УДК 519.6:004.92

https://doi.org/10.52058/2786-6025-2024-1(29)-666-688

Іванов Артем Олександрович здобувач освіти, Житомирський державний університет імені Івана Франка, вул. Велика Бердичівська, 40, м. Житомир, 10008, тел.: (063) 434-81-33, https://orcid.org/0009-0006-2704-9906

Кривонос Олександр Миколайович кандидат педагогічних наук, доцент, доцент кафедри комп'ютерних наук та інформаційних технологій, Житомирський державний університет імені Івана Франка, вул. Велика Бердичівська, 40, м. Житомир, 10008, тел.: (098) 742-02-28, https://orcid.org/ 0000-0002-4211-6541

Жуковський Сергій Станіславович кандидат педагогічних наук, доцент кафедри комп'ютерних наук та інформаційних технологій, Житомирський державний університет імені Івана Франка, вул. Велика Бердичівська, 40, м. Житомир, 10008, тел.: (063) 812-49-81, https://orcid.org/0000-0001-5826-0751

ОСОБЛИВОСТІ РЕАЛІЗАЦІЇ АЛГОРИТМУ ФОРЧУНА ДЛЯ ПОБУДОВИ ДІАГРАМИ ВОРОНОГО НА МОВІ ПРОГРАМУВАННЯ PYTHON

Анотація. Дана робота присвячена вивченню особливостей реалізації алгоритму Форчуна для побудови діаграми Вороного на мові програмування Python. Алгоритм Форчуна є ефективним методом для розділення простору на області, названі областями Вороного, на основі набору точок у просторі. Діаграма Вороного має широкі застосування в областях геометрії, графікі, комп'ютерного зору та інших наукових дисциплінах.

В роботі описано поняття діаграми Вороного, її варіації і їх прикладне застосування. Дане розбиття знайшло широке використання в біології, географічних інформаційних системах, матеріалознавстві, комп'ютерній графіці тощо. Показано, що ця діаграма є досить важливою частиною як для природничих наук, так і для обчислювальної геометрії, представлено зв'язок діаграми з іншими геометричними структурами.

Робота розглядає теоретичні аспекти алгоритму Форчуна та його реалізацію на мові програмування Python. Вона досліджує структуру алгоритму, його етапи та ключові кроки, необхідні для отримання точної діаграми Вороного. Автори розглядають способи оптимізації та врахування особливостей мови Python для покращення продуктивності алгоритму.

Представлений окремий аналіз різних алгоритмів комп'ютерної побудови діаграми Вороного, а саме: поступова вставка сайтів, «розділяй і

володарюй», побудова через тріангуляцію Делоне, алгоритм Форчуна – дано опис алгоритмів, оцінка ефективності по використанню часу і пам'яті, складності реалізації. Запропонована реалізація алгоритму Форчуна для 2-вимірного випадку з асимптотичною складністю $O(n^2)$ на мові програмування Python. Описано основні структури даних та змінні, необхідні для реалізації алгоритму. В даній роботі для реалізації алгоритму використано двозв'язні списки, проте алгоритм можна вдосконалити, використовуючи бінарні збалансовані дерева пошуку, тим самим зменшивши асимптотичну складність до $O(n \log n)$.

Ключові слова: обчислювальна геометрія, діаграма Вороного, алгоритм Форчуна, замітаюча пряма, Python.

Ivanov Artem Oleksandrovych Student, Zhytomyr Ivan Franko State University, 40, Velyka Berdychivska St., Zhytomyr, 10008, tel.: (063) 434-81-33, https://orcid.org/0009-0006-2704-9906

Kryvonos Oleksandr Mykolaiovych Candidate of Pedagogical Sciences (PhD in Pedagogy) Docent Department of Computer Science and Information Technology, Zhytomyr Ivan Franko State University, 40, Velyka Berdychivska St., Zhytomyr, 10008, tel.: (098) 742-02-28, https://orcid.org/0000-0002-4211-6541

Zhukovskyi Serhii Stanislavovych Candidate of Pedagogical Sciences (PhD in Pedagogy) Docent Department of Computer Science and Information Technology, Zhytomyr Ivan Franko State University, 40, Velyka Berdychivska St., Zhytomyr, 10008, tel.: (063) 812-49-81, https://orcid.org/0000-0001-5826-0751

PECULIARITIES OF IMPLEMENTATION OF THE FORTUNE'S ALGORITHM FOR GENERATING A VORONOI DIAGRAM IN THE PYTHON PROGRAMMING LANGUAGE

Abstract. This article deals with the peculiarities of implementing the Fortune's algorithm for generating Voronoi diagrams in Python programming language. The Fortune's algorithm is an efficient method for dividing a space into regions, called Voronoi regions, based on a set of points in space. The Voronoi diagram is widely used in the fields of geometry, graphics, computer vision, and other scientific disciplines.

The paper describes the concept of the Voronoi diagram, its variations and their application. This diagram is widely used in biology, geographic information systems, materials science, computer graphics, etc. It is shown that this diagram is a rather important part of both natural sciences and computational geometry, and the connection of the diagram with other geometric structures is presented.

The study considers the theoretical aspects of the Fortune's algorithm and its implementation in the Python programming language. It explores the algorithm's structure, its stages, and the key steps required to obtain an accurate Voronoi diagram. The authors consider ways to optimise and take into account the peculiarities of the Python language to improve the algorithm's performance.

The article presents a separate analysis of various algorithms for computer construction of Voronoi diagrams, specifically: gradual insertion of sites, divide and conquer, construction through Delaunay triangulation, Fortune's algorithm - the description of algorithms, assessment of efficiency in terms of time and memory usage, and complexity of implementation are given. This paper proposes an implementation of Fortune's algorithm for the 2-D case with asymptotic complexity O(n^2) in Python programming language. It describes the main data structures and variables required to implement the algorithm. In this article, we use two-linked lists to implement the algorithm, but the algorithm can be improved by using balanced binary search trees, thereby reducing the asymptotic complexity to O(n logn).

Keywords: computational geometry, Voronoi diagram, Fortune's algorithm, sweeping line, Python.

Постановка проблеми. Представимо одну з найвідоміших задач в цій області аналітичної геометрії, так звану задачу поштових офісів: на площині задано n фіксованих точок – сайтів (site). Для довільної точки на цій площині необхідно знайти найближчий сайт в евклідовій метриці. Пошук такого сайту «в лоба» – перевіркою всіх відстаней від точки до сайтів – є досить неефективним, особливо при великій кількості сайтів.

Для вирішення цієї задачі будується діаграма Вороного – розбиття площини на області Вороного (region). Кожному сайту діаграми відповідає своя область Вороного, кожна точка якої є ближчою до цього сайту, ніж до інших. Більш формально це визначення виглядає наступним чином:

$$reg(p_i) = \{x \in \mathbb{R}^d | \delta(x, p_i) \leq \delta(x, p_j) \forall j: j \neq i\}$$

де p_i – відповідний сайт, x – довільна точка у d-вимірному просторі. Тоді діаграма Вороного буде множиною всіх цих областей: $\mathbb{V} = \{reg(p_1), reg(p_2), \ldots, reg(p_n)\}$. Пошук найближчого сайту є лише одним з багатьох прикладів застосування діаграми Вороного, вони також активно використовуються для моделювання структур кристалів, побудови найкоротших шляхів, аналізу географічних об'єктів, виявлення колізій, для розв'язання низки задач в обчислювальній геометрії та теорії графів і багато іншого.

Аналіз останніх досліджень і публікацій. Провели комплексний аналіз діаграми Вороного та методів її побудови, також провели опис розв'язання інших завдань з використанням діаграми Вороного в сучасному погляді такі науковці, як F. Aurenhammer, A. Dobrin, S. K. Dey, M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf.

