Dokumentacja

Triangulacja wielokąta prostego – porównanie metod

Jan Chadziński i Emil Żychowicz

3 stycznia 2025

Opisy funkcji i klas - część techniczna

W tej sekcji znajdują się opisy techniczne funkcji, metod oraz klas użytych do implementacji algorytmów. Niektóre metody i atrybuty zostały uznane za nieistotne - wówczas nie umieszczano ich w dokumentacji, aby nie zaciemniały obrazu tego, co istotne.

1 Triangulacja Delaunaya

1.1 Klasa Vector

Atrybuty

1. `x` : współrzędna x wektora
2. `y` : współrzędna y wektora

1.2 Funkcje pomocnicze

1. `sgn(value) -> int` : funkcja signum
2. `orientationDet(v1: Vector, v2: Vector, v3: Vector) -> float` : zwraca wartość wyznacznika używanego do wyznaczania orientacji trzech punktów
3. `orientation(v1: Vector, v2: Vector, v3: Vector) -> int`
   zwraca:
   (a) -1 jeżeli v3 leży po lewej stronie od v1 v2
   (b) 0 jeżeli punkt v3 jest współliniowy z v1, v2
   (c) 1 jeżeli v3 leży po po prawej stronie od v1 v2
4. `onsegment(a: Vector, b: Vector, p: Vector, Eps: float)` : sprawdza czy punkt `p` leży na odcinku `ab`, przy użyciu tolerancji `Eps`
5. `circumcircleTest(a: Vector, b: Vector, c: Vector, p: Vector) -> int` : sprawdza położenie punktu `p` względem okręgu opisanego na punktach `a, b, c`
   zwraca:
   (a) -1 jeżeli `p` znajduje się poza okręgiem opisanym na `a, b, c`
   (b) 0 jeżeli punkt `p` znajduje się na okręgu opisanym na `a, b, c`
   (c) 1 jeżeli punkt `p` znajduje się wewnątrz okręgu opisanego na `a, b, c`
6. `segmentsIntersect(a: Vector, b: Vector, c: Vector, d: Vector)` -> bool : sprawdza czy odcinki

`ab` i `cd` przecinają się

7. `quadConvex(a: Vector, b: Vector, c: Vector, d: Vector)` -> bool : sprawdza czy czworobok `abcd` jest wypukły, to znaczy wszystkie jego kąty wewnętrzne są mniejsze od π

8. `getBoundingTriangle(points: [Vertex]) -> (Vector, Vector, Vector)` : znajduje trójkąt obejmujący wszystkie wejściowe wierzchołki

1.3 Klasa Face

Reprezentuje powierzchnię danego trójkąta, używana w strukturze do znajdywania trójkąta zawierającego dany punkt.

Atrybuty

1. `edge: HalfEdge` : dowolna krawędź należąca do tej powierzchni (używane jedynie w przypadku gdy nie ma dzieci)

2. indeksy wierzchołków, w kolejności CCW:

3. `vertex1: Int`

4. `vertex2: Int`

5. `vertex3: Int`

6. `children: [Int]` : indeksy powierzchni, które powstały na skutek podzielenia tej powierzchni

Metody

1. `AddChild(face: Int)` - dodaje `face` do listy dzieci

2. `contains(point: Vector, vertices: [Vector])` - sprawdza czy powierzchnia zawiera punkt `point`

1.4 Klasa HalfEdge

Reprezentuje pojedynczą krawędź w strukturze Half Edge.

Atrybuty

1. `next: HalfEdge` : Odnośnik do kolejnej krawędzi.

2. `twin: HalfEdge` : Odnośnik do krawędzi symetrycznej, w sąsiadującym trójkąćie.

3. `vertex: int` : Indeks wierzchołka, z którego krawędź wychodzi.

4. `face: int` : Indeks powierzchni, do której przynależy krawędź.

Metody

1. `flip() -> HalfEdge` : Zamienia przekątną czworoboku do którego należy i zwraca nową przekątna.

2. `splitFace(newVertex: Int) -> (HalfEdge, HalfEdge, HalfEdge)` : Rozdziela trójąt na trzy nowe i zwraca krawędź każdego z nich.

3. `splitEdge(newVertex: Int) -> (HalfEdge, HalfEdge, HalfEdge, HalfEdge)` : Rozdziela krawędź na dwie i zwraca krawędzie czterech nowo utworzonych trójkątów.

1.5 Klasa Mesh

Reprezentuje pojedynczą krawędź w strukturze Half Edge.

Atrybuty

1. `vertices: [Vector]` : Lista wierzchołków.
2. `faces: [Face]` : Lista powierzchni.
3. `verticesEdges: [set : HalfEdge]` : Lista zbiorów krawędzi wychodzących z danego wierzchołka.
4. `constrained: set : (Int, int)` : Zbiór krawędzi, które zostały wymuszone, używane podczas znajdowania zewnętrznych/wewnęrznych trójkątów.

Metody

1. `addVertexAndLegalize(edge: HalfEdge, vertexIndex: int)` : Dodaje wierzchołek, a następnie sprawia, by triangulacja spełniała warunki delaunaya.
2. `locate(face: int, vertex: Vector)` : Zwraca indeks powierzchni zawierającej podane punkt.
3. `constrainEdge(index1: int, index2: int)` : Wymusza krawędź pomiędzy wierzchołkami o indeksach, `indeks1 i indeks2`
4. `flipEdge(edge: HalfEdge) -> HalfEdge` : Przekręca przekątną czworoboku i aktualizuje graf powierzchni.
5. `isEdgeLegal(edge: HalfEdge) -> bool` : Sprawdza czy przekątna jest legalna.
6. `legalizeEdge(edge: HalfEdge)` : Jeżeli przękątna nie jest legalna, przekręca ją.
7. `addVertexToEdgeAndLegalize(edge: HalfEdge, vertexIndex: int)` : Dodaje wierzchołek znajdujący się na krawędzi, naprawia triangulację oraz aktualizuje graf powierzchni.
8. `addVertexToFaceAndLegalize(edge: HalfEdge, vertexIndex: int)` : Dodaje wierzchołek znajdujący się na wewnątrz trójkąta, naprawia triangulację oraz aktualizuje graf powierzchni.
9. `faceContains(face: int, vertex: Vector) -> bool` : Sprawdza czy powierzchnia o indeksie `face` zawiera punkt `vertex`.
10. `findIntersectingEdges(index1: int, index2: int) -> deque: HalfEdge` : Znajduje krawędzie aktualnie istniejące w triangulacji, przecinające się z odcinkiem między wierzchołkami o indeksach `index1 i index2`.
11. `findInner() -> [bool]` : Zwraca listę bool-i, dla każdej powierzchni wartość `True`, oznacza, że powierzchnia ta jest wewnętrzna.
12. `usesSuperTriangle(face: int) -> bool` : Sprawdza czy dana powierzchnia zawiera wierzchołek należący do super trójkąta.
13. `toTriangleList(filterSuperTriangle: bool, removeOuter: bool) -> [(int, int, int)]` : Zwraca wynik triangulacji, filterSuperTriangle oznacza, że usunięta zostaną trójąty zawierająca wierzchołki super trójkąta, removeOuter oznacza, że usunięty zostaną zewnętrzne trójkąty.

1.6 Funkcja dt

`dt (points: [Vector], shuffle: bool = False) -> [(int, int, int)]` : Przeprowadza triangulację delaunaya (Delaunay Triangulation). Zwraca wynik z usuniętymi krawędziami należącymi do super trójkąta.

1. `points: [Vector]` : List punktów do triangulacji.
2. `consrains: [(int, int)]` : Lista wymuszonych odcinków, w postaci par indeksów punktów.
3. `shuffle: bool` : Flaga odpowiadająca za proces permutowania kolejności dodawania punktów do triangulacji, losowa kolejność pozwala uzyskaść lepszą złożoność oczekiwaną ($nlogn$) triangulacji bez ograniczeń, jednak utrudnia testowanie programu.

1.7 Funkcja cdt

`cdt (points: [Vector], constrains: [(int, int)], shuffle: bool = False) -> [(int, int, int)]` : Przeprowadza triangulację z ograniczenima (Constrained Delaunay Triangulation). Zwraca wynik z usuniętymi krawędziami należącymi do super trójkąta oraz usuwa zewnętrzne krawędzie, jeżeli została podana przynajmniej jedna wymuszona krawędź.

