Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Mateusz Malczewski

Nr albumu: 1153671

Diagramy Voronoi w geometrii obliczeniowej

Praca licencjacka na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Informatyki Stosowanej

Kraków 2021

1:9869977611

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia   Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia   Podpis kierującego pracą

2:1017950727

Bardzo dziękuję Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu, dzięki któremu napisanie tej pracy stało się o wiele prostszym wyzwaniem.

3:9454444076

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów wyznaczających diagramy Voronoi w geometrii obliczeniowej. Diagram Voronoi dla punktów ze zbioru P to podział płaszczyzny na wypukłe obszary powiązane z poszczególnymi punktami zbioru. Każdy punkt w danym obszarze znajduje się bliżej punktu powiązanego z tym obszarem, niż pozostałych punktów ze zbioru P.

Przygotowano algorytmy generujące diagramy Voronoi dwoma sposobami: bezpośrednio (algorytm Fortune'a) i pośrednio (przekształcając triangulację Delaunaya). Algorytm Fortune'a wykorzystuje technikę zamiatania płaszczyzny, gdzie napotykane punkty generują zdarzenia miejscowe i zdarzenia kołowe, a do tego należy śledzić kształt linii brzegowej. W metodzie pośredniej najpierw wyznacza się triangulację Delaunaya dla zbioru punktów P, a następnie środki okręgów opisanych na trójkątach triangulacji stają się wierzchołkami diagramu Voronoi. Sprawdzono kilka wersji tej metody.

Algorytmy, oraz używane przez nie struktury danych, zostały szczegółowo opisane. Algorytmy zostały przetestowane pod względem poprawności, a ich teoretyczne złożoności obliczeniowe zgadzają się z wynikami testów praktycznych.

Słowa kluczowe: diagram Voronoi, triangulacja Delaunaya, algorytm Fortune'a, drzewo AVL, podwójnie łączona lista krawędzi

4:6853169623

English title: Voronoi diagrams in computational geometry

Abstract

Python implementation of selected algorithms finding Voronoi diagrams in computational geometry is presented. A Voronoi diagram of a given point set P in the plane is a division of the plane into convex areas associated with specific points of the set. Each point in a given area is closer to point associated with that area than to the rest of points from P.

Presented algorithms compute Voronoi diagrams using two methods: a direct method (Fortune's algorithm) and an indirect method (transformation of a Delaunay triangulation). The Fortune's algorithm is a sweep line algorithm, where the input points generate site events and circle events, whereas a beach line moving through the plane is maintained. The indirect method starts with a Delaunay triangulation, and then circumcenters of the triangles create vertices of the Voronoi diagram. Several versions of this method were checked.

All algorithms and data structures used by them are described in detail. Algorithms were carefully tested for correctness and their theoretical time complexities were confirmed by the practical tests.

Keywords: Voronoi diagram, Delaunay triangulation, Fortune's algorithm, AVL tree, doubly connected edge list

5:4601027231

Spis treści

Spis rysunków 3
Listings 4
1. Wstęp 5
1.1. Cele pracy 5
1.2. Organizacja pracy 7
2. Geometria obliczeniowa 8
2.1. Diagram Voronoi 8
2.2. Triangulacja Delaunaya 10
3. Implementacja 11
3.1. Figury geometryczne 11
3.1.1. Nowe metody klasy Triangle 11
3.2. Graf 12
3.3. Klasa BowyerWatson 12
3.4. Klasa QuickHull 12
3.5. Kolejka priorytetowa 12
3.6. Drzewo AVL 13
3.6.1. Nowe metody drzewa AVL 13
3.7. Podwójnie łączona lista krawędzi 14
3.7.1. Półkrawędź 15
3.7.2. Wierzchołek 16
3.7.3. Ściana 16
3.8. Klasa Arc 16
3.9. Klasa Breakpoint 17
3.10. Zdarzenia 18
3.11. Klasy VoronoiArea i VoronoiAreaCollection 19
3.12. Przykładowe użycie algorytmów 21
4. Algorytmy 23
4.1. Algorytm Fortune'a 23
4.2. Diagram Voronoi z triangulacji Delaunaya 28
4.3. Diagram Voronoi z triangulacji Delaunaya w postaci listy odcinków 31
5. Podsumowanie 33
A. Testy algorytmów 34
A.1. Testy algorytmu Fortune'a 34
A.2. Testy wyznaczania diagramu Voronoi z triangulacji Delaunaya . . 34
Bibliografia 38

6:9142624610

Spis rysunków

1.1. Podział powierzchni Poznania na komórki diagramu Voronoi. 6

2.1. Przykładowy diagram Voronoi. 9
2.2. Największy pusty okrąg, którego środkiem jest q. 9
2.3. Wizualizacja warunków wierzchołków i krawędzi diagramu. 9
2.4. Przykładowy diagram Voronoi z triangulacją Delaunaya. 10

3.1. Przykładowa krawędź reprezentowana za pomocą półkrawędzi. 15

4.1. Przykładowa linia brzegowa. 24
4.2. Na pomarańczowo łuk w linii brzegowej i jego pozycja w drzewie. . . . 24
4.3. Wizualizacja dodawania nowego łuku do linii brzegowej w momencie natrafienia przez miotłę na zdarzenie miejscowe. 26
4.4. Wizualizacja znikania łuku z linii brzegowej w momencie natrafienia przez miotłę na aktywne zdarzenie kołowe. 26
4.5. Drzewo przed i po usunięciu przykładowego łuku. 27

A.1. Wydajność algorytmu Fortune'a dla losowych punktów. 35
A.2. Wydajność algorytmu Fortune'a dla punktów współliniowych (oś X). 35
A.3. Wydajność algorytmu Fortune'a dla punktów współliniowych (oś Y). . 36
A.4. Wydajność algorytmu przekształcającego triangulację Delaunaya w diagram Voronoi dla losowych punktów. 36
A.5. Wydajność algorytmu przekształcającego triangulację Delaunaya w listę odcinków diagramu Voronoi. 37

7:4438315105

Listings

3.1 Metody is_neighbor() i circumcenter() z klasy Triangle. 11
3.2 Metody predecessor_by_node() i successor_by_node() z modułu avltree. 13
3.3 Metody predecessor_leaf_by_node() i successor_leaf_by_node() z modułu avltree. 13
3.4 Metoda replace_leaf() z modułu avltree. 14
3.5 Klasa DCEL z modułu dcel. 14
3.6 Klasa Halfedge z modułu dcel. 15
3.7 Klasa Vertex z modułu dcel. 16
3.8 Klasa Face z modułu dcel. 16
3.9 Klasa Arc z modułu arcs. 17
3.10 Klasa Breakpoint z modułu breakpoints. 17
3.11 Klasa Event z modułu events. 18
3.12 Klasa VoronoiArea z modułu voronoi_area. 19
3.13 Klasa VoronoiAreaCollection z modułu voronoi_area_collection. . . 20
4.1 Klasa VoronoiFromDelaunay2 z modułu voronoi_from_delaunay2. . . 29
4.2 Klasa VoronoiFromDelaunay1 z modułu voronoi_from_delaunay1. . . 31

8:4398450511

1. Wstęp

Tematem niniejszej pracy jest wyznaczanie diagramu Voronoi dla zbioru punktów P na płaszczyźnie. Diagram Voronoi jest to podział płaszczyzny na obszary powiązane z punktami ze zbioru P. Każdy punkt w danym obszarze znajduje się bliżej punktu powiązanego z tym obszarem, niż od pozostałych punktów ze zbioru P [1]. Kształty obszarów zależą od funkcji użytej do pomiaru odległości (metryki). Może to być przykładowo odległość euklidesowa lub taksówkowa. Diagramy te pełnią istotną rolę w geometrii obliczeniowej, oraz posiadają wiele zastosowań, między innymi w domenach biologii, medycyny, inżynierii czy planowania.

Przykładem zastosowania diagramu Voronoi może być podział pewnego obszaru przez sieć sklepów. Punktami, dla których obliczamy diagram, są położenia sklepów sieci na mapie, a wygenerowane komórki diagramu w połączeniu z gęstością ich zaludnienia pozwalają określić obszary, w których sklepów jest za mało, a także miejsca na mapie, w których sklepów jest za dużo. Informacje te mogą zostać wykorzystane do określenia położenia budowy nowego sklepu lub do likwidacji oddziału znajdującego się w obszarze z małą liczbą mieszkańców (rys. 1.1) [2].

Diagramy Voronoi wykorzystywane są także przez krystalografów przy analizie struktury niektórych kryształów i metali, przez ekologów w celu zbadania konkurencji między roślinami, czy przez antropologów przy badaniu zasięgu wpływów różnych kultur [3].

Diagramy Voronoi można obliczać dwiema metodami.

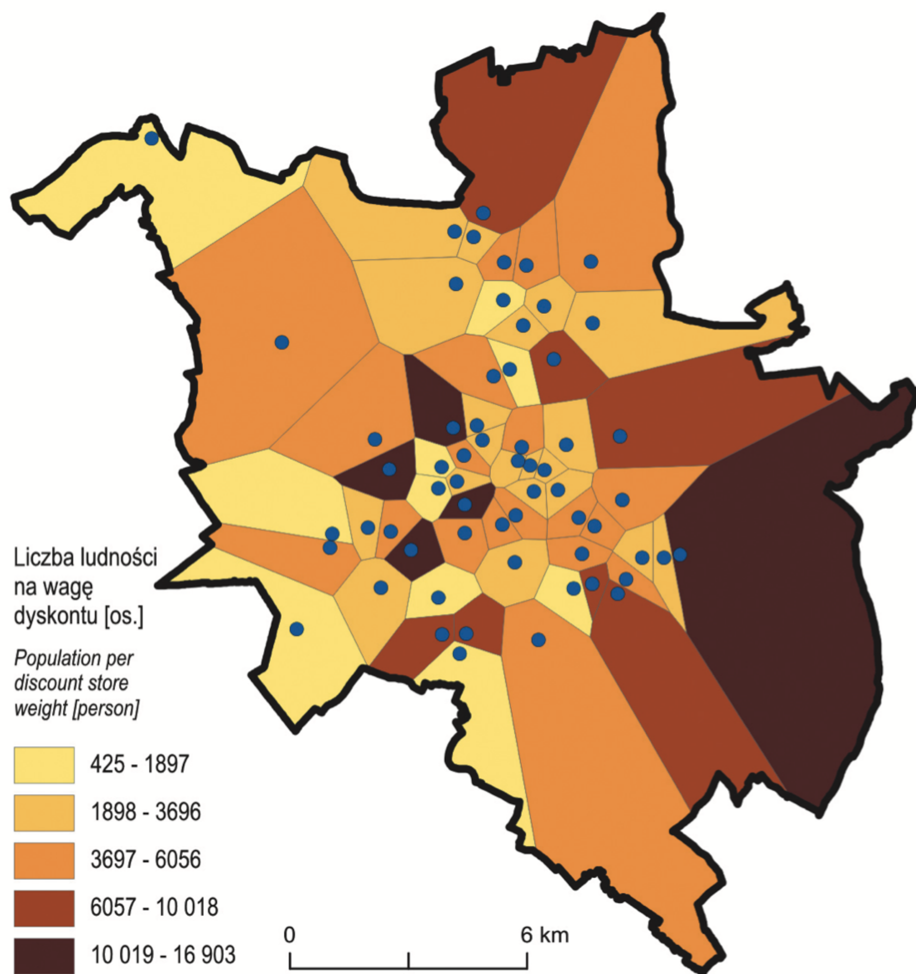
— **Metoda bezpośrednia.** Algorytm otrzymuje na wejściu listę punktów, a następnie generuje dla nich diagram Voronoi. Przykładem algorytmu bezpośredniego jest algorytm Fortune'a.
— **Metoda pośrednia.** Algorytm otrzymuje na wejściu listę punktów, dla których generuje najpierw triangulację Delaunaya, a następnie przekształca ją w diagram Voronoi. Triangulację można obliczyć na przykład za pomocą algorytmu Bowyera-Watsona.