Мета статті – аналіз алгоритмів побудови діаграми Вороного та реалізація алгоритму Форчуна на мові програмування Python.

Матеріали та методи дослідження. Для вирішення поставлених задач були використані такі теоретичні методи, як систематизація наукових знань та їх порівняння, моделювання. Серед загальнонаукових методів були використані наступні методи: аналіз, дедукція та індукція для опрацювання інформації.

Виклад основного матеріалу. Для знаходження найближчого сайту для довільної точки необхідно провести попередній аналіз всього набору сайтів, а саме – побудувати діаграму Вороного, також відома як зони Вігнера-Зейтца (в біології, хімії та кристалографії), полігони Тіссена (в геоінформатиці), або мозаїка Діріхле.

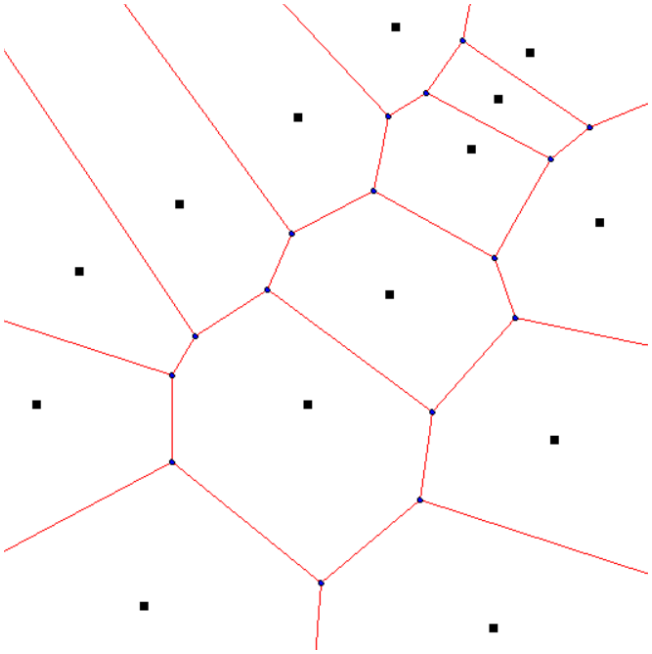

Рис. 1. *Діаграма Вороного та основні її елементи: сайти, ребра та вершини*

Ця діаграма дозволяє за константний час виконати пошук найближчого сайту для довільної точки. Сама діаграма представляє собою набір ребер (edge), які є рівновіддаленими від 2 сайтів. Декілька ребер сходяться у вершині (vertex), які є точками, рівновіддаленими від 3 або більше сайтів (див. рис. 1). Розглянемо основні алгоритми побудови, а також їхні переваги та недоліки.

Алгоритм Гріна-Сібсона (Green-Sibson algorithm), також відомий як метод поступової вставки сайтів (incremental insertion of sites), полягає у тому, щоб поступово збільшувати діаграму додаванням нових вершин, корегуючи її відповідним чином. Для цього для кожної нової точки перебираються всі інші. Кожен цикл додавання сайту виконується $O(i)$ часу, де i – кількість доданих

сайтів на даному етапі. Підсумовуючи, отримуємо очікувану алгоритмічну складність даного методу $O(n^2)$. Можливе покращення алгоритму шляхом простої евристики, рандомізації додаваних сайтів тощо.

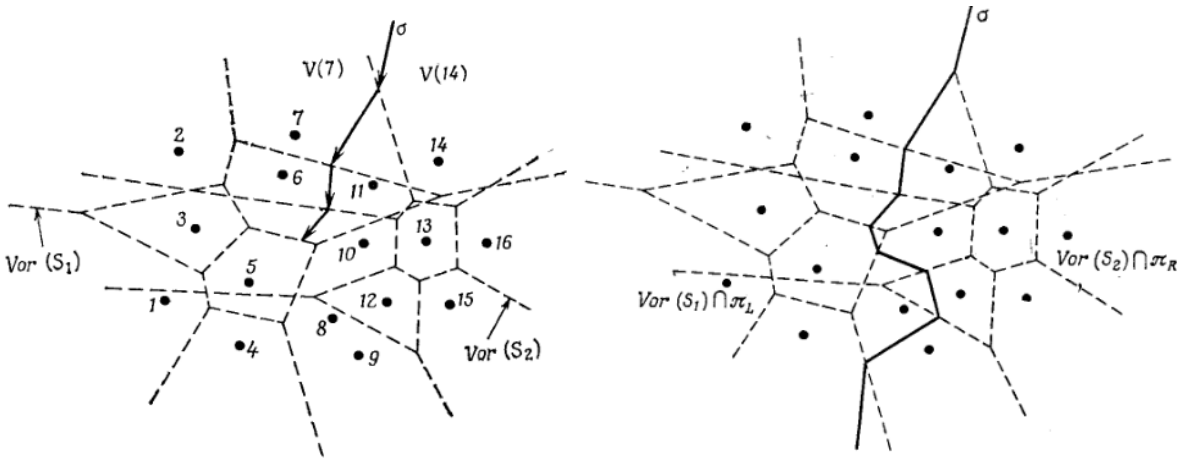

Рис. 2. *Злиття двох діаграм Вороного*

Алгоритм «розділяй та володарюй» («divide and conquer») є одним з найбільш популярних типів ефективних алгоритмів, які мають лінеаритмічну алгоритмічну складність – $O(n \log n)$. Ідея даного алгоритму схожа до алгоритму Гріна-Сібсона, але має деякі відмінності: тут виконується злиття декількох діаграм Вороного (див. рис. 2). Тим самим, впорядкований набір сайтів розділяється на 2 окремих набора, кожен з яких рекурсивно розділяється відповідним чином. Таким чином досягається алгоритмічна складність алгоритму $O(n \log n)$. Мінусом алгоритму є складність розуміння механізму злиття і його реалізація.

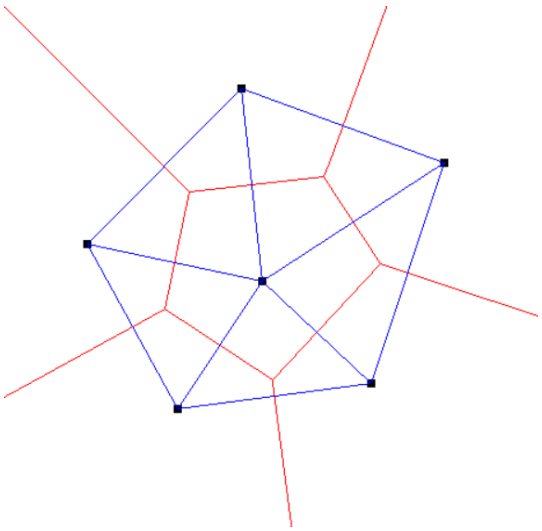

Рис. 3*. Діаграма Вороного (червона) та тріангуляція Делоне (синя)*

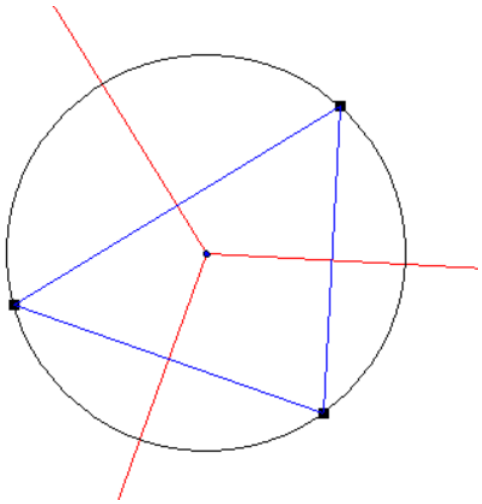

Рис. 4. *Центр описаного кола тріангуляції Делоне в вершині діаграми*

Інший алгоритм базується на попередній побудові дуального графа до Вороного – тріангуляції Делоне (див. рис. 3). Тріангуляція Делоне (Delauney triangulation) – розбиття набору точок на трикутники таким чином, щоб будь-яке описане коло навколо будь-якого трикутника не містило в собі інших точок з набору. Дуальність цієї тріангуляції до діаграми Вороного виражається у тому, що обидві геометричні структури можна легко отримати одне з одного.