1. `points: [Vector]` : List punktów do triangulacji.
2. `consrains: [(int, int)]` : Lista wymuszonych odcinków, w postaci par indeksów punktów.
3. `shuffle: bool` : Flaga odpowiadająca za proces permutowania kolejności dodawania punktów do triangulacji, losowa kolejność pozwala uzyskaść lepszą złożoność oczekiwaną ($nlogn$) triangulacji bez ograniczeń, jednak utrudnia testowanie programu.

1.8 Funkcja triangulatePolygon

`triangulatePolygon (points: [Vector] shuffle: bool = False) -> [(int, int, int)]` : Używa funkcji cdt do przeprowadzenia triangulacji wielokąta.

2 Triangulacja przez dekompozycję do wielokątów monotonicznych

2.1 Klasy struktur danych

2.1.1 Klasa `Vertex`

Klasa reprezentująca wierzchołek w strukturze Half-Edge.

Atrybuty:

1. `x`: Współrzędna x wierzchołka.
2. `y`: Współrzędna y wierzchołka.
3. `id`: Identyfikator wierzchołka (opcjonalny).
4. `outgoingEdge`: Wskaźnik na jedną z wychodzących krawędzi (`HalfEdge`).
5. `type`: Typ wierzchołka (np. `'terminal'`, `'split'` itp.).

Metody:

1. `__repr__()`: Zwraca reprezentację tekstową wierzchołka w formacie `Vertex(x, y)`.

2.1.2 Klasa `Graph`

Klasa implementująca reprezentację grafu poprzez listę sąsiedztwa. Pozwala na łatwe zarządzanie grafem, w tym dodawanie krawędzi, takich jak przekątne w wielokącie.

Atrybuty:

1. `vertices`: Lista wierzchołków (obiekty klasy `Vertex`).
2. `edges`: Lista krawędzi w grafie reprezentowanych jako krotki (`Vertex1, Vertex2`).
3. `G`: Lista sąsiedztwa reprezentująca graf w postaci listy połączeń między wierzchołkami.

Metody:

1. `createAdjacencyList()`: Metoda tworząca listę sąsiedztwa na podstawie podanej listy wierzchołków i krawędzi. Dla każdej krawędzi dodaje jej końce do list sąsiedztwa odpowiednich wierzchołków.
2. `addDiagDiv(v1ID, v2ID)`: Dodaje przekątną między wierzchołkami o podanych identyfikatorach (`v1ID`, `v2ID`). Uaktualnia listę sąsiedztwa o nowe połączenie.

2.1.3 Klasa `HalfEdge`

Klasa reprezentująca krawędź w strukturze Half-Edge.

Atrybuty:

1. `origin`: Wskaźnik na początkowy wierzchołek (`Vertex`).
2. `twin`: Wskaźnik na drugą połowę tej samej krawędzi (`HalfEdge`).
3. `next`: Wskaźnik na następną krawędź w tej samej ścianie (`HalfEdge`).
4. `prev`: Wskaźnik na poprzednią krawędź w tej samej ścianie (`HalfEdge`).
5. `face`: Wskaźnik na ścianę (`Face`), do której należy krawędź.
6. `A`, `B`, `C`: Współczynniki równania prostej $Ax + By + C = 0$.
7. `helper`: Pomocniczy wskaźnik dla algorytmów manipulujących strukturą.

Metody:

1. `currX()`: Oblicza bieżącą współrzędną x dla zadanej wartości `currY`, korzystając z równania prostej.
2. `__repr__()`: Zwraca reprezentację tekstową krawędzi w formacie `HalfEdge(origin=...)`.
3. `__eq__(other)`: Porównuje dwie krawędzie na podstawie ich punktów początkowych i końcowych.
4. `__hash__()`: Generuje hash krawędzi na podstawie identyfikatora początkowego wierzchołka.

2.1.4 Klasa `Face`

Klasa reprezentująca ścianę w strukturze Half-Edge.

Atrybuty:

1. `outerEdge`: Wskaźnik na jedną z otaczających krawędzi (`HalfEdge`).

Metody:

1. `__repr__()`: Zwraca reprezentację tekstową ściany w formacie `Face(outer_edge=...)`.

2.1.5 Klasa `HalfEdgeMesh`

Klasa reprezentująca całą strukturę Half-Edge, w tym wierzchołki, krawędzie i ściany.

Atrybuty:

1. `vertices`: Lista wierzchołków (`Vertex`).
2. `edges`: Lista krawędzi (`HalfEdge`).
3. `faces`: Lista ścian (`Face`).

Metody:

1. `addVertex(x: float, y: float)`: Dodaje nowy wierzchołek o współrzędnych (x, y) do listy `vertices`.
2. `addEdge(v1: Vertex, v2: Vertex)`: Tworzy krawędź między wierzchołkami `v1` i `v2`, dodając dwie `HalfEdge` jako połówki tej krawędzi.
3. `addFace(edgeCycle: List[HalfEdge])`: Tworzy ścianę otoczoną przez podany cykl krawędzi `edgeCycle`.
4. `__repr__()`: Zwraca reprezentację tekstową obiektu w formacie `HalfEdgeMesh(vertices=..., edges=..., faces=...)`.

2.2 Pozostałe klasy

2.2.1 Klasa `Structures`

Klasa odpowiedzialna za przygotowanie danych wejściowych do algorytmu dekompozycji na wielokąty proste. Klasa zawiera metody do tworzenia listy zdarzeń, struktury stanu oraz inicjalizacji zamiatania.

Atrybuty:

1. `points`: Lista obiektów klasy `Vertex`, reprezentująca wierzchołki wielokąta.

Metody:

1. `prepareHalfEdgeMesh()`: Zamienia listę wierzchołków w orientacji przeciwnej do ruchu wskazówek zegara na strukturę DCEL (Half-Edge);
2. `prepareEvents()`: Wykonuje klasyfikacje punktów na typy (początkowy, końcowy, łączący itd.). Typy punktów są przechowywane w strukturze `Vertex`. Sortuje wierzchołki w kolejności przeciwnej do kolejności zamiatania (wierzchołki będą zdejmowane z góry) - po rzędnych, a następnie po odciętych rosnąco;
3. `cmp(HalfEdge: halfedge1, HalfEdge: halfedge2)`: Metoda porównuje wartości odciętych dwóch półkrawędzi w `currY` - danej rzędnej;
4. `prepareSweep()`: Inicjalizuje strukturę `SortedSet` z biblioteki `sortedcontainers` z nadpisanym kluczem, wprowadzonym za pomocą biblioteki `functools` oraz funkcji komparatora `cmp`.

2.2.2 Klasa `Classification`

Klasa odpowiedzialna za klasyfikację wierzchołków wielokąta na podstawie ich położenia względem sąsiednich punktów.

Atrybuty:

1. `points`: Lista obiektów klasy `Vertex`;
2. `start`, `end`, `merge`, `split`, `regular`: Listy indeksów wierzchołków odpowiednich typów.

Metody:

1. `orient(A: Vertex, B: vertex, C: Vertex)`: Funkcja określa położenie punktu C względem prostej AB. Robi to za pomocą wyznacznika liczonego metodą Sarrusa w `det_sarrus` i porównywania go do 0 z tolerancją na zero $\epsilon = 10^{-12}$.
2. `classify()`: Klasyfikuje wierzchołki do następujących kategorii: w. początkowe, w. końcowe, w. łączące, w. dzielące i w. prawidłowe. Ustawia atrybuty typów instancjom `Vertex` z points. Klasyfikacja odbywa się na zasadzie określania wzajemnego położenia sąsiednich trójek punktów z points.
3. `visualizeClassification()`: Wizualizuje klasyfikację wierzchołków na wykresie.

2.2.3 Klasa `Division`

Klasa implementująca algorytm dzielenia wielokąta prostego na wielokąty monotoniczne przy użyciu algorytmu zamiatania.

Atrybuty:

1. `polygon`: Obiekt reprezentujący wielokąt (siatka Half-Edge - HalfEdgeMesh).
2. `Q`: Lista zdarzeń zamiatania.
3. `T`: Zbiór aktywnych krawędzi zamiatania.
4. `D`: Graf, do którego dodawane są krawędzie (przekątne).

Metody:

1. `divide()-> [[Vertex]]`: Główna funkcja dzieląca wielokąt na prostsze regiony. Polega na zdejmowaniu kolejnych zdarzeń ze stosu Q i wykonywaniu odpowiedniej funkcji `handle` w zależności od typu wierzchołka. Po zdjęciu wszystkich elementów ze stosu, tworzy listę list wierzchołków na poszczególnych ścianach. Dzieje się to za pomocą metody `updateFaces`.

2. `updateFaces()-> [[Vertex]]` : Metoda zajmuje się wydzielaniem kolejnych łańcuchów punktów z grafu D. Przechodzi przez graf zawsze wybierając krawędź na lewo od krawędzi, po której weszła do wierzchołka (krawędzie są uprzednio posortowane po kątach). Logika opisana w sprawozdaniu. Funkcja zwraca listę list wierzchołków.

2.2.4 Klasa `Triangulation`

Klasa implementująca algorytm triangulacji wielokąta monotonicznego. Obsługuje dwa warianty wynikowe triangulacji: lista trójek indeksów i dodanych przekątnych.

Atrybuty:

1. `triangles`: Lista wygenerowanych trójkątów.

2. `addedDiags`: Lista dodanych przekątnych.

3. `vertices`: Lista obiektów klasy Vertex w kolejności CCW.

4. `left`: lista booli, gdzie `left[i] == True` oznacza, że punkt w `vertices` na indeksie `i` jest w lewym łańcuchu.

Metody:

1. `branches() -> ([int], [int])`: Dzieli na lewą i prawą gałąź - do prawej należy najniższy punkt, do lewej najwyższy. Modyfikuje atrybut `left` i zwraca listy indeksów punktów należących do lewych i prawych indeksów.

2. `inside(idx1: int, idx2: int, idx3: int) -> bool`: Sprawdza czy trójkąt stworzony z punktów o indeksach idx1, idx2, idx3 jest w środku wielokąta. Najpierw ustawia punkty w kolejności CCW, potem oblicza ich wyznacznik (z `det_sarrus`). Gdy wyznacznik $> \epsilon = 10^{-12}$ kąt wew. tworzony przez te 3 punkty jest $< \pi$, zatem trójkąt należy do wielokąta.

3. `mergeBranches() -> [int]`: Metoda łączy listę lewych indeksów i prawych tak, aby w liście wynikowej indeksy były posortowane tak, że punkty im odpowiadające są posortowane po rzędnych. Dodatkowo, przy konflikcie pierwszeństwo ma lewy łańcuch.

4. `algorithm()`: Algorytm przygotowuje sobie listę indeksów z `mergeBranches()`. Następnie, pobierając elementy z tej listy, za pomocą `left` rozpatruje za każdym razem dwie sytuacje w zależności od położenia punktów na łańcuchach. Aby uniknąć dodawania przekątnych, które są tak naprawdę bokami do `addedDiags`, sprawdza się czy indeksy nie są kolejne oraz czy nie są pierwszym i ostatnim indeksem (czy nie są to indeksy sąsiednich wierzchołków).

5. `algorithmTriangles()`: Algorytm wykonujący te same czynności co `algorithm`, ale zamiast dodawania przekątnych dodaje trójki indeksów punktów w kolejności przeciwnej do ruchu wskazówek zegara do `triangles`.

2.3 Pozostałe pliki

1. `animations.py` - plik pomocniczny do animacji,

2. `MD_generate_animations.py` - plik generujący animacje dla triangulacji przez podział na wielokąty monotoniczne,

3. `compare_times.py` - plik do porównywania czasów działania dla dwóch metod,

4. `compare_quality.py` - plik porównujący jakość triangulacji dwóch metod.

5. `generate_sun_like_figure.py` - plik generujący listę punktów CCW losowo. Najpierw losuje kąty z zakresu 0 do 2π, następnie umieszcza punkty na zmianę na szerszym i węższym okręgu. W ten sposób tworzy się figura przypominające słońce/gwiazdkę.

Instrukcja użytkowania

- **interactive()**
  Ta funkcja umożliwia interaktywne dodawanie wierzchołków do siatki triangulacji za pomocą kliknięć myszą. Po dodaniu nowego punktu funkcja automatycznie przeprowadza proces triangulacji Delaunaya, a wyniki są natychmiast wizualizowane na wykresie.

  Zdarzenia obsługiwane przez funkcję:

  – Kliknięcie lewym przyciskiem myszy dodaje nowy wierzchołek.

  – Ruch kursora powoduje podświetlenie trójkąta, w którym znajduje się kursor.

  – Naciśnięcie klawisza numerycznego tworzy krawędź pomiędzy wskazanymi punktami.

- **inputPolygon()**
  Funkcja umożliwia użytkownikowi narysowanie wielokąta na wykresie interaktywnym, który następnie jest triangulowany przez algorytm.

- **specificTest()**
  Ładuje plik JSON zawierający zestaw punktów oraz krawędzi wymuszonych, a następnie wykonuje triangulację z ograniczeniami. Wynik jest wyświetlany na wykresie.

- **specificTestNoConstrains()**
  Podobnie jak `specificTest()`, ale nie uwzględnia krawędzi wymuszonych podczas triangulacji.

2.4 Menu główne

Funkcja `__main__` zawiera prosty system menu, który pozwala użytkownikowi wybrać jedną z dostępnych funkcji:

A – Interaktywne dodawanie wierzchołków i triangulacja,

B – Automatyczne testy,

C – Rysowanie wielokąta i zapisywanie wyników triangulacji,

D – Wykonanie konkretnego testu z pliku,

E – Wykonanie konkretnego testu bez krawędzi wymuszonych,

2.5 Triangulacja przez podział na wielokąty monotoniczne

Aby striangulować wielokąt należy wejść w plik `monotonic_division.py` i go uruchomić. Następnie należy wybrać 1 z 3 opcji.
W celu wygenerowania animacji należy wejść w plik `MD_generate_animations.py` i wybrać jedną z 3 opcji.

3 Spis bibliotek użytych w projekcie

1. `sortedcontainers.SortedSet`: Biblioteka zewnętrzna zapewniająca szybkie, dynamicznie sortowane kontenery, takie jak zbiory i listy. W tym przypadku użyta do struktury stanu (miotły) w algorytmie zamiatania.

2. `functools.cmp_to_key`: Moduł standardowy Pythona umożliwiający przekształcenie funkcji porównującej na klucz używany w sortowaniu. Użyta do stworzenia klucza w strukturze stanu algorytmu zamiatania.

3. `matplotlib.pyplot`: Biblioteka do tworzenia wykresów w Pythonie. Posłużyła jako narzędzie wszelkich wizaulizacji.

4. `json`: Moduł standardowy Pythona służący do obsługi formatu JSON. Służy jako most między wprowadzonymi interaktywnie danymi, a algorytmami.

4 Wymagania techniczne

Algorytmy i struktury danych zostały zaimplementowane w języku Python, wykorzystując zarówno wbudowane funkcje, jak i zewnętrzne biblioteki. Kod źródłowy znajduje się w publicznie dostępnym repozytorium na platformie GitHub.

Poniżej zamieszczono tabelę przedstawiającą użyte biblioteki oraz narzędzia wraz z ich wersjami. Ponieważ program był testowany na dwóch różnych komputerach, niektóre informacje, takie jak rodzaj procesora, zostały uwzględnione dwukrotnie.

System operacyjny n1	Microsoft Windows 11 Pro 10.0.22631
CPU n1	13th Gen Intel(R) Core(TM) i7-13700H 2.40 GHz
System operacyjny n2	Linux Mint 21
CPU n2	AMD Ryzen 5 3600
Python interpreter	Python 3.11.9 64-bit
Numpy version	2.1.1
Matplotlib version	3.9.2
Sortedcontainers version	2.4.0

Tabela 1: Dane techniczne

Sprawozdanie

4.1 Triangulacja przez dekompozycję do wielokątów monotonicznych

4.1.1 Część teoretyczna

Do algorytmu triangulacji potrzebne jest zdefiniowanie poniższych pojęć.
Klasyfikacja wierzchołków w wielokącie
W algorytmach triangulacji pomocny jest następujący podział wierzchołków:

- w. początkowy - obaj sąsiedzi leżą poniżej, a kąt wewnętrzny mniejszy od π,
- w. końcowy - obaj sąsiedzi leżą powyżej i kąt wewnętrzny mniejszy od π,
- w. łączący - obaj sąsiedzi leżą powyżej i kąt wewnętrzny większy od π,
- w. dzielący - obaj sąsiedzi leżą poniżej i kąt wewnętrzny większy od π,
- w. prawidłowy - pozostałe przypadki (jeden sąsiad powyżej, drugi poniżej).