1.1. Cele pracy

Głównym celem pracy jest implementacja algorytmów obliczających diagramy Voronoi zarówno bezpośrednio (algorytm Fortune'a) jak i pośrednio. Zastosowanie języka Python [4], posiadającego prostą i czytelną składnię, ułatwia zrozumienie dość skomplikowanych algorytmów.

Praca wykorzystuje i rozwija pakiet planegeometry [5]. Szczególnie wykorzystywanymi elementami pakietu są podstawowe obiekty geometryczne (punkt, odcinek, trójkąt, wielokąt), struktury danych wykorzystywane przy algorytmach opierających się na technice zamiatania płaszczyzny (drzewo AVL)

9:1969292539



Rysunek 1.1. Podział powierzchni Poznania na komórki diagramu Voronoi dla punktów będącymi położeniami sklepów sieci Biedronka. Uwzględniona została także liczba mieszkańców przypadająca na wagę sklepu (wielkość dyskontu) [2].

10:3100305751

oraz algorytmy obliczające triangulację Delaunaya, opracowane odpowiednio w pracach Marcina Permusa [6], Wojciecha Chrobaka [7] i Moniki Wiech [8]. Ogólne informacje z dziedziny geometrii obliczeniowej i algorytmów geometrycznych zostały zaczerpnięte z książek [9], [10], [11], [12].

1.2. Organizacja pracy

Praca została zorganizowana w następujący sposób. Rozdział 1 wprowadza w tematykę i prezentuje cele pracy. Rozdział 2 zawiera opis istotnych dla pracy pojęć z dziedziny geometrii obliczeniowej. Rozdział 3 przedstawia implementacje użytych struktur danych oraz przykładowe zastosowanie algorytmów. Rozdział 4 prezentuje implementację algorytmów obliczających diagramy Voronoi oraz ich opis. Rozdział 5 podsumowuje pracę. Dodatek A zawiera wyniki testów omówionych algorytmów.

11:9022470427

2. Geometria obliczeniowa

Rozdział ten zawiera opis podstawowych definicji i twierdzeń z dziedziny geometrii obliczeniowej używanych w tej pracy.

2.1. Diagram Voronoi

Definicja [9]: Niech $\text{dist}(p, q)$ będzie odległością euklidesową między punktami p i q. Na płaszczyźnie otrzymujemy:

$$\text{dist}(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}. \tag{2.1}$$

Niech $P = \{p_1, p_2, ..., p_n\}$ będzie zbiorem n różnych punktów na płaszczyźnie. Są to punkty dla których tworzymy diagram. Diagram Voronoi dla zbioru P definiujemy jako podział płaszczyzny na n komórek w taki sposób, że punkt q należy do komórki odpowiadającej punktowi p_i ze zbioru P tylko, gdy $\text{dist}(q, p_i) < \text{dist}(q, p_j)$ dla każdego $p_j \in P$ takiego, że $j \neq i$.

Nazwa diagram Voronoi pochodzi od nazwiska ukraińskiego matematyka Georgy'a Voronoy, który jako pierwszy go zdefiniował.

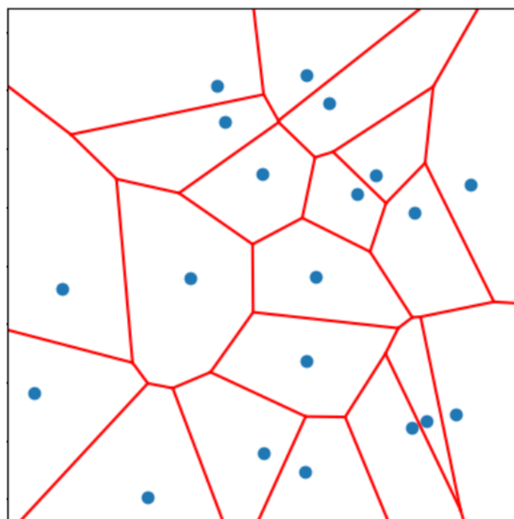
Twierdzenie [9]: Niech P będzie zbiorem n różnych punktów na płaszczyźnie. Jeżeli wszystkie punkty z P są współliniowe, to wynikowy diagram Voronoi składa się z $n - 1$ równoległych linii. W przeciwnym wypadku, krawędzie diagramu są połączonymi odcinkami lub półprostymi.

Twierdzenie [9]: Niech P będzie zbiorem n różnych punktów na płaszczyźnie. Dla $n \geqslant 3$ liczba wierzchołków w diagramie Voronoi wynosi maksymalnie $2n - 5$, a liczba krawędzi wynosi maksymalnie $3n - 6$ (poza sytuacją, gdy wszystkie punkty ze zbioru P są współliniowe).

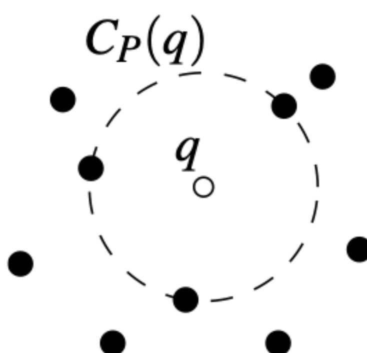
Twierdzenie [9]: Niech $P = \{p_1, p_2, ..., p_n\}$ będzie zbiorem n różnych punktów na płaszczyźnie. Dla diagramu Voronoi utworzonego na podstawie zbioru P zachodzi:

— Punkt q jest wierzchołkiem diagramu Voronoi wtedy i tylko wtedy, gdy na największym pustym okręgu, którego jest środkiem, leżą trzy lub więcej punkty ze zbioru P (rys. 2.2 i 2.3).
— Dwusieczna pomiędzy punktami p_i i p_j definiuje krawędź diagramu Voronoi wtedy i tylko wtedy, gdy istnieje punkt q leżący na dwusiecznej taki, że na największym pustym okręgu, którego jest środkiem, jako jedyne leżą punkty p_i i p_j (rys. 2.3).

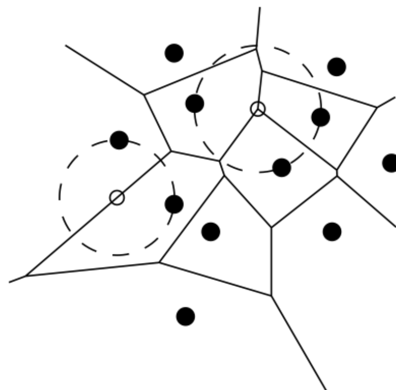
12:1391053421



Rysunek 2.1. Przykładowy diagram Voronoi.



Rysunek 2.2. Największy pusty okrąg, którego środkiem jest q [9].



Rysunek 2.3. Wizualizacja warunków wierzchołków i krawędzi diagramu [9].

13:4108104662

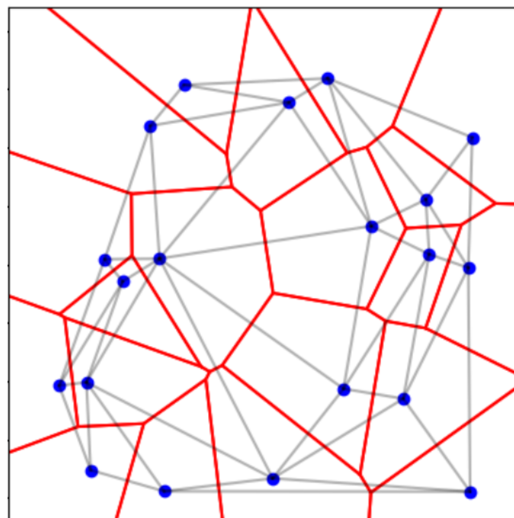
2.2. Triangulacja Delaunaya

Definicja [13]: Triangulacja Delaunaya to podział przestrzeni R^{n+1} na triangulację złożoną ze skończonej ilości $(n+1)$-sympleksów w taki sposób, że spełnione są następujące warunki:

— każde dwa sympleksy posiadają wspólną ścianę lub wierzchołek albo nie mają części wspólnej,
— każdy zbiór ograniczony w R^{n+1} ma część wspólną ze skończoną liczbą sympleksów,
— wnętrze kuli opisanej na dowolnym sympleksie nie zawiera wierzchołków żadnego sympleksu.

Triangulacja Delaunaya ma wiele zastosowań, między innymi w modelowaniu powierzchni czy w planowaniu. Została wymyślona przez rosyjskiego matematyka Borysa Delone w 1934 roku.

Istotną z punktu widzenia diagramów Voronoi właściwością triangulacji Delaunaya jest fakt, że graf dualny triangulacji odpowiada diagramowi Voronoi dla tego samego zbioru punktów [14]. Środki okręgów opisanych na trójkątach triangulacji odpowiadają wierzchołkom diagramu Voronoi, a odpowiednie krawędzie między tymi wierzchołkami można uzyskać biorąc pod uwagę sąsiedztwo trójkątów (rys. 2.4).



Rysunek 2.4. Przykładowy diagram Voronoi z triangulacją Delaunaya.

3. Implementacja

Rozdział ten poświęcony jest omówieniu podstawowych obiektów, struktur danych i metod pomocniczych, stosowanych w opisanych w tej pracy algorytmach.

3.1. Figury geometryczne

Algorytmy obliczające diagramy Voronoi wykorzystują podstawowe elementy geometrii obliczeniowej zdefiniowane w pakiecie planegeometry. Są nimi klasy: Point (reprezentująca punkt), Segment (reprezentująca odcinek), Triangle (reprezentująca trójkąt) i Polygon (reprezentująca wielokąt), a także zdefiniowane w module geomtools funkcje pomocnicze.

3.1.1. Nowe metody klasy Triangle

Do klasy Triangle została dodana metoda is_neighbor(), sprawdzająca, czy dwa trójkąty sąsiadują ze sobą (czy mają wspólny bok), oraz metoda circumcenter(), obliczająca środek okręgu opisanego na trójkącie. Metody te wykorzystywane są w algorytmie przekształcającym triangulacje Delaunaya w diagram Voronoi. Warto zauważyć, że metoda circumcenter() zwraca współrzędne środka okręgu w postaci liczb wymiernych (wynik jest dokładny numerycznie), jeżeli wierzchołki trójkąta mają współrzędne wymierne.