Доведення. Ребро діаграми Вороного це відрізок/промінь/пряма, що рівновіддалений від двох сайтів; як відомо, геометричне місце точок, рівновіддалених від двох заданих є серединний перпендикуляр відрізку, що з'єднує ці точки. Отже, ребра Вороного перпендикулярні ребрам трикутників тріангуляції Делоне, і ділять їх навпіл. Також зауважимо, що вершина Вороного це точка, рівновіддалена від 3 або більше сайтів, тобто, нема ні одного сайту, який знаходиться ближче, що відсилає нас до визначення тріангуляції Делоне. Ефективні алгоритми тріангуляції Делоне працюють за час $O(n \log n)$. Слід зазначити, що для невироджених діаграм Вороного (non-degenerate Voronoi diagram) – степінь вершин яких рівна 3 – існує єдина тріангуляція Делоне, на відміну від вироджених діаграм Вороного (degenerate Voronoi diagram) (див. рис. 5).

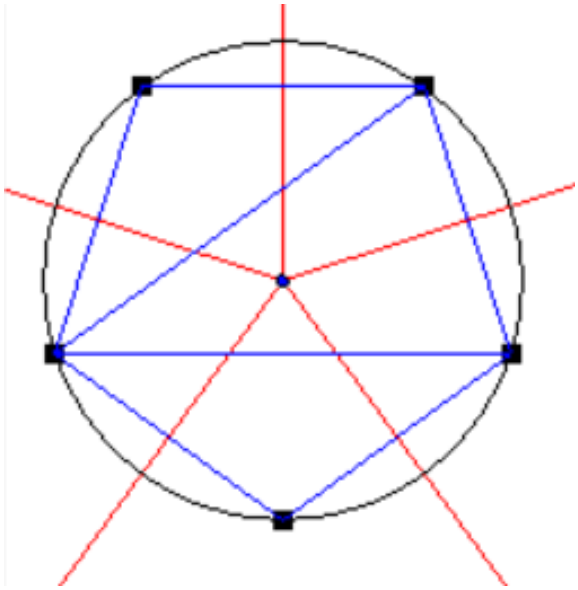

Рис. 5. *Одна з тріангуляцій* Делоне у виродж еній діаграмі Вороного

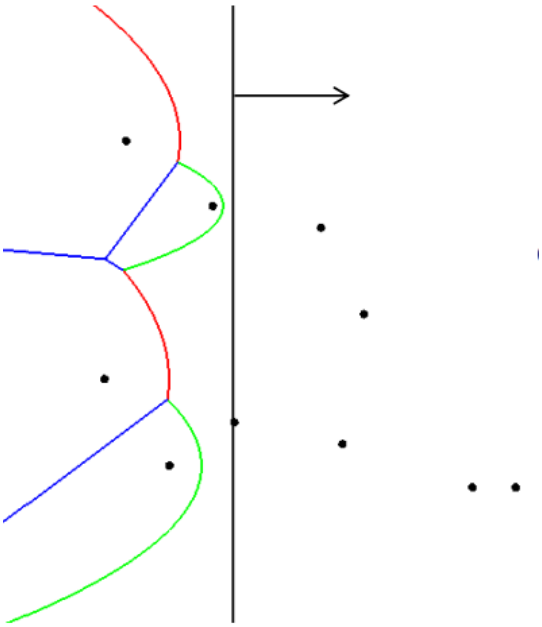

Рис. 6. *Рух замітаючої лінії*

Алгоритм Форчуна (Fortune's agorithm), також відомий як метод замітаючої прямої (line sweep algorithm) бере свої початки з відомого алгоритму Бентлі-Оттманна (Bentley-Ottmann algorithm) для знаходження перетину відрізків на площині. В цьому алгоритмі вперше описано

використання «замітаючої лінії», яка розділяє набір вхідних даних на два: оброблені, і ті, які ще потребують обробки. Стівен Форчун взяв дану ідею за основу алгоритму побудови діаграми Вороного.

Нехай замітаюча лінія буде вертикальною лінією, яка, рухаючись направо (див. рис. 6), розділяє всі сайти на три окремих набори: сайти, які вже додані до діаграми Вороного; сайти, які в процесі обробки; і сайти, які очікують обробки. Замітаюча лінія являє собою елемент діаграми, і впродовж свого руху вона бере безпосередню участь в її формуванні.

Як відомо, геометричним місцем точок, рівновіддалених від заданої точки і прямої, є парабола. Саме цю фігуру утворюють крайні точки зліва від замітаючої прямої, формуючи собою берегову лінію діаграми, яка складається виключно з дуг парабол. Для цих парабол замітаюча лінія виступає директрисою, таким чином, відстань від неї до вершини параболи рівна відстані від вершини параболи до сайту. При проходженні замітаючої лінії через новий сайт, утворюється нова парабола, яка спочатку є променем, оскільки лінія проходить через сайт. Як бачимо (див. рис. 8), при додаванні нової дуги параболи, вона може розділити дугу за собою на дві окремі дуги, які мають однаковий фокус.

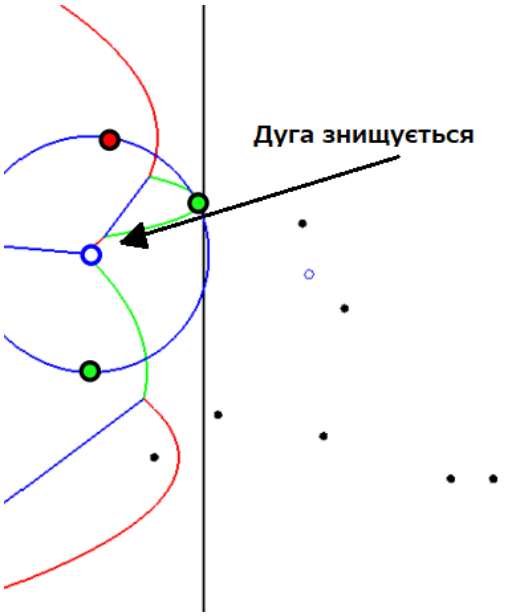

Рис. 7. *Подія круга, при якій знищується дуга*

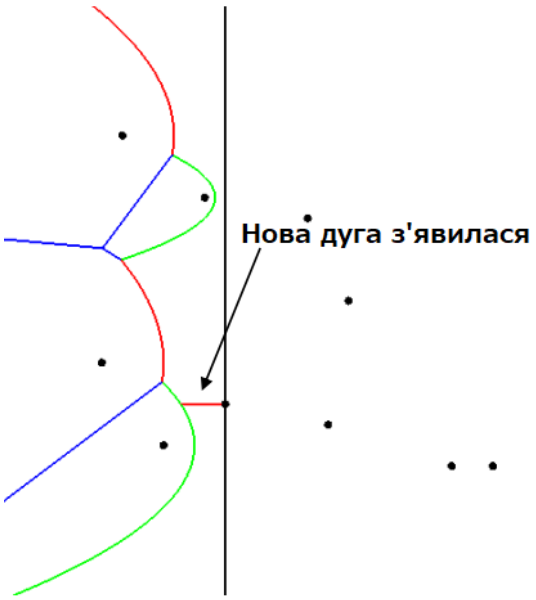

Рис. 8. *Подія точки, при якій з'являється нова дуга*

Неважко помітити, що кожна пара дуг перетинаються в точці, яка рівновіддалена від обох фокусів дуг і від замітаючої лінії. Таким чином, при переміщенні замітаючої лінії, на місці перетину дуг утворюються ребра Вороного. Кожна дуга (окрім крайніх) обмежена двома ребрами Вороного, які можуть сходитись одне до одного. В такому випадку довжина дуги прямує до

нуля, і в точці, де довжина дуги буде нуль, вона буде рівновіддалена від 3 або більше сайтів, що свідчить про те що ми досягли вершини Вороного (див. рис. 7).

