Diagramy Voronoi

Jan Dąbrowski, Jakub Karoń

Algorytmy Geometryczne 2023/2024

Rozdział 1

Dane techniczne

1.1 Architektura komputera

Sprzęt 1:

System operacyjny: Windows 11 Pro ver. 22H2
Procesor: 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz
RAM: 16GB

Sprzęt 2:

System operacyjny: macOS Sonoma 14.0
Procesor: Apple M1
RAM: 16GB

1.2 Dane środowiskowe

Języki: Python
Interpreter: Python 3.11.7 64-bit

Rozdział 2

Wprowadzenie

2.1 Temat projektu

Celem projektu jest implementacja algorytmów znajdujących diagram Woronoja dla danych punktów na dwa różne sposoby:

1. Algorytmem Bowyer'a-Watson'a (algorytm inkrementacyjny)
2. Algorytmem Fortune'a

2.2 Struktura projektu

Projekt zawiera następujące pliki:

- `README.md` - wstęp do projektu, instrukcja obsługi plików
- `main.ipynb` - prosty plik stworzony do obsługi przez użytkownika przedstawiający wizualizacje algorytmów
- `delaunay` - pakiet z klasami i funkcjami pomocniczymi potrzebnymi do algorytmu Bowyer'a-Watson'a
  - `__init__.py` - inicjacja pakietu
  - `point.py` - klasa odpowiadająca za punkty wraz z pomocniczymi funkcjami
  - `edge.py` - klasa odpowiadająca za krawędzie wraz z pomocniczymi funkcjami
  - `triangle.py` - klasa odpowiadająca za trójkąty wraz z pomocniczymi funkcjami
  - `neighbours.py` - klasa odpowiadająca za reprezentację sąsiadów
  - `delaunay_triangulation.py` - klasa odpowiadająca za triangulację Delaunay'a wraz z funkcją pomocniczą
  - `voronoi_diagram.py` - klasa odpowiadająca za stworzenie diagramu Voronoi
- `fortunes` - pakiet z klasami i funkcjami pomocniczymi potrzebnymi do algorytmu Fortune'a
- `visualizer` - pakiet zawierający wszelkie potrzebne narzędzia wizualizacji geometrycznej obiektów

Rozdział 3

Zawartość plików

3.1 Pakiet `delaunay`

3.1.1 `point.py`

Punkty w projekcie reprezentowane są klasą `Point`. Każdy z nich ma atrybuty `x` oraz `y`, odpowiadające im współrzędnym. Punkt jest obiektem niezmiennym. Do klasy zostały załączone metody:

- `is_in_circumcircle_of(self,triangle)` - zwraca `True` lub `False`, w zależności od tego, czy punkt znajduje się wewnątrz okręgu opisanego przez trójkąt `triangle`
- inicjująca, hashująca, porównująca oraz zwracająca punkt jako `String`

W pliku znajdują się również: funkcja `orientation(p1,p2,p3)`, która sprawdza, czy dane 3 punkty są współliniowe, tworzą lewy lub prawy skręt; oraz `distSq(p1,p2)`, która oblicza odległość między punktami.

3.1.2 `triangle.py`

Trójkąty w projekcie reprezentowane są klasą `Triangle`. Każdy trójkąt, oprócz atrybutów `a`, `b`, `c`, odpowiadających jego wierzchołkom, ma również atrybut `edges`, który przechowuje jego boki. Punkty w trójkącie przechowywane są zgodnie z kierunkiem przeciwnym do ruchu wskazówek zegara. Trójkąt jest również obiektem niezmiennym. Do klasy zostały załączone metody:

- inicjująca, haszująca, porównująca oraz zwracająca trójkąt jako `String`
- `circumcircle_contains(self,point)` - zwraca `True`, gdy trójkąt zawiera punkty `point` lub `False` w p.p.
- `point_in_triangle(self, p)` - zwraca `True`, gdy punkt `p` znajduje się w trójkącie lub `False` w p.p.
- `contains(self,p)` - zwraca `True` lub `False`, w zależności od tego, czy punkt `p` jest jednym z wierzchołków trójkąta
- `find_circumcenter(self)` - zwraca środek okręgu opisanego na trójkącie
- funkcje konwertujące trójkąt na obiekt klasy `Polygon` i listę wierzchołków

Dodatkowe funkcje zawarte w pliku wykorzystywane są jedynie jako funkcje pomocnicze metod w klasie.

3.1.3 edge.py

Krawędzie w projekcie reprezentowane są klasą `Edge`. Każda krawędź definiowana jest jej dwoma końcami, które są atrybutami klasy. Krawędź jest również obiektem niezmiennym. Do klasy zostały załączone następujące metody:

- inicjująca, haszująca, porównująca oraz zwracająca krawędź jako `String`. Aby krawędzie były jednoznacznie wyszukiwane w zbiorach i słownikach, funkcja haszująca zwraca haszowanie krawędzi z punktami posortowanymi leksykograficznie. Krawędzie są równe, jeśli mają takie same wierzchołki, ale ich kolejność nie ma znaczenia
- `get_centre(self)` - zwraca punkt w środku krawędzi
- metody pomocnicze do obliczania okręgów opisanych

Została również zaimplementowana funkcja `find_intersection(a1,b1,c1,a2,b2,c2)`, która znajduje punkt przecięcia dwóch prostych zadanych w postaci $Ax + By + C = 0$ za pomocą wzorów Cramera.

