Wieloboki Voronoi
porównanie metod konstrukcji

Jakub Ciszewski, Olaf Fertig

Styczeń 2024

1 Dane techniczne

- Język: Python 3.10.11
- System operacyjny: Windows 10
- Procesor: Intel Core i5-6300HQ 2.30 GHz
- Pamięć RAM: 16GB
- Architektura procesora: Skylake
- Wykorzystane biblioteki: numpy, pandas, matplotlib, queue, bitalg, scipy

1.1 Oznaczenia

Jako atrybuty klasy przyjmuję również metody oznaczone dekoratorem @property.
W dokumentacji atrybuty, właściwości i metody opisywane są w poniższy sposób:

- Atrybut: <nazwa>: <typ> - <opis>

- Metoda: <nazwa_metody>(<nazwa_parametru>:<typ_parametru>) - <opis>.

Stosowane są również skróty:

- **BL** - (*Beachline*) skrót przyjęty dla struktury danych przechowującej linie brzegową w algorytmie Fortune'a
- **EQ** - (*Event Queue*) skrót przyjęty dla struktury zdarzeń w algorytmie Fortune'a.
- **np** - skrót przyjęty dla biblioteki numpy
- **plt** - skrót przyjęty dla modułu pyplot z biblioteki matplotlib

2 Dokumentacja

2.1 Aplikacja graficzna

Sekcja ta zawiera informacje dotyczące prostej aplikacji graficznej stworzenej w ramach projektu w celu ułatwienia tworzenia zbiorów testowych.

Aplikacja umożliwia:

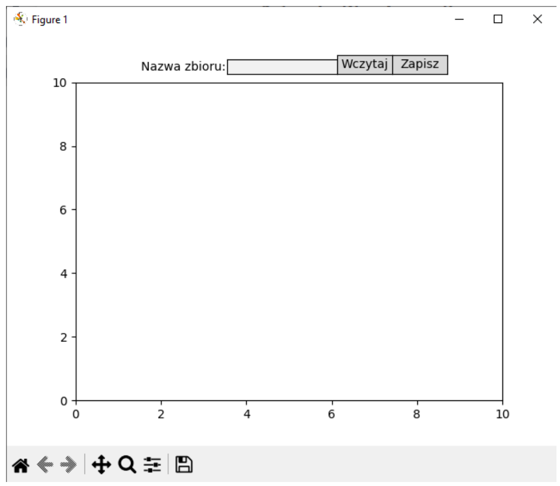
- Zadanie punktów za pomocą myszki.
- Zapis zadanych punktów do pliku tekstowego.
- Wczytanie punktów z pliku tekstowego i ich zwizualizowanie w aplikacji.

2.1.1 Instrukcja obsługi aplikacji:

Uruchomienie
Aby uruchomić aplikacje należy uruchomić pierwszą komórkę pod napisem **Aplikacja** w pliku `main.ipynb`.

Rysunek 1: Miejsce uruchomienia aplikacji graficznej w pliku `main.ipynb`

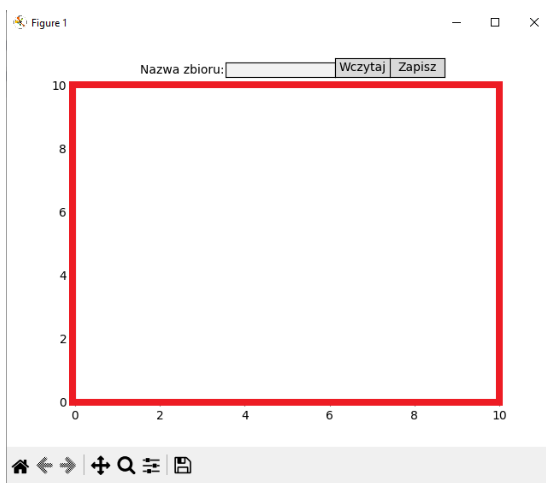
Po uruchomieniu aplikacji na ekranie wyświetli się okno zawierające aplikacje.



Rysunek 2: Wygląd aplikacji w systemie Windows 10

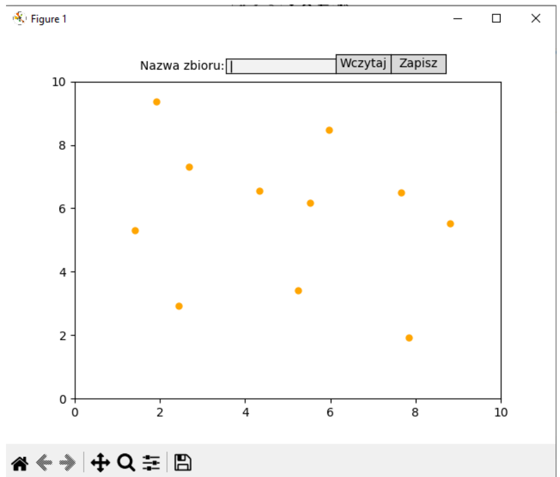
Dodawanie punktów

Aby dodać punkt należy kliknąć myszką w dowolnym punkcie w obszarze zaznaczonym na czerwono na poniższym rysunku.



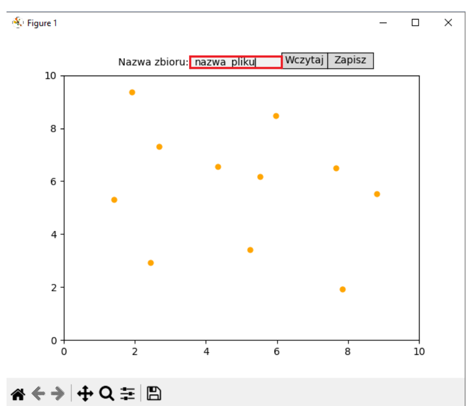
Rysunek 3: Obszar, w którym można zadawać punkty myszką w aplikacji graficznej

Zadane punkty są reprezentowane jako pomarańczowe koła.



Rysunek 4: Wygląd punktów w aplikacji

Zapis punktów
Aby zapisać punkty do pliku należy wprowadzić nazwę pliku do pola tekstowego zaznaczonego na czerwono w poniższym rysunku. Nazwa pliku nie musi kończyć się rozszerzeniem `.txt`.



Rysunek 5: Pole tekstowe służące do przekazania nazwy pliku, z którego program ma wczytać punkty lub do którego ma je zapisać

Po wpisaniu nazwy pliku należy kliknąć przycisk **Zapisz** oznaczony na czerwono w poniższym rysunku. Spowoduje to zapisanie punktów do pliku tekstowego do folderu `points`. Punkty zapisywane są w formacie:

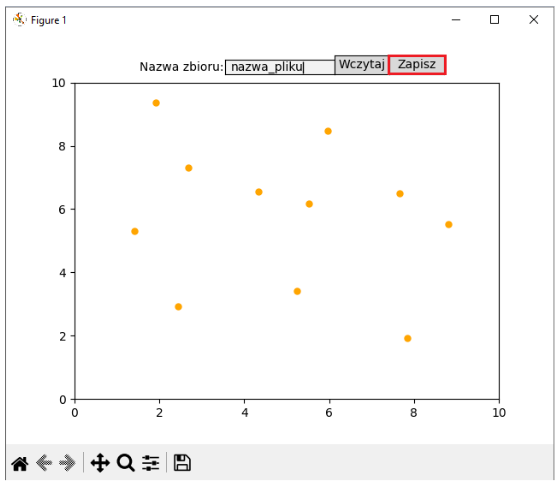
$$x_1 \; y_1$$

$$x_2 \; y_2$$

$$\vdots$$

$$x_n \; y_n$$

, gdzie x_i to współrzędna x i-tego punktu, a y_i to współrzędna y i-tego punktu.



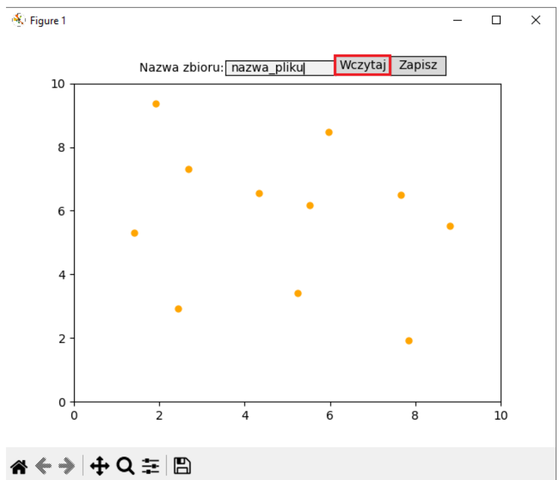
Rysunek 6: Przycisk służący do zapisu punktów do pliku

```
2.681451612903225534e+00 7.310606060606060552e+00
1.411290322580645018e+00 5.308441558441558961e+00
5.241935483870967971e+00 3.414502164502164483e+00
7.661290322580644130e+00 6.498917748917749648e+00
4.334677419354838079e+00 6.553030303030303649e+00
2.439516129032258451e+00 2.927489177489177585e+00
7.842741935483870108e+00 1.926406926406926345e+00
8.810483870967741993e+00 5.524891774891775853e+00
5.524193548387096087e+00 6.174242424242424754e+00
5.967741935483870108e+00 8.474025974025975572e+00
1.915322580645161477e+00 9.366883116883117921e+00
```

Rysunek 7: Zawartość pliku nazwa_pliku.txt

Odczyt punktów
Aby odczytać punkty z pliku plik, zawierający punkty, które chcemy odczytać musi znajdować się w folderze `points` oraz być w formacie opisanym w sekcji poświęconej zapisowi.

W celu odczytania punktów należy do pola tekstowego oznaczonego w 27 wprowadzić nazwę pliku, z którego chcemy wczytać punkty.
Uwaga! Nie należy podawać rozszerzenia pliku. Tylko nazwę.
Następnie należy kliknąć przycisk **Wczytaj** oznaczony na czerwono na poniższym rysunku.



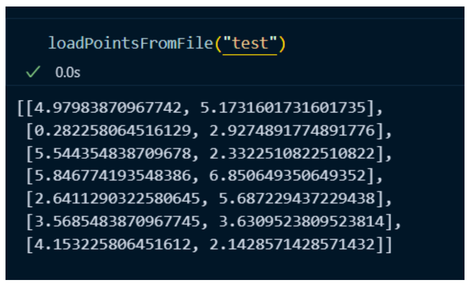
Rysunek 8: Przycisk służący do odczytu punktów z pliku.

Po wciśnięciu przycisku aplikacja wyświetli punkty zapisane w podanym pliku.