Listing 3.1. Metody is_neighbor() i circumcenter() z klasy Triangle.

```
class Triangle:

    def is_neighbor(self, other):
        """Test if triangles have a common segment."""
        set1 = set([self.pt1, self.pt2, self.pt3])
        set2 = set([other.pt1, other.pt2, other.pt3])
        set3 = set1 & set2
        return len(set3) == 2

    def circumcenter(self):
        """Return the circumcenter for the triangle.

        https://en.wikipedia.org/wiki/Circumscribed_circle#Circumcircle_equations
        """
        a, b, c = self.pt1, self.pt2, self.pt3
        d = 2 * ( a.cross(b) - a.cross(c) + b.cross(c) )
        x = ((a*a)*(b.y - c.y) + (b*b)*(c.y - a.y) + (c*c)*(a.y - b.y))
        y = ((a*a)*(c.x - b.x) + (b*b)*(a.x - c.x) + (c*c)*(b.x - a.x))
        if isinstance((x+y+d), float):
            return Point(x / float(d), y / float(d))
        else:
```


15:7531553008

```
return Point(Fraction(x, d), Fraction(y, d))
```

3.2. Graf

Struktura danych z pakietu planegeometry, która razem z klasą Edge, reprezentująca krawędź grafu, pozwala na przechowywanie danych w formie grafu, co w wielu przypadkach korzystnie wpływa na złożoność obliczeniową algorytmów. Stosowana jest między innymi do przechowywania informacji o triangulacji Delaunaya w algorytmie przekształcającym ją w diagram Voronoi. Najczęściej wierzchołkami grafu są punkty na płaszczyźnie (hashowalne instancje klasy Point), a krawędziami odcinki łączące te punkty. Warto zauważyć, że interfejs klasy Segment celowo zawiera w sobie atrybuty klasy Edge, dzięki czemu odcinki mogą być bezpośrednio wstawiane do grafu.

3.3. Klasa BowyerWatson

Klasa BowyerWatson z modułu planegeometry oblicza triangulację Delaunaya za pomocą algorytmu Bowyera-Watsona. Umożliwia zapisanie triangulacji w postaci grafu, co jest wykorzystywane w algorytmie przekształcającym ją w diagram Voronoi.

3.4. Klasa QuickHull

Klasa QuickHull z modułu planegeometry oblicza otoczkę wypukłą podanego zbioru punktów [15]. Wykorzystywana w algorytmie przekształcającym triangulacje Delaunaya w diagram Voronoi w celu utworzenia poprawnego wielokąta otaczającego daną komórkę Voronoi.

3.5. Kolejka priorytetowa

Kolejka priorytetowa [16] to struktura danych posiadająca dwie funkcje: dodanie elementu do kolejki oraz zwrócenie (i usunięcie z kolejki) elementu o najmniejszym priorytecie (lub największym priorytecie, w zależności od implementacji). Nie ma możliwości pobrania z kolejki elementu innego niż ten, który znajduje się w niej na pierwszym miejscu. Te ograniczenia pozwalają jednak osiągać złożoność obliczeniową $O(\log n)$, zarówno w przypadku dodawania elementu do kolejki, jak i w przypadku zwracania elementu o najmniejszym priorytecie. Użyta w pracy implementacja kolejki priorytetowej pochodzi z pakietu planegeometry, a jej zastosowanie w algorytmie Fortune'a jest konieczne w celu uzyskania końcowej złożoności obliczeniowej $O(n \log n)$. Ta implementacja kolejki posiada zabezpieczenie przed wstawieniem dupliaktu elementu, który już jest w kolejce.

16:5044469441

3.6. Drzewo AVL

Drzewo AVL [17] to struktura danych reprezentująca binarne drzewo poszukiwań, w którym każdy węzeł posiada dodatkowy parametr nazywany współczynnikiem wyważenia, kontrolujący wysokość lewego i prawego poddrzewa tego węzła. W momencie wykrycia przechylenia drzewa lub dowolnego jego poddrzewa w którąś stronę (gdy wysokość lewego i prawego poddrzewa rozpatrywanego węzła różni się o więcej niż 1), następuje zrównoważenie drzewa poprzez wykonanie odpowiednich rotacji. Zastosowanie tego rodzaju ochrony przed utworzeniem niezrównoważonego drzewa zwiększa koszt pojedynczej modyfikacji, ale gwarantuje pesymistyczną złożoność obliczeniową wyszukiwania elementów w drzewie rzędu $O(\log n)$. Użyta w pracy implementacja drzewa AVL pochodzi z pakietu planegeometry, a zastosowanie w algorytmie Fortune'a zrównoważonego drzewa przeszukiwań binarnych jest konieczne w celu uzyskania końcowej złożoności obliczeniowej $O(n \log n)$.

Implementacja drzewa AVL z pakietu planegeometry jest oparta na kodzie opublikowanym na Rosetta Code [18] i bibliotece PyBST [19].

3.6.1. Nowe metody drzewa AVL

Aby umożliwić i ułatwić użycie klasy AVLTree w implementacji algorytmu Fortune'a, klasa została wzbogacona o nowe metody. Potrzebne węzły drzewa w algorytmie Fortune'a przechowywane są w postaci referencji, dlatego dodane zostały metody odnajdujące poprzednika i następnika danego węzła, gdy posiadamy do niego referencję.

Listing 3.2. Metody predecessor_by_node() i successor_by_node() z modułu avltree.

```
class AVLTree:

    def predecessor_by_node(self, node):
        return node and node.predecessor()

    def successor_by_node(self, node):
        return node and node.successor()
```

Dodatkowo, dodane zostały także metody odnajdujące poprzednika i następnika danego węzła, który jest liściem drzewa, również przyjmujące jako argument referencję do interesującego nas węzła.

Listing 3.3. Metody predecessor_leaf_by_node() i successor_leaf_by_node() z modułu avltree.

```
class AVLTree:

    def predecessor_leaf_by_node(self, node):
        if node.left:
            return node.left.find_max()
        current = node
        while current.parent and current is current.parent.left:
            current = current.parent
        if current.parent is None or current.parent.left is None:
            return None
```


17:2346504844

```
        return current.parent.left.find_max()

    def successor_leaf_by_node(self, node):
        if node.right:
            return node.right.find_min()
        current = node
        while current.parent and current is current.parent.right:
            current = current.parent
        if current.parent is None or current.parent.right is None:
            return None
        return current.parent.right.find_min()
```

Drzewo przeszukiwań binarnych w algorytmie Fortune'a przechowuje inne informacje w węzłach będących węzłami wewnętrznymi, a inne w węzłach będącymi liśćmi. Aby zapobiec sytuacji, w której na skutek rotacji równoważących drzewo, węzeł będący liściem zostałby węzłem wewnętrznym (lub na odwrót), zamieniamy wyszukany liść drzewa odpowiednio już zbudowanym poddrzewem i dopiero po tej operacji dokonujemy (jeżeli to konieczne) zrównoważenia drzewa, zamiast dodawać do drzewa pojedyncze węzły. By móc tego dokonać, do klasy AVLTree dodana została metoda replace_leaf().

Listing 3.4. Metoda replace_leaf() z modułu avltree.

```
class AVLTree:

    def replace_leaf(self, node, sub_tree):
        if node.left is not None or node.right is not None:
            return Exception("Not a leaf")
        if node.parent is None:
            self.root = sub_tree
        elif node.parent.left == node:
            parent = node.parent
            parent.left = sub_tree
            sub_tree.parent = parent
        elif node.parent.right == node:
            parent = node.parent
            parent.right = sub_tree
            sub_tree.parent = parent
```

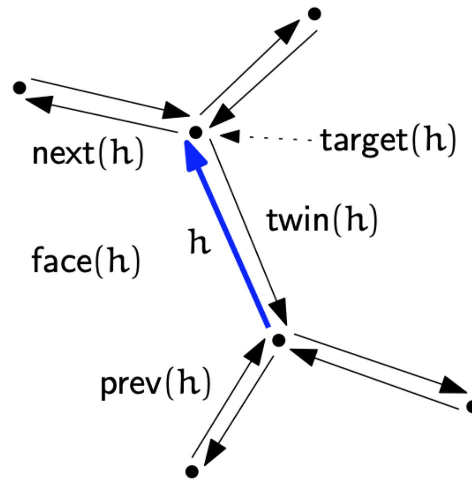
3.7. Podwójnie łączona lista krawędzi

Podwójnie łączona lista krawędzi (ang. *doubly connected edge list*) to struktura danych, w której przechowywane są w pełni obliczone już fragmenty diagramu Voronoi. Wykorzystanie jej w algorytmie Fortune'a pozwala na osiągnięcie końcowej złożoności obliczeniowej $O(n \log n)$. W skład tej struktury wchodzą trzy obiekty: półkrawędzie, wierzchołki oraz ściany (ang. *face*). Podwójnie łączona lista krawędzi zawiera trzy listy odpowiadające tym obiektom: listę półkrawędzi, listę wierzchołków i listę ścian.

Listing 3.5. Klasa DCEL z modułu dcel.

```
class DCEL:
```


18:6682530804



Rysunek 3.1. Przykładowa krawędź reprezentowana za pomocą półkrawędzi [20].

```
    def __init__(self):
        self.vertices = []
        self.halfedges = []
        self.faces = []
```

3.7.1. Półkrawędź

Każda półkrawędź zawiera referencję do poprzedniej i następnej półkrawędzi, dzięki którym można poruszać się po krawędziach otaczających daną komórkę diagramu. Krawędź diagramu często jednak oddziela dwie komórki, dlatego aby uzyskać dostęp do obu komórek (i ich krawędzi) każda krawędź diagramu Voronoi reprezentowana jest przez dwie półkrawędzie nazywane bliźniakami (ang. *twins*) (stąd nazwa półkrawędź), ułożonymi w ten sposób, że półkrawędź wskazywana przez rozważaną półkrawędź jako następna otacza komórkę w kierunku przeciwnym do ruchu wskazówek zegara (rys. 3.1). Każda półkrawędź zawiera również informację o wierzchołku na który wskazuje, jak i o wierzchołku z którego się zaczyna. Dodatkowo, półkrawędź zawiera także informację o punkcie, do którego należy otaczana przez nią komórka diagramu.

Listing 3.6. Klasa Halfedge z modułu dcel.

```
class Halfedge:

    def __init__(self, source=None, target=None, next=None,
            prev=None, face_point=None):
        self.source = source
        self.target = target
        self.next = next
        self.prev = prev
        self.face_point = face_point
        self.twin = None

    def __str__(self):
```


19:8555030979

```
        return "Halfedge({} -> {})".format(self.source, self.target)

    @staticmethod
    def create_twins():
        h1 = Halfedge()
        h2 = Halfedge()
        h1.twin = h2
        h2.twin = h1
        return h1, h2
```

3.7.2. Wierzchołek

Wierzchołek w podwójnie łączonej liście krawędzi zawiera informację o swoim położeniu na płaszczyźnie, oraz referencję do dowolnej półkrawiędzi, której jest początkiem.

Listing 3.7. Klasa Vertex z modułu dcel.

```
class Vertex:

    def __init__(self, point, halfedge=None):
        self.point = point
        self.halfedge = halfedge

    def __str__(self):
        return "Vertex({})".format(self.point)
```

3.7.3. Ściana

Ściana w podwójnie łączonej liście krawędzi zawiera informację o punkcie, do którego należy reprezentowana przez tą ścianę komórka diagramu, oraz referencję do dowolnej otaczającej ją półkrawiędzi.