3.1.4 neighbours.py

Instancja klasy `Neighbours` przechowuje jedynie atrybut `edges`. Jest to słownik, z elementami następującej postaci:

`edge` : `[trójkąt1,trójkąt2]`

Zatem, kluczami elementów słownika są krawędzie, a wartościami listy trójkątów. Powyższy przykład oznacza, że `trójkąt1` oraz `trójkąt2` mają wspólną krawędź `edge`. Przyjmiemy nazewnictwo, że `trójkąt1` *sąsiaduje* z `trójkąt2` *poprzez* `edge` lub `[trójkąt1, trójkąt2]` jest *listą sąsiedztwa dla krawędzi* `edge`. Może się zdarzyć, że dany trójkąt nie ma sąsiada przez daną krawędź - wtedy lista trójkątów w słowniku jest jednoelementowa. Klasa ma dołączone następujące metody:

- `put(self,triangle)` - dodaje wszystkie boki trójkąta `triangle` do słownika (dodaje nowy element słownika jeśli boku w nim nie ma, lub dopisuje trójkąt do listy sąsiedztwa danej krawędzi)
- `find_neighbour(self, edge, triangle)` - zwraca sąsiada trójkąta `triangle` poprzez `edge`
- `remove_neighbour(self, edge, triangle)` - usuwa `triangle` z listy sąsiedztwa krawędzi `edge`
- inicjująca oraz zwracająca `edges` jako `String`

3.1.5 delaunay_triangulation.py

Plik zawiera klasę `DelaunayTriangulation`. Klasa ta przechowuje wszelkie potrzebne metody do obliczenia triangulacji Delaunay'a dla danego zbioru punktów. Zawiera następujące atrybuty:

- `points` - punkty, na których ma być zrobiona triangulacja
- `super_triangle` - początkowy trójkąt, wewnątrz którego znajdują się wszystkie punkty z `points`
- `triangulation` - zbiór trójkątów - obecny stan triangulacji
- `neighbours` - lista sąsiadów, reprezentowana klasą `Neighbours`
- `inittriangle` - trójkąt startowy (którego użycie wytłumaczone poniżej)

Do klasy załączone są metody:

- inicjująca instancję

- `find_triangle(self,p)` - przeszukuje trójkąty z `triangulation` w poszukiwaniu tego, który zawiera punkt `p`, zaczynając od `inittriangle`. Dla każdego boku (w_i, w_{i+1}) trójkąta sprawdza, jaki skręt tworzy trójka w_i, w_{i+1}, p. Jeśli jest to prawy skręt, oznacza to, że `p` leży w półpłaszczyźnie niezawierającej tego trójkąta, a więc przejście do trójkąta sąsiadującego z obecnie rozważanym poprzez bok (w_i, w_{i+1}) zbliży nas do `p`. Taka implementacja znajdowania trójkątów działa dlatego, że wierzchołki w trójkącie są w kolejności przeciwnej do ruchu wskazówek zegara.
- `find_neighbourhood(self, p, curr)` - zwraca *sąsiedztwo* oraz *otoczkę sąsiedztwa* danego punktu. Sąsiedztwem `p` nazywamy wszystkie trójkąty, których okrąg opisany zawiera `p`, a otoczką sąsiedztwa krawędzie ograniczające to sąsiedztwo. Innymi słowy, otoczką sąsiedztwa nazywamy taki podzbiór wszystkich krawędzi z sąsiedztwa, że żaden inny trójkąt z sąsiedztwa nie ma sąsiada w sąsiedztwie poprzez daną krawędź. Działanie funkcji jest rekurencyjne - dla danego trójkąta sprawdzamy, czy jego sąsiedzi również należą do sąsiedztwa
- `delete_neighbourhood(self,neighbourhood)` - usuwa trójkąty należące do sąsiedztwa z `triangulation` oraz z list sąsiedztwa w `neighbours`.
- `rebuild_neighbourhood(self,p,hull)` - odbudowuje sąsiedztwo, dodając krawędzie między punktami otoczki a `p` oraz odpowiadające im trójkąty
- zwracająca listę odcinków w triangulacji (wykorzystywana przy wizualizacji)
- `run(self)` - korzysta z pozostałych metod aby zwrócić pełną triangulację

W pliku znajduje się również pomocnicza funkcja `get_init_triangles(points)`, która wykorzystywana jest jedynie przy inicjacji instancji klasy. Zwraca ona trójkąt, wewnątrz którego znajdują się wszystkie punkty z `points`.

3.1.6 voronoi_diagram.py

W pliku znajduje się klasa `VoronoiDiagram` z metodami obliczającymi diagram Woronoja. Jej atrybutami są: `points` (punkty wejściowe), `neighbours` oraz `triangles` (opisujące triangulację). Klasa zawiera następujące metody:

- `create_point_to_edges_mapping(self)` - zwraca słownik z elementami postaci:

  ```
  p : [krawędź1, krawędź2, ...],
  ```

  gdzie każda krawędź ma jakiś koniec w punkcie `p`. Nazywamy to *mapowaniem krawędzi dla punktu* `p`
- `create_polygon_for_vertex(self, vertex, point_to_edges)` - zwraca komórkę diagramu Voronoi, której centrum jest punkt `vertex`. Dzięki mapowaniu krawędzi dla punktu znamy trójkąty, które mają wierzchołek w tym punkcie. Wybieramy losowo jeden z nich, a następnie przechodzimy "dookoła" tego punktu po trójkątach (funkcją `find_neighbour`), otrzymując wielokąt
- `create_diagram(self)` - zwraca pełny diagram Woronoja jako listę obiektów `Polygon`

3.2 Pakiet `fortunes`

3.2.1 structures.py

Plik `structures.py` zawiera implementacje klas będące niezbędne do przejrzystego zaimplementowania algorytmu Fortune'a.

Klasa `Point`

Klasa ta jest reprezentacją punktu na płaszczyźnie dwuwymiarowej o współrzędnych (x, y). W swoim konstruktorze przyjmuje kolejno współrzędne x oraz y oraz inicjalizuje punkt z tymi koordynatami.