Funkcja pomocnicza

Aby umożliwić wygodne przekazanie punktów z pliku tekstowego do konstruktora klasy odpowiedzialnej za uruchomienia algorytmu Fortune'a stworzono funkcję `loadPointsFromFile(filename: str)`, która zwraca punkty z pliku o nazwie `filename.txt` z folderu `points` w postaci listy krotek formatu: $[(x_1, y_1),(x_2, y_2),...,(x_n, y_n)]$, gdzie (x_i, y_i) to współrzędno odpowiednio x i y i-tego punktu.

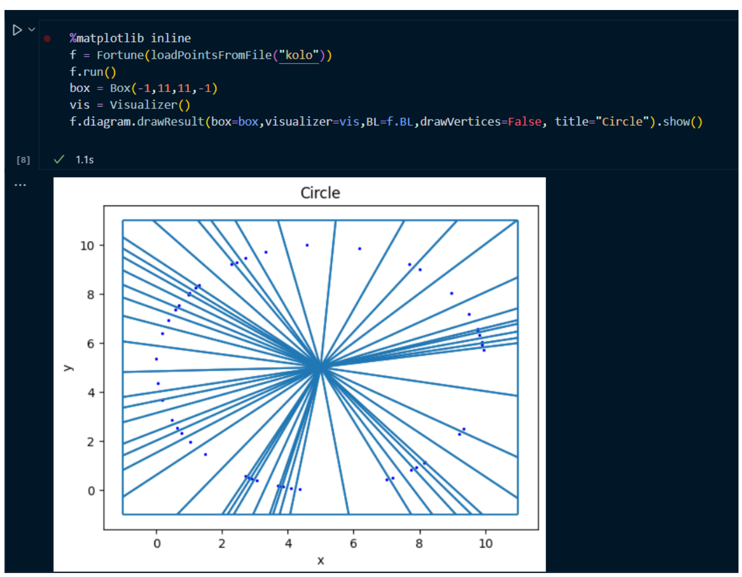
Uwaga! Należy podać samą nazwę pliku bez rozszerzenia.



```
loadPointsFromFile("test")
```
✓ 0.0s

```
[[4.97983870967742, 5.1731601731601735],
 [0.282258064516129, 2.9274891774891776],
 [5.544354838709678, 2.3322510822510822],
 [5.846774193548386, 6.850649350649352],
 [2.6411290322580645, 5.687229437229438],
 [3.5685483870967745, 3.6309523809523814],
 [4.153225806451612, 2.1428571428571432]]
```

Rysunek 9: Przykład wartości zwracanej przez funkcje `loadPointsFromFile`



```
%matplotlib inline
f = Fortune(loadPointsFromFile("kolo"))
f.run()
box = Box(-1,11,11,-1)
vis = Visualizer()
f.diagram.drawResult(box=box,visualizer=vis,BL=f.BL,drawVertices=False, title="Circle").show()
```


Rysunek 10: Przykład użycia funkcji `loadPointsFromFile` do wyświetlenia diagramu Voronoi.

2.2 Dodatkowe funkcje pakietu `Visualizer`

Do pakietu `Visualizer` stworzonego przez koło naukowe **BIT** dodano dwie dodatkowe funkcjonalności:

- Metoda `add_parabola()` klasy `Visualizer` pozwala na dodanie paraboli do obiektu klasy `Visualizer`. Metoda przyjmuje krotkę (`np.ndarray`, `np.ndarray`), które reprezentują odpowiednia argumenty i wartości funkcji opisującej parabole.
- Dodano dodatkową funkcjonalność metody `show_gif()` oraz `show()`. przyjmują one teraz dodatkowe parametry `x_lim: tuple[float,float], y_lim: tuple[float, float]`, które pozwalają ograniczać wyświetlany wykres do podanych rozmiarów.

2.3 Algorytm Fortune'a

Sekcja ta zawiera informację o klasach i ich metodach, które są stosowane przy pracy algorytmu Fortune'a.

2.3.1 Klasa Vector2d

Klasa ta reprezentuje dwuwymiarowy wektor na płaszczyźnie kartezjańskiej. W algorytmie klasa ta częściej interpretowana jest jako punkt.

Atrybuty:

- `x: float` - współrzędna x wektora.
- `y: float` - współrzędna y wektora.
- `orthogonal: Vector2d` - wektor prostopadły, czyli wektor o współrzędnych: $(-y, x)$
- `norm: float` - norma(długość) wektora. Obliczana ze wzroru: $\sqrt{x^2 + y^2}$

Metody:

- `__init__(x: float, y: float)` - konstruktor klasy tworzący nowy obiekt typu `Vector2d`.
- `__add__(other: Vector2d)` - definiuje sposób dodawania dwóch obiektów klasy `Vector2d`. Zwraca obiekt klasy `Vector2d` o współrzędnych `x` = `self.x` + `other.x`, `y` = `self.y` + `other.y`.
- `__sub__(other: Vector2d)` - definiuje sposób odejmowania dwóch obiektów klasy `Vector2d`. Zwraca obiekt klasy `Vector2d` o współrzędnych `x` = `self.x - other.x`, `y` = `self.y - other.y`.

- `mulByScalar(scalar: float)` - zwraca nowy obiekt klasy `Vector2d` o współrzędnych
  `x = scalar * self.x`, `y = scalar * self.y`
- `det(other: Vector2d)` - zwraca wartość wyznacznika dwóch wektorów dwuwymiarowych obliczonego ze wzoru:

$$det(u, v) = u_x * v_y - u_y * v_x$$

- `asTuple()` - zwraca wektor w postaci krotki `(x, y)`

2.3.2 Klasa Vertex

Klasa ta reprezentuje pojedynczy wierzchołek stworzony przy zdarzeniu kołowym w algorytmie Fortune'a.

Atrybuty:

- `point: Vector2d` - współrzędne wierzchołka.

Metody:

- `__init__(point: Vector2d)` - konstruktor klasy tworzący nowy obiekt typu Vertex.

2.3.3 Klasa Site

Klasa reprezentująca pojedynczy punkt zadany na wejściu algorytmu Fortune'a. Wobec punktów reprezentowanych przez obiekty tej klasy algorytm tworzy diagram Voronoi.

Atrybuty:

- `point: Vector2d` - współrzędne punktu.
- `face: Face` - komórka diagramu Voronoi, do której należy punkt reprezentowany przez obiekt klasy `Site`.

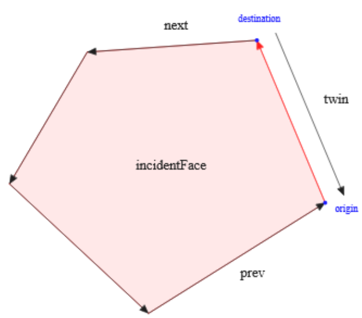
Metody:

- `__init__(point: Vector2d, face: Face)` - konstruktor klasy tworzący nowy obiekt klasy `Site`.

2.3.4 Klasa HalfEdge

Klasa ta reprezentuje pojedynczą półkrawędź. Półkrawędź to struktura danych, reprezentująca jedną z dwóch krawędzi skierowanych, które tworzą pełną krawędź w wielokącie. Klasa to wykorzystywana jest przez algorytm w celu przechowywania informacji o tworzonych komórkach diagramu Voronoi. Ułatwia ona poruszanie się krawędziach komórek diagramu. Półkrawędź przechowuje informacje o:

- następniku - następnej krawędzi
- poprzedniku - poprzedniej krawędzi
- wierzchołku startowym
- wierzchołku końcowym
- bliźniaku (z ang. *twin*) - czyli o półkrawędzi skierowanej w przeciwną stronę.
- komórce, której jest krawędzią



Rysunek 11: Przykładowa komórka zbudowana z półkrawędzi

Atrybuty:

- `next: HalfEdge` - następnik półkrawędzi
- `prev: HalfEdge` - poprzednik półkrawędzi
- `origin: Vertex` - wierzchołek początkowy
- `destination: Vertex` - wierzchołek końcowy
- `twin: HalfEdge` - bliźniacza półkrawędź
- `incidentFace: Face` - komórka otaczana przez krawędź

Metody:

- `__init__()` - konstruktor klasy tworzący nowy obiekt typu `HalfEdge`. Ustawia wszystkie atrybuty na wartość `None`.

2.3.5 Klasa Face

Klasa reprezentująca pojedynczą komórkę diagramu Voronoi.

Atrybuty:

- `site: Site` - punkt, który zawiera się w komórce
- `outerComponent: HalfEdge` - jedna z półkrawędzi otaczających te komórke

Metody:

- `__init__(site: Site, outerComponent: HalfEdge)` - konstruktor klasy tworzący nowy obiekt typu `Face`.

2.3.6 Klasa Event

Klasa reprezentująca zdarzenie. Klasa zawiera dwa pola statyczne: `CIRCLE = 0` oraz `SITE = 1`. Używane są do rozróżnienia zdarzenia punktowego oraz zdarzenie kołowego. Wszystkie zdarzenia punktowe znamy na samym początku algorytmu – są to punkty, dla których tworzony jest diagram Voronoi. Zdarzenia kołowe powstają podczas działania algorytmu. Są one tworzone, gdy 3 punkty tworzą koło.

Atrybuty:

- `type: CIRCLE/SITE` - typ zdarzenia
- `site: Site` - punkt, w którym wystąpiło zdarzenie. Dla zdarzenia punktowego jest to po prostu punkt, a dla zdarzenia kołowego środek okręgu stworzonego przez 3 punkty.
- `arc: Arc` - (zdarzenie kołowe) łuk zanikający podczas zdarzenia.
- `lowest: float` - (zdarzenie kołowe) współrzędna y najniższego punktu w kole stworzonym przez 3 punkty.
- `active: bool` - (zdarzenie kołowe) status aktywności zdarzenia: `True` - aktywne; `False` - nieaktywne.
- `radius: float` - długość promienia koła stworzonego przez 3 punkty.
- `y_to_comp` - zmienna wykorzystywana do porównania dwóch obiektów klasy `Event`. W przypadku, gdy typ zdarzenia to `SITE` zwracana jest współrzędna y punktu wystąpienia zdarzenia. Gdy zdarzenie jest typu `CIRCLE` to zwracana jest wartość atrybutu `lowest`.