Listing 3.8. Klasa Face z modułu dcel.

```
class Face:

    def __init__(self, halfedge, point):
        self.halfedge = halfedge
        self.point = point

    def __str__(self):
        return "Face({}, {})".format(self.halfedge, self.point)
```

3.8. Klasa Arc

Klasa Arc to struktura służąca do przechowywania danych o parabolach budujących linię brzegową (ang. *beach line*) w algorytmie Fortune'a. Struktura ta przechowuje informacje o punkcie, dla którego obliczana jest parabola, oraz referencję do zdarzenia kołowego w kolejce priorytetowej odpowiadającego tej paraboli. Obiekty klasy Arc są liśćmi w drzewie AVL. Klasa ta wykorzystywana jest w algorytmie Fortune'a.

20:5291141941

Listing 3.9. Klasa Arc z modułu arcs.

```
class Arc:

    def __init__(self, point, circle_event=None):
        self.point = point
        self.circle_event = circle_event

    def __str__(self):
        return "Arc({}, {})".format(self.point, self.circle_event)

    def __hash__(self):
        return hash((self.point.x, self.point.y))
```

3.9. Klasa Breakpoint

Klasa Breakpoint to struktura odpowiadająca węzłom wewnętrznym drzewa AVL. Przechowuje ona dwie informacje: dwuelementową krotkę zawierającą informację o punkcie przerwania na linii brzegowej, gdzie pierwszy element krotki zawiera informację o paraboli po lewej stronie, a drugi o paraboli po prawej stronie od tego punktu, oraz referencję do półkrawędzi z podwójnie łączonej listy krawędzi, odpowiadającej za krawędź diagramu Voronoi oddzielającą te dwa punkty. Klasa Breakpoint wykorzystywana jest w algorytmie Fortune'a.

Listing 3.10. Klasa Breakpoint z modułu breakpoints.

```
class Breakpoint:

    def __init__(self, pair, halfedge=None):
        self.data = pair
        self.halfedge = halfedge

    def __str__(self):
        return "Breakpoint({})".format(self.data)

    def __lt__(self, other):
        return self.data[0].x < other.data[0].x

    def __le__(self, other):
        return self.data[0].x <= other.data[0].x

    def __gt__(self, other):
        return self.data[0].x > other.data[0].x

    def __ge__(self, other):
        return self.data[0].x >= other.data[0].x

    def __eq__(self, other):
        return (self.data[0].x == other.data[0].x and
                self.data[0].y == other.data[0].y and
                self.data[1].x == other.data[1].x and
                self.data[1].y == other.data[1].y)

    def __ne__(self, other):
```


21:1609701277

```
        return not self == other

    def __hash__(self):
        return hash(self.data)
```

3.10. Zdarzenia

Zdarzenia to punkty na płaszczyźnie, w których zatrzymuje się miotła algorytmów wykorzystujących technikę zamiatania płaszczyzny. Zdarzenia przechowywane są w kolejce priorytetowej, dzięki czemu rozpatrywane są w odpowiedniej kolejności. W przypadku tej implementacji miotła porusza się po płaszczyźnie z góry na dół. W algorytmie Fortune'a wyróżniamy dwa typy zdarzeń: zdarzenia miejscowe (ang. *site event*) i zdarzenia kołowe (ang. *circle event*). Oba te typy reprezentuje klasa Event. Zdarzenia miejscowe występują w punktach, dla których obliczamy diagram Voronoi i przechowują one jedynie informację o punkcie ich wystąpienia. Zdarzenia kołowe występują w punktach znikania paraboli z linii brzegowej. Miejsca wystąpień tych zdarzeń nie są znane na początku pracy algorytmu i dodawane są do kolejki priorytetowej w momencie ich wykrycia. Zdarzenia tego typu przechowują więcej informacji: poza punktem ich wystąpienia zawierają także informację o punkcie zniknięcia paraboli, do której się odnoszą (punkt wystąpienia zdarzenia różni się od punktu zniknięcia paraboli), referencję do liścia drzewa AVL opisującego tą parabolę, a także flagę, opisującą czy zdarzenie to jest aktywne czy nie.

Struktura klasy Event została zaprojektowana tak, aby mogła być użyta w innych algorytmach stosujących technikę zamiatania płaszczyzny. Można rozszerzać listę atrybutów klasy, które odpowiadają typom zdarzeń, a także można dodać w konstruktorze kod tworzący nowe atrybuty instancji klasy Event potrzebne nowym typom zdarzeń.

Listing 3.11. Klasa Event z modułu events.

```
class Event:
    SITE = 7    # Fortune
    CIRCLE = 8    # Fortune

    def __init__(self, point, event_type, *sequence):
        self.point = point
        self.type = event_type
        if self.type == Event.SITE:
            pass
        elif self.type == Event.CIRCLE:
            self.center = sequence[0]
            self.leaf_pointer = sequence[1]
            self.is_valid = sequence[2]

    def __str__(self):
        return "Event({}, {})".format(self.point, self.type)

    # reverse comparisons because PriorityQueue is minimum priority queue,
    # while sweep line goes from top to bottom
```


22:6244745743

```
    def __lt__(self, other):
        return (-self.point.y, self.point.x) < (-other.point.y, other.point.x)

    def __le__(self, other):
        return (-self.point.y, self.point.x) <= (-other.point.y, other.point.x)

    def __gt__(self, other):
        return (-self.point.y, self.point.x) > (-other.point.y, other.point.x)

    def __ge__(self, other):
        return (-self.point.y, self.point.x) >= (-other.point.y, other.point.x)

    def __eq__(self, other):
        # equal check must distinguish events because priority queue
        # deletes duplicates
        if self.type == other.type:
            if self.type == Event.CIRCLE:
                return (self.point == other.point and
                        self.center == other.center and
                        self.leaf_pointer == other.leaf_pointer and
                        self.is_valid == other.is_valid)
            else: # Event.SITE
                return self.point == other.point
        else:
            return False

    def __ne__(self, other):
        return not self == other

    def __hash__(self):
        return hash((self.point.x, self.point.y, self.type))
```

3.11. Klasy VoronoiArea i VoronoiAreaCollection

Klasa VoronoiArea reprezentuje komórkę diagramu Voronoi za pomocą wielokąta, co pozwala na wykorzystywanie metod z klasy Polygon z pakietu planegeometry i posiada informację o punkcie, do którego należy ta komórka.

Klasa VoronoiAreaCollection przechowuje kolekcję obiektów klasy VoronoiArea i posiada metody ją obsługujące. Pozwala także na zapisanie kolekcji w postaci grafu z pakietu planegeometry.

Listing 3.12. Klasa VoronoiArea z modułu voronoi_area.

```
#!/usr/bin/python

from points import Point
from polygons import Polygon

class VoronoiArea:
    """The class defining a Voronoi area."""

    def __init__(self, point, polygon):
        if isinstance(point, Point) and isinstance(polygon, Polygon):
            self.voronoi_point = point
            self.area_polygon = polygon
```


23:6487720611

```
        else:
            raise ValueError("Bad argument type")

    def __repr__(self):
        """String representation of a voronoi area."""
        return "VoronoiArea({}, {})".format(
            self.voronoi_point, self.area_polygon)

    def __eq__(self, other):
        """Comparison of Voronoi areas (va1 == va2)."""
        return (self.voronoi_point == other.voronoi_point and
                self.area_polygon == other.area_polygon)

    def __ne__(self, other):
        """Comparison of Voronoi areas (va1 != va2)."""
        return not self == other

    def __hash__(self):
        return hash((self.voronoi_point, self.area_polygon))

    def copy(self):
        """Return a copy of a Voronoi area."""
        return VoronoiArea(self.voronoi_point, self.area_polygon)
```

Listing 3.13. Klasa VoronoiAreaCollection z modułu voronoi_area_collection.

```
#!/usr/bin/python

from points import Point
from graphs import Graph
from voronoi_area import VoronoiArea

class VoronoiAreaCollection:
    """The class defining a voronoi collection."""

    def __init__(self):
        """Make a collection of voronoi areas."""
        self.items = dict()

    def __str__(self):
        """String representation of a voronoi area collection."""
        return str(list(self.items.values()))

    def __len__(self):
        """Return the number of voronoi areas in the collection."""
        return len(self.items)

    def iterareas(self):
        """Generate voronoi areas on demand."""
        for voronoi_point in self.items:
            yield self.items[voronoi_point]

    def insert(self, voronoi_area):
        """Insert a voronoi to the collection."""
        if not isinstance(voronoi_area, VoronoiArea):
            raise ValueError("not a voronoi area")
        self.items[voronoi_area.voronoi_point] = voronoi_area
```


24:4407736652

```
    def remove(self, voronoi_area):
        """Remove a voronoi area from the collection."""
        if isinstance(voronoi_area, VoronoiArea):
            point = voronoi_area.voronoi_point
        elif isinstance(voronoi_area, Point):
            point = voronoi_area
        else:
            raise ValueError("not a voronoi area and not a point")
        del self.items[point]

    def search(self, point):
        """Finding voronoi areas containing a point."""
        result = []
        for voronoi_point in self.items:
            if point in self.items[voronoi_point].area_polygon:
                result.append(self.items[voronoi_point])
        return result

    def get_area(self, voronoi_point):
        """Finding the Voronoi area for a Voronoi point."""
        return self.items[voronoi_point]    # O(1) time

    def __contains__(self, other):
        """Test if a voronoi area is in a collection."""
        if isinstance(other, VoronoiArea):
            return other.voronoi_point in self.items
        else:
            raise ValueError("not a voronoi area")

    def to_graph(self):
        """Return the voronoi diagram as a graph."""
        graph = Graph()
        for voronoi_point in self.items:
            for segment in self.items[voronoi_point].area_polygon.itersegments():
                if not graph.has_edge(segment):
                    graph.add_edge(segment)
        return graph
```

3.12. Przykładowe użycie algorytmów

Przykładowa sesja interaktywna pokazuje zastosowanie przygotowanych algorytmów. Diagram Voronoi w postaci grafu lub listy odcinków można narysować używając biblioteki Matplotlib. Przykłady znajdują się w plikach źródłowych.

```
>>> from points import Point
>>> from fortune import Fortune
>>> from voronoi_from_delaunay1 import VoronoiFromDelaunay1
>>> from voronoi_from_delaunay2 import VoronoiFromDelaunay2

# Utworzenie listy punktow.
>>> point_list = [Point(3, 4), Point(-2, 3), Point(7, 7)]

# Uzycie algorytmu Fortune'a.
```


25:1051751476

```
>>> algorithm = Fortune(point_list)
>>> algorithm.run()
>>> G = algorithm.make_voronoi_area_collection().to_graph()
>>> G.show()
...

# Uzycie algorytmu przeksztalcajacego triangulacje Delaunaya w diagram Voronoi
# (wersja z lista odcinkow)
>>> point_list = [Point(3, 4), Point(-2, 3), Point(7, 7)]
>>> algorithm = VoronoiFromDelaunay1(point_list)
>>> algorithm.run()
>>> print(algorithm.segment_list)
[ ... ]

# Uzycie algorytmu przeksztalcajacego triangulacje Delaunaya w diagram Voronoi
# (wersja z VoronoiAreaCollection)
>>> point_list = [Point(3, 4), Point(-2, 3), Point(7, 7)]
>>> algorithm = VoronoiFromDelaunay2(point_list)
>>> algorithm.run()
>>> G = algorithm.vac.to_graph()
>>> G.show()
...
```

4. Algorytmy

W tym rozdziale przedstawione zostaną algorytmy obliczające diagramy Voronoi.