Klasa `Event`

Klasa ta reprezentuje zdarzenie w algorytmie Fortune'a oraz wiąże dane zdarzenie z obiektami klas, których to zdarzenie dotyczy. Klasa ta zawiera następujące atrybuty:

- `x` - współrzędna x danego zdarzenia
- `p` - powiązany ze zdarzeniem punkt, który jest obiektem klasy `Point`
- `a` - powiązany ze zdarzeniem łuk, który jest obiektem klasy `Arc`
- `valid` - flaga wskazująca na to, czy zdarzenie jest nadal ważne

Konstruktor tej klasy przyjmuje parametry `x`, `p` oraz `a`, a następnie inicjalizuje zdarzenie z tymi parametrami.

Klasa `Arc`

Klasa `Arc` służy reprezentacji paraboli w algorytmie Fortune'a. Klasa przechowuje następujące atrybuty:

- `p` - punkt centralny łuku będący obiektem klasy `Point`
- `pprev` - poprzedni łuk w strukturze drzewa
- `pnext` - następny łuk w strukturze drzewa
- `e` - powiązane zdarzenie będące obiektem klasy `Event`
- `s0, s1` - obiekty klasy `Segment` powiązane z tym łukiem

W konstruktorze klasa przyjmuje punkt centralny `p` oraz opcjonalnie poprzedni oraz następny łuk.

Klasa `Segment`

Klasa ta reprezentuje odcinek będący częścią granicy komórki Woronoja oraz posiada następujące atrybuty:

- `start, end` - początkowy oraz końcowy punkt danego odcinka opisane za pomocą klasy `Point`
- `done` - flaga wskazująca, czy odcinek został już zakończony

Ponadto klasa posiada metodę `finish(self, p)`, która pozwala na zakończenie odcinka w punkcie `p`.

Klasa `PriorityQueue`

Klasa ta implementuje strukturę kolejki priorytetowej wykorzystanej do zarządzania zdarzeniami w algorytmie.

Inicjalizacja i atrybuty

Podczas tworzenia instancji klasy dodajemy również następujące atrybuty:

- `pq` - lista, która będzie służyć jako kopiec do przechowywania elementów
- `entry_finder` - słownik do śledzenia istnienia elementów w kolejce, używany do szybkiego sprawdzania, czy element już istnieje
- `counter` - licznik, który zapewnia unikalność każdego wpisu w kopcu oraz zapobiega konfliktom priorytetów

Metody

Klasa implementuje również metody, które są kluczowe do sprawnego zarządzania tą strukturą:

- `push(self, item)` - dodaje nowy element do kolejki. Wpisy są krotkami `(item.x, count, item)`, gdzie kolejko `item.x` jest priorytetem (współrzędną x), `count` to unikalna wartość oraz `item` to właściwy element
- `remove_entry(self, item)` - usuwanie elementu z kolejki. Z uwagi na nieefektywność usuwania elementu z kopca, zostaje on oznaczony jako `Removed` i jest pomijane podczas operacji `pop`
- `pop(self)` - usuwanie oraz zwracanie elementu o najwyższym priorytecie, czyli najmniejszym `item.x`
- `top(self)` - zwracanie elementu o najwyższym priorytecie bez usuwania go
- `empty` - sprawdzanie, czy kolejka jest pusta

Do implementacji tej klasy posłużono się modułem `heapq`, który efektywnie zarządza kopcem `min-heap` oraz pozwala na dodawanie i usuwanie elementów o najwyższym priorytecie w czasie $O(logn)$. W kontekście algorytmu Fortune'a, struktura ta pozwala na przetwarzanie zdarzeń w takiej kolejności, w jakiej pojawiają się na osi x.

3.2.2 fortune_algorithm.py

Klasa FortuneAlgorithm

Klasa `FortuneAlgorithm` to kluczowa część algorytmu Fortune'a do generowania diagramów Woronoja.

Inicjalizacja i atrybuty

Konstruktor `__init__(self, points, bound)` inicjalizuje instancję klasy `FortuneAlgorithm` z danymi wejściowymi oraz przygotowuje struktury, które będą niezbędne do wykonania algorytmu. Parametr `points` to lista punktów o klasie `Point`, dla których będzie tworzony diagram Woronoja.

Atrybuty

- `output` - lista, która będzie przechowywać odcinki wynikowego diagramu Woronoja
- `init_points` - zapisuje listę punktów wejściowych
- `arc` - w dalszym etapie algorytmu atrybut ten będzie wykorzystywany do przechowywania binarnego drzewa łuków parabol, co pozwoli na efektywne zarządzania łukami podczas przemieszczania się miotły

- `points` - kolejka priorytetowa do przechowywania zdarzeń związanych z punktami, czyli momentami, kiedy nowe łuki dodawane są do rozważanych
- `event` - kolejka priorytetowa do przechowywania zdarzeń związanych z kołami, czyli tzw. zdarzeniami kołowymi, które polegają na zanikaniu jednego łuku na rzecz dwóch łuków sąsiadujących
- `bound` - służy do poprawnej wizualizacji powstałego diagramu