Таким чином, маємо дві події, які змінюють структуру берегової лінії: подія точки, і подія круга. Події точки можна визначити одразу, адже положення сайтів відомі. А для визначення подій круга достатньо після кожної події перевірити три послідовні дуги і побудувати коло по трьом відповідним сайтам. Тоді подія круга відбудеться у крайній правій точці, оскільки саме в цьому випадку ребра Вороного зійдуться в вершині.

Для збереження берегової лінії в програмній реалізації було використано таку структуру даних як двозв'язний список, в якому кожен елемент має посилання на попередній і наступний, якщо такі є. Це доволі проста структура даних, але на заходження в ньому певного елементу витрачається до $O(n)$ операцій (в залежності від місця положення елементу). Інша структура даних, збалансоване бінарне дерево пошуку – також відоме як АВЛ-дерево – є більш складною, але дозволяє знайти елемент за $O(n \log n)$ в найгіршому випадку. Цей факт залишає простір для вдосконалення реалізації даного алгоритму.

Для початку подивимось, які структури даних використані в ході написання програми. Вони представлені на рисунку 9.

class Point
x: float
y: float
getCircleCenter(a: Point, b: Point, c: Point): Point

class Event
x: float
point: Point
arc: Arc
valid: boolean

class PriorityQueue
queue: array<Point>
count: int
entries: dict<int, {x: int, count: int, item: int}>
push(item: int)
pop()
top(): int

class Edge
start: Point
end: Point | None
verts: {Arc, Arc} | None
setEnd(end: Point)

class Arc
center: Point
prev: Arc | None
next: Arc | None
event: Event | None
contour1: Edge | None
contour2: Edge | None

Рис. 9. *Структури даних, використані в програмі*

Слід зазначити, шо ребро представлене у вигляді початкової і кінцевої точки, а також його клас має поле, яке містить посилання на дві дуги, щоб визначати, з якими сайтами межує дане ребро, а клас дуги має в собі посилання на перше і друге ребро, яким вона обмежена. Визначивши ці класи, можемо перейти до огляду самого алгоритму. Ось загальний опис алгоритму:

1: створення порожньої берегової лінії
2: пріоритетної черги для подій точки і круга: відповідно $point\_e$ і $circle\_e$
3: створення масиву ребер для занесення результату $result$

4: занесення всіх вхідних точок до $point\_e$
5: поки $point\_e$ не порожня:
6: якщо перша за пріоритетом подія $point\_e$ має більший пріоритет ніж $circle\_e$:
7: додати нову дугу до діаграми
8: за необхідності розділити дугу позаду на дві
9: прибрати поточну подію з черги
10: перевірити наявність нових подій круга, занести їх до $circle\_e$
11: інакше:
12: прибрати дугу з двозв'язного списку
13: створити нове ребро, яке буде продовженням вершини
14: занести закінчені ребра до $result$
15: прибрати поточну подію з черги
16: поки $circle\_e$ не порожня:
17: прибрати дугу з двозв'язного списку
18: створити нове ребро, яке виходить з вершини
19: занести закінчені ребра до $result$
20: прибрати поточну подію з черги
21: продовжити ребра з $result$, які не мають кінцевої точки, до нескінченності

Розглянемо більш детально безпосередньо програмну реалізацію на мові програмування Python.

```
1   import heapq
2
3   BORDER = [-(2 << 14), -(2 << 14), (2 << 14), (2 << 14)]
4   INF = 2 << 16
5
```

Рис. 10. *Імпорт бібліотек та визначення констант*

Спочатку (див. рис. 10) ми імпортуємо вбудовану бібліотеку для роботи з пріоритетними чергами $heapq$. Також ми визначаємо межі робочого простору, і якесь достатньо велике число у якості нескінченності.

```python

class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    @classmethod
    def getCircleCenter(cls, a, b, c):
        if ((b.x - a.x) * (c.y - a.y) - (c.x - a.x) * (b.y - a.y)) > 0:
            return None, None
        x1, y1, x2,y2, x3, y3 = a.x, a.y, b.x, b.y, c.x, c.y
        if x1 == x2:
            x2, y2, x3, y3 = x3, y3, x2, y2
        elif x2 == x3:
            x1, y1, x2, y2 = x2, y2, x1, y1
        if x1 == x2:
            return None, None
        m1, m2 = (y2-y1)/(x2-x1), (y3-y2)/(x3-x2)

        if m1 == m2:
            return None, None
        x = (m1*m2*(y1-y3)+m2*(x1+x2)-m1*(x2+x3))/2/(m2-m1)
        try:
            y = -1/m2 * (x-(x2+x3)/2) + (y2+y3)/2
        except ZeroDivisionError:
            y = -1/m1 * (x-(x1+x2)/2) + (y1+y2)/2

        rad = ((x-x1)**2+(y-y1)**2)**0.5
        return cls(x, y), x+rad

```

Рис. 11. *Опис класу точки*

Далі (див. рис. 11) визначаємо клас точки, в середині якого описуємо метод $getCircleCenter$. Цей метод визначає центр кола по 3 заданим точкам. Це робиться за допомогою проведення серединних перпендикулярів між двома парами точок. $m1$ та $m2$ – коефіцієнти нахилу відрізків, які визначаються за формулою:

$$m_1 = \frac{y_2 - y_1}{x_2 - x_1};\ m_2 = \frac{y_3 - y_2}{x_3 - x_2}$$

Коефіцієнти нахилу серединних перпендикулярів є відповідними коефіцієнтами нахилу прямих, помножених на -1. Тоді рівняння серединних перпендикулярів визначатимуться за наступною формуло:

$$y_1' = -\frac{1}{m_1}x + \frac{x_1 + x_2}{2} \cdot \frac{1}{m_1} + \frac{y_1 + y_2}{2}$$

$$y_2' = -\frac{1}{m_2}x + \frac{x_2 + x_3}{2} \cdot \frac{1}{m_2} + \frac{y_2 + y_3}{2}$$

Залишилось розв'язати систему рівнянь двох серединних перпендикулярів і визначити координати центру кола. В програмі одразу розраховується координати точки:

$$x = \frac{m_1 m_2 (y_1 - y_3) + m_2 (x_1 + x_2) - m_1 (x_2 + x_3)}{2(m_2 - m_1)}; y = y_1'(x)$$

Звісно, номери точок вибираються адаптовано. З метою запобігання ділення на 0 в процесі, в рядках 18-23 визначаємо кращу комбінацію точок. Якщо точки лежать на одній прямій, то такого кола не існує.

```
37
38  class Event:
39
40      def __init__(self, x, point, arc):
41          self.x = x
42          self.point : Point = point
43          self.arc : Arc = arc
44          self.valid = True
45
46
47  class Arc:
48
49      def __init__(self, center, prev=None, next=None):
50          self.center = center
51          # попередня та наступна дуга
52          self.prev = prev
53          self.next = next
54          # подія круга
55          self.event = None
56
57          self.contour1 : Edge = None
58          self.contour2 : Edge = None
59
60
61  class Edge:
62
63      def __init__(self, point):
64          self.start = point
65          self.end = None
66
67          self.verts = [None, None]
68
69      def setEnd(self, end):
70          if self.end:
71              return
72          self.end = end
73
```

