Diagramy Voronoi

Algorytmy Geometryczne, rok 2023/2024

Jakub Karoń
Jan Dąbrowski

Opis projektu

Celem projektu jest napisanie algorytmu wizualizującego diagram Voronoi dla danych punktów na dwa sposoby:

- metodą inkrementacyjną
- algorytmem Fortune'a.

Diagramem Voronoi (lub Woronoja) dla danych *n* punktów nazywamy taki podział płaszczyzny *n* obszarów (komórek) w taki sposób, że każdy z punktów w danej komórce jest bliżej jednego z wejściowych punktów niż pozostałych punktów wejściowych.

Metoda inkrementacyjna

Metoda inkrementacyjna

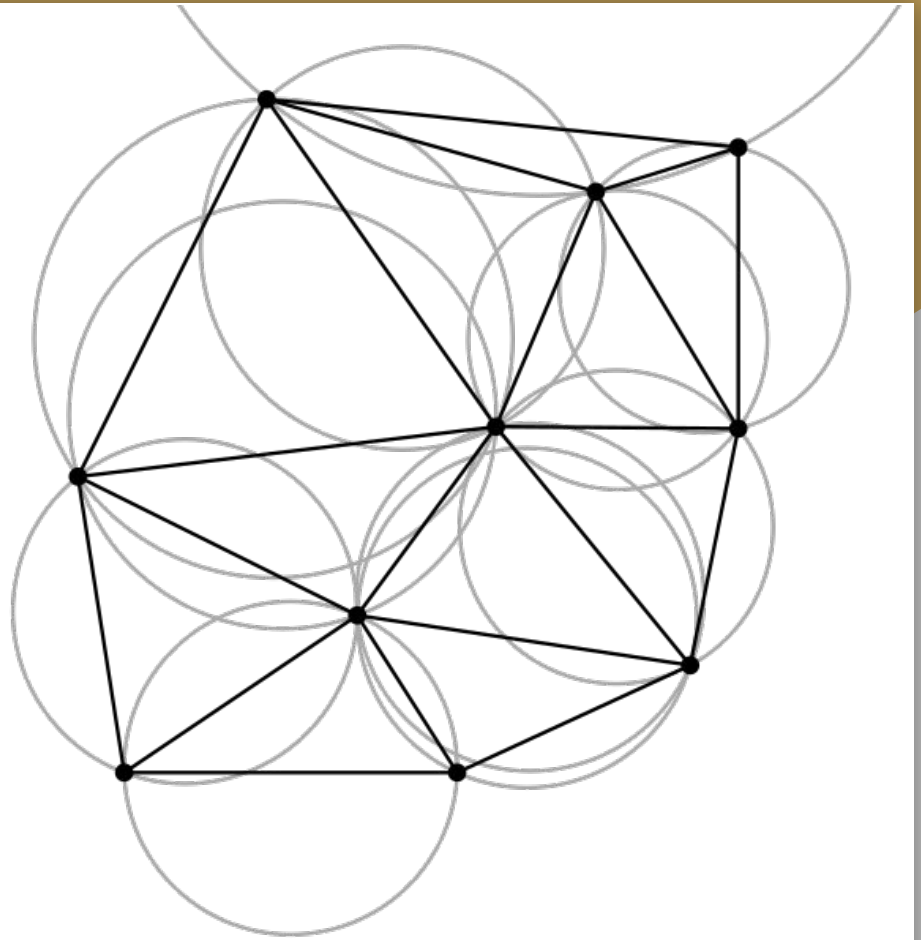
Polega ona na znalezieniu triangulacji Delaunay’a (czyt. Delone), a następnie zbudowania na niej diagramu Voronoi.

Triangulacja Delaunay’a jest triangulacją punktów spełniającą kryterium max-min, lub równoważnie, kryterium okręgów opisanych. W naszej implementacji będziemy kierować się tym drugim. Kryterium okręgów opisanych mówi, że okrąg opisany na każdym z trójkątów z triangulacji nie może zawierać ani jednego punktu z wejściowego zbioru punktów w swoim wnętrzu.