Metody

- `process(self)` - metoda ta jest główną pętlą algorytmu, gdzie odbywa się przetwarzanie zdarzeń. W zależności od potrzeby przetworzenia zdarzenia kołowego, bądź punktowego, wywoływane są odpowiednio metody `process_event` lub `process_point`
- `process_point(self)` - przetwarzanie zdarzeń punktowych. Metoda ta pobiera oraz usuwa następne zdarzenie punktowe z kolejki priorytetowej. Dla danego punktu wstawia nową parabolę do struktry linii brzegowej za pomocą funkcji `arc_insert`
- `process_event(self)` - metoda odpowiedzialna za przetwarzanie zdarzeń kołowych. Jeśli zdarzenie tego wymaga, rozpoczynana jest nowa krawędź w punkcie zdarzenia kołowego oraz dodawana do listy wynikowej. Usuwany jest również powiązany ze zdarzeniem łuk.
- `arc_insert(self, p)` - metoda dodaje nowy łuk do drzewa binarnego na podstawie punktu zdarzenia. Jeśli drzewo nie jest puste, sprawdzane są przecięcia nowej paraboli z innymi, już istniejącymi. Jeśli przecięcie zostanie znalezione, tworzone oraz wstawiane są nowe łuki, a także odpowiednie odcinki dodawane są do listy wynikowej.
- `check_circle_event(self, i, x0)` - metoda odpowiedzialna za sprawdzenie, czy dla danego łuku może wystąpić nowe zdarzenie kołowe. Sprawdzane jest czy łuk posiada sąsiadujące łuki oraz czy trzy sąsiadujące punkty tworzą koło. Jeśli warunki są spełnione, do kolejki priorytetowej dodaje się nowe zdarzenie kołowe.
- `circle(a: Point, b: Point, c: Point)` - metoda przyjmuje trzy punkty oraz, na podstawie orientacji oraz sprawdzania współliniowości, zwraca wartość mówiącą, czy punkty te tworzą okrąg
- `intersect(p: Point, i: Arc)` - metoda sprawdza, czy parabola o danym punkcie skupienia przecina się z danym łukiem. Dla sąsiednich łuków danej paraboli wykorzystywana jest funkcja `intersection`, aby sprawdzić punkty przecięcia z parabolą.
- `intersection(self, p0: Point, p1: Point, l)` - metoda oblicza punkt przecięcia dwóch parabol reprezentowanych przez punkty skupienia za pomocą wzoru kwadratowego z uwzględnieniem przypadków brzegowych
- `finish_edges(self)` - metoda kończąca przetwarzanie krawędzi diagramy wynikowego. Dla każdego łuku i jego następnika obliczany jest punkt przecięcia ich parabol i ustawiany jako końcowy dla odpowiedniego odcinka
- `get_output(self)` - metoda zwraca przetworzony diagram Woronoja z uwzględnieniem określonych granic wynikających z funkcji `bound`

Funkcje pomocnicze

- `bound(n: int, seg: tuple[tuple[float, float], tuple[float, float]])` - funckja ograniczająca dany odcinek do obszaru o określonym rozmiarze zdefiniowanym przez wartość 10^n.
- `find_intersection_with_horizonal_line(segment, y_coordinate)` - funkcja służąca do znajdowania punktu przecięcia między odcinkiem, a poziomą linią
- `find_intersection_with_vertical_line(segment, x_coordinate)` - funkcja służąca do znajdowania punktu przecięcia między odcinkiem, a pionową linią

3.3 Pakiet `visualizer`

Pakiet jest przygotowanym pakietem do wizualizacji geometrycznej obiektów dostarczonym na laboratoriach.

3.4 `main.ipynb` oraz `README.md`

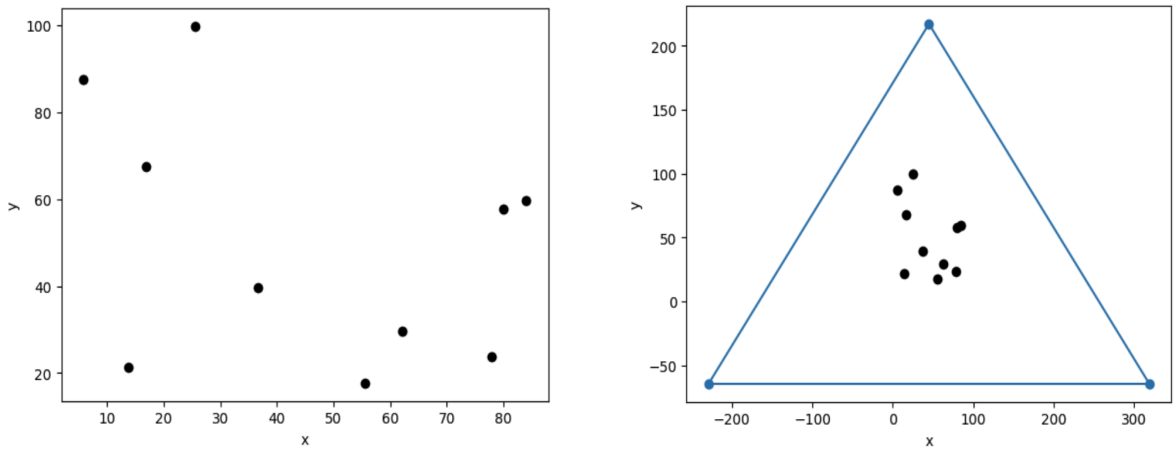
Plik `README.md` jest plikiem tekstowym z instrukcją wpisywania danych wejściowych i obsługi algorytmu. Plik `main.ipynb` jest natomiast prostym plikiem przeznaczonym dla użytkownika do dostosowania danych wejściowych oraz wizualizacji algorymtu.