Рис. 12. *Опис класів події кола; дуги; та ребра Вороного*

На рисунку 12 бачимо опис базових класів, який повністю відповідає схемі, зазначеній на рисунку 9. Визначено клас події, який сам по собі є подією кола, яка має два стани: валідний та інвалідний (відпрацьована подія, яку надалі слід ігнорувати). Клас дуги реалізований як елемент двозв'язного

списку, який має посилання на попередній і наступний елемент, які за замовчуванням рівні *None*. Для класу ребра визначено метод *setEnd*, який дозволяє зручно встановити кінцеву точку ребра.

```
74
75  class PriorityQueue:
76
77      def __init__(self):
78          self.queue = []
79          self.count = 0
80          self.entries = dict()
81
82      def push(self, item):
83          if item in self.entries:
84              return
85          self.count += 1
86          entry = [item.x, self.count, item]
87          self.entries[item] = entry
88          heapq.heappush(self.queue, entry)
89
90      def pop(self):
91          while self.queue:
92              p, count, item = heapq.heappop(self.queue)
93              if item:
94                  del self.entries[item]
95                  return item
96          raise Exception("empty queue")
97
98      def top(self):
99          item = self.pop()
100         self.push(item)
101         return item
102
```

Рис. 13. *Опис класу пріоритетної черги*

На рисунку 13 показана реалізація пріоритетної черги. Вона реалізована з використанням допоміжної вбудованої бібліотеки *heapq*, яка надає інструменти для роботи з пріоритетними чергами, а саме додавання і видалення елементів звідти. Клас *PriorityQueue* адаптований для комфортної роботи з подіями.

На рисунку 15 вже визначається клас діаграми Вороного. В його ініціалізації визначаються ребра, берегова лінія, а також пріоритетні черги для кожного типу подій. До пріоритетної черги подій точки додаються всі точки, які передаються конструктору при ініціалізації. Попередньо ці точки додатково сортуються по осі y для запобігання помилок в обробці точок, що лежать на одній вертикалі.

Для початку побудови діаграми Вороного потрібно викликати метод *process*, який починає обробку подій. В ході обробки вибираються найлівіші події і відповідно обробляються. Після закінчення подій точки обробляються залишкові події круга, після чого крайові ребра продовжуються до нескінченності і повертаються самі ребра Вороного.

На рисунку 16 показана реалізація методів обробки відповідно подій точки і кола. Обробка події кола доволі проста. Для початку в 140 рядку створюється нове ребро Вороного яке буде виходити з точки, в яку стягується відповідна дуга (ця точка дістається з відповідної події). Далі в рядках 144-151 відбувається видалення дуги з двозв'язного списку: для цього редагуються попередня та наступна дуга таким чином, щоб вони були сусідніми одне з одним. Після закінчення ребер, які зійшлись в точці, яку описувала подія, в 153-156 рядках, сусідні дуги, що залишились, перевіряються на наявність подій кола (події кола для дуги може не бути, якщо обмежуючі ребра розходяться, а дуга збільшується).

```
103
104 class Vor:
105
106     def __init__(self, points):
107         self.result = [] # ребра Вороного
108         self.arc = None # берегова лінія у вигляді двозв'язного списку дуг
109         self.points = points
110
111         # пріоритетні черги для подій точки та круга
112         self.point_e = PriorityQueue()
113         self.circle_e = PriorityQueue()
114
115         for pts in sorted(points, key=lambda x: x[1]):
116             point = Point(pts[0], pts[1])
117             self.point_e.push(point)
118
119     def process(self):
120         while self.point_e.queue:
121             if self.circle_e.queue and (self.point_e.top().x >= self.circle_e.top().x):
122                 self.process_circle()
123             else:
124                 self.process_point()
125
126         while self.circle_e.queue:
127             self.process_circle()
128
129         self.finish_edges()
130
131         return self.result
132
```

Рис. 15. *Конструктор класу діаграми Вороного та метод для її побудови*

```
136     def process_circle(self):
137         event = self.circle_e.pop()
138         if event.valid:
139             # створюється нове ребро Вороного
140             edge = Edge(event.point)
141             self.result.append(edge)
142 
143             arc = event.arc
144             if arc.prev:
145                 arc.prev.next = arc.next
146                 arc.prev.contour2 = edge
147                 edge.verts[0] = arc.prev
148             if arc.next:
149                 arc.next.prev = arc.prev
150                 arc.next.contour1 = edge
151                 edge.verts[1] = arc.next
152 
153             if arc.contour1:
154                 arc.contour1.setEnd(event.point)
155             if arc.contour2:
156                 arc.contour2.setEnd(event.point)
157 
158             if arc.prev:
159                 self.check_circle_event(arc.prev, event.x)
160             if arc.next:
161                 self.check_circle_event(arc.next, event.x)
162 
```

***Рис. 16**. Обробка події кола*

А от обробка події точки більш складна, так як вимагає вставлення нової дуги і виявлення її взаємного розташування з сусідніми дугами (див. рис. 17)

```
163     def insert_arc(self, point):
164         if self.arc == None:
165             self.arc = Arc(point)
166         else:
167             cur = self.arc
168             while cur:
169                 inter = self.crossing_lp(point, cur)
170                 if inter:
171                     inter_next = self.crossing_lp(point, cur.next)
172                     if cur.next and not inter_next:
173                         cur.next.prev = Arc(cur.center, prev=cur, next=cur.next)
174                         cur.next = cur.next.prev
175                     else:
176                         cur.next = Arc(cur.center, cur)
177                     cur.next.contour2 = cur.contour2
178 
179                     cur.next.prev = Arc(point, prev=cur, next=cur.next)
180                     cur.next = cur.next.prev
181                     cur = cur.next
182 
183                     edge_1 = Edge(inter)
184                     self.result.append(edge_1)
185                     cur.prev.contour2 = cur.contour1 = edge_1
186                     edge_1.verts = [cur.prev, cur.next]
187 
188                     edge_2 = Edge(inter)
189                     self.result.append(edge_2)
190                     cur.next.contour1 = cur.contour2 = edge_2
191                     edge_2.verts = [cur.next, cur]
192 
193                     self.check_circle_event(cur, point.x)
194                     self.check_circle_event(cur.prev, point.x)
195                     self.check_circle_event(cur.next, point.x)
196 
197                     return
198 
199                 cur = cur.next
200 
201             cur = self.arc
202             while cur.next:
203                 cur = cur.next
204             cur.next = Arc(point, cur)
205 
206             x = BORDER[0]
207             y = (cur.next.center.y + cur.center.y)/2
208             start = Point(x, y)
209 
210             edge = Edge(start)
211             cur.contour2 = cur.next.contour1 = edge
212             edge.verts = [cur, cur.next]
213             self.result.append(edge)
214 
```

Рис. 17. *Обробка події точки*

Якщо є хоча б одна дуга, то ми починаємо перебирати всі дуги на пошук дуги, яку перетне горизонтальна пряма, проведена в точці, на якій знаходиться замітаюча лінія. У точці перетину прямої і дуги почнуться два променя, які представляють собою дві частини одного ребра Вороного, і розходяться в обидва боки, поки кожна з них або не зійдеться в вершині, або не піде в нескінченність.