Metody:

- `__init__(type: CIRCLE/SITE, site: Site, lowest: float, arc: float, radius: float)` - konstruktor klasy tworzący nowy obiekt klasy `Event`. Parametry `lowest, arc, radius` są domyślnie ustawione na `None`.
- `remove()` - metoda zmienia atrybut `active` na `False` tym samym dezaktywując zdarzenie
- `__lt__(other: Event)` - metoda definiująca sposób sprawdzenia czy obiekt jest mniejszy od innego obiektu klasy `Event`. Sprawdza czy atrybut `-self.y_to_comp` jest mniejszy od `-other.y_to_comp`. Jeśli oba atrybuty są sobie równe to porównuje `self.site.point.x` i `other.site.point.x`. Porównywanie zdefiniowane jest w ten sposób z racji na to, że zdarzenia w algorytmie Fortune'a są pobierane ze struktury zdarzeń w kolejności malejącej względem współrzędnej y zdarzeń. Jeśli natomiast oba zdarzenia mają taką samą współrzędną y to struktura zdarzeń zwraca zdarzenie o mniejszej współrzędnej x. Pozostałe metody używane do porównania dwóch obiektów klasy `Event` zostały stworzone w tym samym celu.
- `__gt__(other: Event)` - metoda definiująca sposób sprawdzenia czy obiekt jest większy od innego obiektu klasy `Event`. Sprawdza czy atrybut `-self.y_to_comp` jest większy od `-other.y_to_comp`. Jeśli oba atrybuty są sobie równe to porównuje `self.site.point.x` i `other.site.point.x`.
- `__le__(other: Event)` - metoda definiująca sposób sprawdzenia czy obiekt jest mniejszy bądź równy od innego obiektu klasy `Event`. Sprawdza czy atrybut `-self.y_to_comp` jest mniejszy bądź równy od `-other.y_to_comp`.
- `__ge__(other: Event)` - metoda definiująca sposób sprawdzenia czy obiekt jest wiekszy bądź równy od innego obiektu klasy `Event`. Sprawdza czy atrybut `-self.y_to_comp` jest większy bądź równy od `-other.y_to_comp`.
- `__eq__(other: Event)` - metoda definiująca sposób sprawdzenia czy dwa obiekty klasy `Event` są sobie równe. Zwraca `True` tylko w przypadku gdy obie współrzędne atrybutu `site` są równe.
- `__ne__(other: Event)` - metoda definiująca sposób sprawdzenia czy dwa obiekty klasy `Event` nie są sobie równe. Metoda ta wywołuje metodę `__ge__` i zwraca jej zanegowany wynik.

2.3.7 Klasa Arc

Klasa reprezentująca łuk tworzący linię brzegową. Jest ona węzłem(z ang. *Node*) w strukturze drzewiastej odpowiadającej za przechowywanie informacji o linii brzegowej(z ang. *Beachline*). Posiada 4 pola statyczne:
`RED = 0, BLACK = 1, LEFT = 2, RIGHT = 3`. `RED` oraz `BLACK` odpowiadają za rozróżnienie kolorów węzła w linii brzegowej z racji na to, że w implementacji wykorzystałem drzewo czerwono-czarne. `LEFT` oraz `RIGHT` odpowiadają za rozróżnienie

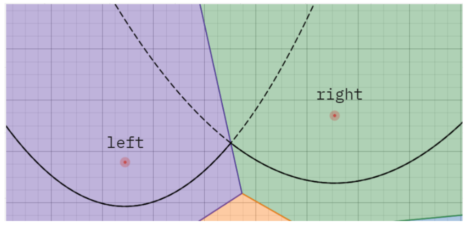
czy łuk zmierza do ∞ czy do $-\infty$ 12.

Atrybuty:

- `parent: Arc` - rodzic łuku w strukturze stanu(BL)
- `left: Arc` - lewo dziecko łuku w strukturze stanu
- `right: Arc` - prawe dziecko łuku w strukturze stanu
- `site: Site` - centrum łuku, od którego się rozchodzi. Centrami łuków są punkty według, których tworzymy diagram Voronoi.
- `leftHalfEdge: HalfEdge` - lewa półkrawędź tworzona przez łuk
- `rightHalfEdge: HalfEdge` - prawo półkrawędź tworzona przez łuk
- `next: Arc` - następnik łuku w strukturze stanu, czyli najbliższy łuk po prawej stronie
- `prev: Arc` - poprzednik łuku w strukturze stanu, czyli najbliższy łuk po lewej stronie
- `color: BLACK/RED` - kolor węzła reprezentowanego przez łuk w strukturze stanu
- `side: LEFT/RIGHT` - strona, w którą rozchodzi się dany łuk

Metody:

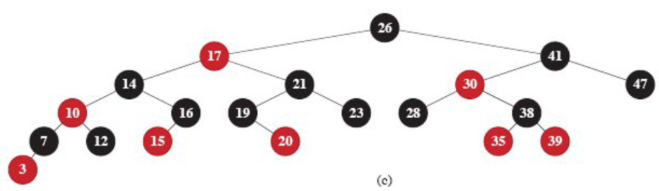
- `__init__()` - konstruktor klasy tworzący nowy obiekt klasy `Arc`
- `get_plot(x: np.ndarray, sweepline: float)` - metoda oblicza aktualny wzór paraboli opisującej łuk, a następnie dla każdego argumentu z `x` oblicza wartość funkcji kwadratowej opisującej ten łuk i przypisuje jej wartość do tablicy `y`. Zwraca krotkę `(x,y)`. y obliczany wg wzoru: $a =$ `self.site.point.x`
  $b =$ `self.site.point.y`
  $k =$ `sweepline`
  $$y = \frac{x^2 - 2ax + a^2 + b^2 - k^2}{2(b-k)}$$
  Więcej na ten temat tutaj.
  Jeśli `sweepline = self.site.point.y` wtedy zwraca `None`



Rysunek 12: Łuk `left` jest nieograniczony z lewej strony - `LEFT`, łuk `right` jest nieograniczony z prawej strony - `RIGHT`

2.3.8 Klasa Beachline

Klasa reprezentuje drzewo czerwono-czarne opisujące linie brzegową tworzoną przez łuki, czyli obiekty klasy `Arc`. Wybrałem te implementację ze względu na prostszą obsługę względem popularniejszego podejścia, czyli drzewa AVL z rozróżnieniem dwóch typów węzłów: wewnętrznych, które przechowują najczęściej obiekty klasy `Breakpoint` reprezentujące punkty załamania oraz liści, czyli najczęściej obiekty klasy `Arc` reprezentujące parabole. Przy wybranym przeze mnie podejściu wszystkimi węzłami są obiekty klasy `Arc`.
Klasa posiada jedno pole statyczne `eps`=10^{-9}, które przechowuje precyzję z jaką będą wykonywane porównania współrzędnych w obiekcie klasy `Beachline`.
Pomysł został zaczerpnięty z tego artykułu oraz dobrze opisany w tej prezentacji.
Większość metod drzewa została zaczęrpnięta z tej książki.



Rysunek 13: Przykładowe drzewo czerwono-czarne

Atrybuty:

- `guardian: Arc` - atrybut przechowujący węzeł pusty. Jest on zawsze czarny zgodnie z zasadami drzewa czerwono-czarnego.
- `root: Arc` - korzeń drzewa.
- `isEmpty: bool` - informacja czy drzewo jest puste. Drzewo uważane jest za puste jeśli `root = guardian`.

Metody:

- `__init__()` - konstruktor klasy tworzący nowy obiekt klasy `Beachline`. Tworzy nowy obiekt klasy `Arc`, ustawia jego kolor na czarny i przypisuje go atrybutowi `guardian`.

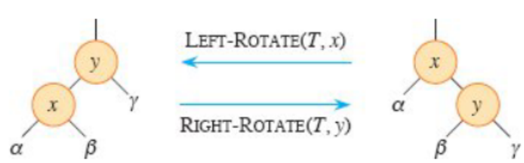
- `createArc(site: Site, side: LEFT/RIGHT)` - metoda tworzy nowy łuk(`Arc`) w punkcie `site` idący w strone `side`. Przypisuję jego atrybutom: `left,right,parent,prev,next` wartość `self.guardian`. Ustawia kolor łuku na czerwony. Zwraca nowo powstały łuk.
- `setRoot(x: Arc)` - ustawia łuk `x` jako korzeń drzewa oraz upewnia się, że korzeń zgodnie z zasadami drzewa czerwono-czarnego jest czarny.
- `getMostLeft()` - zwraca łuk znajdujący się najbardziej na lewo względem innych łuków. Przykładowo (patrz 13) funkcja dla tego drzewa zwróciłaby węzeł 3.
- `minimum(x: Arc)` - zwraca najmniejszy łuk w poddrzewie, którego korzeniem jest `x`. Przykładowo (patrz 13) funkcja dla węzła 41 zwróciłaby węzeł 28.
- `leftRotate(x: Arc)` - obraca poddrzewo, którego korzeniem jest `x` w lewo 14.
- `rightRotate(x: Arc)` - obraca poddrzewo, którego korzeniem jest `x` w prawo 14.
- `removeFixup(x: Arc)` - naprawia strukture drzewa, zaczynając od `x`, po ówczesnym usunięciu węzła z drzewa tak aby zachowane zostały zasady drzewa czerwono-czarnego.
- `insertFixup(x: Arc)` - naprawia strukture drzewa, zaczynając od `x`, po ówczesnym dodaniu węzła do drzewa, tak aby zachowane zostały zasady drzewa czerwono-czarnego.
- `transplant(u: Arc, v: Arc)` - zamienia poddrzewo `u` na poddrzewo `v`, jednak **nie** utrzymuje zasad drzewa czerwono-czarnego. 15
- `insertBefore(x: Arc, y: Arc)` - wstawia łuk `x` jak poprzednik(największy mniejszy łuk) łukiem `y` utrzymując zasady drzewa czerwono-czarnego. 18
- `insertAfter(x: Arc, y: Arc)` - wstawia łuk `x` po łuku `y` utrzymując zasady drzewa czerwono-czarnego. 17
- `replace(x: Arc, y: Arc` - zamienia pojedynczy węzeł `x` w drzewie na węzeł `y`.16
- `remove(z: Arc)` - usuwa z drzewa węzeł `z`.
- `locateArcAbove(point: Vector2d, sweepline: float)` - znajduję i zwraca łuk znajdujący się nad punktem `point` dla położenia miotły `sweepline`. Wykorzystuję metodę `computeBreakpoint` jako klucz w procesie szukania w drzewie.