Rozdział 4

Działanie algorytmu iteracyjnego

W pliku `Main.ipynb` znajduje się algorytm iteracyjny do znajdowania i wizualizacji diagramu Woronoja dla danego zbioru punktów. Zaczyna się on od inicjacji instancji klasy `Delaunay_Triangulation`, na której będziemy przeprowadzać większość operacji. Aby pokazać działanie algorytmu krok po kroku, stworzymy zbiór 10 punktów (`points`) o współrzędnych z zakresu $(0, 100)$. Na rys. 4.1 pokazany jest zbiór punktów oraz początkowy stan triangulacji: jak widać, zawiera ona trójkąt, którego wierzchołki nie należą do zbioru `points`. Jest to trójkąt wygenerowany funkcją `gen_init_triangle`. W następnym kroku będziemy, dla każdego punktu:

1. Znajdować trójkąt, do którego dany punkt należy
2. Znajdować sąsiedztwo punktu
3. Usuwać sąsiedztwo punktu
4. Odbudowywać sąsiedztwo punktu

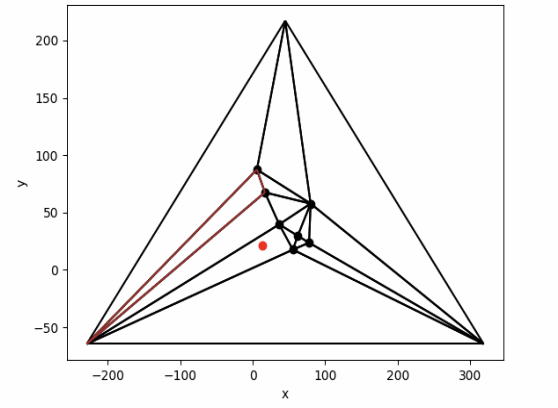


(a) Wygenerowane punkty

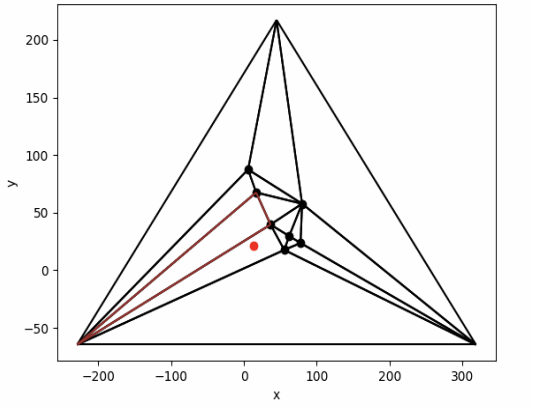
(b) Początkowy stan triangulacji

Rysunek 4.1: Pierwszy krok algorytmu

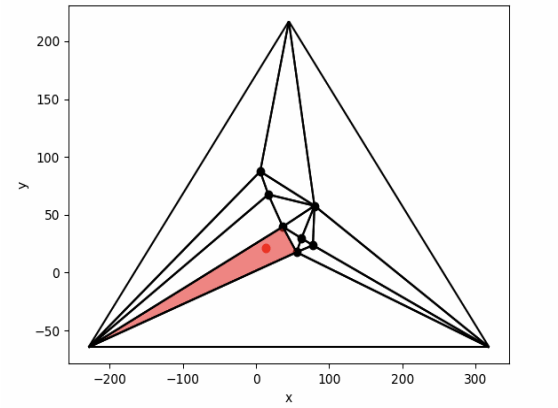
Aby czytelnie wizualizować działanie tych funkcji, przeniesiemy się do przodu, gdy już zostało dodane 6 punktów i jesteśmy w procesie dodawania siódmego (rys. 4.2). Naszym startowym trójkątem (atrybut `inittriangle`) jest trójkąt zaznaczony na rys. 4.2b, więc, jak widać, nie zawiera on nowego punktu (również zaznaczonego na czerwono). Rozpoczynamy zatem przeszukiwanie. Funkcja `find_triangle` wywoła dwa razy funkcję `find_next_triangle`, co przedstawione jest na rysunkach 4.2b i 4.2c.



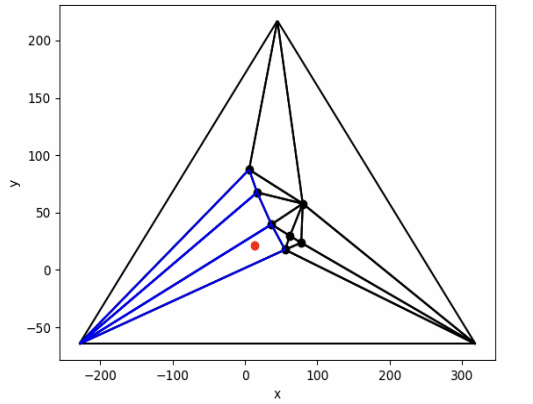
(a) Trójkąt, od którego zaczynamy przeszukiwanie po dodaniu ósmego punktu do triangulacji



(b) Przeszukiwanie następnego trójkąta



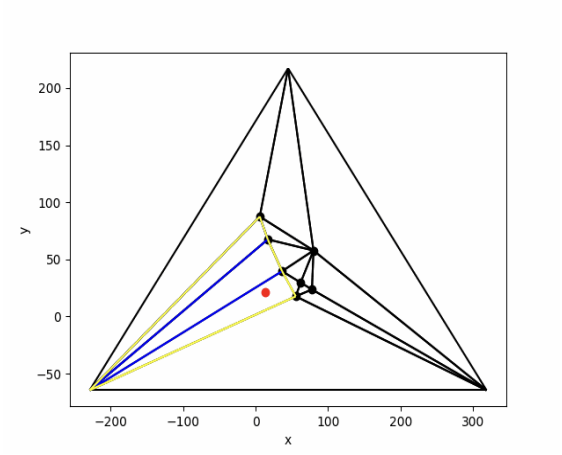
(c) Znaleziony trójkąt zawierający punkt



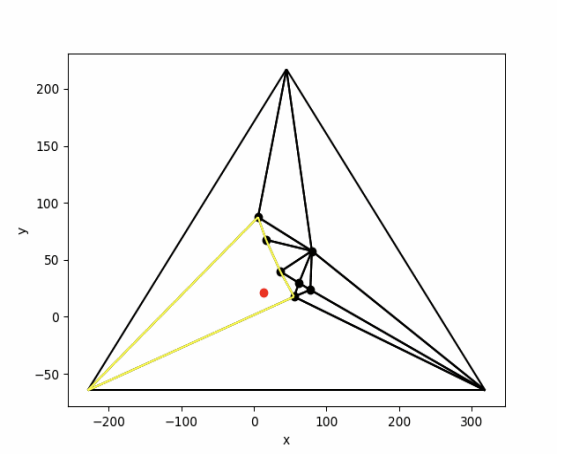
(d) Sąsiedztwo obecnie rozważanego punktu