В рядках 171-176 перевіряється, чи є у поточної дуги наступна, і чи перетинає її промінь, який випускається горизонтально з точки. Це потрібно для коректного розділення дуги на дві частини і переадресації посилань сусідніх дуг. В рядках 177-191 створюється нова дуга та нові ребра, які її обмежують. Після цього в рядках 193-195 всі задіяні дуги перевіряються на наявність події круга. Починаючи з 201 рядка починається обробка варіанту коли замітаюча лінія в першу чергу стикається з декількома точками, розташованими вздовж вертикалі. Як ми побачимо пізніше, цей варіант є винятковим, і його треба обробляти окремо.

```
215     # перетин пряма-парабола
216     def crossing_lp(self, point : Point, arc : Arc):
217         if arc == None or point.x == arc.center.x:
218             return None
219 
220         sides = [None, None]
221         if arc.prev:
222             sides[0] = self.crossing_pp(arc.prev, arc, point.x).y
223         if arc.next:
224             sides[1] = self.crossing_pp(arc, arc.next, point.x).y
225 
226         if (arc.prev == None or sides[0] <= point.y) and (arc.next == None or sides[1] >= point.y):
227             crossing = [None, None]
228             crossing[1] = point.y
229             crossing[0] = (arc.center.x**2 + (arc.center.y-crossing[1])**2 - point.x**2)/2/(arc.center.x-point.x)
230             return Point(*crossing)
231         return None
232 
```

Рис. 18. *Метод для визначення точки перетину горизонтальної прямої і параболи, гілки якої напрямлені вліво*

На рисунку 18 визначено метод для знаходження точки перетину між прямою та параболою. Метод нічого не поверне у випадку якщо йому дано неіснуючу дугу, або якщо фокус дуги та поточна точка знаходяться на одній вертикалі, оскільки це призведе до того, попередня дуга не буде розділена на дві – вона буде обрізаною з однієї сторони. Саме тому для вертикального випадку необхідна окрема обробка.

Для того щоб перевіряти перетин прямої тільки з дугою параболи, використовується перевірка, здійснена в рядках 220-226. Для цього знаходяться точки перетину цієї дуги з сусідніми і порівнюються їх y-координату з заданою прямою (оскільки вона горизонтальна). І якщо пряма перетинає дугу параболи, визначається точка перетину. Вже відома y-координата точки перетину, і залишається визначити x-координату:

$$2(x_0 - L)\left(x - \frac{x_0 + L}{2}\right) = (y - y_0)^2 \Rightarrow x = \frac{(y - y_0)^2 + x_0^2 - L^2}{2(x_0 - L)}$$

де L – x-координата директриси параболи (замітаючої лінії), $\left(\frac{x_0+L}{2}; y_0\right)$ – вершина параболи, $x_0 - L$ – параметр параболи, гілки якої напрямлені вліво.

```
233     # перетин парабола-парабола
234     def crossing_pp(self, arc1, arc2, L):
235         # для перетину парабола-парабола використаємо форму x = ay^2 + by + c
236         crossing = [None, None]
237         if arc1.center.x == L:
238             crossing[1] = arc1.center.y
239             crossing[0] = (arc2.center.x ** 2 + (arc2.center.y - crossing[1]) ** 2 - L ** 2)/2/(arc2.center.x - L)
240             return Point(*crossing)
241         elif arc2.center.x == L:
242             crossing[1] = arc2.center.y
243             crossing[0] = (arc1.center.x ** 2 + (arc1.center.y - crossing[1]) ** 2 - L ** 2) / (2 * arc1.center.x - 2 * L)
244             return Point(*crossing)
245         eq1 = [1/2/(arc1.center.x-L), -arc1.center.y/(arc1.center.x-L),
246                1/2/(arc1.center.x-L)*(arc1.center.x**2+arc1.center.y**2-L**2)]
247         eq2 = [1/2/(arc2.center.x-L), -arc2.center.y/(arc2.center.x-L),
248                1/2/(arc2.center.x-L)*(arc2.center.x**2+arc2.center.y**2-L**2)]
249         final_eq = [eq1[0]-eq2[0], eq1[1]-eq2[1], eq1[2]-eq2[2]]
250         if final_eq[0] == 0:
251             crossing[1] = -final_eq[2]/final_eq[1]
252             crossing[0] = eq1[0]*crossing[1]**2 + eq1[1]*crossing[1] + eq1[2]
253             return Point(*crossing)
254         crossing[1] = (-final_eq[1] - (final_eq[1]**2 - 4*final_eq[0]*final_eq[2])**0.5) / 2 / final_eq[0]
255         crossing[0] = eq1[0]*crossing[1]**2 + eq1[1]*crossing[1] + eq1[2]
256         return Point(*crossing)
257
```

***Рис. 19**. Метод для визначення точки перетину двох парабол, гілки яких напрямлені вліво і які мають спільну директрису*

На рисунку 19 показано метод для знаходження точки перетину двох парабол, які мають одну директрису – замітаючу лінію. Окремими випадками є випадки коли одна з вершин знаходиться на директрисі, і відповідна парабола являє собою промінь. Для знаходження точки перетину двох парабол будується та розв'язується система їх рівнянь. Для цього рівняння зводиться до вигляду $x = ay^2 + by + c$ оскільки парабола знаходиться «на боку»:

$$2(x_1 - L)\left(x - \frac{x_1 + L}{2}\right) = (y - y_1)^2 \Rightarrow x = \frac{y^2 - 2y_1 y + y_1^2}{2(x_1 - L)} + \frac{x_1 + L}{2} \Rightarrow x$$
$$= \frac{1}{2(x_1 - L)}y^2 - \frac{y_1}{2(x_1 - L)}y + \frac{y_1^2 + x_1^2 - L^2}{2(x_1 - L)}$$

Звідки отримуємо відповідні коефіцієнти:

$$\begin{cases} a = \dfrac{1}{2(x_1 - L)} \\ b = -\dfrac{y_1}{2(x_1 - L)} \\ c = \dfrac{y_1^2 + x_1^2 - L^2}{2(x_1 - L)} \end{cases}$$

Маючи два таких рівняння, можемо відняти їх, і отримати рівняння вигляду:

$$a'y^2 + b'y + c' = 0$$

де $a' = a_2 - a_1$, $b' = b_2 - b_1$, $c' = c_2 - c_1$. Це звичайне квадратне рівняння яке легко розв'язується. Оскільки обидві параболи знаходяться по одну сторону спільної директриси, таке рівняння завжди матиме розв'язок. Окремим випадком є випадок коли $a' = 0$, що значить, що дві параболи знаходяться на одній вертикалі, і мають єдину точку перетину, цей випадок обробляється в 250 рядку.

```
258     def check_circle_event(self, arc : Arc, x):
259         if arc.event and arc.event.x != x:
260             arc.event.valid = False
261         arc.event = None
262 
263         if not (arc.prev and arc.next):
264             return
265 
266         center, max_x = Point.getCircleCenter(arc.prev.center, arc.center, arc.next.center)
267         if center and max_x > x:
268             arc.event = Event(max_x, center, arc)
269             self.circle_e.push(arc.event)
270 
271     def finish_edges(self):
272         l = BORDER[2] + (BORDER[2] - BORDER[0]) + (BORDER[3] - BORDER[1])
273         cur = self.arc
274         while cur.next:
275             if cur.contour2:
276                 p = self.crossing_pp(cur, cur.next, 2*l)
277                 cur.contour2.setEnd(end=p)
278             cur = cur.next
279 
```

Рис. 20. *Методи для перевірки дуги на подію круга, і для закінчення ребер.*

На рисунку 20 представлені останні два метода класу діаграми Вороного. Перший метод здійснює перевірку дуги на наявність події круга шляхом знаходження центру кола, проведеного через центр цієї дуги, і її сусідніх дуг. Якщо поточна дуга є крайньою, то для неї не існує події круга. Отримана подія заноситься у відповідну пріоритетну чергу.