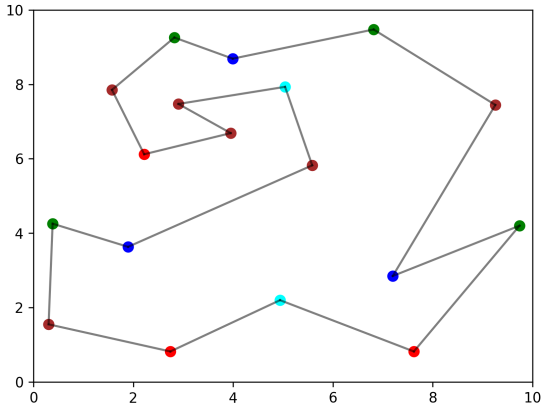
Relacje powyżej i poniżej są jednak uszczegółowione, aby ułatwić obsługę wielokątów z wieloma punktami na jednej rzędnej. punkt p znajduje się **poniżej** punktu q, gdy $p_y < q_y$, lub $p_y = q_y$ i $p_x > q_x$. Analogicznie, punkt p jest **powyżej** punktu q, jeśli $p_y > q_y$, lub $p_y = q_y$ i $p_x < q_x$.

(Można wyobrazić sobie delikatny obrót układu współrzędnych zgodnie z ruchem wskazówek zegara, dzięki czemu żadne dwa punkty nie będą miały identycznej wartości współrzędnej y. Relacja „powyżej" i „poniżej", którą właśnie zdefiniowano, odpowiada relacji „powyżej" i „poniżej" na lekko obróconej płaszczyźnie.)

Na Rysunku 1. przedstawiono przykładową klasyfikację. Legenda kolorów jest następująca:

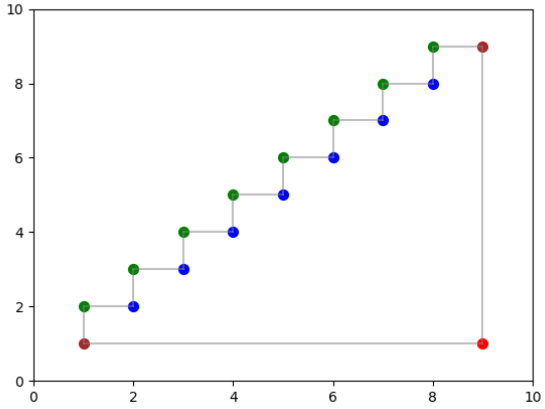
- w. początkowy - zielony,
- w. końcowy - czerwony,
- w. łączący - ciemnoniebieski,
- w. dzielący - jasnoniebieski,
- w. prawidłowy - brązowy.

Warto dodać, że wielokąty monotoniczne można zdefiniować również jako wielokąty, które nie posiadają wierzchołków dzielących ani łączących. Można zauważyć, że wielokąt na Rysunku 2. z pewnością nie jest monotoniczny.



Rysunek 1: Przykładowa klasyfikacja wierzchołków wielokąta

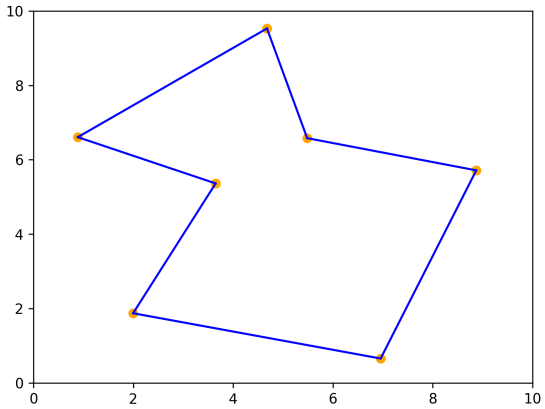
Na Rysunku 2. pokazano klasyfikację, w przypadku gdy punkty leżą na tej samej rzędnej. Widać, że punkty bardziej po lewej są uznawane za początkowe, a te po prawej za łączące - dokładnie tak jakbyśmy obrócili delikatnie wielokąt zgodnie ze wskazówkami zegara i dopiero wtedy klasyfikowali.



Rysunek 2: Klasyfikacja wierzchołków wielokąta nie ściśle monotonicznego

Monotoniczność wielokąta
Wielokąt nazywany jest monotonicznym względem prostej l, gdy jego brzeg można podzielić na dwa spójne łańcuchy, takie że przecięcia prostych prostopadłych do l z oboma łańcuchami są spójne. Przecięcie wielokąta z prostą jest spójne, wtedy i tylko wtedy gdy jest odcinkiem, punktem lub jest zbiorem pustym. Wielokąt

y - monotoniczny jest specyficznym przypadkiem wielokąta monotonicznego. Jest to wielokąt monotoniczny względem osi rzędnych. Przykład takiego wielokąta widnieje na Rysunku 3.

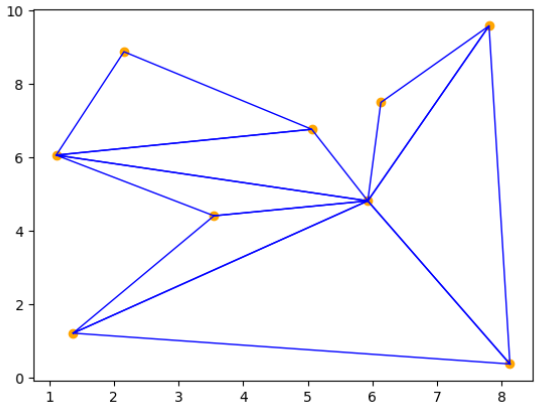


Rysunek 3: Przykład wielokąta y-monotonicznego

Triangulacja wielokąta prostego
Triangulacja wielokąta to podział wnętrza prostego wielokąta P na zbiór trójkątów $T_1, T_2, \ldots, T_k$ za pomocą przekątnych, przy czym:

- każda krawędź trójkąta jest albo krawędzią wielokąta, albo przekątną wielokąta,
- przekątne nie przecinają się wewnątrz wielokąta,
- suma obszarów wszystkich trójkątów $T_1, T_2, \ldots, T_k$ pokrywa cały obszar wielokąta P.

Na Rysunku 4. przedstawiono graficznie przykładową triangulację.



Rysunek 4: Graficzne przedstawienie triangulacji wielokąta prostego

Algorytmy zamiatania
Algorytmy zamiatania (ang. sweep algorithms) to techniki stosowane w informatyce, które wykorzystują wirtualną hiperpłaszczyznę w celu rozwiązywania problemów geometrycznych lub problemów związanych z porządkowaniem danych. Główna idea polega na systematycznym przetwarzaniu elementów przestrzeni w uporządkowany sposób za pomocą tzw. **miotły** (np. prostej), wzdłuż **kierunku zamiatania** (zwykle wzdłuż jednej osi). Pozycje, w których miotła się zatrzymuje, nazywa się **zdarzeniami**. W zdarzeniach aktualizowana jest struktura stanu. Na obszarze „zamiecionym" przez algorytm rozwiązanie badanego problemu jest już znane.

W algorytmie zamiatania do podziału na wielokąty monotoniczne:

- kierunek zamiatania: wzdłuż osi OX, w dół,
- zdarzenia: punkty wielokąta odpowiednio posortowane (opisane w implementacji poniżej),
- miotła: prosta postaci $y = k$,
- stan: zbiór aktywnych (opisane w implementacji poniżej) krawędzi posortowanych po x.

W algorytmie zamiatania do podziału na wielokąty monotoniczne zdefiniować można dodatkowo tzw. **pomocnika**. Pomocnikiem krawędzi e jest najniższy wierzchołek powyżej miotły, taki że poziomy odcinek łączący ten wierzchołek z e leży wewnątrz zamiatanego wielokąta.

4.1.2 Metodyka i implementacja algorytmów

Implementacja klasyfikacji wierzchołków
Klasyfikacja wierzchołków polega na iteracyjnym przetwarzaniu listy punktów, gdzie analizowane są trójki sąsiednich wierzchołków w celu określenia ich wzajemnego położenia. Wierzchołki początkowe i końcowe uwzględniono poprzez zastosowanie operacji modulo (`%N`, gdzie `N` oznacza długość listy), co umożliwia cykliczne odwoływanie się do elementów na liście przy przekroczeniu jej granic. Każdy punkt jest klasyfikowany zgodnie z klasyfikacją opisaną we wstępie teoretycznym. Sprawdzanie kąta tworzonego przez 3 sąsiednie punkty opiera się na funkcji `orient` obliczającej wyznacznik metodą Sarrusa. W zależności od znaku wyznacznika (z tolerancją dla zera $\epsilon = 10^{-12}$) określa ona, czy kąt wewnętrzny jest wklęsły, czy wypukły. Należy pamiętać, że funkcja ta zakłada uporządkowanie listy punktów w kierunku przeciwnym do ruchu wskazówek zegara. Dzięki temu kąt wewnętrzny zawsze znajduje się po lewej stronie wzdłuż kierunku iteracji zgodnej z tym uporządkowaniem.

Implementacja algorytmu podziału na wielokąty monotoniczne
Zanim zacznie się właściwa część algorytmu, należy przygotować konieczne struktury danych.

Strukturą stanu `T` jest `SortedSet`. W `T` przechowywane będą posortowane po odciętej aktywne półkrawędzi. Aktywna półkrawędź to taka, która przecina się z miotłą, a ponadto $intP$ (niech P oznacza triangulowaną figurę) jest na prawo od niej.

Zdarzenia przechowywane są na stosie `Q`. Algorytm zdejmował będzie kolejne zdarzenia ze stosu i je analizował. `Q` składa się z wierzchołków P (obiektów `Vertex`) posortowanych po y-kach rosnąco, **a w razie konfliktu** dodatkowo po x-ach malejąco. W efekcie, skoro zdejmujemy zdarzenia od końca, pobieramy punkty w kolejności od największego y-ka i najmniejszego x-a.

Wielokąt przechowywany jest jako obiekt `HalfEdgeMesh`. W funkcji `prepareHalfEdgeMesh` zamieniana jest lista punktów na siatkę Half-Edge. Wejściowy wielokąt jest prosty, zatem siatka będzie po prostu jedną ścianą o odpowiednio połączonych krawędziach i ustawionych polach `outgoingEdge` dla wierzchołków.

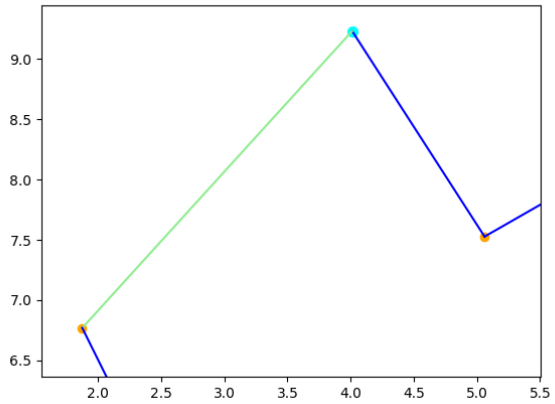
Do przechowywania rezultatu algorytmu przygotowywany jest graf nieskierowany (obiekt `Graph`). Wierzchołkami grafu są pola `id` wierzchołków. Na początku połączenia są dokładnie takie jak w wejściowym wielokącie. Z czasem, za pomocą `addDiagDiv` dodawane są kolejne krawędzie.

Główna część programu jest pętlą, która zdejmuje zdarzenie `event` (obiekt `Vertex`) z `Q`, ustawia wysokość miotły na `event.y`. Następnie w zależności od typu zdarzenia (`event.type`) zaczyna się odpowiednia procedura:

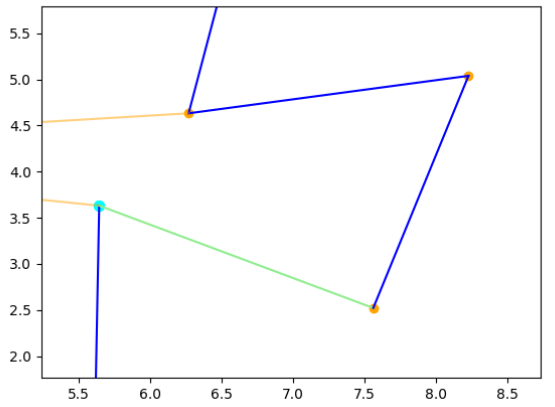
1. `handleStartVertex(v)`: Dodaje półkrawędź po lewej do `T` i ustawia pomocnika tej półkrawędzi na `v`. (Rysunek 5.)
2. `handleEndVertex(v)`: Usuwa półkrawędź po lewej z `T`. (Rysunek 6.)
3. `handleSplitVertex(v)`: Dodaje przekątną między `v`, a pomocnikiem krawędzi bezpośrednio po lewej. Ponadto dodaje półkrawędź po prawej do `T`. (Rysunek 7.)
4. `handleMergeVertex(v)`: Usuwa prawą półkrawędź powyżej i ustawia `v` na pomocnika krawędzi bezpośrednio na lewo. (Rysunek 8.)
5. `handleRegularVertex(v)`: Zachowuje się inaczej w zależności od tego, po której stronie od krawędzi leży

intP. Jeśli leży po prawej to: usuwa krawędź powyżej i dodaje tą poniżej do `T`. Jeśli zaś po lewej: ustawia pomocnika krawędzi bezpośrednio po lewej na `v`. (Rysunek 9.)

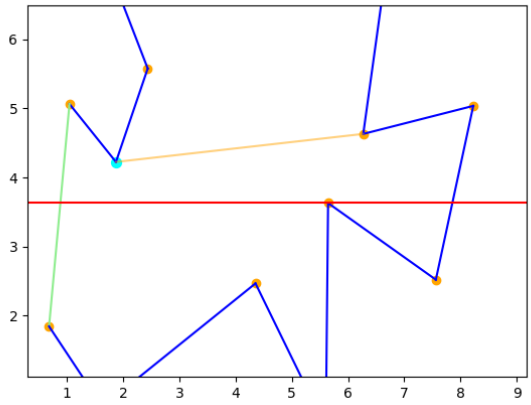
Ponadto w każdym z przypadków uwzględnia się przypadek, gdy pomocnikiem krawędzi bezpośrednio po lewej lub krawędzi poprzedzającej wierzchołek (w CCW) jest wierzchołek typu łączącego. Na poniższych Rysunkach 5-9. przedstawione są poszczególne przypadki. Na jasnoniebiesko zaznaczeni są pomocnicy, na zielono aktywne krawędzie. Na Rysunku 9. pokazano sytuację, w której należy dodać krawędź z wierzchołka prawidłowego do łączącego, który jest powyżej (widać bowiem, że pomocnik krawędzi, którą usuwamy, jest właśnie krawędzią łączącą). Aby sprawdzić, jaka krawędź jest bezpośrednio na lewo od danej, można dodać do struktury `T` odpowiednią "sondę" a następnie sprawdzić, co stoi na poprzedzającym je indeksie. W większości przypadków nie było to jednak konieczne, bo wystarczyło sprawdzić poprzednika odpowiedniej krawędzi znajdującej się już w strukturze (np. na Rysunku 7. chcąc sprawdzić krawędź na lewo od analizowanego wierzchołka, można sprawdzić poprzednika dopiero co włożonej krawędzi po prawej).