Rysunek 4.2: Wizualizacja przeszukiwania trójkątów oraz sąsiedztwo

Znaleźliśmy więc trójkąt, do którego należy punkt. Zauważmy, że samo dodanie odcinków między wierzchołkami trójkąta przedstawionego na rys. 4.2c do nowego punktu utworzy poprawną triangulację, jednak nie będzie ona spełniać kryterium okręgów opisanych, więc nie będzie triangulacją Delaunay'a. Musimy więc znaleźć wszystkie trójkąty, których okręgi opisane zawierają nowy punkt - czyli jego sąsiedztwo. Służy do tego funkcja `find_neighbourhood`. Znalezione sąsiedztwo oraz jego otoczka przedstawione są na rys. 4.2d oraz 4.3a.



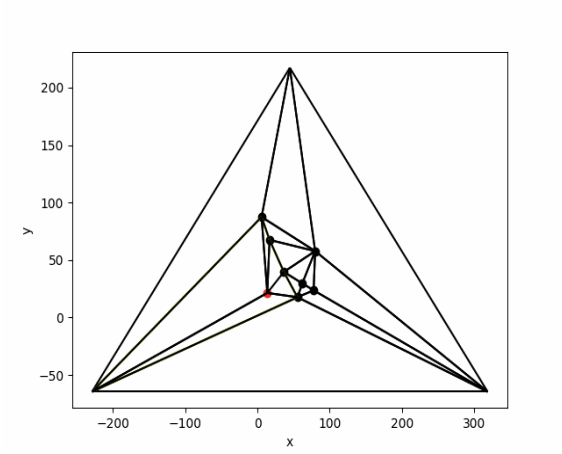
(a) Otoczka sąsiedztwa dla danego punktu



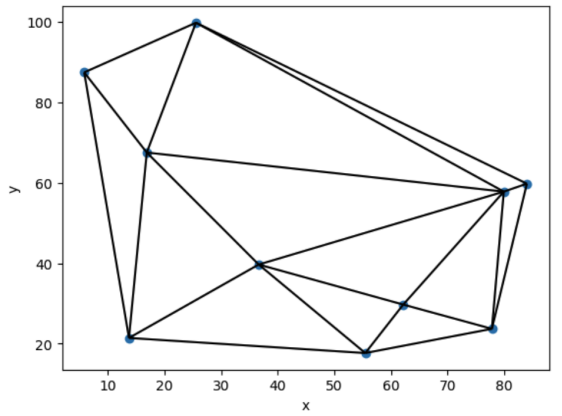
(b) Usunięte krawędzie sąsiedztwa z pominięciem otoczki

Rysunek 4.3: Otoczka sąsiedztwa oraz usuwanie sąsiedztwa

Następnym krokiem jest usunięcie sąsiedztwa. Służy do tego funkcja `delete_neighbourhood`. Jej działanie przedstawione jest na rys. 4.3b. Ostatnim krokiem iteracji jest odbudowanie nowego sąsiedztwa wokół punktu. Na rys. 4.4a pokazane jest, w jaki sposób funkcja `rebuild_neighbourhood` dodaje nowe krawędzie i trójkąty.



(a) Odbudowane sąsiedztwo punktu

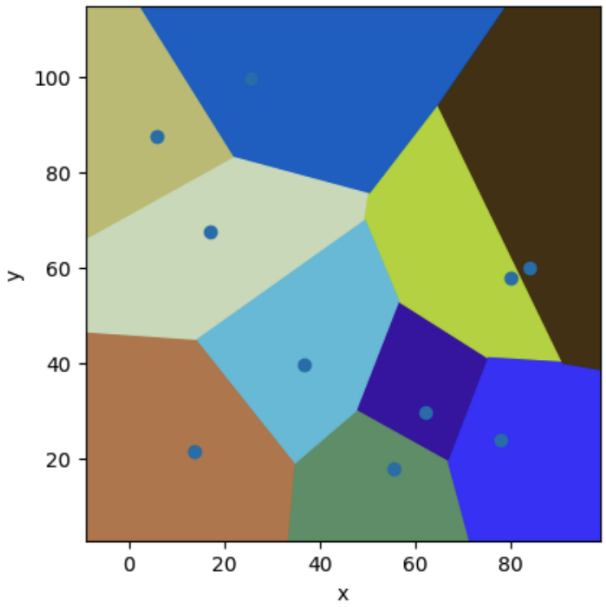


(b) Pełna triangulacja po usunięciu trójkąta pomocniczego

Rysunek 4.4: Ostatni krok w każdej iteracji oraz wynik triangulacji

Zauważmy, że odbudowane sąsiedztwo tworzy poprawną triangulację punktów. Spełnia ono również kryterium okręgów opisanych, jednak łatwiej jest dostrzec spełnianie równoważnego kryterium `max-min`, które polega na maksymalizacji minimum szcześciu kątów w trójkątach sąsiadujących ze sobą. Powtórzenie tej sekwencji dla wszystkich punktów skutkuje pełną triangulacją Delaunay'a, co zaprezentowane jest na rys. 4.4b.

Mając gotową triangulację, otrzymamy diagram Voronoi, odpowiednio łącząc środki okręgów opisanych na trójkątach tej triangulacji. Inicjujemy więc instancję klasy `VoronoiDiagram`, a następnie znajdujemy diagram metodą `create_diagram`. Końcowy efekt zaprezentowany jest na rys. 4.5



Rysunek 4.5: Gotowy diagram Woronoja dla danego zbioru punktów

Rozdział 5

Działanie algorytmu Fortune'a

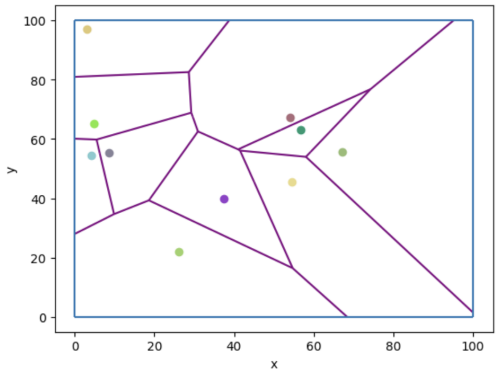
5.1 Ogólny opis

Algorytm Fortune'a wykorzystuje podejście z "linią zamiatającą", nazywaną również "miotłą". Rozważamy każdy punkt kolejno od lewej do prawej i "rozwijamy" komórki w miarę przesuwania się linii. Gdy dana komórka w otoczeniu punktu zostanie całkowicie otoczona przez inne, oznaczać to będzie, że nie może już dalej "rosnąć". Widzimy zatem, że wystarczy śledzić tylko te komórki (łuki) będące blisko "miotły". Zbiór ten nazywać będziemy "linią brzegową". W celu zapewnienia efektywności procedury wstawiania i wyszukiwania, linia brzegowa jest zbiorem uporządkowanym.

Rosnące komórki reprezentowane są jako parabole, które rosną wokół swojego punktu skupienia. Podczas działania algorytmu miotła napotyka na dwa typy zdarzeń. Pierwszym jest zdarzenie punktowe polegające na dodawaniu nowego łuku do linii brzegowej. Drugim typem zdarzenia jest zdarzenie przecięcia krawędzi, nazywane również zdarzeniem kołowym.

5.2 Przebieg algorytmu

Algorytm rozpoczyna od ustawienia linii miotły po lewej stronie wszystkich punktów oraz inicjalizacji pustej kolejki zdarzeń. Gdy miotła napotyka na nowy punkt dodaje nowy łuk do linii brzegowej oraz sprawdza i aktualizuje zdarzenia kołowe związane z nowo dodanym łukiem. Gdy natomiast przetwarzane jest zdarzenie kołowe, łuk jest usuwany z linii brzegowej, dodawany jest punkt w miejscu przecięcia się łuków oraz następuje aktualizacja łuków sąsiednich i ewentualne stworzenie nowych zdarzeń kołowych. Przeszedłszy wszystkie punkty i wydarzenia, algorytm kończy swoje działanie, dokańczając odcinki, które jeszcze nie mają dwóch końców. Na rys. 5.1 znajduje się powstały diagram Voronoi z użyciem tego algorytmu.



Rysunek 5.1: Diagram Voronoi uzyskany algorytmem Fortune'a

5.3 Problem znajdowania punktów przecięć dwóch paraboli

Zważając na fakt, że parabole powstałe podczas przesuwania miotły są zadane za pomocą ogniska i dorywczej, może pojawić się wątpliwość jak znajdować punkt przecięcia dwóch parabol zadanych właśnie w ten sposób.
Załóżmy, że mamy daną parabolę z osią symetrii równoległą do osi x oraz wierzchołkiem o współrzędnych (h, k), ogniskiem $(h, k + p)$ i dorywczą $y = k - p$. Taką parabolę możemy zapisać wówczas jako:

$$(y - h)^2 = 4p(x - k)$$

Po przekształceniach, możemy zapisać dwie rozważane parabole jako odpowiednio:

$$x = \frac{1}{4p_1}y^2 - \frac{1}{2p_1}y + \frac{h_1^2}{4p_1} + k_1$$

$$x = \frac{1}{4p_2}y^2 - \frac{1}{2p_2}y + \frac{h_2^2}{4p_2} + k_2$$

Przyrównując do siebie te dwa równania możemy obliczyć współrzędną y punktu przecięcia. Wystarczy rozwiązać następujące równanie kwadratowe korzystając z wyróżnika:

$$(\frac{1}{4p_1} - \frac{1}{4p_2})y^2 + (\frac{h_2}{2p_2} - \frac{h_1}{2p_1})y + \frac{h_1^2}{4p_1} + k_1 - \frac{h_2^2}{4p_2} - k_2 = 0$$

Następnie, mając współrzędną y, wystarczy wstawić ją do dowolnego równania wyznaczonego wcześniej i w ten sposób otrzymać współrzędną x.

5.4 Struktury wykorzystane w implementacji

Do przechowywania zdarzeń punktowych oraz kołowych została wykorzystana implementacja kolejki priorytetowej, która pozwala na optymalny dostęp do zdarzeń o najmniejszej współrzędnej x.
Jeśli chodzi o strukturę stanu miotły, implementacja powinna opierać się na drzewie binarnym przechowującym punkty oraz łuki, natomiast ze względu na złożoność tej struktury zdecydowaliśmy się skorzystać z listy dwukierunkowej, skupiając się w pierwszej kolejności na poprawności algorytmu. Niestety rozwiązanie to ma wpływ na złożoność obliczeniową, co opisane jest w kolejnym rozdziale.

Rozdział 6

Podsumowanie

W ramach podsumowania projektu warto pochylić się nad różnicami w podejściu podczas implementacji obu algorytmów oraz przeanalizować czas działania tych podejść dla różnych zbiorów danych.