Для закінчення ребер, які уходять в нескінченність, берегова лінія просто віддаляється на достатньо велику відстань і потрібні ребра добудовуються.

```
280 if __name__ == "__main__":
281     count = int(input())
282     points = []
283     for i in range(count):
284         points.append(list(map(int, input().split())))
285 
286     vor = Vor(points)
287     vor.process()
288     print()
289     for edge in vor.result:
290         print((edge.start.x, edge.start.y), (edge.end.x, edge.end.y))
291 
```

Рис. 21. *Безпосередній виклик програми*

При безпосередньому виклику програми (див. рис. 21) пропонується ввести кількість сайтів та координати сайтів з клавіатури. Далі програма побудує по ним діаграму Вороного і виведе координати ребер.

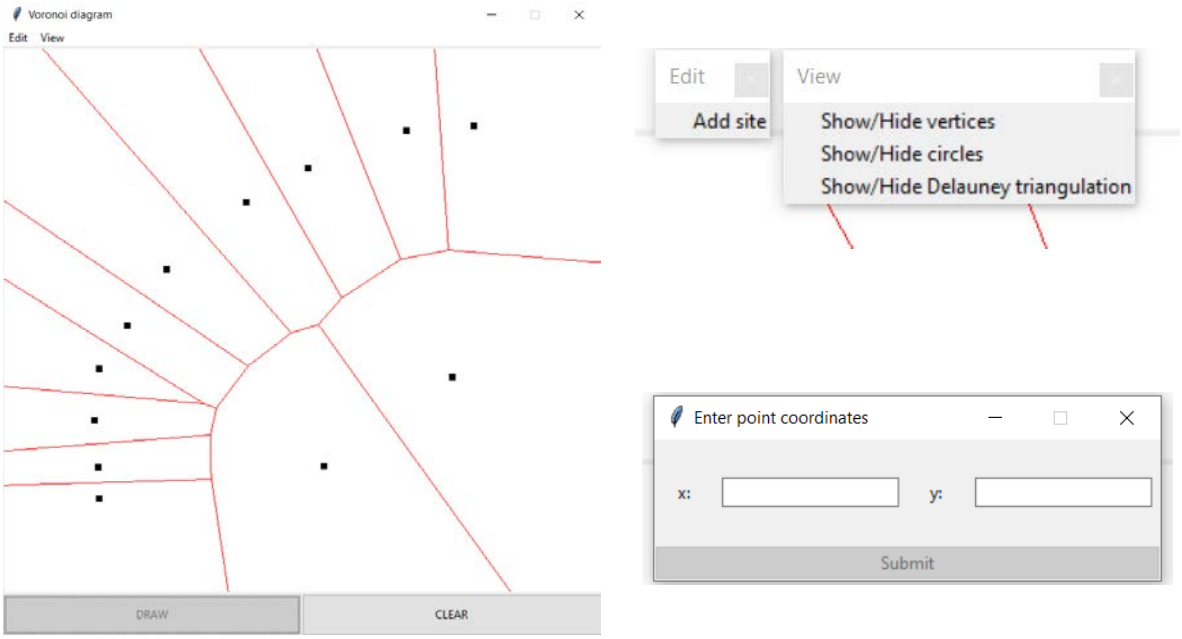

***Рис. 22**. Графічний інтерфейс для роботи з діаграмами* Вороного

На рисунку 22 представлений спеціально розроблений десктопний додаток для комфортної роботи та візуалізації діаграми Вороного. Цей додаток пропонує наступний функціонал:

- Додавання сайтів можливе натисканням ПКМ по робочій області / вибрати в меню *Edit* → *Add site* для введення координати сайту вручну.
- Для побудови діаграми Вороного треба натиснути кнопку DRAW. Після цього можливість ставити точки буде заблоковано
- Після побудови діаграми в меню *View* доступна візуалізація відповідно вершин, кругів (з центром в вершинах діаграми, які проходять по декільком сайтам), тріангуляції Делоне.
- Для очищення робочої області необхідно натиснути кнопку CLEAR. Після цього знову можна ставити точки

Подивимось, як цей додаток реалізовано програмно.

```
1   from fortune import Vor
2   import tkinter as tk
3   from tkinter import ttk
4
5   WIDTH = 640
6
```

***Рис. 23**. Імпортування класу діаграми Вороного та бібліотеки tkinter.*

На рисунку 23 імпортуються всі необхідні модулі і бібліотеки, а також визначається константа розміру екрану (він квадратний).

На рисунку 24 бачимо визначення класу вікна і його конструктор, де в 10-14 рядку визначаються допоміжні змінні для збереження стану робочої області та збереження геометричних структур. Далі описується саме вікно та його конфігурація. В рядках 22-31 створюється меню, його елементи, в врешті решт це меню закріплюється за головним вікном. Також ми створюємо саму робочу область. Далі для позиціонування віджетів, які розставляються у вікні за допомогою методу *pack*, створюється фрейм, в якому описано дві кнопки. Це забезпечує відповідний зовнішній вигляд.

```
7   class Window:
8   
9       def __init__(self, w_master : tk.Tk):
10          self.DISABLE_DRAWING = False
11          self.vor = None
12          self._verts = []
13          self._circles = []
14          self._delauneys = []
15  
16          self.master = w_master
17          self.master.title("Voronoi diagram")
18  
19          self.master.geometry(f"{WIDTH}x{WIDTH}+100+100")
20          self.master.resizable(width=False, height=False)
21  
22          self.main_menu = tk.Menu(self.master)
23          self.main_menu__view = tk.Menu()
24          self.main_menu__view.add_command(label="Show/Hide vertices", command=self.verts)
25          self.main_menu__view.add_command(label="Show/Hide circles", command=self.circles)
26          self.main_menu__view.add_command(label="Show/Hide Delauney triangulation", command=self.delauney)
27          self.main_menu__edit = tk.Menu()
28          self.main_menu__edit.add_command(label="Add site", command=self.add_site)
29          self.main_menu.add_cascade(label="Edit", menu=self.main_menu__edit)
30          self.main_menu.add_cascade(label="View", menu=self.main_menu__view)
31          self.master.config(background="white", menu=self.main_menu)
32  
33          self.canvas = tk.Canvas(self.master, width=WIDTH, height=WIDTH-50, background="white")
34          self.canvas.bind("<Button-3>", lambda event : self.setpoint(event.x, event.y))
35          self.canvas.pack()
36  
37          self.button_frame = tk.Frame()
38          self.button_frame.pack(fill="both", expand=1)
39  
40          self.draw_button = ttk.Button(self.button_frame, text="DRAW", command=self.draw)
41          self.draw_button.pack(side="left", expand=1, fill="both")
42          self.draw_button["state"] = "disabled"
43          self.clear_button = ttk.Button(self.button_frame, text="CLEAR", command=self.clear)
44          self.clear_button.pack(side="right", expand=1, fill="both")
45  
46          self.master.mainloop()
```

***Рис. 24**. Ініціалізація вікна додатку*

```python
def verts(self):
    if self.vor and self.DISABLE_DRAWING and not self._verts:
        for edge in self.vor.result:
            if edge.end:
                self._verts.append(self.canvas.create_oval(edge.end.x-2, edge.end.y-2, edge.end.x+2, edge.end.y+2, fill="blue"))
    elif self._verts:
        for point in self._verts:
            self.canvas.delete(point)
        self._verts = []

def circles(self):
    if self.vor and self.DISABLE_DRAWING and not self._circles:
        for edge in self.vor.result:
            if edge.end and not (edge.end.x < -100000 or edge.end.y < -100000 or edge.end.x > 100000 or edge.end.y > 100000):
                radius = ((edge.end.x-edge.verts[0].center.x)**2+(edge.end.y-edge.verts[0].center.y)**2)**0.5
                self._circles.append(self.canvas.create_arc(edge.end.x-radius, edge.end.y-radius,
                                                            edge.end.x+radius, edge.end.y+radius, style="arc", extent=359.9))
    elif self._circles:
        for circle in self._circles:
            self.canvas.delete(circle)
        self._circles = []

def delauney(self):
    if self.vor and self.DISABLE_DRAWING and not self._delauneys:
        for edge in self.vor.result:
            self._delauneys.append(self.canvas.create_line(edge.verts[0].center.x, edge.verts[0].center.y,
                                                           edge.verts[1].center.x, edge.verts[1].center.y, fill="blue"))
    elif self._delauneys:
        for point in self._delauneys:
            self.canvas.delete(point)
        self._delauneys = []