4.1. Algorytm Fortune'a

Algorytm Fortune'a to algorytm obliczający diagram Voronoi dla podanych punktów, wykorzystujący technikę zamiatania płaszczyzny (ang. *sweep line technique*). Technika ta polega na wykorzystaniu prostej (nazywanej miotłą) przesuwającej się w jakimś określonym kierunku (na przykład z góry na dół) po zbiorze punktów i przetwarzającej zdobyte w ten sposób informacje.

Dane wejściowe: Zbiór punktów P.

Problem: Obliczenie diagramu Voronoi dla podanych danych wejściowych.

Opis algorytmu: Algorytm wykorzystuje kilka istotnych struktur danych, bez których uzyskanie złożoności czasowej $O(n \log n)$ nie byłoby możliwe.

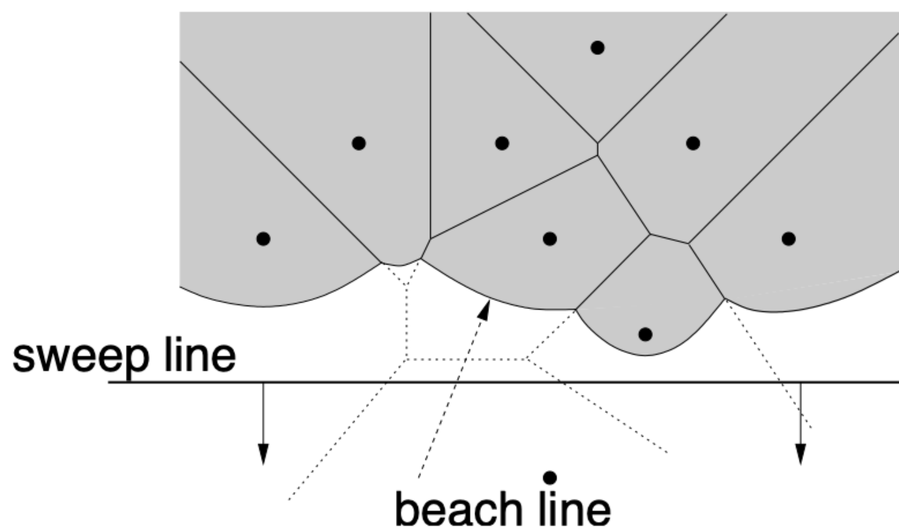
Pierwszą z nich jest kolejka priorytetowa przechowująca informacje o uporządkowanych zdarzeniach (zarówno o zdarzeniach miejscowych jak i o znanych już zdarzeniach kołowych), które nie zostały jeszcze rozpatrzone. Na początku działania algorytmu kolejka priorytetowa wypełniana jest wszystkimi zdarzeniami miejscowymi, czyli punktami podanymi jako dane wejściowe, a podczas pracy algorytmu dodawane do niej są zdarzenia kołowe. Po rozparzeniu zdarzenia zostaje ono usunięte z kolejki priorytetowej. Algorytm kończy działanie w momencie, gdy kolejka priorytetowa stanie się pusta.

Drugą strukturą danych wykorzystaną w algorytmie jest drzewo AVL, które służy do przechowywania informacji o linii brzegowej (rys. 4.1).

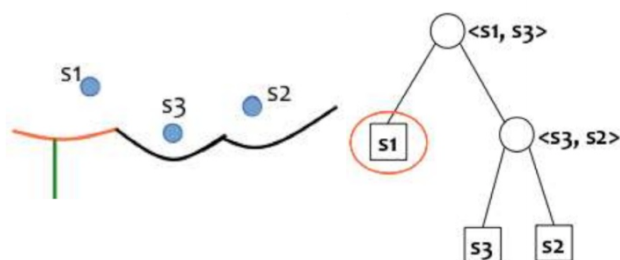
Liście drzewa odpowiadają łukom parabolicznym, uporządkowanym od lewej strony w taki sposób, że skrajnie lewy liść drzewa odpowiada najbardziej lewemu łukowi na płaszczyźnie (rys. 4.2).

Każdy liść zawiera informacje o punkcie, którego łuk jest przez niego reprezentowany. Wewnętrzne węzły drzewa (nie będące liśćmi) zawierają informację o punktach przerwania, reprezentowanych za pomocą krotek zwierających dwa punkty: pierwszy element krotki zawiera informację o paraboli po lewej stronie, a drugi o paraboli po prawej stronie od punktu przerwania. W praktyce nie przechowujemy w drzewie w jawny sposób informacji o parabolach, a jedynie punkty przerwań w węzłach wewnętrznych i punkty, które reprezentują łuk w liściach. Taka implementacja linii brzegowej pozwala odnaleźć łuk, który zostanie rozdzielony na dwa przy rozpatrywaniu nowego zdarzenia miejscowego w czasie $O(\log n)$.

27:7871370318



Rysunek 4.1. Przykładowa linia brzegowa [21].



Rysunek 4.2. Na pomarańczowo łuk w linii brzegowej i jego pozycja w drzewie [22].

28:3832838972

Dodatkowo, każdy liść zawiera referencje do zdarzenia kołowego z kolejki priorytetowej (lub None, jeżeli zdarzenie takie nie zostało jeszcze wykryte), w wyniku którego łuk, który jest przez ten liść reprezentowany zniknie. Referencja ta jest potrzebna, ponieważ niektóre zdarzenia kołowe mogą być "fałszywymi alarmami" i należy je zignorować. Ponadto, każdy wewnętrzny węzeł drzewa zawiera referencje do krawędzi z podwójnie łączonej listy krawędzi.

Ostatnią używaną strukturą danych jest podwójnie łączona lista krawędzi, która przechowuję informację o już obliczonych fragmentach diagramu Voronoi. Węzły wewnętrzne drzewa AVL przechowują referencję do odpowiadających im krawędzi, możemy więc szybko pozyskać z niej potrzebne informacje.

Główną częścią algorytmu jest przetwarzanie dwóch rodzajów zdarzeń: zdarzeń miejscowych i zdarzeń kołowych. Jako, że algorytm Fortune'a jest algorytmem wykorzystującym technikę zamiatania płaszczyzny, przesuwamy miotłę po płaszczyźnie z góry na dół natrafiając na interesujące nas zdarzenia. Zdarzenia miejscowe są znane na początku pracy algorytmu: są to punkty podane jako dane wejściowe. Odpowiadają one miejscom dodawania nowych paraboli do linii brzegowej (rys. 4.3).

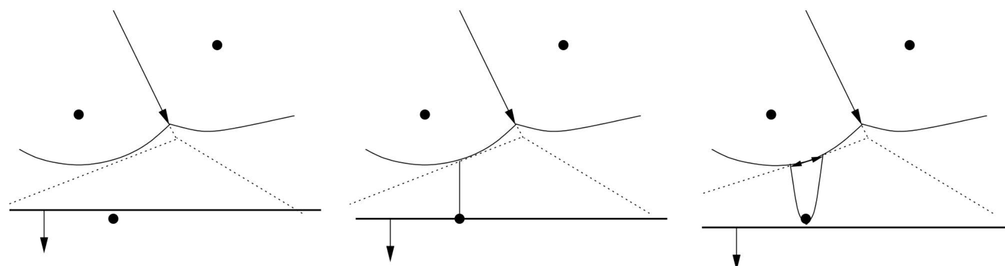
W momencie natrafienia przez miotłę na zdarzenie miejscowe następuje wyszukanie w drzewie AVL paraboli, która zostanie rozdzielona przez nową parabolę odpowiadającą rozpatrywanemu zdarzeniu miejscowemu, a następnie dodanie informacji o nowej paraboli do drzewa, po czym, jeżeli jest to konieczne, należy zrównoważyć drzewo. Informację o nowej paraboli dodajemy do drzewa w następujący sposób: tworzymy nowe drzewo, którego korzeniem jest węzeł wewnętrzny zawierający krotkę, której pierwszym elementem jest punkt odpowiadający za starą (rozdzielaną) parabole, a drugim elementem jest punkt odpowiadający za nową parabolę (aktualnie rozważany punkt zdarzenia miejscowego). Lewym dzieckiem tego korzenia ustalamy lewą część rozdzielonego łuku, a prawym kolejny węzeł wewnętrzny zawierający krotkę, której elementy są w odwrotnej kolejności niż poprzednio (pierwszym elementem jest punkt odpowiadający za nową parabolę, a drugim elementem jest punkt odpowiadający za starą parabolę). Lewym dzieckiem tego węzła wewnętrznego ustalamy nową parabolę, a prawym prawą część rozdzielanej paraboli. Tak utworzonym poddrzewem zastępujemy odnaleziony poprzednio w drzewie liść odpowiadający za rozdzielaną parabolę.

Jeżeli parabola, która została rozdzielona, zawierała referencję do zdarzenia kołowego znajdującego się w kolejce priorytetowej, zdarzenie to jest "fałszywym alarmem" i trzeba je dezaktywować.

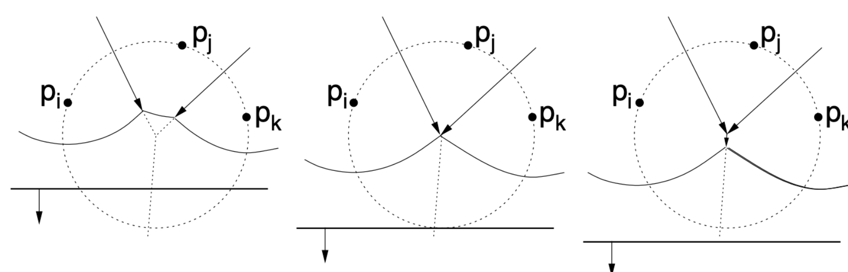
Po utworzeniu nowej paraboli i dodaniu informacji o niej do drzewa AVL, należy uaktualnić podwójnie łączoną listę krawędzi. Do listy półkrawędzi dodajemy dwie nowe półkrawędzie, a referencję do nich przypisujemy odpowiednio do utworzonych poprzednio węzłów wewnętrznych drzewa.

Ostatnim krokiem wykonywanym podczas rozpatrywania zdarzenia miejscowego jest sprawdzenie, czy po dodaniu nowej paraboli nie jesteśmy w stanie wykryć wystąpienia nowego zdarzenia kołowego. Aby tego dokonać wybieramy z drzewa trójkę sąsiadujących łuków. Należy sprawdzić dwie możliwości: trójkę, gdzie nowo dodana parabola leży na płaszczyźnie najbardziej

29:1874701127



Rysunek 4.3. Wizualizacja dodawania nowego łuku do linii brzegowej w momencie natrafienia przez miotłę na zdarzenie miejscowe [21].