Triangulacja Delaunay’a

Począwszy od startowego trójkąta, dodajemy po kolei punkt i wykonujemy następujące operacje:

1. Znajdujemy trójkąt, do którego należy punkt
2. Znajdujemy sąsiedztwo punktu
3. Usuwamy sąsiedztwo
4. Odbudowujemy sąsiedztwo



Znajdowanie trójkąta

Począwszy od jednego z trójkątów, przechodzimy po jego sąsiadach w kierunku punktu dopóki nie znajdziemy trójkąta zawierającego punkt.

```
def find_triangle(self, p: Point) -> Triangle:

    curr_triangle = self.inittriangle
    visited = {curr_triangle}

    def _next_triangle(t, p):
        ns = []
        for edge in t.edges:
            neighbour = self.neighbours.find_neighbour(edge, t)
            if neighbour is not None and neighbour not in visited:
                ns.append(neighbour)
                if orientation(edge.p1, edge.p2, p) == 1:
                    return neighbour

        raise Exception("Here we go again")

    i = 0
    while not curr_triangle.point_in_triangle(p):
        print("Next triangle search, step: ", i)
        curr_triangle = _next_triangle(curr_triangle, p)
        visited.add(curr_triangle)
        i += 1

    return curr_triangle
```

Sąsiedztwo

Sąsiedztwem punktu nazywamy wszystkie trójkąty triangulacji, których okrąg opisany zawiera dany punkt. Znalezienie sąsiedztwa przeprowadzamy rekurencyjnie, przeszukując sąsiadów danego trójkąta, począwszy od tego zawierającego punkt.

Otoczką sąsiedztwa nazywamy taki podzbiór wszystkich krawędzi z sąsiedztwa, że żaden inny trójkąt z sąsiedztwa nie ma sąsiada w sąsiedztwie poprzez daną krawędź.

```python
def find_neighbourhood(
    self,
    p: Point,
    curr: Triangle,
) -> tuple[list[Triangle], list[Edge]]:
    neighbourhood: set[Triangle] = set()
    hull: list[Edge] = []

    def _rec(t: Triangle):
        neighbourhood.add(t)
        for edge in t.edges:
            next_t = self.neighbours.find_neighbour(edge, t)
            if next_t is None:
                hull.append(edge)
                continue

            if next_t not in neighbourhood:
                if next_t is not None and next_t.circumcircle_contains(p):
                    _rec(next_t)
                else:
                    hull.append(edge)

    _rec(curr)

    return neighbourhood, hull
```

Sąsiedztwo - usunięcie i odbudowa

Usuwamy z sąsiedztwa wszystkie trójkąty będące jego częścią. Odbudowanie sąsiedztwa polega na połączeniu krawędzią wszystkich punktów otoczki z dodawanym punktem. Otrzymujemy w ten sposób poprawną triangulację, spełniającą warunek okręgów opisanych.

```
def delete_neighbourhood(self, neighbourhood) -> None:
    for triangle in neighbourhood:
        for i in range(3):
            self.neighbours.remove_neighbour(triangle.edges[i], triangle)
        self.triangulation.remove(triangle)

def rebuild_neighbourhood(self, p : Point, hull):
    for edge in hull:
        t = Triangle(p, edge.p1, edge.p2)
        self.triangulation.add(t)
        self.neighbours.put(t)
    self.inittriangle = t
```

Budowanie komórki diagramu Voronoi

Mając pełną triangulację, połączenie środków okręgów opisanych na trójkątach sąsiadujących ze sobą tworzy pełen diagram Voronoi. Dla danego punktu, połączone środki okręgów opisanych na trójkątach mających wierzchołek w danym punkcie skutkuje powstaniem komórki diagramu Voronoi, której centrum jest ten punkt.

```
def create_polygon_for_vertex(
        self,
        vertex: Point,
        point_to_edges: dict
        ) -> Optional[list[Point]]:
    polygon = []
    edges = list(point_to_edges[vertex])
    first_edge = edges[0]
    edge = first_edge
    triangle = self.neighbours.edges[edge][0]
    circumcenter = triangle.find_circumcenter()
    polygon.append(circumcenter)

    while True:
        next_triangle = self.neighbours.find_neighbour(edge, triangle)
        circumcenter = next_triangle.find_circumcenter()
        polygon.append(circumcenter)

        for temp_edge in next_triangle.edges:
            if (vertex == temp_edge.p1 or vertex == temp_edge.p2) \
                and temp_edge != edge:
                next_edge = temp_edge
        if next_edge == first_edge:
            break
        edge = next_edge
        triangle = next_triangle

    return polygon
```

Zbudowanie pełnego diagramu Voronoi

Użycie poprzedniej funkcji dla każdego punktu ze zbioru wejściowego prowadzi do powstania pełnego diagramu Voronoi.

```
def create_diagram(self) -> list[Polygon]:
    point_to_edges = self.create_point_to_edges_mapping()
    diagram = []
    for vertex in self.points:
        polygon = self.create_polygon_for_vertex(vertex, point_to_edges)
        if polygon is not None:
            diagram.append(Polygon(
                data=[(p.x, p.y) for p in polygon],
                options={}
            ))

    return diagram
```

Struktury użyte w implementacji

Dokładny opis struktur i ich metod znajduje się w dokumentacji

```python
class Point:
    def __init__(self, x, y) -> None:
        self.x = x
        self.y = y
```

```python
class Neighbours:
    def __init__(self):
        self.edges = defaultdict(list)
```

```python
class Edge:
    def __init__(self, p1: Point, p2: Point):
        self.p1 = p1
        self.p2 = p2
```

```python
class Triangle:
    def __init__(self, a, b, c) -> None:
        # coutnerclockwise
        if orientation(a, b, c) == 2:
            self.a = a
            self.b = b
            self.c = c
        else:
            self.a = a
            self.b = c
            self.c = b

        self.edges = [Edge(self.a, self.b),
                      Edge(self.b, self.c),
                      Edge(self.c, self.a)]
```

```python
class DelaunayTriangulation:
    def __init__(self, points) -> None:
        # list of Triangles
        self.points = points
        self.super_triangle = gen_init_triangle(points)

        self.triangulation = {self.super_triangle}
        self.neighbours = Neighbours()
        self.neighbours.put(self.super_triangle)
        self.inittriangle = self.super_triangle
```

```python
class VoronoiDiagram:
    def __init__(self, points : List[Point], neighbours : Neighbours, triangulation : List[Triangle]) -> None:
        self.points = points
        self.neighbours = neighbours
        self.triangles = triangulation
```

Algorytm Fortune’a

Algorytm Fortune’a

Jest to algorytm zamiatania. Miotła porusza się przeciwnie do kierunku osi y. Ponad miotłą znajdują się jedynie elementy, które nie mogą być zmienione przez punkty poniżej miotły - zbiór takich elementów, będącymi ścianami komórek Voronoi, jest listą odcinków.

Obecnym stanem miotły jest drzewo binarne przechowujące niedokończone krawędzi oraz łuki (parabole). Przeszukiwanie elementów w tym drzewie jest poprzez x-współrzędną.

Wydarzenia są reprezentowane przez dwie kolejki priorytetowe: jedna przechowuje punkty, a druga za tzw. circle events - są to punkty, w których jakiś łuk zostaje przykryty łukami sąsiadującymi z nim.