Różnice w działaniu algorytmów

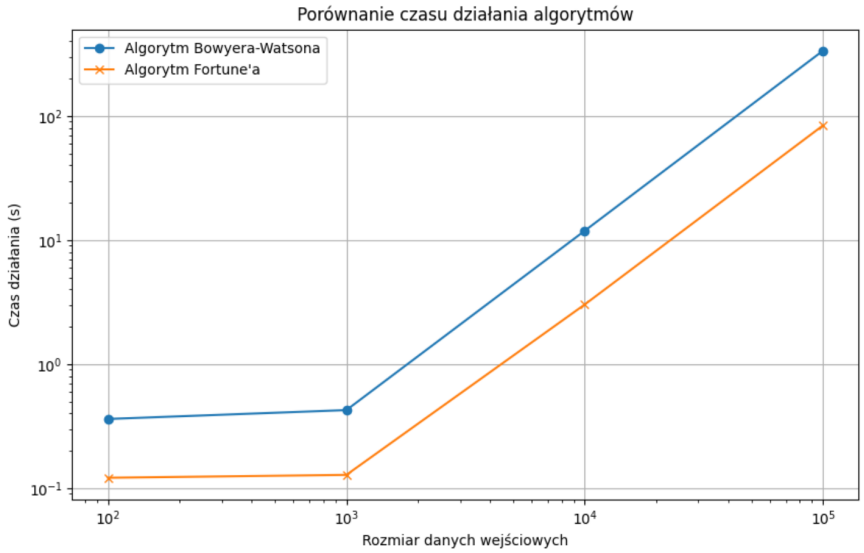
Algorytm Bowyera-Watsona korzysta z podejścia iteracyjnego. Konstruowanie diagramu odbywa się stopniowo wraz z dodawaniem kolejnych punktów i modyfikowaniem istniejącej struktury. Poprzez implementacje wydajnej funkcji rekurencyjnej znajdującej trójkąt, w którym znajduje się nowo dodany punkt, średni czas działania funkcji został zmniejszony. Najgorszy przypadek przeszukania wszystkich trójkątów jest natomiast niezmienny i wpływa na to, że złożoność implememntacji to $O(n^2)$. Na czas obliczeń może wpływać natomiast losowość przy dodawaniu nowych punktów, przez co mogą być przypadki, w których będzie należało sprawdzić wszystkie trójkąty w triangulacji.

Algorytm Fortune'a wykorzystuje natomiast "zamiatanie", aby efektywnie tworzyć diagram Woronoja. Przy odpowiednim wykorzystaniu struktur do przechowywania zdarzeń oraz stanu miotły można osiągnąć złożoność na poziomie $O(n \log n)$. W przypadku naszego algorytmu, ze względu na wykorzystanie struktury listy dwukierunkowej do przechowywania stanu miotły, złożoność może teoretycznie wzrastać do $O(n^2)$, natomiast przeprowadzone testy, które będą prezentowane w tym rozdziale, pokazały, że algorytm radzi sobie bardzo dobrze nawet z dużymi zestawami danych.

Porównanie wydajności

Liczba punktów	**Czas działania algorytmu [s]**	
	Algorytm Bowyera-Watsona	Algorytm Fortune'a
10^2	0.046	0.016
10^3	0.42	0.12
10^4	11.27	2.79
10^5	321.47	74.62

Tabela 6.1: Porównanie czasu działania algorytmów dla różnej wielkości danych wejściowych



Rysunek 6.1: Porównanie czasu działania algorytmów w formie graficznej, wykres w skali logarytmicznej

Punkty generowane do testów posiadały współrzędne x oraz y z przedziału $[0, 1000]$. Jak możemy zauważyć w tabeli 6.1, algorytm Fortune'a jest szybszy dla każdego zestawu danych wejściowych, natomiast największe różnice pojawiają się dla najbardziej licznych zestawów danych wejściowych.

Łatwość implementacji i użytkowania

Porównując te dwa algorytmy można bez wahania stwierdzić, że algorytm Frotune'a wymaga większej sprawności programistycznej oraz wiedzy z zakresu geometrii obliczeniowej, ze względu na zaawansowane struktury wykorzystywane podczas implementacji. Struktura stanu miotły, która zapewnia optymalność, czyli odpowiednie drzewo binarne, jest skomplikowaną strukturą trudną do implementacji.

W przypadku algorytmu Bowyera-Watsona, implementacja w dobrej złożoności również wymaga dużej wiedzy i pomysłów, natomiast sam kod może być bardziej zrozumiały dla mniej zaawansowanych użytkowników. Optymalizacja polegająca na zminimalizowaniu liczby przeszukiwanych trójkątów nie jest najprostsza, natomiast pozwala na zmniejszenie czasu działania dla wielu przypadków.

Bibliografia

Główna idea działania algorytmów została zaczerpnięta z wykładów z przedmiotu Algorytmy Gometryczne. Pakiet `visualizer` został przygotowany do laboratoriów z tego przedmiotu przez koło naukowe BIT AGH. Poniżej znajdują się dodatkowe źródła, które pomogły nam w implementacji algorytmów:

- de Berg, Mark, et al. "Diagramy Voronoi." Geometria obliczeniowa: algorytmy i zastosowania, Przekład Mirosław Kowaluk, 2. ed., Wydawnictwo Naukowo-Techniczne, 2007, ss. 177-195.
- Jacques Heunis. "Fortunes Algorithm: An intuitive explanation". 28 marca 2018, https://jacquesheunis.com/post/fortunes-algorithm/.
- Brubeck, Matt. "Fortune's Algorithm in C++". 2002, https://www.cs.hmc.edu/ mbrubeck/voronoi.html.
- Malczewski, Michał. "Implementacja algorytmów obliczających diagramy Voronoi w języku Python." Diagramy Voronoi w geometrii obliczeniowej, red. A. Kapanowski, Wydawnictwo Uniwersytetu w Białymstoku, 2021, ss. 105-122.
- GeeksforGeeks. https://www.geeksforgeeks.org/