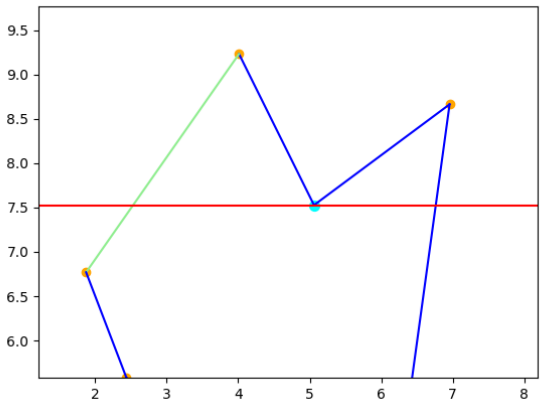
Rysunek 5: Graficzne przedstawienie `handleStartVertex`



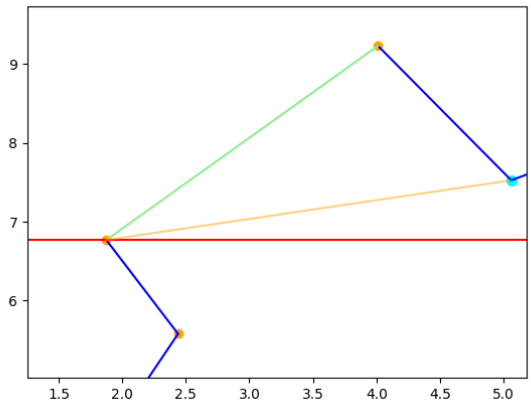
Rysunek 6: Graficzne przedstawienie `handleEndVertex`



Rysunek 7: Graficzne przedstawienie `handleSplitVertex` (przed dodaną krawędzią)

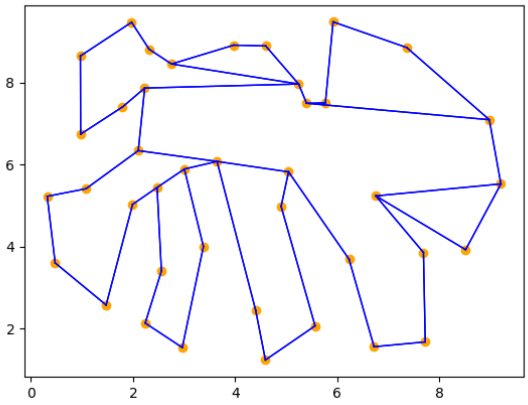


Rysunek 8: Graficzne przedstawienie `handleMergeVertex`



Rysunek 9: Graficzne przedstawienie `handleRegularVertex`

Efekt końcowy algorytmu podziału wygląda tak jak na Rysunku 10. Można zauważyć, że rzeczywiście każdy pomniejszy wielokąt jest y-monotoniczny.



Rysunek 10: Przykładowy wielokąt po podziale na wielokąty monotoniczne

Wyodrębnianie ścian i przekazywanie wyniku dekompozycji algorytmowi triangulacji wielokąta monotonicznego
W funkcji `updateFaces` graf `D` zamieniany jest na listę list wierzchołków (CCW) tworzących kolejne ściany. Proces rozpoczyna się od aktualizacji listy sąsiedztwa dla każdego wierzchołka. Lista ta jest sortowana w oparciu o kąty skierowane między osią OX a wektorem łączącym rozważany wierzchołek z jego sąsiadem. Punkt startowy dla obliczeń przyjmuje układ współrzędnych, którego początek znajduje się w aktualnie przetwarzanym wierzchołku.

Następnie tworzona jest lista `visited`, w której zaznaczane będą odwiedzone krawędzie. W `visited` osobno traktuje się krawędzie w obie strony. Krawędzi wejściowe (krawędzi pierwotnego wielokąta), które są skierowane CW zaznacza się jako odwiedzone, aby później nie przechodzić po zbędnej ścianie zewnętrznej.

Następnie zaczyna się przechodzenie po grafie. W każdym wierzchołku wychodzi się po następnych, w uprzednio ustalonym porządku, krawędziach. Jeśli krawędź nie jest odwiedzona, wchodzi się do następnego wierzchołka. Teraz wybiera się następną krawędź do wyjścia. Następną krawędzią wyjściową jest ta po lewej od krawędzi wejściowej (tej, po której algorytm wszedł do wierzchołka, zakładając skierowanie do wierzchołka). Wybranie tej krawędzi odbywa się w `prevIdx`. Procedura odbywa się tak długo, jak odwiedzane są nowe krawędzie. Na bieżąco dodawane są wierzchołki do listy `face`. Gdy procedura się kończy, face zapisywane jest w liście list `faces`.

Implementacja algorytmu triangulacji wielokąta monotonicznego
Algorytm rozdziela punkty na dwa łańcuchy - znajduje dowolny najwyższy punkt i dowolny najniższy punkt i odpowiednio klasyfikuje punkty na te "po lewej" i te "po prawej". Algorytm sortuje punkty wg wartości rzędnych malejąco. Potem tworzony jest stos z dwoma najwyższymi punktami i rozpoczyna się główna pętla. W głównej pętli odbywa się iteracja po uprzednio posortowanej liście punktów. Wewnątrz pętli algorytm rozpatruje dwa przypadki:

1. ostatni element stosu jest na tej samej gałęzi co aktualnie rozpatrywany wierzchołek,

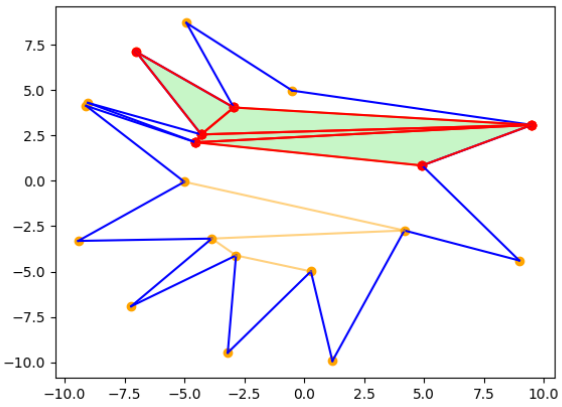
2. ostatni element stosu jest na drugiej gałęzi.

W przypadku 2. sytuacja jest prostsza: każdy kolejny trójkąt składający się z dwóch ostatnich elementów stosu i aktualnie rozpatrywanego punktu jest dokładany do wyniku triangulacji.

W przypadku 1. należy sprawdzić, czy trójkąt powstający w ten sposób jest w środku wielokąta. Zaimplementowane jest to bardzo podobnie do weryfikowania kąta wewnętrznego przy klasyfikacji punktów. Najpierw punkty są układane w kolejności przeciwnej do ruchu wskazówek zegara, a następnie obliczany jest odpowiedni wyznacznik za pomocą metody Sarrusa i sprawdza się jego znak (z tolerancją na zero $\epsilon = 10^{-12}$). Dopóki trójkąty otrzymywane są w środku wielokąta, zdejmowane są kolejne elementy ze stosu. Gdy jednak otrzymano trójkąt poza wielokątem, procedura jest przerywana.

Algorytm obsługujący triangulację

Algorytm obsługujący triangulację najpierw przepuszcza listę wierzchołków do algorytmu dekompozycji (uprzednio przygotowując wcześniej opisane struktury), a następnie każdą z otrzymanych ścian (jako listę wierzchołków CCW) przepuszcza przez algorytm triangulacji wielokątów monotonicznych. Z każdej ściany zapisuje dodane trójkąty jako trójki indeksów punktów do listy `allTriangles` przechowującej triangulację. Na Rysunku 11. pokazano wizualizację stanu wielokąta po podziale na wielokąty monotoniczne i striangulowaniu jednego z nich.



Rysunek 11: Wizualizacja stanu wielokąta po striangulowaniu jednego wielokąta monotonicznego.

4.1.3 Struktury danych - zalety i wady

SortedSet
Struktura stanu przechowywana jako `SortedSet` umożliwia szybkie operacje dodawania i usuwania krawędzi (złożoność $O(logn)$), a jednocześnie utrzymywania rosnącego porządku. Dzięki temu, nie trzeba co zdarzenie na nowo tworzyć całego porządku, a jedynie odpowiednio usuwać i dodawać elementy.`SortedSet` możnaby zastąpić dowolnym zrównoważonym drzewem wyszukiwań binarnych, jednak wydłużyłoby to znacznie implementację. Wadą zastosowanego rozwiązania były większe trudności ze znalezieniem błędów.

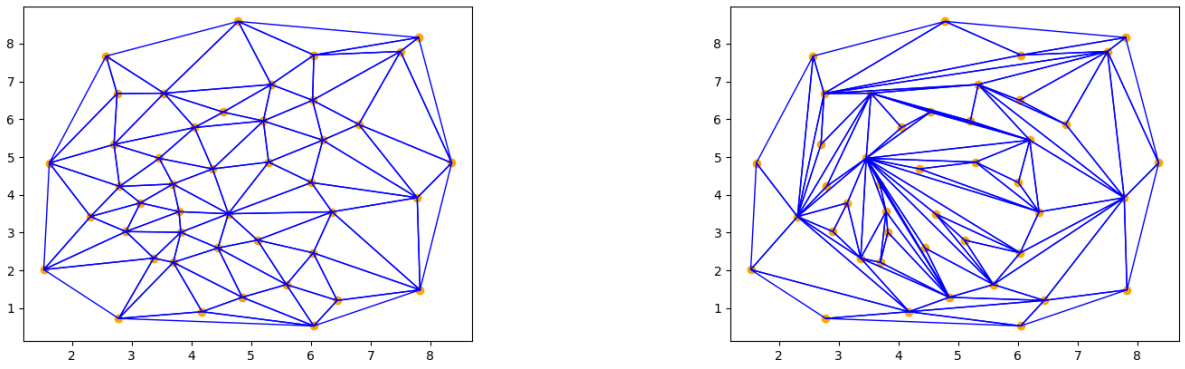
DCEL (Doubly-Connected Edge List) - Half-Edge
Struktura ta była uciążliwa w obsłudze. Przez trudności z aktualizowaniem odpowiednich pól przy dodawaniu przekątnych w strukturze Half-Edge, została ona zastąpiona przez zwykły graf (klasa `Graph`). Pozostała ona jedynie jako struktura opisująca wielokąt wejściowy w algorytmie `divide` podziału na wielokąty monotoniczne. Umożliwiła wygodny dostęp do informacji takich jak poprzednia krawędź dla danego wierzchołka.

4.2 Triangulacja Delaunaya

4.2.1 Część teoretyczna

Triangulacją zbioru punktów nazywamy maksymalny planarny podział punktów. Maksymalny planarny podział punktów, to taki podział, do którego nie da się dodać żadnej krawędzi tak, żeby nie przecinała się z żadną inną krawędzią.

Triangulacja Delaunaya to taka triangulacja, która maksymalizuje minimalny kąt występujący w triangulacji, w drugiej kolejności drugi najmniejszy itd. Formalnie: $A(T)$ to wektor niemalejąco uporządkowanych kątów występujących w triangulacji T: $A(T) = (\alpha_1, \alpha_2, \alpha_3, ...)$. Triangulacja T jest triangulacją Delaunaya, jeżeli $\forall_{T'} A(T) >= A(T')$ - wektory są porównywane leksykograficznie. Na Rysunku 12, widoczne jest porównanie triangulacji chmury punktów z użyciem triangulacji Delaunaya i przeciwnie. Można zauważyć, że trójkąty w triangulacji Delaunaya wyglądają "lepiej", są mniej wąskie i wynikowa triangulacja jest bardziej przejrzysta.

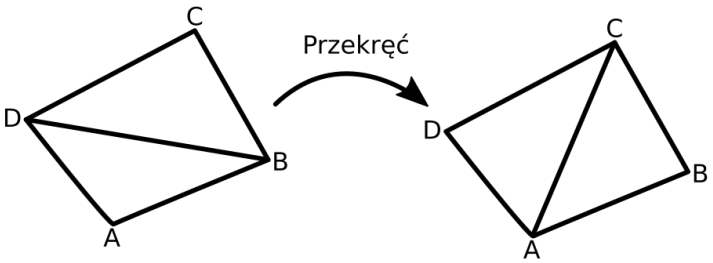


(a) Triangulacja Delaunaya

(b) Triangulacja jakakolwiek

Rysunek 12: Przedstawienie własności triangulacji Delaunaya

Przekręcanie krawędzi To operacja polegająca, na zamienianiu przekątnej w czworokącie wypukłym. (Rysunek 13)



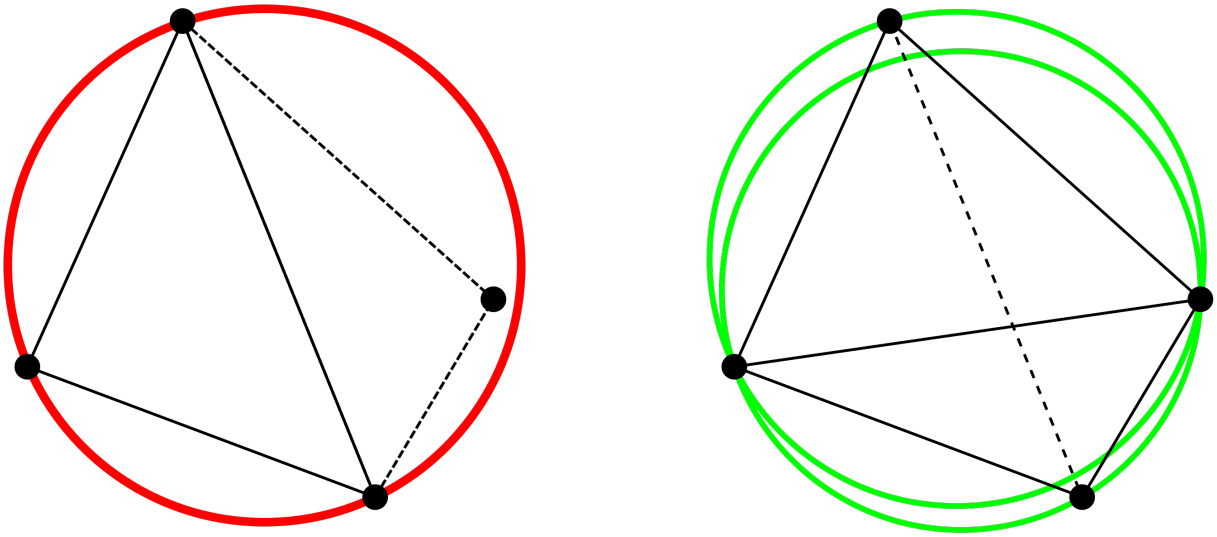
Rysunek 13: Ilustracja operacji przekręć

Krawędź nielegalna to taka krawędź, której przekręcenie zwiększy lokalnie najmniejszy kąt. Natomiast krawędź legalna to krawędź, której przekręcenie nie zmieni bądź zmniejszy najmniejszy kąt. Można udowodnić, że jeżeli wszystkie krawędzie w triangulacji są legalne, to jest to triangulacja Delaunaya.

Predykat do sprawdzenia legalności krawędzi
Aby sprawdzić czy dana krawędź jest legalna, wystarczy sprawdzić czy okrąg opisany na dwóch punktach tej krawędzi oraz którymś z dwóch pozostałych punktów zawiera czwarty punkt. Natomiast aby to sprawdzić wystarczy policzyć znak odpowiedniego wyznacznika(Rysunek 14). Rysunek 15 ilustruje działanie tego testu.

$$\begin{bmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ p_x & p_y & p_x^2 + p_y^2 & 1 \end{bmatrix}$$

Rysunek 14: Wyznacznik



(a) Krawędź jest nielegalna, punkt wewnątrz koła opisanego

(b) Krawędź legalna(po przekręceniu), punkt poza kołem opisanym

Rysunek 15: Przedstawienie sprawdzania legalności krawędzi za pomocą testu z okręgiem opisanym. Źródło: `https://en.wikipedia.org/wiki/Delaunay_triangulation`