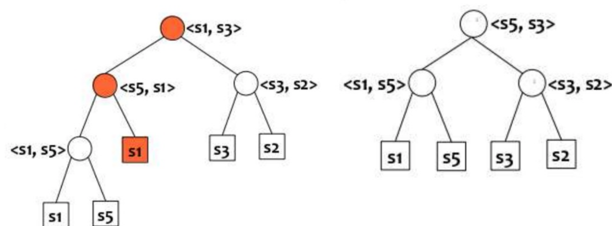
Rysunek 4.4. Wizualizacja znikania łuku z linii brzegowej w momencie natrafienia przez miotłę na aktywne zdarzenie kołowe [21].

na lewo i trójkę, gdzie nowo dodana parabola leży na płaszczyźnie najbardziej na prawo. Dla obu możliwości sprawdzamy, czy punkty odpowiadające tym parabolom są zorientowane przeciwnie z ruchem wskazówek zegara. Jeżeli nie, kończymy rozważanie podanych punktów, ponieważ nie tworzą one zdarzenia kołowego. Jeżeli tak, obliczamy środek okręgu utworzonego przez te trzy punkty. Jest to punkt, w którym zakończy się krawędź oddzielająca poszczególne komórki diagramu (jeżeli nie okaże się, że zdarzenie to jest "fałszywym alarmem"). Obliczamy również punkt leżący na okręgu o minimalnej wartości y, ponieważ jest to punkt wystąpienia zdarzenia kołowego. Na koniec dodajemy do kolejki priorytetowej zdarzenie kołowe występujące w obliczonym punkcie, które zawiera również informację o wyliczonym centrum i referencję do liścia w drzewie odpowiadającego za parabolę środkowego z rozważanych punktów. To samo robimy w drugą stronę, czyli dodajemy referencję do zdarzenia kołowego z kolejki priorytetowej liściowi w drzewie, czego używamy przy wykrywaniu "fałszywych alarmów".

Drugim rodzajem zdarzeń przetwarzanych przez algorytm są zdarzenia kołowe. W przeciwieństwie do zdarzeń miejscowych, punkty ich wystąpień nie są znane od początku, dlatego trudnością algorytmu jest ich przewidywanie, ponieważ w momencie wystąpienia zdarzenia kołowego miotła przekroczyła już punkt oznaczający koniec (lub początek) krawędzi oddzielającej komórki diagramu Voronoi (rys. 4.4).

W momencie natrafienia przez miotłę na aktywne zdarzenie kołowe, łuk

30:3268289061



Rysunek 4.5. Drzewo przed i po usunięciu przykładowego łuku [22].

do którego jest ono przypisane znika i należy go usunąć z drzewa AVL. Aby tego dokonać, usuwamy odpowiedni liść odpowiadający tej paraboli z drzewa (mamy łatwy dostęp do tej paraboli, ponieważ rozpatrywane zdarzenie kołowe przechowuje referencję do liścia), a następnie odpowiednio uaktualniamy węzły wewnętrzne drzewa (rys. 4.5). Gdy zachodzi potrzeba, po tej operacji należy zrównoważyć drzewo.

Po usunięciu paraboli z drzewa, należy uznać wszystkie pozostałe biorące ją pod uwagę zdarzenia kołowe za "fałszywe alarmy" i dezaktywować je. Są to zdarzenia sąsiednich łuków.

W następnym kroku należy utworzyć nowy wierzchołek i dodać go do listy wierzchołków w podwójnie łączonej liście krawędzi. Wierzchołek ten tworzymy w punkcie przechowywanym w zdarzeniu kołowym będącym środkiem okręgu tego zdarzenia. Tworzymy także nowe półkrawędzie, odpowiadające za nową krawędź oddzielającą komórki diagramu, a następnie uaktualniamy referencję odpowiednich półkrawędzi do poprzednich i następnych półkrawędzi w podwójnie łączonej liście krawędzi.

Na koniec, należy sprawdzić, czy po usunięciu paraboli, sąsiadujące teraz ze sobą parabole nie tworzą nowego zdarzenia kołowego. Robimy to dokładnie w taki sam sposób, jak w przypadku sprawdzania wystąpienia zdarzenia kołowego przy rozpatrywaniu zdarzania miejscowego, sprawdzając również dwa przypadki: gdy lewy sąsiad usuniętej paraboli jest środkowym łukiem trzech sąsiadujących łuków i gdy prawy sąsiad usuniętej paraboli jest środkowym łukiem trzech sąsiadujących łuków.

Główna część algorytmu działa w następujący sposób: tworzymy zdarzenia miejscowe dla każdego punktu podanego jako dane wejściowe i dodajemy je do pustej kolejki priorytetowej. Dopóki kolejka nie jest pusta, pobieramy z niej element i w zależności, czy jest to zdarzenie miejscowe czy kołowe wykonujemy odpowiednie operacje. W momencie rozpatrzenia wszystkich zdarzeń (gdy kolejka jest pusta), otrzymujemy gotowy diagram Voronoi. Pozostaje jedynie zakończenie prostych lub półprostych zmierzających do nieskończoności, ponieważ nie jesteśmy w stanie ich reprezentować. Dostęp do nich możemy uzyskać poprzez węzły wewnętrzne drzewa AVL, które zostały w nim po zakończeniu pracy algorytmu. Następnie zamykamy diagram w ramce ograniczającej lub przedłużamy krawędzie zmierzające do nieskończoności do odpowiedniej długości imitującej nieskończoność, a następnie łączymy z sąsiadującymi krawędziami nieskończonymi. Na koniec, dodajemy do listy ścian w podwójnie łączonej liście krawędzi każdą komórkę

31:7638630148

diagramu, dokonując tego poprzez dodanie do listy referencji do dowolnej z krawędzi otaczającej tą komórkę.

W kodzie źródłowym zostały wykorzystane metody utworzone w oparciu o źródła: [23], [24] i [25].

Złożoność: Złożoność czasowa algorytmu wynosi $O(n \log n)$.

Uwaga 1: Algorytm Fortune'a jest optymalnym algorytmem obliczania diagramów Voronoi pod względem złożoności czasowej, ponieważ problem posortowania n liczb rzeczywistych sprowadza się do problemu obliczania diagramów Voronoi, co oznacza, że każdy algorytm obliczający diagramy Voronoi musi w najlepszym przypadku zająć $\Omega(n \log n)$ czasu [9].

Uwaga 2: Zdarzenia miejscowe i kołowe muszą być od siebie odróżnialne, inaczej kolejka priorytetowa nie będzie w stanie przechowywać obu zdarzeń jednocześnie, gdy występują one w tym samym punkcie.

Uwaga 3: Jeżeli na początku pracy algorytm natrafi na dwa (lub więcej) współliniowe względem osi, po której porusza się miotła punkty, występuje szczególny przypadek algorytmu, ponieważ parabole odnoszące się do tych punktów są w tym momencie równoległymi prostymi, co oznacza, że nigdy się nie przetną. Problem ten jest rozwiązywany szczególnym kodem.

Uwaga 4: Szczególny przypadek algorytmu występuje również wówczas, gdy dane wejściowe składają się z dwóch punktów, lub gdy wszystkie punkty w danych wejściowych są współliniowe. Należy wówczas w inny sposób połączyć krawędzie symulujące nieskończoność.

4.2. Diagram Voronoi z triangulacji Delaunaya

Diagram Voronoi może być utworzony na bazie triangulacji Delaunaya. Do wyznaczenia triangulacji Delaunaya został wykorzystany algorytm Bowyera-Watsona, który jest dostępny w bibliotece planegeometry. Transformacja triangulacji Delaunaya w diagram Voronoi jest możliwa, ponieważ graf dualny triangulacji odpowiada diagramowi Voronoi dla tego samego zbioru punktów. Istnieje więc zależność, którą można wykorzystać: środki okręgów opisanych na trójkątach triangulacji odpowiadają wierzchołkom diagramu Voronoi. Krawędzie między tymi wierzchołkami uzyskujemy łącząc je zgodnie z sąsiedztwem trójkątów w triangulacji.

Dane wejściowe: Zbiór punktów P.

Problem: Obliczenie triangulacji Delaunaya, a następnie przekształcenie jej w diagram Voronoi dla podanych danych wejściowych.

Opis algorytmu: Na początku pracy algorytm dodaje do zbioru punktów wejściowych cztery punkty tworzące kwadrat zawierający w sobie wszystkie pozostałe punkty i generuje dla tych punktów triangulacje Delaunaya,

32:1050502612

wykorzystując w tym celu algorytm Bowyera-Watsona. Triangulacja ta zapisywana jest w postaci grafu. Dla każdego wierzchołka w tym grafie (pomijając wierzchołki budujące ograniczający kwadrat) tworzymy listę środków okręgów i listę sąsiadujących z nim wierzchołków, a następnie sprawdzamy każdy możliwy trójkąt zbudowany z wierzchołka rozważanego i dwóch sąsiadujących z nim wierzchołków. Jeżeli w grafie istnieje krawędź pomiędzy dwoma aktualnie rozważanymi wierzchołkami z listy sąsiadów i jeżeli tak utworzony trójkąt nie zawiera w sobie żadnego innego wierzchołka z listy sąsiadów, obliczamy środek okręgu opisanego na tym trójkącie i dodajemy go do listy środków okręgów.

Po rozpatrzeniu wszystkich możliwych trójkątów, lista środków okręgów zawiera punkty tworzące wielokąt ograniczający komórkę diagramu Voronoi należącą do punktu reprezentowanego przez rozważany wierzchołek. Po powtórzeniu tych operacji dla każdego wierzchołka ze zbioru danych wejściowych otrzymujemy cały diagram Voronoi.

Złożoność: Złożoność obliczeniowa użytego algorytmu Bowyera-Watsona wynosi $O(n^2)$, więc końcowa złożoność czasowa algorytmu również wynosi $O(n^2)$.

Uwaga: Krawędzie zmierzające do nieskończoności nie są w rzeczywistości reprezentowane jako nieskończone, lecz mają swój koniec w pewnym punkcie. Należy więc dobrać wystarczająco duży kwadrat ograniczający punkty podane jako dane wejściowe, aby uniknąć ewentualnego błędu związanego z wyjściem poza granicę diagramu.

Listing 4.1. Klasa VoronoiFromDelaunay2 z modułu voronoi_from_delaunay2.

```
try:
    integer_types = (int, long)
except NameError:  # Python 3
    integer_types = (int,)
    xrange = range

from points import Point
from triangles import Triangle
from polygons import Polygon
from segments import Segment
from bowyerwatson import BowyerWatson
from quickhull import QuickHull
from voronoi_area import VoronoiArea
from voronoi_area_collection import VoronoiAreaCollection

class VoronoiFromDelaunay2:
    """Algorithm for creating polygons containing Voronoi area
    of Voronoi diagram from Delaunay triangulation."""

    def __init__(self, point_list):
        self.point_list = point_list
        self.vac = VoronoiAreaCollection()
        self.infinity = 0

    def insert_big_square(self):
```


33:1206833213

```
        big = max(max(abs(point.x), abs(point.y))
            for point in self.point_list)
        m = 20
        self.infinity = m * big
        self.point_list.append(Point(m * big, m * big))
        self.point_list.append(Point(-m * big, m * big))
        self.point_list.append(Point(-m * big, -m * big))
        self.point_list.append(Point(m * big, -m * big))

    # checks if any other vertex is inside triangle
    def _contains_other_vertex(self, triangle, vertices, i, j):
        for k in range(len(vertices)):
            if k == i or k == j:
                continue
            if vertices[k] in triangle:
                return True
        return False

    def run(self):
        # add points in infinity to create infinite edges
        self.insert_big_square()

        algorithm = BowyerWatson(self.point_list)
        algorithm.run()

        triangulation_graph = algorithm.tc.to_graph()
        for vertex in triangulation_graph.iternodes():
            # no need to calculate for artificial infinity points
            if vertex.x == self.infinity or vertex.x == -self.infinity:
                continue
            circumcircles = []
            vertices = list(triangulation_graph.iteradjacent(vertex))
            for i in range(len(vertices) - 1):
                for j in range(i + 1, len(vertices)):
                    # check if points generate a proper triangle
                    if triangulation_graph.has_edge(Segment(
                        vertices[i], vertices[j])):
                        triangle = Triangle(vertex, vertices[i], vertices[j])
                        # if other vertex is inside triangle then it's not a
                        # proper triangle
                        if not self._contains_other_vertex(
                            triangle, vertices, i, j):
                            circumcircles.append(triangle.circumcenter())

            # create proper polygon from point list
            hull = QuickHull(circumcircles)
            hull.run()
            polygon_points = hull.convex_hull

            self.vac.insert(VoronoiArea(vertex, Polygon(*polygon_points)))
```


34:4414572610

4.3. Diagram Voronoi z triangulacji Delaunaya w postaci listy odcinków

Dane wejściowe: Zbiór punktów P.