- `static computeBreakpoint(point1: Vector2d, point2: Vector2d, sweepline: float, side: LEFT/RIGHT)` - metoda statyczna obliczająca współrzędną x punktu przecięcia dwóch łuków w punktach `point1, point2` dla pozycji miotły `sweepline` oraz strony rozchodzenia się łuku zadanego punktem `point1 - side`. Metoda rozpatruje przypadki:
`point1.x` = x_1, `point1.y` = y_1, `point2.x` = x_2, `point2.y` = y_2, `sweepline` = l

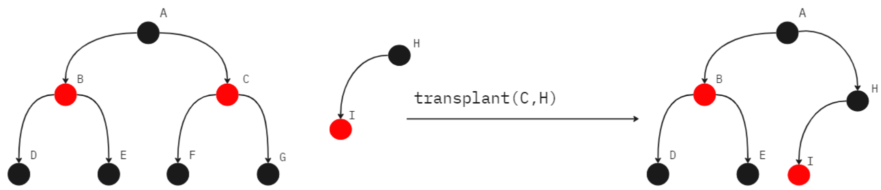
  - $y_1 = y_2$ – łuki się nigdy nie przetną 19. W tej sytuacji metoda zwróci $(x_1 + x_2)/2$ jeśli $x_1 < x_2$. W przeciwnym wypadku zwróci $-\infty$ jeśli parabola generowana przez `point1` rozchodzi się w lewo lub ∞ gdy w prawo.
  - $y_1 =$ `sweepline` - punkt `point1` leży na miotle. Metoda zwróci x_1.20
  - $y_2 =$ `sweepline` - punkt `point2` leży na miotle. Metoda zwróci x_2.20
  - Jeśli nie występuje żaden z powyższych przypadków metoda oblicza punkt przecięcia.21

$$d_1 = 1/(2 * (y_1 - l))$$
$$d_2 = 1/(2 * (y_2 - l))$$
$$a = d_1 - d_2$$
$$b = 2 * (x_2 * d_2 - x_1 * d_1)$$
$$c = (y_1^2 + x_1^2 - l^2) * d_1 - (y_2^2 + x_2^2 - l^2) * d_2$$
$$\Delta = b^2 - 4 * a * c$$
$$x_0 = \frac{-b + \sqrt{\Delta}}{2 * a}$$

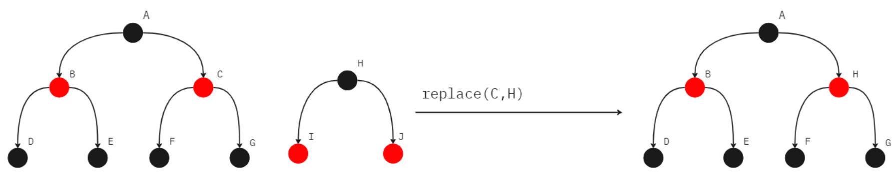
Zwraca x_0 – prawe przecięcie.



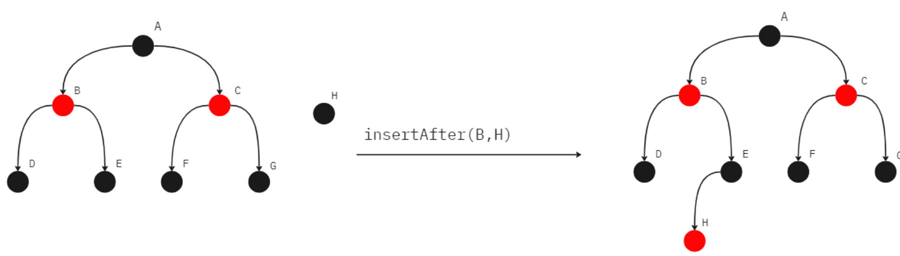
Rysunek 14: Zobrazowanie operacji `leftRotate` oraz `rightRotate`



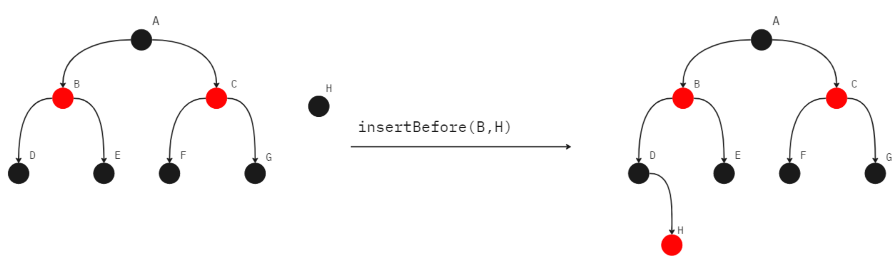
Rysunek 15: Zobrazowanie operacji `transplant`



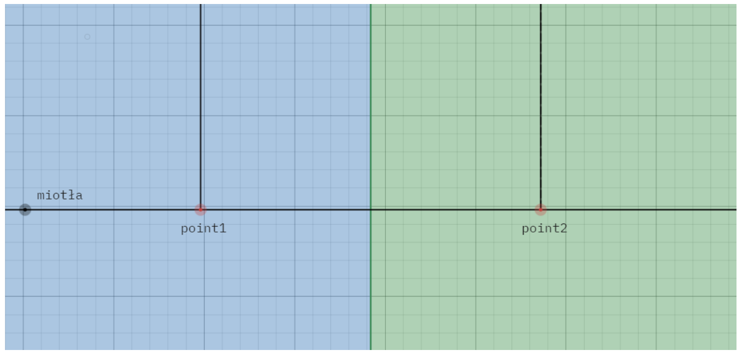
Rysunek 16: Zobrazowanie operacji `replace`



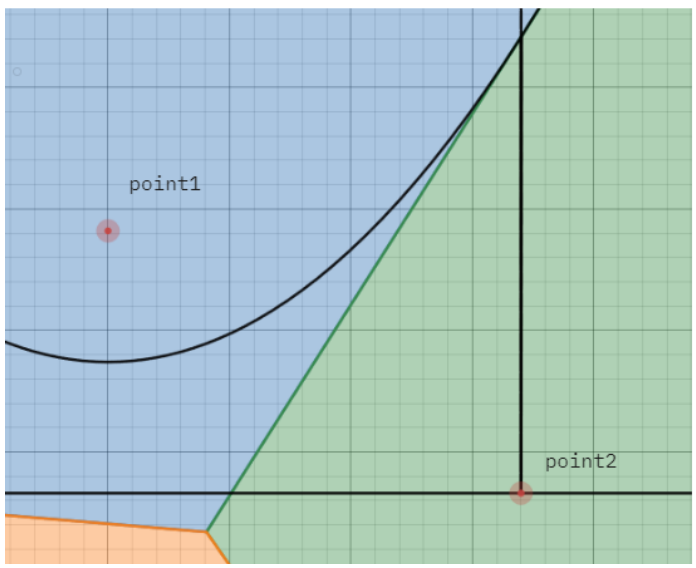
Rysunek 17: Zobrazowanie operacji `insertAfter`



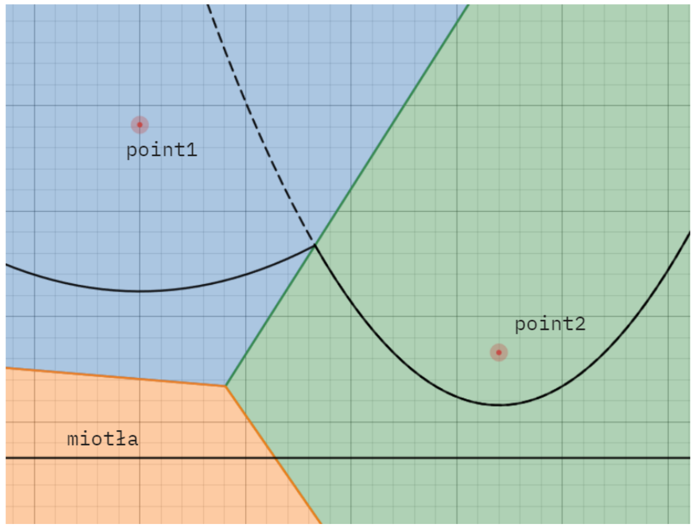
Rysunek 18: Zobrazowanie operacji `insertBefore`



Rysunek 19: Zobrazowanie przypadku gdy dwa punkty mają tą samą współrzędną y



Rysunek 20: Zobrazowanie przypadku gdy jeden punkt leży na miotle



Rysunek 21: Przecięcie dwóch paraboli

2.3.9 Klasa Intersection

Klasa reprezentująca punkt przecięcia półprostej z prostokątem reprezentowanym przez obiekt klasy `Box`.

Atrybuty:

- `point: Vector2d` - współrzędne przecięcia.

2.3.10 Klasa Box

Klasa reprezentująca prostokąt na płaszczyźnie. Używana przy tworzeniu diagramu w algorytmie Fortune'a do ograniczenia nieskończonych krawędzi.

Atrybuty:

- `left: float` - lewy bok prostokąta interpretowany jako prosta o równaniu $x =$ `left`.
- `right: float` - prawy bok prostokąta interpretowany jako prosta o równaniu $x =$ `right`.
- `top: float` - górny bok prostokąta interpretowany jako prosta o równaniu $y =$ `top`.
- `bottom: float` - dolny bok prostokąta interpretowany jako prosta o równaniu $y =$ `bottom`.

Metody:

- `contains(point: Vector2d)` - zwraca `True`, gdy punkt znajduję się w środku prostokąta reprezentowanego przez te klase.
- `getIntersection(origin: Vector2d, direction: Vector2d)` - metoda oblicza punkt przecięcia półprostej zadanej jako punkt początkowy `origin` i wektora kierunkowego `direction` z prostokątem reprezentowanym przez te klase. Metoda zwraca obiekt klasy `Intersection`.

2.3.11 Klasa Diagram

Klasa odowiedzialna za przechowywanie tworzonego diagramu Voronoi oraz za jego wizualizacje.

Atrybuty:

- `halfEdges: list[HalfEdge]` - lista krawędzi diagramu.
- `vertices: list[Vertex]` - lista wierzchołków diagramu.
- `faces: list[Face]` - lista komórek diagramu.
- `sites: list[Site]` - lista punktów wejściowych diagramu.
- `arcs: list[Arc]` - lista łuków tworzących linie brzegową diagramu.
- `visibleParabolas: list[Parabola]` - lista wszystkich aktualnie wyświetlanych paraboli przez `Visualizer`.
- `visibleHalfEdges: list[LineSegment]` - lista wszystkich aktualnie wyświetlanych krawędzi przez `Visualizer`.
- `visibleHalfEdgesCoord: list[tuple[tuple[float, float]]]` - lista wszystkich aktualnie wyświetlanych krawędzi przez `Visualizer` w formacie $(x_i, y_i), (x_j, y_j)$, gdzie x_i, y_i to współrzędne pierwszej początka odcinka, a (x_j, y_j) to współrzędne końca odcinka. Początek rozumiany jest jako punkt opisywany przez atrybut `origin` obiektu klasy `HalfEdge`, a koniec jako punkt opisywany przez atrybut `destination` lub przez punkt przecięcia(`Intersection`) z prostokątem(`Box`).
- `visibleSweepline: LineSegment` - aktualnie wyświetlana przez `Visualizer` prosta wizualizująca aktualne położenie miotły.
- `circle: tuple[Vector2d, float]` - krotka formatu: $((x_i, y_i), r)$, gdzie (x_i, y_i) - współrzędne środka koła; r - promień koła, opisująca koło do zwizualizowania.
- `visisbleCircle: Circle` - aktualnie wyświetlane koło przez `Visualizer`.

Metody:

- `__init__(points: list[tuple[float, float]])` - konstruktor klasy `Diagram`. Na podstawie listy punktów `points` tworzy listę punktów początkowych(`Site`) oraz listę komórek diagramu(`Face`).
- `createHalfEdge(face: Face)` - tworzy nową krawędź diagramu(`HalfEdge`) oraz nadaje przynależność nowej krawędzi do komórki `face`. Jeślia atrybut `outerComponent` komórki `face` jest równy `None`, przypisuję temu atrybutowi nowo powstałą krawędź. Na koniec dopisuję nową krawędź do listy krawędzi diagramu oraz zwraca krawędź.
- `createVertex(point: Vector2d, add: bool)` - tworzy nowy wierzchołek diagramu(`Vertex`) w punkcie `point` oraz jeśli `add=True` to dodaję nowy wierzchołek do listy wszystkich wierzchołków diagramu.
- `drawResult(box: Box, visualizer: Visualizer, BL: Beachline, drawVertices: bool, title: str)` - metoda tworzy wizualizacje końcowego diagramu Voronoi. Parametr `box` określa prostokąt, którym ma być ograniczony diagram. Parametr `visualizer` to obiekt klasy `Visualizer`, który jest odpowiedzialny za wyświetlanie całego diagramu to on też będzie przechowywał wizualizacje. Parametr `BL` to linia brzegowa, z której metoda uzyskuję kluczowe informację o krawędziach diagramu idących w nieskończoność. Parametr `drawVertices` określa czy `visualizer` wyświetlać również wierzchołki diagramu. Parametr `title` ustala tytuł diagramu. Ta metoda powinna być wywołana po skończeniu działania algorytmu Fortune'a. Metoda zwraca `visualizer` przechowujący wizualizacje diagramu.
- `drawStep(box: Box, sweepline: float, vis: Visualizer, BL: Beachline, message: str)` - metoda wyświetlająca wizualizacje aktualnego stanu diagramu dla aktualnego położenia miotły `sweepline` oraz dla aktualnego stanu linii brzegowej `BL`. Parametr `vis` to obiekt klasy `Visualizer`, który będzie użyty do wyświetlenia aktualnego stanu diagramu. Parametr `box` służy do ograniczenia diagramu prostokątem reprezentowanym przez obiekt klasy `Box`. Parametr `message` ustala tytuł diagramu. Metoda w przeciwności do metody `drawResult` nie zwraca nic, lecz wywołuję metode `show` klasy `Visualizer` i wyświetla wizualizacje.
- `drawArcs(box: Box, sweepline: float, vis: Visualizer, guardian: Arc)` - metoda usuwająca z `vis` oraz `visibleParabolas` wszystkie dotychczasowo narysowane parabole. Metoda następnie dodaje do `vis` wszystkie parabole z listy `arcs`. Parametr `box` służy do ograniczenia paraboli. Parametr `guardian` jest do sprawdzenia czy parabola ma poprzednika i następnika.
- `drawSites(vis: Visualizer)` - metoda dodaje do `vis` wszystkie punkty z listy `sites`.
- `drawVertices(box: Box, vis: Visualizer)` - metoda dodająca do `vis` wszystkie wierzchołki diagramu, które zawierają się w prostokącie reprezentowanym przez `box` do `vis`.

- `drawHalfEdges(box: Box, vis: Visualizer)` - metoda dodająca do `vis` i `visibleHalfEdges` wszystkie krawędzie diagramu przechowywane w liście `halfEdges`, których znane są już punkty początkowy(`origin`) i końcowy(`destination`).
- `drawSweepline(sweepline: float, box: Box, vis: Visualizer)` - metoda usuwająca(jeśli istnieje) aktualny odcinek reprezentujący miotłe z `vis` oraz dodająca do `vis` nowy odcinek wizualizujący położenie miotły `sweepline`. Parametr `box` ogranicza odcinek reprezentujący miotłę tak, aby zawierała się w prostokącie reprezentowanym przez ten obiekt.
- `drawCircle(center: Vector2d, radius: float, vis: Visualizer, box: Box)` - metoda usuwa(o ile istnieją) wszystkie okręgi listy `visibleCircle` z `vis`, a następnie dodaje do `vis` okrąg o środku w punkcie reprezentowanym przez `center` oraz o promieniu `radius`. Metoda wywołuje również metodę `drawVertices` i z racji na to musi przekazać jej parametr `box`.

2.3.12 Klasa Fortune

Klasa odpowiadająca za stworzenie diagramu Voronoi przy pomocy algorytmu Fortune'a.

Atrybuty:

- `BL: Beachline` - struktura stanu przechowująca aktualny stan linii brzegowej.
- `EQ: PriorityQueue` - kolejka priotytetowa z biblioteki `queue` będąca struktura zdarzeń algorytmu.
- `diagram: Diagram` - klasa przechowująca tworzony diagram Voronoi.
- `sweepline: float` - aktualna pozycja miotły.
- `vertices: list[Vertex]` - wierzchołki diagramu.
- `edges: list[HalfEdge]` - krawędzie diagramu.
- `faces: list[Face]` - komórki diagramu.

Metody:

- `run(visualize: bool, box: Box, vis: Visualizer)` - metoda uruchamiająca działanie algorytmu Fortune'a. Na początku tworzy zdarzenia i dodaje je do `EQ`, a następnie uruchamia algorytm. Parametr `visualize` decyduje o tym czy kroki algorytmu będą wizualizowane. Jeśli zostanie on ustawiony na `True` wtedy działanie algorytmu zostanie zwizualizowane. W takim przypadku należy też podać parametr `box` – prostokąt, który będzie ograniczać diagram oraz `vis` – obiekt, który będzie odpowiedzialny za wizualizacje procesu tworzenia diagramu.

- `handleSiteEvent(event: Event)` - metoda obsłgująca zdarzenie punktowe opisane przez `event`.
- `handleCircleEvent(event: Event)` - metoda obsługująca zdarzenie kołowe opisane przez `event`.
- `breakArc(arc: Arc, site: Site)` - metoda rozdzielająca łuk `arc` łukiem rozchodzącym się od punktu `site`. Zwraca łuk rozchodzący się punktu `site`. 22
- `removeArc(arc: Arc, vertex: Vertex)` - metoda usuwa łuk `arc` z linii brzegowej oraz ustawia `vertex` jako punkty końcowe i początkowe odpowiednim półkrawędziom wyznaczanym przez usuwany łuk. Następnie tworzy dwie nowe krawędzie wychodzące z tego punktu. 23
- `addEdge(left: Arc, right: Arc)` - metoda tworzy nowe półkrawędzie tworzone przez łuki `left` i `right` ustawia nowe łuki jako swoje bliźniaki(`twin`).
- `setOrigin(left: Arc, right: Arc, vertex: Vertex)` - metoda ustawia punkt początkowy(`origin`) prawej krawędzi(`rightHalfEdge`) łuku `left` oraz punkt końcowy(`destination`) lewej krawędzi(`leftHalfEdge`) łuku `right` na `vertex`.
- `setDestination(left: Arc, right: Arc, vertex: Vertex)` - metoda ustawia punkt końcowy(`destination`) prawej krawędzi(`rightHalfEdge`) łuku `left` oraz punkt początkowy(`origin`) lewej krawędzi(`leftHalfEdge`) łuku `right` na `vertex`.
- `setPrevHalfEdge(prev: HalfEdge, next: HalfEdge)` - ustanawia `next` następnikiem `prev` i `prev` poprzednikiem `next`.
- `isMovingRight(left: Arc, right: Arc)` - sprawdza czy krawędź tworzona przez przecięcie łuków `left` i `right` zmierza w prawą strone. Sprawdza to porównując współrzędne y punktów, od których rozchodzą się parabole. Jeśli krawędź rozchodzi się w prawo zwraca `True`.24, 25
- `getInitialX(left: Arc, right: Arc, movingRight: bool)` - zwraca współrzędną x centrum łuku `left` jeśli krawędź tworzona przez łuki `left` i `right` rozchodzi się w prawo lub współrzędną x łuku `right` jeśli krawędź rozchodzi się w lewo. Ta metoda jest przydatna w metodzie `addEvent`. Przykładowo w sytuacji zobrazowanej rysunku 24 funkcja zwróci współrzędną x punktu centralnego prawej paraboli, a w sytuacji zobrazowanej na rysunku 25 lewej.
- `computeConvergencePoint(point1: Vector2d, point2: Vector2d, point3: Vector2d)` - metoda wyznaczająca środek, promień oraz najniższy punkt okręgu zadanego punktami `point1, point2, point3`. W przypadku gdy punkty są współliniowe zwraca $((\infty, \infty), 0, \infty)$.

- `addEvent(left: Arc, middle: Arc, right: Arc)` - metoda sprawdzająca czy okrąg stworzony przez centra łuków `left, middle, right` tworzą okrąg, który generuje zdarzenie kołowe. Jeśli tak to dodaje do strukury zdarzeń nowe zdarzenie kołowe stworzone przez ten okrąg. Warunki, które muszą zostać spełnione, aby okrąg został zdarzeniem kołowym:

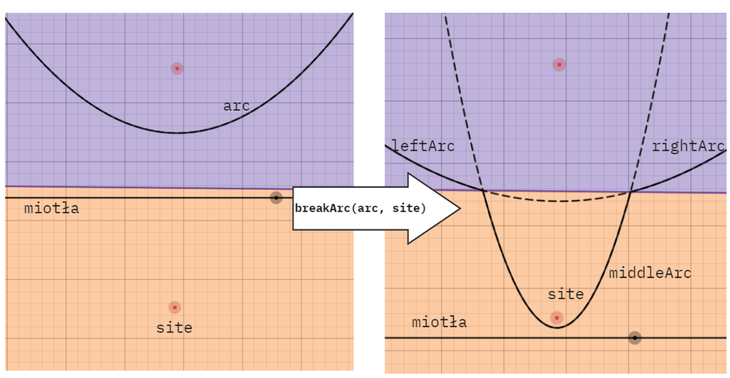
  lewy punkt rozchodzi się w _ – `isMovingRight(left, middle)`
  prawy punkt rozchodzi się w _ – `isMovingRight(middle, right)`
  lewy punkt początkowy – `getInitialX(left, middle)`
  prawy punkt początkowy – `getInitialX(middle, right)`
  punkt zbieżności – najniższy punkt okręgu wyznaczonego przez metodę `computeConvergencePoint`

  I lewy punkt załamania nie rozchodzi się w lewą kiedy prawy punkt załamania rozchodzi się w prawą

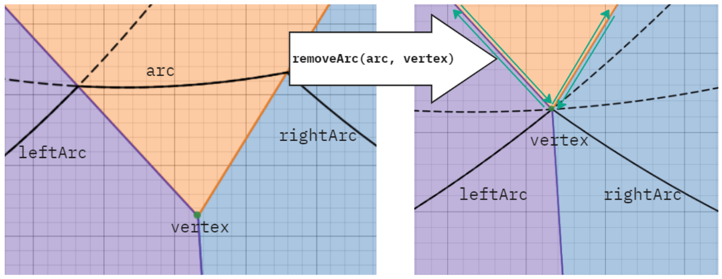
  II lewy punkt załamania rozchodzi się w prawą strone oraz lewy punkt początkowy leży po lewej stronie obliczonego punktu zbieżności lub lewy punkt załamania rozchodzi się w lewo oraz lewy punkt początkowy leży po prawej stronie punktu zbieżności

  III prawy punkt załamania rozchodzi się w prawo oraz prawy punkt początkowy leży po lewej stronie punktu zbieżności lub prawy punkt załamania rozchodzi się w lewą strone i prawy punkt początkowy leży po prawej stronie punktu zbieżności.

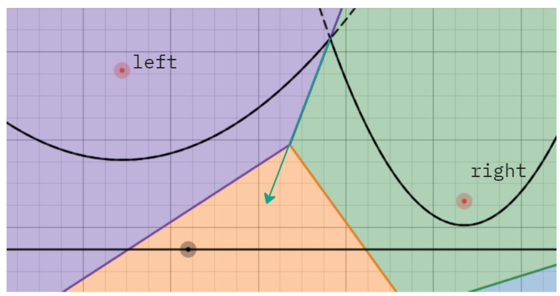
  IV najniższy punkt okręgu przechodzącego przez 3 punkty leży poniżej aktualnej pozycji miotły.



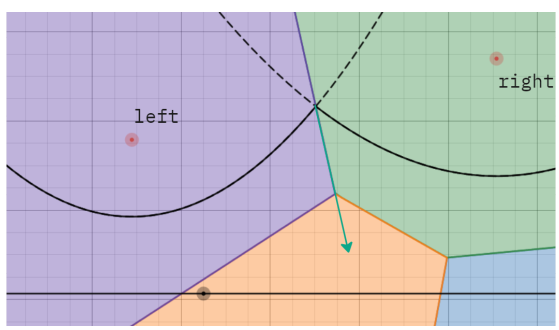
Rysunek 22: Zobrazowanie funkcji `breakArc`



Rysunek 23: Zobrazowanie funkcji `removeArc`



Rysunek 24: Przykład krawędzi poruszającej się w lewo



Rysunek 25: Przykład krawędzi poruszającej się w prawo

2.4 Algorytm Naiwny

Sekcja ta zawiera informację o klasach i ich metodach, które są stosowane przy pracy naiwnego algorytmu wyznaczającego diagram Voronoi.

2.4.1 Klasa HPlane

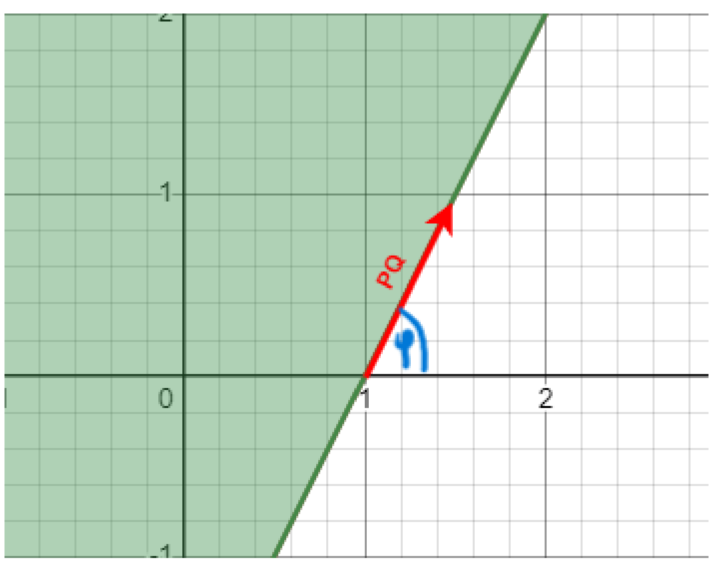
Reprezentacja półpłaszczyzny. Klasa przechowuje punkt P oraz wektor v wyznaczający kierunek prostej wykrawającej półpłaszczyznę przechodzącej przez P.Każda półpłaszczyzna obejmuje obszar po lewej stronie wektora kierunkowego.

Atrybuty:

- `P : np.array([1,2])` - punkt P należący do prostej wyznaczającej półpłaszczyznę
- `PQ : np.array([1,2])` - wektor definiujący kierunek prostej
- `angle : float` - kąt ϕ między wektorem **PQ** a prostą **OX** gdzie $-\pi < \phi \leq \pi$

Metody:

- `out(P : np.array[1,2])` - metoda sprawdzająca czy punkt leży poza półpłaszczyzną. Jeżeli punkt P nie należy do płaszczyzny zwracana jest wartość `True`
- `__lt__(hp : HPlane)` - metdoa $<$ porównująca dwie półpłaszczyzny na podstawie ich kąta `angle`
- `inter(hp : HPlane)` - metoda wyzaczająca punkt przecięcia brzegów dwóch nierównoległych półpłaszczyzn.



Rysunek 26: półpłaszczyzna o wzorze $y \geq 2x - 2$

2.4.2 Funkcje

- `hplanes_intersection(H : [HPlane], M : float` - funkcja wyznaczają część wspólną przecięcia zbioru `H` półpłaszczyzn. Parametr `M` używany jest do zbudowania obszaru ograniczajęcego złożonego z 4 półpłaszczyzn wyznaczonych przez proste o równoważniach : $\{y = M, y = -M, x = M, x = -M\}$. Ograniczenie gwarantuje, że obszar zwracany przez funkcje `hplanes_intersection` będzie punktem, odcinkiem lub wielokątem wypukłym

- `naive_voronoi(points : np.array([n,2],) inf : float)` - funkcja wyznaczająca diagram voronoi dla zbioru punktów `points`. Parametr `inf` stanowi o wielkości ramki ograniczającej segmenty diagramu voronoi będące półprostymi. Funckja zwraca zbiór wielokoątów (kolejne komórki diagramu)

- `draw_voronoi_poly(polygons , points)` - funkcja pomocnicza rysująca diagram voronoi zwrócony przez funkcje `naive_voroni`

2.5 Diagram Voronoi z triangulacji Delaunay

2.5.1 Funkcje

- `center(vertices)` - oblicza środek okręgu opisanego na trójkącie złożonego z punktów z tablicy vertices

- `voronoiDelenuay(points)` - wyznacza diagram voronoi punktów ze zbioru points wykorzystując triangulacje Delaunay. Dane o triangulacji uzyskiwane są z klasy `scipy.spatial.Delaunay` (atrybuty: `Delaunay.simplices`, `Delunay.neighbours`). Funckja zwraca listę odcinków budujących diagram.

- `draw_voronoi_edges(E, P)` - funkcja pomocnicza służąca do wizualazacji diagramu zwróconego przez funkcję `voronoiDelenuay`. Parametr E to zbiór odcinków, a P to zbiór środków

3 Sprawozdanie

3.1 Cel projektu

Zapoznanie się z różnymi metodami konstrukcji diagramu Voronoi oraz ich porównanie.

3.2 Opis projektu

W ramach projektu zaimplementowano i porównano 3 algorytmu tworzenia diagramu Voronoi:

- Algorytm Steven'a Fortune'a

- Graf dualny do Triangulacji Delaunay

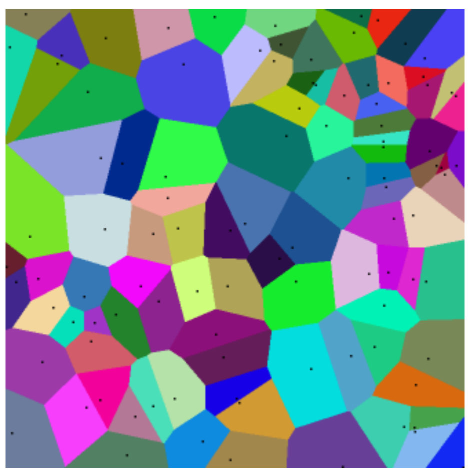
- Algorytm naiwny

Dodatkowo stworzono prostą aplikacje graficzną umożliwiającą zadawanie punktów za pomocą myszki oraz ich odczyt i zapis z pliku tekstowego.

3.3 Diagram Voronoi

3.3.1 Definicja

Podział płaszczyzny z n punktami na wielokąty wypukłe w taki sposób, że każdy wielokąt zawiera dokładnie jeden punkt generujący, a każdy punkt w danym wielokącie jest bliżej swojego punktu generującego niż do jakiegokolwiek innego. Diagram Voronoi jest czasami nazywany też tessellacją Dirichleta. Komórki tego diagramu są nazywane obszarami Dirichleta, wielokątami Thiessena lub wielokątami Voronoi.



Rysunek 27: Przykład diagramu Voronoi, `https://pl.wikipedia.org/wiki/Diagram_Woronoja`

3.3.2 Algorytm Fortune'a

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

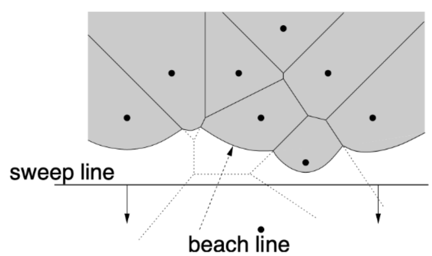
- Wejście: Zbiór punktów A, $|A| > 1$, $\forall a \in A,\ a \in \mathbb{R}^2$
- Wyjście: Diagram Voronoi opisany za pomocą krawędzi i wierzchołków.
- Złożoność obliczeniowa: $O(n \log n)$

- Złożoność pamięciowa: $O(n)$

Algorytm wykorzystuje kilka istotnych struktur danych, które są kluczowe w uzyskaniu złożoności $O(n \log n)$.

Pierwszą z nich jest kolejka priorytetowa, która przechowuje wszystkie zdarzenia. Pozwala nam ona pobierać kolejne zdarzenia w określonej kolejności w czasie logarytmicznym. W tej implementacji rozpatruje zdarzenia idąc z "góry"do "dołu", czyli malejąco względem współrzędnej y. Zamiast implementować własną kolejkę posłużyłem się gotową implementacją `PriorityQueue` z biblioteki `queue`.

Drugą strukturą jest struktura stanu zwana też *linią brzegową*. Przechowuje ona informacje o łukach tworzących diagram Voronoi. Tradycyjnie stosuje się drzewo AVL, w którym rozróżnia się węzły wewnętrzne opisujące punkty załamania dwóch paraboli i węzły zewnętrzne opisujące łuki tworzące linie brzegową co może skomplikować np. proces balansowania drzewa, który jest kluczowy w uzyskaniu końcowej złożoności $O(n \log n)$. Po kilku nieudanych próbach implementacji wykorzystujących podejście "tradycyjne"zdecydowałem się wykorzystać rozwiązanie z drzewem czerwono-czarnym nierozróżniającym węzłów na punkty załamania i łuki, lecz same łuki. Dzięki własnościom drzewa czerwono-czarnego łatwiej się je balansuje. Struktura ta pomaga nam osiągnąć oczekiwaną złożoność z racji na to, że operacje usuwania, dodawania oraz wyszukiwania w drzewie mają złożoność $O(\log n)$.