```

***Рис. 25**. Опис функціоналу елементів вкладки меню View*

На рисунку 25 показано реалізацію функціоналу, який дозволяє візуалізувати деякі елементи діаграми. Відповідні геометричні структури отримуються з діаграми вороного та зберігаються у допоміжні змінні, за допомогою яких їх можна буде прибрати з вікна.

```python
    def add_site(self):
        def var_check():
            if x.get().isnumeric() and y.get().isnumeric():
                submit_["state"] = "normal"
            else:
                submit_["state"] = "disabled"
        def submit(e):
            self.setpoint(float(x.get()), float(y.get()))
            sub_window.destroy()
        sub_window = tk.Toplevel(self.master)
        sub_window.title("Enter point coordinates")
        sub_window.geometry("360x100")
        sub_window.resizable(width=False, height=False)

        frame = ttk.Frame(sub_window)
        frame.pack(side=tk.TOP, expand=1, fill=tk.BOTH)
        x_var = tk.StringVar()
        x_var.trace_add("write", lambda n, i, m : var_check())
        x = ttk.Entry(frame, textvariable=x_var)
        ttk.Label(frame, text="x:", padding=10).pack(side=tk.LEFT, expand=1)
        x.pack(side=tk.LEFT, expand=1)

        y_var = tk.StringVar()
        y_var.trace_add("write", lambda n, i, m : var_check())
        y = ttk.Entry(frame, textvariable=y_var)
        ttk.Label(frame, text="y:", padding=10).pack(side=tk.LEFT, expand=1)
        y.pack(side=tk.RIGHT, expand=1)

        submit_ = ttk.Button(sub_window, text="Submit", width=100)
        submit_.pack(side=tk.TOP)
        submit_.bind("<Button-1>", submit)
        submit_["state"] = "disabled"

        sub_window.mainloop()

```

***Рис. 26.** Функціонал вкладки меню Edit*

На рисунку 26 показана реалізація можливості додавання сайту шляхом задання координат. В цьому методі описані допоміжні функції для валідації форми для введення координат, і функція яка викликається при натисненні клавіші Submit Тут також створюється фрейм, який вістить в собі дві кнопки та дві мітки, під фреймом розташовується кнопка для створення відповідної точки. При її натисканні, вікно знищується а точка додається на екран.

Висновки. Проведений аналіз алгоритмів побудови діаграми Вороного дає зрозуміти, що на теперішній час найефективнішим практичним алгоритмом побудови діаграми є алгоритм Форчуна. Хоча таку ж ефективність дозволяє досягти і алгоритм «розділяй і володарюй», він є більш складним в реалізації, вимагає передбачення низки частинних випадків та досконалого розуміння механізму злиття діаграм.

Даний алгоритм може бути удосконаленим наступними шляхами: використання збалансованих бінарних дерев пошуку, оптимізація використаних структур даних та структури даних самої діаграми Вороного, що потребує додаткових досліджень. Розробка ефективного алгоритму побудови діаграми Вороного відкриває шляхи для ефективного вирішення багатьох інших задач, а також для досліджень в багатьох наукових галузях.

***Література*:**

1. H. Alt, O. Schwarzkopf, “The Voronoi Diagram of Curved Objects”, Available: https://dl.acm.org/doi/pdf/10.1145/220279.220289

2. F. Aurenhammer, “Voronoi Diagrams — A Survey of a Fundamental Geometric Data Structure”, Institute fur Informationsverarbeitung Technische Universitat Graz, Sch iet!stattgasse 4a, Austria, 1991

3. J.R. Sack, J. Urrutia, “Handbook of Computational Geometry”, Elsevier Science B.V., Amsterdam, Netherlands, 2000, pp. 203-280

4. A. Dobrin, “A review of properties and variations of Voronoi diagrams”, Available: https://www.whitman.edu/documents/academics/mathematics/dobrinat.pdf

5. Rex A. Dwyer, “Higher-Dimensional Voronoi Diagrams in Linear Expected Time (Extended Abstract)”, North Carolina State University, 1989

6. M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf, Computational Geometry: algorithms and applications, 2nd rev. ed., Berlin, Germany: Springer-Verlag Berlin Heidelberg, 2000, pp. 147-162

7. W. Pokojski and P. Pokojska, “Voronoi diagrams – inventor, method, applications”, *Polish Cartographical Review*, Val. 50, no. 3, pp. 141-150, Oct. 2018, doi:10.2478/pcr-2018-0009

8. S. K. Dey, “Voronoi diagrams in the max-norm: Algorithms, implementation, and applications”, PhD thesis, Università della Svizzera Italiana, Lugano, Switzerland, 2015

9. L. Kucera, “Visualization of Abstract Algorithmic Ideas”, Available: https://pdfs.semanticscholar.org/ 2cbb/9df169b20ee81f2af676a32702190fcb2283.pdf

10. O. O. Svitlychnyi and S. V. Plotnytskyi, “Osnovy heoinformatyky: navchalnyi posibnyk [Fundamentals of geoinformatics: a study guide]”, Sumy, Ukraine: ВТД “Університетська книга”, 2006

***References*:**

1. H. Alt, O. Schwarzkopf, “The Voronoi Diagram of Curved Objects”, Available: https://dl.acm.org/doi/pdf/10.1145/220279.220289
2. F. Aurenhammer, “Voronoi Diagrams — A Survey of a Fundamental Geometric Data Structure”, Institute fur Informationsverarbeitung Technische Universitat Graz, Sch iet!stattgasse 4a, Austria, 1991
3. J.R. Sack, J. Urrutia, “Handbook of Computational Geometry”, Elsevier Science B.V., Amsterdam, Netherlands, 2000, pp. 203-280
4. A. Dobrin, “A review of properties and variations of Voronoi diagrams”, Available: https://www.whitman.edu/documents/academics/mathematics/dobrinat.pdf
5. Rex A. Dwyer, “Higher-Dimensional Voronoi Diagrams in Linear Expected Time (Extended Abstract)”, North Carolina State University, 1989
6. M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf, Computational Geometry: algorithms and applications, 2nd rev. ed., Berlin, Germany: Springer-Verlag Berlin Heidelberg, 2000, pp. 147-162
7. W. Pokojski and P. Pokojska, “Voronoi diagrams – inventor, method, applications”, *Polish Cartographical Review*, Val. 50, no. 3, pp. 141-150, Oct. 2018, doi:10.2478/pcr-2018-0009
8. S. K. Dey, “Voronoi diagrams in the max-norm: Algorithms, implementation, and applications”, PhD thesis, Università della Svizzera Italiana, Lugano, Switzerland, 2015
9. L. Kucera, “Visualization of Abstract Algorithmic Ideas”, Available: https://pdfs.semanticscholar.org/2cbb/9df169b20ee81f2af676a32702190fcb2283.pdf
10. O. O. Svitlychnyi and S. V. Plotnytskyi, “Osnovy heoinformatyky: navchalnyi posibnyk [Fundamentals of geoinformatics: a study guide]”, Sumy, Ukraine: