, która tworzy z 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 przechowywać będziemy informacje o łukach i przecięć 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 drzewa nie udało się napisać. 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 krawędzie, które zostały i dodanie ich do diagramu Voronoi, odpowiednio obcinając nie były nieskończone – 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 *ll*, *lp*) i wstawiamy między nie łuk *ln* o ognisku w p. Tak utworzoną sekwencję <*ll*, *lp*, *ln*> 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 *ll* i *lp.* Sprawdzenie to polega na pobraniu sąsiadujących do danego łuku krawędzi i sprawdzić czy się one przetną. Przecięcie tych krawędzi będzie oznaczało, że dwa sąsiadujące łuki wyprą w linii brzegowej sprawdzany łuk, czyli pojawienie się zdarzenia okręgowego.

**Przetworzenie zdarzenia okręgowego**

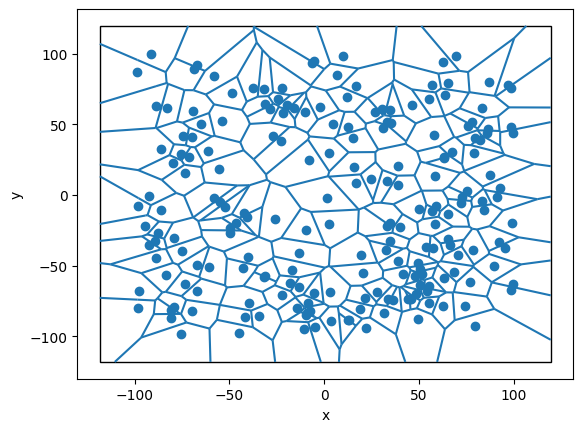
W zdarzeniu okręgowym, które dotyczy łuku *l,* pobieramy krawędzie sąsiadujące z łukiem *l (*oznaczmy je przez *el* i *er*). Zdarzenie to oznacza, że te krawędzie będzie można zakończyć i dodać do finalnego diagramu. Następnie usuwamy z *T* łuk *l* i krawędzie *el*, *er,* a zamiast nich wstawiamy węzeł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

**Zakończenie algorytmu**

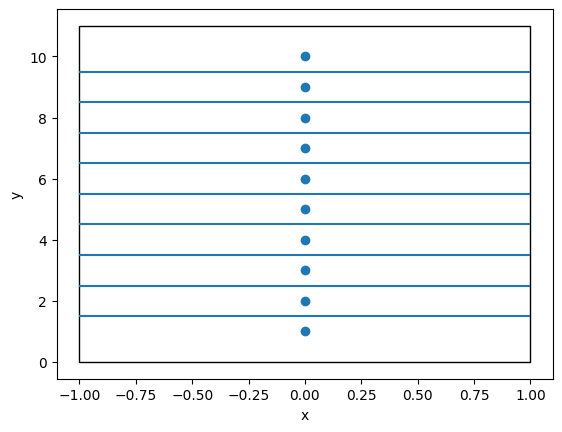
Kiedy w kolejce skończą się zdarzenia, powinniśmy jeszcze zdjąć z *T* pozostałe krawędzie i dodać je do diagramu. Ponieważ te krawędzie są nieskończone musimy je ograniczyć, a robimy to przez pudełko, które jest prostokątem zawierającym wszystkie punktu zwiększonym o dany margines.

**Przykłady działania algorytmu**

|  |  |
| --- | --- |
|  |  |
| Rysunek 2 – przykład działania algorytmu dla 10 punktów | Rysunek 3 – przykład działania algorytmu dla 10 punktów |
|  |  |
| Rysunek 4 – przykład działania algorytmu dla 10 punktów | Rysunek 5 – przykład działania algorytmu dla 10 punktów |



Rysunek 6 - przykład działania algorytmu dla 200 punktów



Rysunek 7 - przykład działania algorytmu dla 10 punktów ustawionych w pionowej linii

# Źródła

Źródła wykorzystywane przy implementacji algorytmu Fortune’a:

* Mark de Berg - "Computational Geometry - Algorithms and Applications"
* <https://jacquesheunis.com/post/fortunes-algorithm/>
* <https://pvigier.github.io/2018/11/18/fortune-algorithm-details.html>