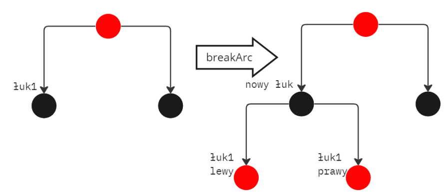
Rysunek 28: Przykładowa linia brzegowa

Trzecią strukturą jest lista podwójnie łączonych półkrawędzi(z ang. *Doubly connected edge list*). Przechowuje ona informacje o obliczonych fragmentach diagramu Voronoi.

Główną częścią algorytmu jest przetwarzanie dwóch rodzajów zdarzeń: punktowych i kołowych. Zdarzenia punktowe znamy od początku działania algorytmu. Są nimi punkty podane na wejściu. Odpowiadają one miejscom dodania nowych paraboli do linii brzegowej. Przy przetwarzaniu zdarzenia punktowego algorytm rozpatruje 2 przypadki:

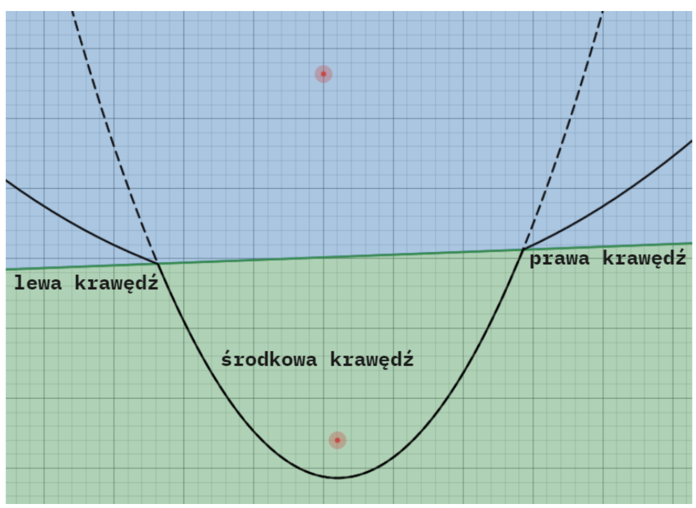
1. Przetworzenie pierwszego punktu:
   W tym przypadku w linia brzegowa jest pusta (`root=guardian`). Algorytm tworzy nowy łuk o centrum w pierwszym punkcie i ustanawia go korzeniem drzewa. Przypadek występuje dokładnie raz, na samym początku.

2. Linia brzegowa nie jest pusta:
   Algorytm znajduję łuk znajdujący się centralnie nad nim - czyli taki łuk, którego punkt centralny ma najbardziej zbliżoną współrzędną x do aktualnie przetwarzanego punktu. Następnie deaktywuje zdarzenie powiązane z tym łukiem(o ile łuk jest juz powiązany z tym zdarzeniem), czyli określa je jako fałszywy alarm. Dalej rozbija znaleziony łuk na dwa metodą `breakArc`.



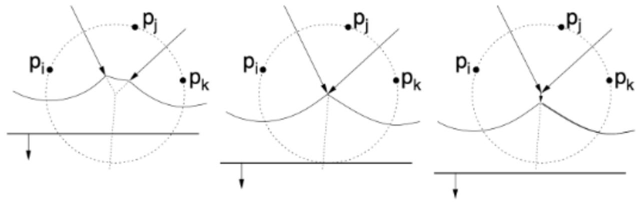
Rysunek 29: Drzewo po rozbiciu łuku

Przyjmuje określenia jak na rysunku poniżej:



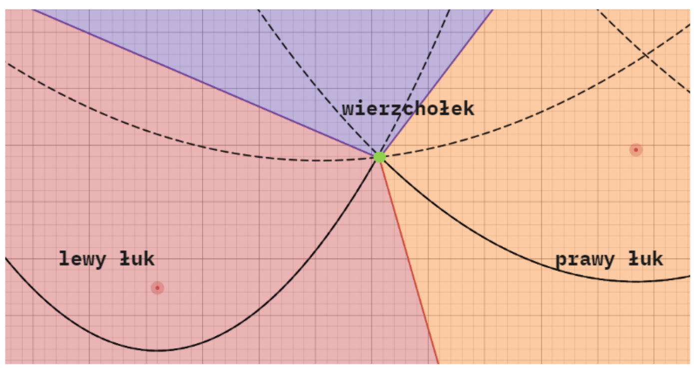
Rysunek 30: Dwa łuki po rozbiciu

Dodaje nowe krawędzie tworzone przez lewy i środkowy łuk oraz środkowy i prawy łuk. Sprawdza punkty centralne środkowego łuku, lewego łuku oraz (o ile istnieje) poprzednika lewego łuku tworzą zdarzenie kołowe oraz sprawdza czy punkty centralne środkowego łuku, prawego łuku oraz (o ile istnieje) następnika prawego łuku tworzą zdarzenie kołowe(`addEvent`). Tym sposobem algorytm znajduję zdarzenia kołowe. Zdarzenia kołowe określają punkty zanikania paraboli.



Rysunek 31: Wizualizacja zanikania paraboli podczas zdarzenia kołowego

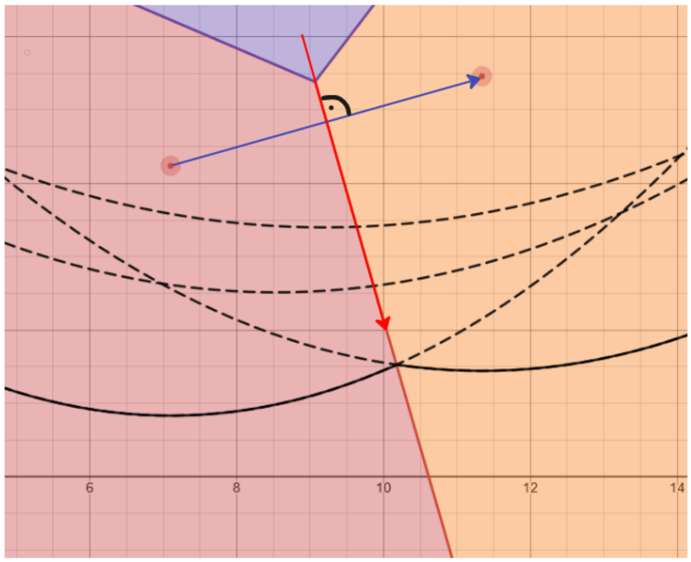
Punktem zaniknięcia paraboli jest środek pustego okręgu stycznego do trzech punktów początkowych. Algorytm podczas rozpatrywania zdarzeń kołowych działa następująco. Tworzy nowy wierzchołek diagramu Voronoi na środku okręgu opisującego zdarzenie kołowe.



Rysunek 32: Łuki sąsiednie po zaniknięciu łuku między nimi

Dezaktywuję ewentualne zdarzenia kołowe powiązane ze swoim poprzednikiem oraz następnikiem(łuku po prawej i po lewej). Usuwa łuk zanikający w aktualnie rozpatrywanym zdarzeniu. Sprawdza ewentualne wystąpienie zdarzeń kołowych stworzonych przez koła styczne do centrów łuku lewego, łuku prawego, poprzednika łuku lewego oraz centrów łuku lewego, łuku prawego, następnika łuku prawego.

Algorytm na początku inicjalizuje wszystkie zdarzenia punktowe w strukturze zdarzeń, a następnie dopóki kolejka priorytetowa nie będzie pusta wyciąga kolejne zdarzenia i je przetwarza. Przed wyświetleniem diagramu algorytm ogranicza wszystkie krawędzie zmierzające do nieskończoności. Uzyskuję o nich informacje bazując na pozostałych łukach w strukturze stanu. Iteruje po wszystkich pozostałych krawędziach tworzących linie brzegową i oblicza wektor kierunkowy, który określa kierunek, w którym zmierza krawędź. Jest to wektor prostopadły do wektora $\overrightarrow{P_l P_r}$, gdzie P_l to centrum lewej paraboli, a P_r to centrum prawej paraboli.

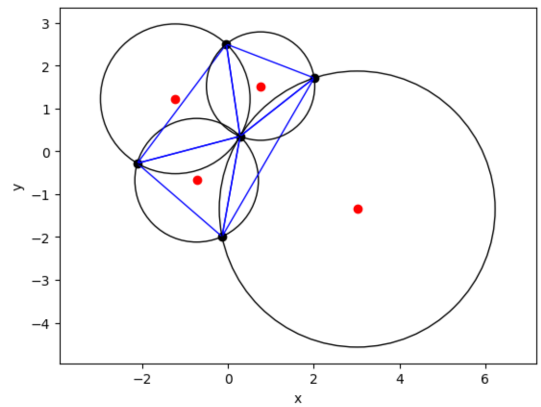


Rysunek 33: Zobrazowanie sposobu znajdywania kierunku rozchodzenia krawędzi nieskończonych

Na podstawie tego wektora oraz punktu początkowego leżącego na środku odcinka $P_l P_r$ wyznacza przecięcie prostokątem ograniczającym.