Problem: Obliczenie triangulacji Delaunaya, a następnie przekształcenie jej w diagram Voronoi dla podanych danych wejściowych.

Opis algorytmu: Jeżeli interesuje nas jedynie wykres diagramu Voronoi i nie potrzebujemy poza nim żadnych informacji o obliczonym diagramie, algorytm transformacji triangulacji Delaunaya w diagram Voronoi jest dużo prostszy. Na początku, podobnie jak w poprzednim algorytmie, program dodaje do zbioru punktów wejściowych cztery punkty tworzące kwadrat zawierający w sobie wszystkie pozostałe punkty i generuje dla tych punktów triangulacje Delaunaya wykorzystując w tym celu algorytm Bowyera-Watsona. Dla każdego trójkąta w triangulacji należy obliczyć środek opisanego na nim okręgu, a następnie połączyć te punkty odcinkami dla sąsiadujących ze sobą trójkątów. Otrzymujemy w ten sposób listę odcinków, którą można następnie wykorzystać do wygenerowania wykresu.

Złożoność: Złożoność obliczeniowa użytego algorytmu Bowyera-Watsona wynosi $O(n^2)$, więc końcowa złożoność czasowa algorytmu również wynosi $O(n^2)$.

Uwaga: Krawędzie zmierzające do nieskończoności nie są w rzeczywistości reprezentowane jako nieskończone, lecz mają swój koniec w pewnym punkcie. Należy więc dobrać wystarczająco duży kwadrat ograniczający punkty podane jako dane wejściowe aby uniknąć ewentualnego błędu związanego z wyjściem poza granicę diagramu.

Listing 4.2. Klasa VoronoiFromDelaunay1 z modułu voronoi_from_delaunay1.

```
try:
    integer_types = (int, long)
except NameError:   # Python 3
    integer_types = (int,)
    xrange = range

import itertools
from points import Point
from triangles import Triangle
from segments import Segment
from bowyerwatson import BowyerWatson

class VoronoiFromDelaunay1:
    """Algorithm for creating segments of Voronoi diagram
    from Delaunay triangulation."""

    def __init__(self, point_list):
        self.point_list = point_list
        self.segment_list = []
```


35:5195438330

```python
def insert_big_square(self):
    big = max(max(abs(point.x), abs(point.y))
        for point in self.point_list)
    m = 20
    self.point_list.append(Point(m * big, m * big))
    self.point_list.append(Point(-m * big, m * big))
    self.point_list.append(Point(-m * big, -m * big))
    self.point_list.append(Point(m * big, -m * big))

def run(self):
    # add points in infinity to create infinite edges
    self.insert_big_square()

    algorithm = BowyerWatson(self.point_list)
    algorithm.run()

    # for each triangle in triangulation, connect it's circumcircle
    # point with it's neighbours circumcircle points
    for (t1, t2) in itertools.combinations(algorithm.tc.items, 2):
        if t1.is_neighbor(t2):
            p1 = t1.circumcenter()
            p2 = t2.circumcenter()
            if p1 != p2:
                self.segment_list.append(Segment(p1, p2))
```

5. Podsumowanie

Celem tej pracy było przygotowanie implementacji w języku Python wybranych algorytmów obliczających diagram Voronoi dla podanego zbioru n punktów na płaszczyźnie. Opracowano algorytmy wykorzystujące dwie metody: bezpośrednią i pośrednią.

Algorytmem bezpośrednim jest algorytm Fortune'a, stosujący technikę zamiatania płaszczyzny, opublikowany w 1986 roku przez Stevena Fortune. Algorytm jest uważany za optymalny, a jego czas działania jest rzędu $O(n \log n)$. Do osiągnięcia teoretycznej złożoności obliczeniowej wymagane jest użycie kilku specjalistycznych struktur danych, które zaczerpnięto z pakietu planegeometry. Z drugiej strony, niniejsza praca przyczyniła się do rozwoju tego pakietu. W implementacji wykorzystuje się struktury danych zbliżone do tych, które stosuje się dla grafów planarnych. Podwójnie połączona lista krawędzi pozwala obiegać granice obszaru Voronoi, tak jak ściany w grafie planarnym.

Algorytm pośredni polega na przekształceniu triangulacji Delaunaya w diagram Voronoi. Tutaj również przydatny okazał się pakiet planegeometry, zawierający implementację algorytmu Bowyera-Watsona do wyznaczania triangulacji Delaynaya. Dostępna implementacja algorytmu Bowyera-Watsona działa w czasie $O(n^2)$, co wpływa na złożoność wyznaczania diagramu Voronoi. W literaturze znane są trudniejsze wersje algorytmu Bowyera-Watsona, które mają złożoność $O(n \log n)$. Użycie takiej wersji algorytmu Bowyera-Watsona prowadziłoby do całkowitej złożoności obliczeniowej metody pośredniej takiej jak dla algorymu Fortune'a.

Wszystkie algorytmy zostały skonstruowane zgodnie z filozofią pakietu planegeometry, gdzie dla danych wejściowych w postaci liczb typu **int** lub Fraction obliczenia prowadzone są dokładnie. Jeżeli dane wejściowe zawierają liczby typu **float**, wtedy wyniki też będą typu **float**, co może prowadzić do pojawienia się błędów zaokrągleń. Wyjątkiem jest algorytm Fortune'a, gdzie wyniki końcowe muszą być typu **float**.

Praca zawiera również testy sprawdzające praktyczną złożoność obliczeniową każdego z zaimplementowanych algorytmów obliczających diagramy Voronoi.

37:3466509835

A. Testy algorytmów

W dodatku tym przedstawione zostaną wyniki testów wydajnościowych omówionych algorytmów w celu potwierdzenia ich teoretycznych złożoności obliczeniowych. Algorytmy zostały przetestowane na zbiorach losowych puntów.

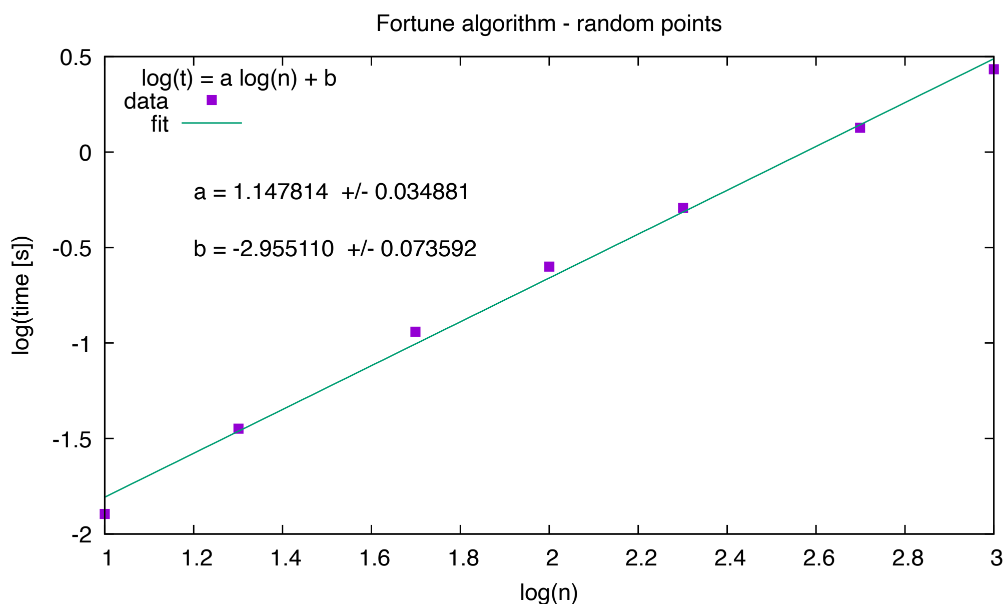
A.1. Testy algorytmu Fortune'a

Testy implementacji algorytmu Fortune'a potwierdzają, że jego praktyczna złożoność obliczeniowa zgadza się z teoretyczną $O(n \log n)$. Dowodzą tego wykresy A.1, A.2 i A.3.

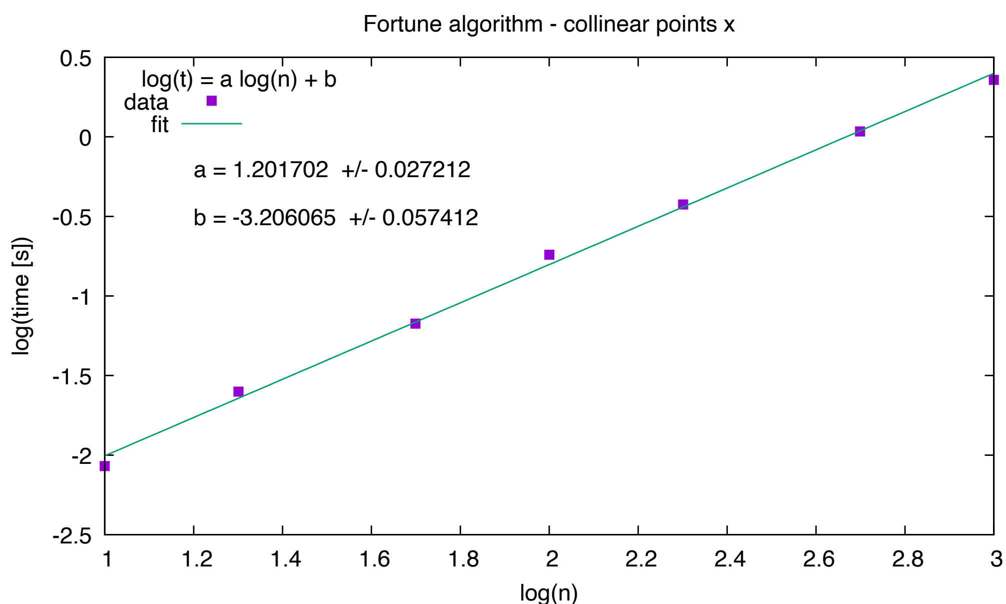
A.2. Testy wyznaczania diagramu Voronoi z triangulacji Delaunaya

Złożoność obliczeniowa algorytmów przekształcających triangulację Delaunaya na diagramy Voronoi zależy od złożoności algorytmu wyznaczającego triangulację. Używana implementacja algorytmu Bowyera-Watsona działa w czasie $O(n^2)$, więc niemożliwe jest osiągnięcie lepszej złożoności. Testy algorytmów przekształcających triangulację Delaunaya w diagramy Voronoi potwierdzają złożoność obliczeniową wynoszącą $O(n^2)$. Dowodzą tego wykresy A.4 i A.5.