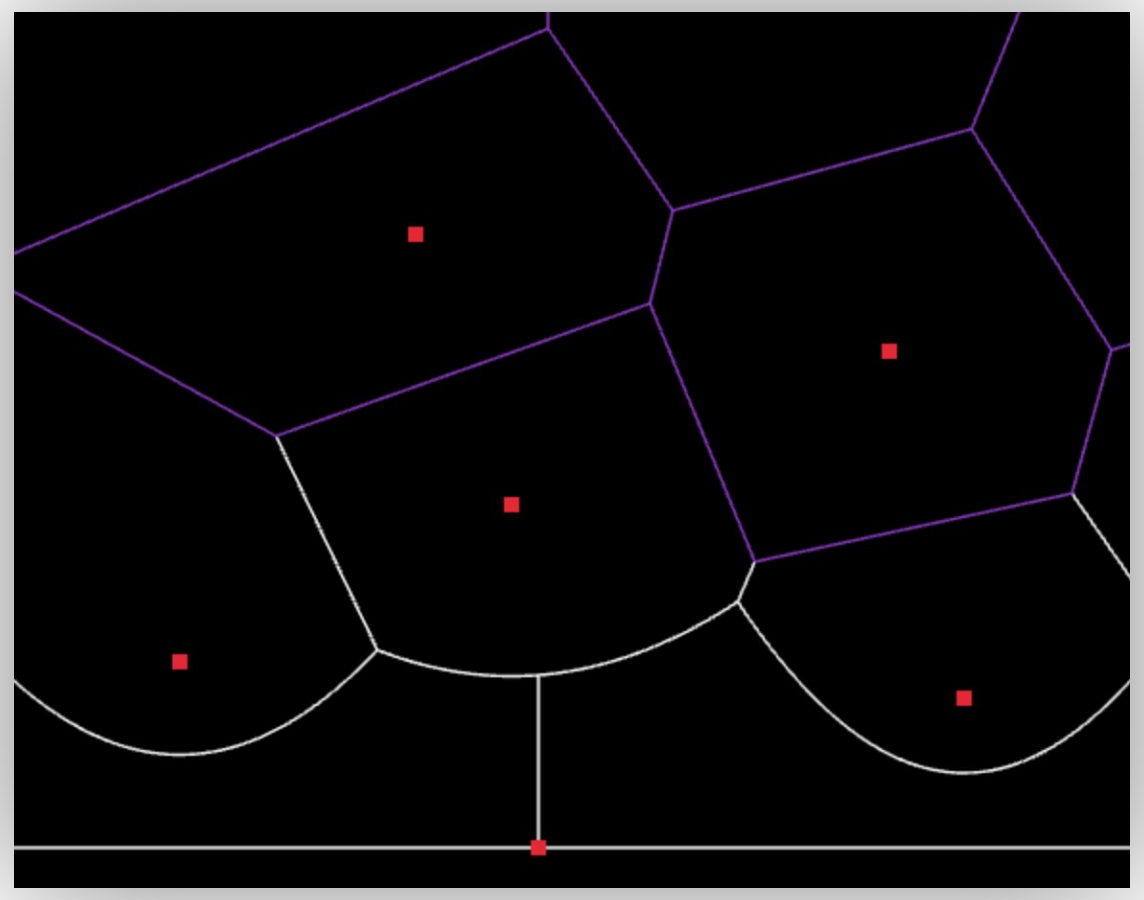
```
def process(self):
    while not self.points.empty():
        if not self.event.empty() and (self.event.top().x <= self.points.top().x):
            self.process_event() # handle circle event
        else:
            self.process_point() # handle site event

    # after all points, process remaining circle events
    while not self.event.empty():
        self.process_event()
```

Wydarzenie - nowy punkt

Gdy miotła natrafia na nowy punkt, musi zostać do stanu miotły nowa parabola, definiowana tym punktem. Sprawdzane jest przecięcie paraboli z poprzednią i następną (jeśli istnieją), a wydarzenia te, jeśli wykryte, są dodawane do kolejki wydarzeń.

```
# if p never intersects an arc, append it to the list
i = self.arc
while i.pnext is not None:
    i = i.pnext
i.pnext = Arc(p, i)

# insert new segment between p and i
x = self.x0
y = (i.pnext.p.y + i.p.y) / 2.0
start = Point(x, y)

seg = Segment(start)
i.s1 = i.pnext.s0 = seg
self.output.append(seg)
```

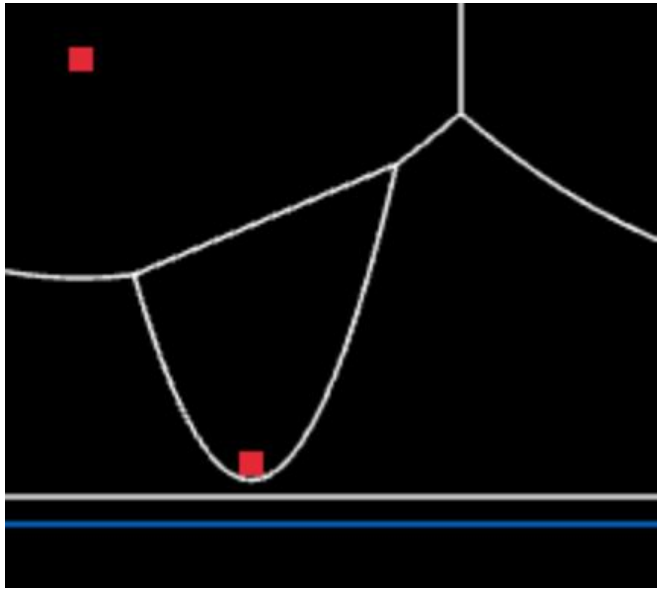
```
i = self.arc
while i is not None:
    flag, z = self.intersect(p, i)
    if flag:
        # new parabola intersects arc i
        flag, zz = self.intersect(p, i.pnext)
        if (i.pnext is not None) and (not flag):
            i.pnext.pprev = Arc(i.p, i, i.pnext)
            i.pnext = i.pnext.pprev
        else:
            i.pnext = Arc(i.p,i)
        i.pnext.s1 = i.s1

        # add p between i and i.pnext
        i.pnext.pprev = Arc(p, i, i.pnext)
        i.pnext = i.pnext.pprev

        i = i.pnext # now i points to the new arc

        # add new half-edges connected to i's endpoints
        seg = Segment(z)
        self.output.append(seg)
        i.pprev.s1 = i.s0 = seg

        seg = Segment(z)
        self.output.append(seg)
        i.pnext.s0 = i.s1 = seg

        # check for new circle events around the new arc
        self.check_circle_event(i, p.x)
        self.check_circle_event(i.pprev, p.x)
        self.check_circle_event(i.pnext, p.x)

        return

    i = i.pnext
```