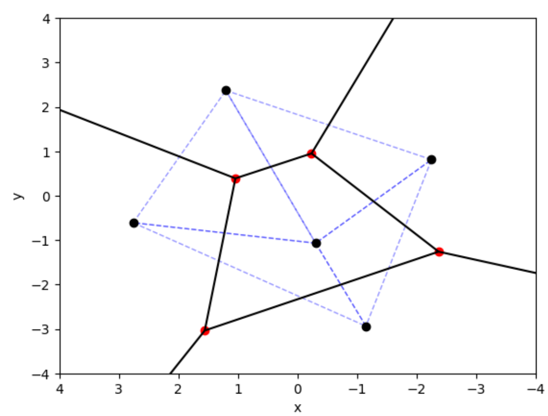
3.3.3 Graf dualny do triangulacji Delaunay

Triangulacja Delaunaya jest to taka triangulacja zbioru punktów P na płaszczyźnie, że żaden punkt ze zbioru P nie znajduje się wewnątrz okręgu opisanego na trójkącie należącym do triangulacji



Rysunek 34: Przykład triangulacji Delaunay

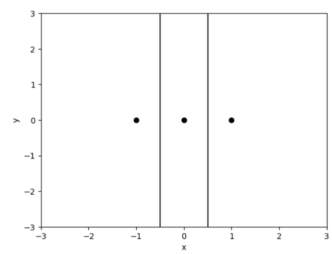
Istotną z punktu widzenia diagramów Voronoi właściwością triangulacji Delaunaya jest fakt, że graf dualny triangulacji odpowiada diagramowi Voronoi dla tego samego zbioru punktów. Środki okręgów opisanych na trójkątach triangulacji odpowiadają wierzchołkom diagramu Voronoi, a odpowiednie krawędzie między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



Na czarno oznaczone są punkty startowe. Linia przerywana to linie siatki triangulacji. Na czerwono oznaczone zostały środki okręgów opisanych na trójkątach, które jednocześnie odpowiadają wierzchołkom diagramu Voronoi

- Wejście: Zbiór puntów P, $|P| = n$, $\forall p \in P, p \in \mathbf{R}^2$
- Złożoność $O(n^2)$
- Wyjście: Zbiór krawędzi (odcinków) E wchdzacych w skład diagramu Voronowi

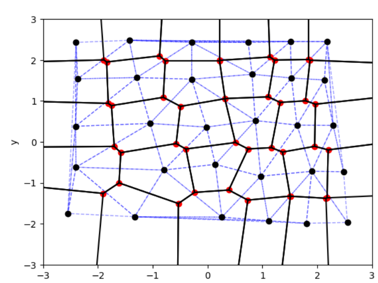
1. Na początku sprawdzana jest współliniowość punktów ze zbioru P. Jeżeli wszystkie leżą na jednej prostej triangulacja nie będzie mogła zostać przeprowadzona. W takim wypadku diagram składa się z $(n-1)$ symetralnych kolejnych par punktów i algorytm zostaje zakończony

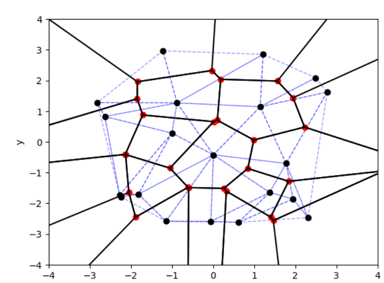


Rysunek 35: Przykład dla punktów współliniowych

2. Przeprowadzona jest triangulacja Delaunaya punktów P
3. Dla każdego trójkąta w triangulacji należy obliczyć środek opisanego na nim okręgu. Środki sąsiadujących ze sobą trójkątów tworzą odcinki, które dodawane są do zbioru wynikowego E. Trójkąty należące do otoczki wypukłej mają maksymalnie 2 sąsiadów. Dla tych trójkątów krawędzie diagramu będą nieskończonymi półprostymi pokrywającymi się z symetralną boku bez sąsiada. Problem ten można obejść dodając odcinek składający się z ortocentrum trójkąta z otoczki wypukłej i punktu leżącego na jego symetralnej w "nieskończoności".

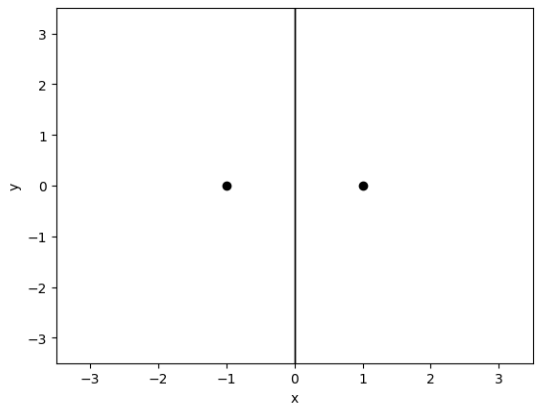
Przykłady





3.3.4 Algorytm Naiwny

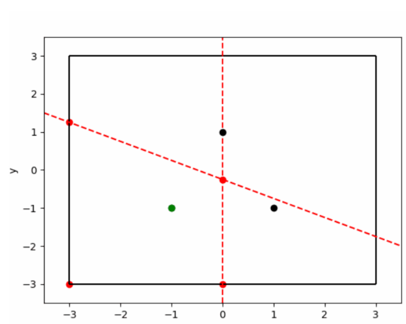
Diagram Voronoi dla dwóch punktów p_1, p_2 można prosto skonstruować. Jest to symetralna odcinka p_1p_2.



W ogólności dla punktów $p_1..p_n$ komórki diagramu Voronoi $V(p_i)$ można skonstruować jako przecięcie n-1 półpłaszczyzn zawierających p_i wyznaczonych przez symetralne odcinków p_ip_j dla $j \neq i$

Funkcja wyliczają przecięcie półpłaszczyzn `hplanes_intersection` działa w złożoności $O(n \log n)$. Złożoność naiwnego algorytmu to zatem $O(n^2 \log n)$,ponieważ dla każdego punktu p_i buduje komórkę $V(p_i)$

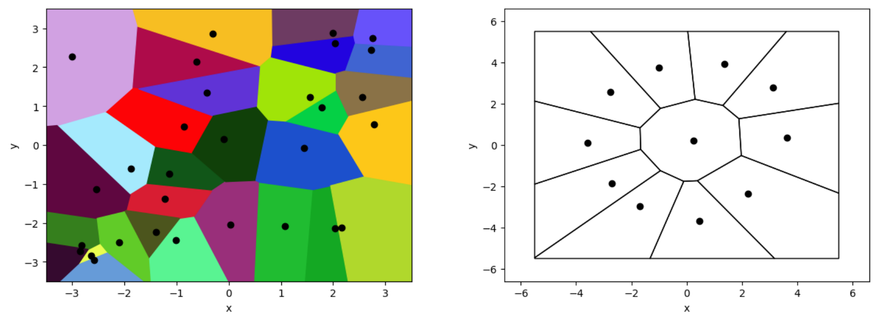
- Wejście: Zbiór puntów P, $|P| = n$, $\forall p \in P, p \in \mathbf{R}^2$
- Złożoność $O(n^2 \log n)$
- Wyjście: Zbiór wielokątów P wchdzacych w skład diagramu Voronowi



Rysunek 36: Klatka animacji wykonania algorytmu naiwnego

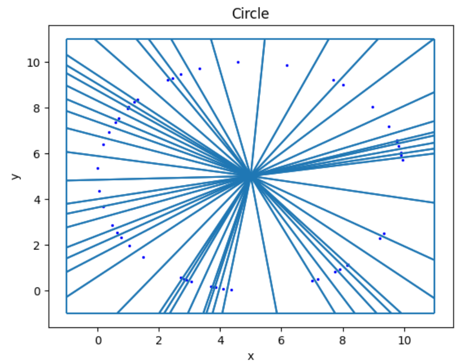
Na zielono oznaczono punkt dla którego wyliczana jest komórka. Czerwone proste to symetralne a punkty czerwone to wierzchołki komórki.

Przykłady

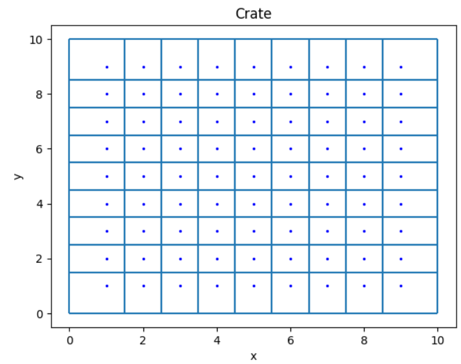


Rysunek 37: Dzięki informacji o wierzchołkach kontentych komórek diagram można pokolorować

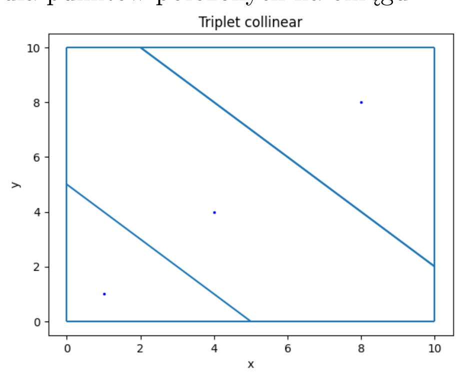
3.3.5 Wizualizacja



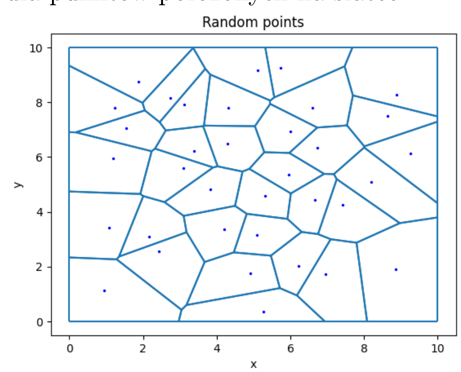
Rysunek 38: Diagram Voronoi
dla punktów położonych na okręgu



Rysunek 39: Diagram Voronoi
dla punktów położonych na siatce



Rysunek 40: Diagram Voronoi
dla 3 współliniowych punktów



Rysunek 41: Diagram Voronoi
dla losowego zbioru punktów

3.3.6 Porównanie czasowe algorytmów

	Liczba punktów	Naiwny [s]	Delauney [s]	Fortune [s]
0	10.0	0.03	0.0	0.0
1	100.0	0.74	0.01	0.01
2	1000.0	73.72	0.08	0.06
3	10000.0	Zbyt długo	0.59	0.64
4	100000.0	Zbyt długo	6.04	13.98

Tabela 1: Zestawienie czasów działania algorytmów

Z porównania wynika, że algorytmy Fortune'a i tworzącego diagram na podstawie traingulacji Delauney'a są znacząco szybsze od algorytmu naiwnego. Co ciekawe dla 100000 punktów algorytm triangulacji okazał się szybszy od Fortune'a. Wynikać to może z tego, że dla większości przypadków algorytm ten ma średnią złozoność obliczeniową $O(n \log n)$ i tylko w pesymistycznych przypadkach osiąga złozoność $O(n^2)$.

4 Bibliografia

- https://ufkapano.github.io/download/Mateusz_Malczewski_2021.pdf
- https://jacquesheunis.com/post/fortunes-algorithm/
- https://github.com/fewlinesofcode/FortunesAlgorithm/tree/master
- https://www.slideshare.net/OleksandrGlagoliev/fortunes-algorithm
- https://en.wikipedia.org/wiki/Doubly_connected_edge_list
- https://jacquesheunis.com/post/fortunes-algorithm-implementation/
- https://github.com/fewlinesofcode/FortunesAlgorithm/blob/master/Sources/FortunesAlgorithm/FortuneSweep.swift
- https://github.com/Yatoom/foronoi
- https://pl.wikipedia.org/wiki/Diagram_Woronoja
- https://pl.khanacademy.org/computing/pixar/pattern/dino/e/constructing-a-voronoi-partit
- http://www.multimedia.edu.pl/for_students/teaching_resources/biometry/files/cwiczenie6b.pdf
- https://www.desmos.com/calculator/ejatebvup4?lang=pl