38:7107885377

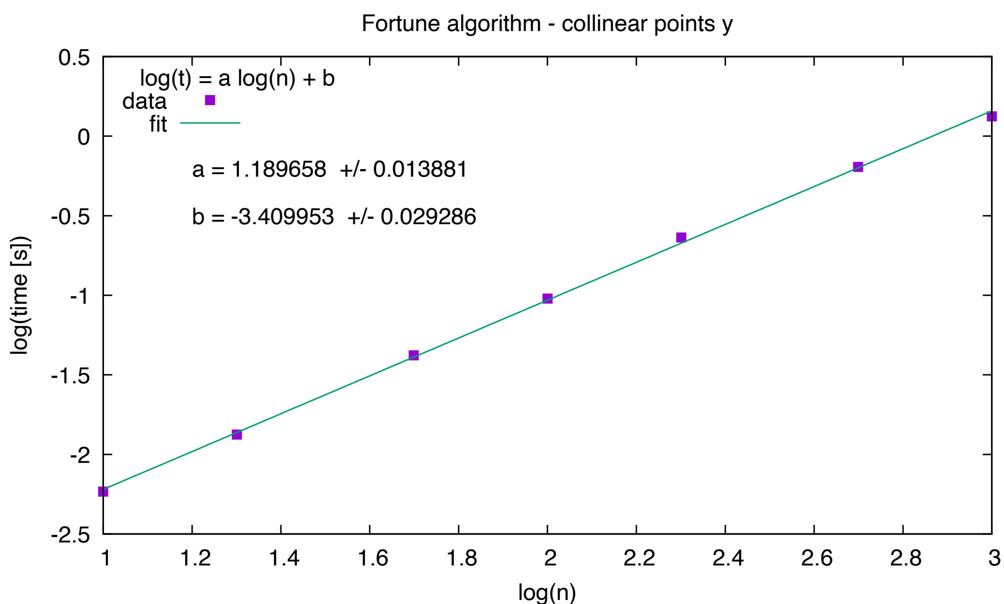


Rysunek A.1. Wykres wydajności algorytmu Fortune'a dla losowych punktów. Współczynnik $a = 1.148(34)$ potwierdza, że praktyczna złożoność obliczeniowa implementacji to $O(n \log n)$.

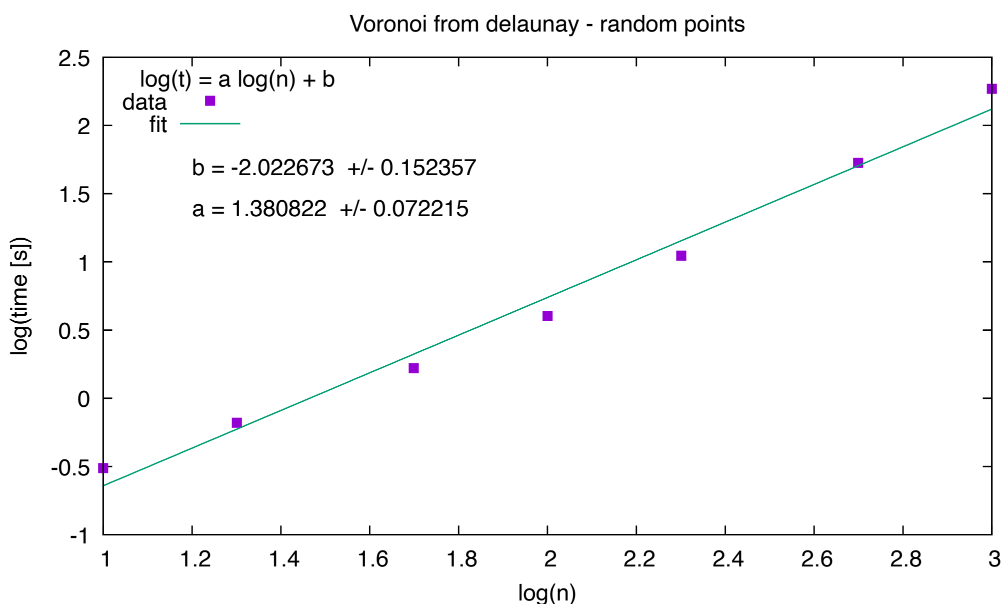


Rysunek A.2. Wykres wydajności algorytmu Fortune'a dla punktów współliniowych względem osi X. Współczynnik $a = 1.202(27)$ potwierdza, że praktyczna złożoność obliczeniowa implementacji to $O(n \log n)$.

39:9081170642

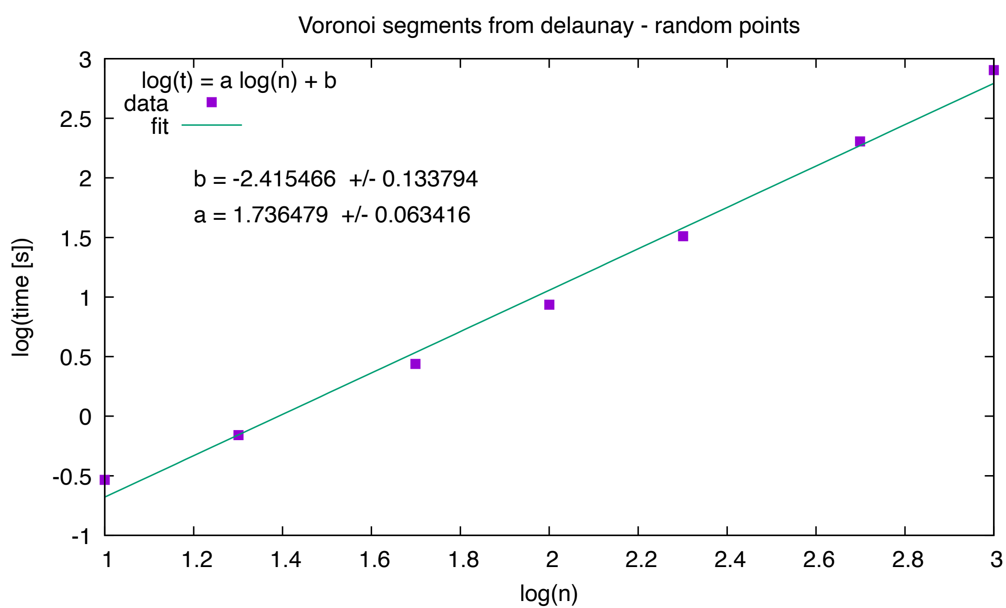


Rysunek A.3. Wykres wydajności algorytmu Fortune'a dla punktów współliniowych względem osi Y. Współczynnik $a = 1.190(14)$ potwierdza, że praktyczna złożoność obliczeniowa implementacji to $O(n \log n)$.



Rysunek A.4. Wykres wydajności algorytmu przekształcającego triangulację Delaunaya w diagram Voronoi dla losowych punktów. Współczynnik $a = 1.381(72)$ potwierdza, że praktyczna złożoność obliczeniowa implementacji to $O(n^2)$.

40:7051972074



Rysunek A.5. Wykres wydajności algorytmu przekształcającego triangulację Delaunaya w listę odcinków, za pomocą której można utworzyć wykres diagramu Voronoi dla losowych punktów. Współczynnik $a = 1.736(63)$ potwierdza, że praktyczna złożoność obliczeniowa implementacji to $O(n^2)$.

41:2950101757

Bibliografia

[1] Wikipedia, Voronoi diagram, 2021,
https://en.wikipedia.org/wiki/Voronoi_diagram.

[2] Wojciech Kisiała, Magdalena Rudkiewicz, *Zastosowanie diagramu Woronoja w badaniu przestrzennych wzorców rozmieszczenia i dostępności sklepów dyskontowych*, Katedra Ekonomiki Przestrzennej i Środowiskowej, Uniwersytet Ekonomiczny w Poznaniu, 2017,
https://rcin.org.pl/igipz/Content/63054/WA51_82399_r2017-t89-z2_Przeg-Geogr-Kisiala.pdf

[3] David Austin, *Voronoi Diagrams and a Day at the Beach*, Grand Valley State University, 2006,
https://www.ams.org/publicoutreach/feature-column/fcarc-voronoi

[4] Python Programming Language - Official Website,
https://www.python.org/.

[5] Andrzej Kapanowski, planegeometry, GitHub repository, 2021,
https://github.com/ufkapano/planegeometry/.

[6] Marcin Permus, *Algorytmy geometryczne w języku Python*, Uniwersytet Jagielloński, Kraków 2018.

[7] Wojciech Chrobak, *Technika zamiatania płaszczyzny*, Uniwersytet Jagielloński, Kraków 2019.

[8] Monika Wiech, *Triangulacja Delaunaya w geometrii obliczeniowej*, Uniwersytet Jagielloński, Kraków 2020.

[9] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, *Geometria obliczeniowa. Algorytmy i zastosowania*, WNT, Warszawa 2007.

[10] Franco P. Preparata, Michael Ian Shamos, *Geometria obliczeniowa. Wprowadzenie*, Helion, Gliwice 2003.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Wprowadzenie do algorytmow*, Wydawnictwo Naukowe PWN, Warszawa 2012.

[12] J. O'Rourke, *Computational Geometry In C 2nd ed.*, Cambridge University Press, Cambridge 1998.

[13] Wikipedia, Triangulacja Delone, 2021,
https://pl.wikipedia.org/wiki/Triangulacja_Delone.

[14] Wikipedia, Delaunay triangulation, 2021,
https://en.wikipedia.org/wiki/Delaunay_triangulation.

[15] Wikipedia, Convex hull, 2021,
https://en.wikipedia.org/wiki/Convex_hull.

[16] Wikipedia, Priority queue, 2021,
https://en.wikipedia.org/wiki/Priority_queue.

[17] Wikipedia, AVL Tree, 2021,
https://en.wikipedia.org/wiki/AVL_tree.

[18] Rosetta Code, *AVL Tree*, 2019,
https://rosettacode.org/wiki/AVL_tree#Python

[19] Tyler Sandman, *PyBST*, 2019,
https://github.com/TylerSandman/py-bst

42:8484972294

[20] Bernd Gärtner, Michael Hoffmann, *Plane Graphs and the DCEL*, Institute of Theoretical Computer Science, 2012,
https://www.ti.inf.ethz.ch/ew/courses/CG12/lecture/Chapter%205.pdf

[21] http://people.math.gatech.edu/~randall/Algs07/mount.pdf

[22] https://www.slideshare.net/KhalidFaisal2/2-voronoi-diagram-construction

[23] Matt Brubeck, *Fortune's Algorithm in C++*, 2002,
https://www.cs.hmc.edu/~mbrubeck/voronoi.html

[24] Jeroen van Hoof, Volker Jaenisch, *Foronoi*, GitHub repository, 2021,
https://github.com/Yatoom/foronoi

[25] Richard Turner, Karl Hobley, Alek Ratzloff, *rust_ voronoi*, GitHub repository, 2021,
https://github.com/petosegan/rust_voronoi

43:3616402066