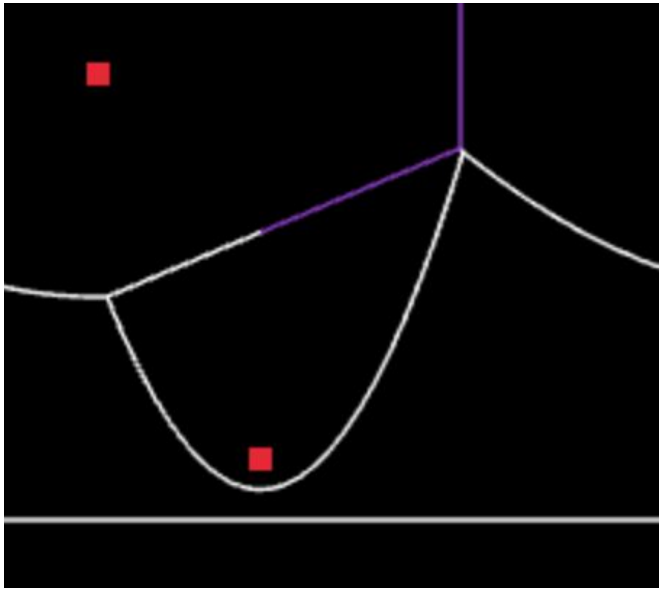


Wydarzenie - “przykrycie” paraboli

Przykrycie paraboli a wydarza się wtedy, gdy odległość od jej centrum do centrów jej poprzedniczki i następniczki jest taka sama. Przykrycie paraboli oznacza, że znika ona ze stanu miotły, a punkt ten jest jednym z wierzchołków diagramu Voronoi.

```
def process_event(self):
    # get next event from circle pq
    e = self.event.pop()

    if e.valid:
        # start new edge
        s = Segment(e.p)
        self.output.append(s)

        # remove associated arc (parabola)
        a = e.a
        if a.pprev is not None:
            a.pprev.pnext = a.pnext
            a.pprev.s1 = s
        if a.pnext is not None:
            a.pnext.pprev = a.pprev
            a.pnext.s0 = s

        # finish the edges before and after a
        if a.s0 is not None: a.s0.finish(e.p)
        if a.s1 is not None: a.s1.finish(e.p)

        # recheck circle events on either side of p
        if a.pprev is not None: self.check_circle_event(a.pprev, e.x)
        if a.pnext is not None: self.check_circle_event(a.pnext, e.x)
```





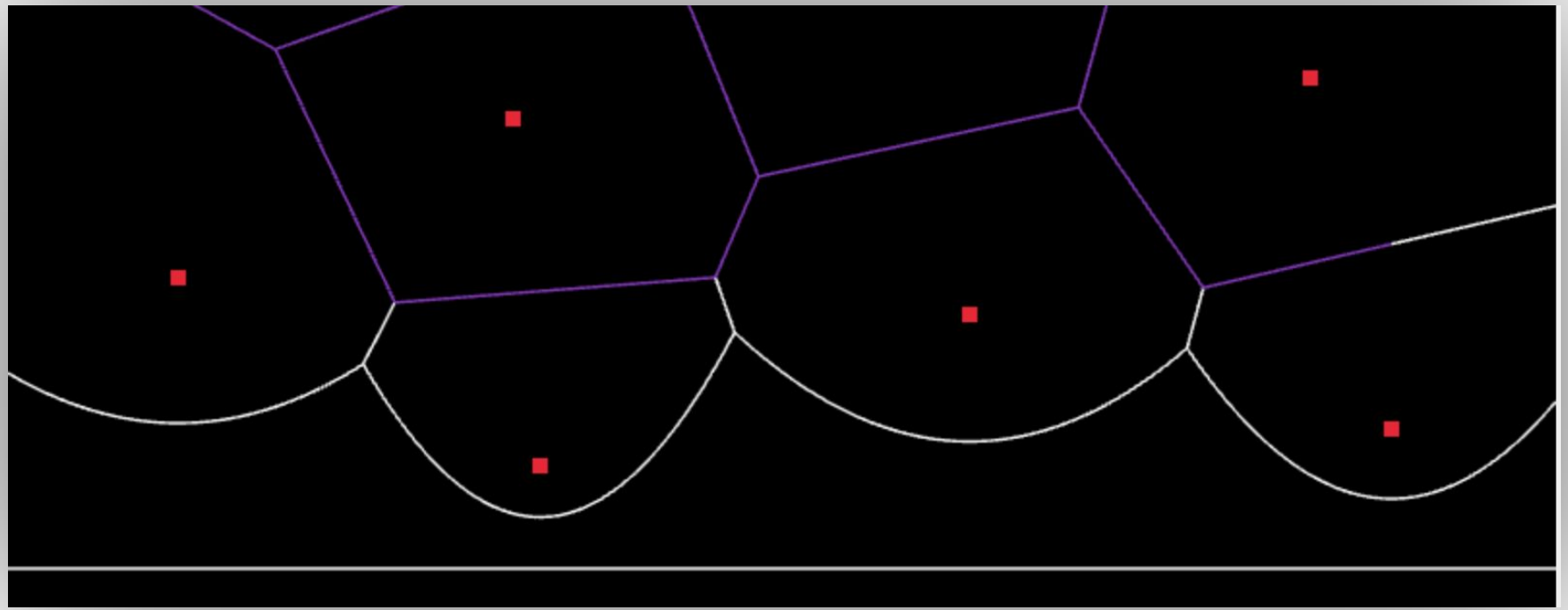
“Przykrycie” łuku. Zaznaczone odcinki kolorem fioletowym są już zakończymi odcinkami, które są częścią diagramu Voronoi.

Ostatni krok - dokończenie nieskończonych odcinków

Przeszliśmy miotłą przez wszystkie punkty i wydarzenia. Zostało zatem tylko dokończenie nieskończonych odcinków.

```
def finish_edges(self):
    l = self.x1 + (self.x1 - self.x0) + (self.y1 - self.y0)
    i = self.arc
    while i.pnext is not None:
        if i.s1 is not None:
            p = self.intersection(i.p, i.pnext.p, l*2.0)
            i.s1.finish(p)
        i = i.pnext
```

Po przejściu wszystkich punktów należy dokończyć pozostałe, niedokończone odcinki (zaznaczone kolorem białym).



Struktury użyte w implementacji

Dokładny opis klas i ich metod znajduje się w dokumentacji.

```
class Point:
    x = 0.0
    y = 0.0
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
class Event:
    x = 0.0
    p = None
    a = None
    valid = True
    def __init__(self, x, p, a):
        self.x = x
        self.p = p
        self.a = a
        self.valid = True
```

```
class Segment:
    start = None
    end = None
    done = False
    def __init__(self, p):
        self.start = p
        self.end = None
        self.done = False

    def finish(self, p):
        if self.done: return
        self.end = p
        self.done = True
```

```
class Arc:
    p = None
    pprev = None
    pnext = None
    e = None
    s0 = None
    s1 = None
    def __init__(self, p, a=None, b=None):
        self.p = p
        self.pprev = a
        self.pnext = b
        self.e = None
        self.s0 = None
        self.s1 = None
```

```
class PriorityQueue:
    def __init__(self):
        self.pq = []
        self.entry_finder = {}
        self.counter = itertools.count()
```

Porównanie wydajności i złożoność obliczeniowa

Algorytm Bowyera-Watsona ma złożoność $O(n^2)$, ponieważ iteruje po n punktach, a wszystkie operacje wewnątrz iteracji są złożoności O(n). Algorytm Fortune'a, ponieważ nie zaimplementowaliśmy drzewa binarnego, ma również złożoność $O(n^2)$, jednak z mniejszą stałą.

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

Bibliografia

- 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. "Fortune’s 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.
- GeeksforGeeks. https://www.geeksforgeeks.org/
- 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.