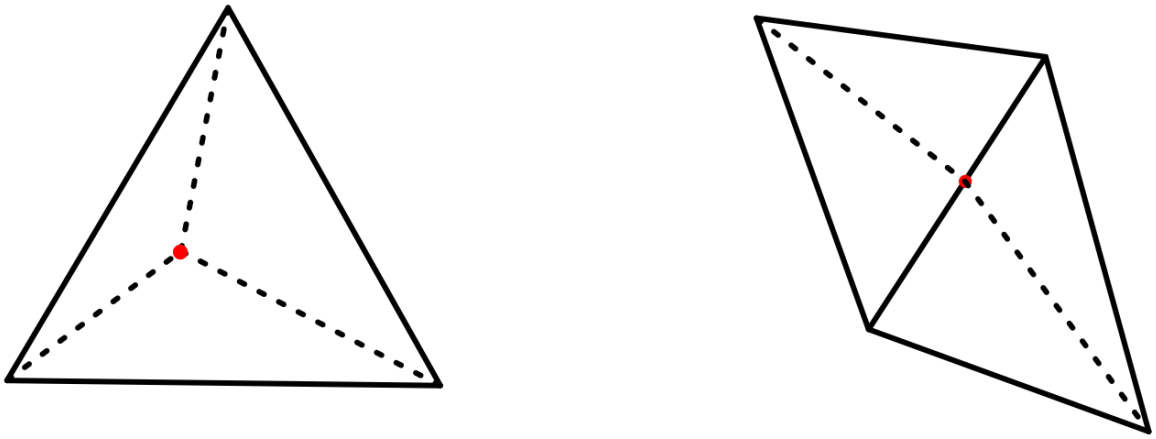
Algorytm naiwny (Local Optimization Procedure)
Triangulacje Delaunaya można uzyskać poprzez stworzenia dowolnej triangulacji zbioru punktów, a następnie przekręcaniu nielegalnych krawędzi, tak długo, aż wszystkie będą legalne. Ponieważ zmiana krawędzi na legalną zwiększa leksykograficznie wektor kątów triangulacji, to proces ten na pewno kiedyś się skończy.

Algorytm inkrementalny

Algorytm inkrementalny polega na dodawaniu kolejnych punktów i utrzymywaniu triangulacji Delaunaya. Zaczynamy od stworzenia supertrójkąta, w którym zmieszczą się wszystkie punkty.

Następnie operacja dodania punktu polega na:

1. Odnalezieniu trójkąta, w którym znajduje się punkt.
2. Dodaniu odpowiednich trójkątów powstałych na skutek wstawienia punktu. (Rysunek 16)
3. Przywrócenie własności Delaunaya poprzez rekurencyjne przekręcanie krawędzi, które mogły przestać być legalne.



(a) Dodawanie nowego punktu w środku trójkąta (b) Dodawanie nowego punktu na krawędzi

Rysunek 16: Dodawanie nowego punktu, w algorytmie inkrementacyjnym

Sposób znajdowania trójkąta zawierającego zadany punkt

Do lokalizacji trójkąta zawierającego punkt używany jest acykliczny graf skierowany, w którym występują trójkąty, które historycznie istniały podczas triangulacji, a krawędzie prowadzą od wcześniej istniejących trójkątów do nowo powstałych na ich miejscu. Aby odnaleźć szukany trójkąt przechodzimy przez graf zaczynając od wierzchołka supertrójkąta i przechodzimy dowolną krawędzią prowadzącą do trójkąta zawierającego dodawany punkt. Ostatecznie gdy stopień wyjściowy wierzchołka równy jest zero znaleziony został odpowiedni trójkąt.

Złożoność

Dodatkowo aby poprawić oczekiwaną złożoność tego algorytmu, punkty dodawne są w losowej kolejności. Skutkuje to oczekiwaną złożonością $\Theta(nlogn)$.

Triangulacja Delaunaya z ograniczeniami

Triangulacja Delaunaya z ograniczeniami dodaje jeszcze możliwość wymuszenia istnienia niektórych krawędzi. Wymuszone krawędzie nie mogą się przecinać (inaczej wymagałoby to dodania punktów). Triangulacja Delaunaya z ograniczeniami nie jest triangulacją Delaunaya, jednak również maksymalizuje minimalne kąty, biorąc pod uwagę zadane ograniczenia.

1. Przeprowadzamy zwykłą triangulację Delaunaya.
2. Dla każdej wymuszonej krawędzi:

    1. Jeżeli zadana krawędź znajduje się już w triangulacji, nie trzeba nic robić.

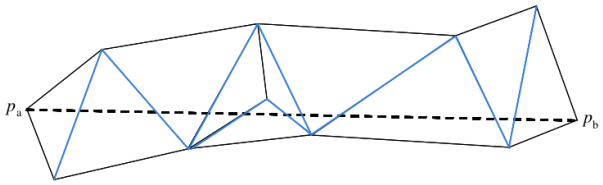
    2. Znajdujemy krawędzie przecinające się z krawędzią wymuszoną.

    3. Odpowiednio przekręcamy krawędzie, tak żeby przestały przecinać krawędź wymuszoną.

    4. Przywracamy triangulację Delaunaya, nie przekręcamy wymuszonej krawędzi. Można udowodnić, że wystarczy jedynie sprawdzić legalność przekręconych krawędzi. Używamy do tego algorytmu LOP.

Znajdowanie krawędzi przecinających się z zadaną krawędzią
Aby znaleźć krawędzie przecinające się z zadaną krawędzią najpierw przechodzimy przez krawędzie wychodzące z pierwszego wierzchołka krawędzi, aby znaleźć pierwszą przecinającą krawędź, a następnie przechodzimy po

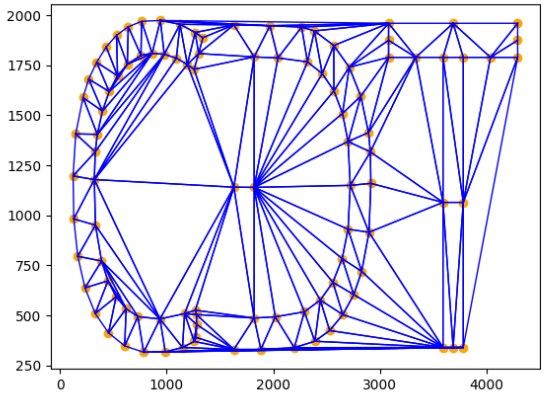
siatce do drugiego wierzchołka znajdując po drodze wszystkie krawędzie przecinające zadaną krawędź.



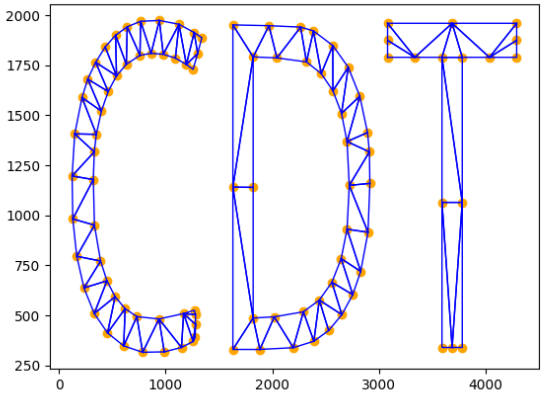
Rysunek 17: Rysunek przedstawiający krawędzie przecinające się z krawędzią $p_a p_b$

Złożoność tego rozwiązania ciężko określić, jednak w praktycznych przykładach zazwyczaj liczba krawędzi przecinających krawędź wymuszoną jest niewielka. Na dodatek nawet w przypadku gdy liczba przecinających się krawędzi nie jest bardzo mała algorytm wypada relatywnie dobrze. W pracy naukowej S. W. Sloan A Fast Algorithm For Generating Constrained Delaunay Triangluations 1992, empiryczne pomiary przeprowadzone przez autora pokazywały średnią złożoność około $O(N^{1.12})$, muszę jednak zaznaczyć, że nie jest to złożoność pesymistyczna, a jedynie zaobserwowane działanie dla losowych zbiorów testowych.

Usuwanie zewnętrznych
Przy usuwaniu trójkątów zewnętrznych, zakładamy, że z lewej strony od wymuszonych krawędzi znajdują się wnętrza wielokątów, natomiast z prawej zewnętrza. Takie podejście pozwala na triangulację wielokątów podanych w kierunku przeciwnym do ruchu wskazówek zegara, mogących zawierać dziury (podane w kolejności przeciwnej do ruchu wskazówek zegara). Aby znaleźć trójkąty wewnętrzne używamy algorytmu BFS z wielu źródeł - z każdego trójkąta, będącego po lewej stronie od wymuszonej krawędzi. Następnie przechodzimy po sąsiadujących trójkątach, jednak nie przekraczamy wymuszonych krawędzi. Wynik działania procedury usuwania trójkątów zewnętrznych widoczny jest na rysunkach 18 i 19.



Rysunek 18: Wynik bez usuwania trójkątów zewnętrznych



Rysunek 19: Wynik z usuwaniem trójkątów zewnętrznych

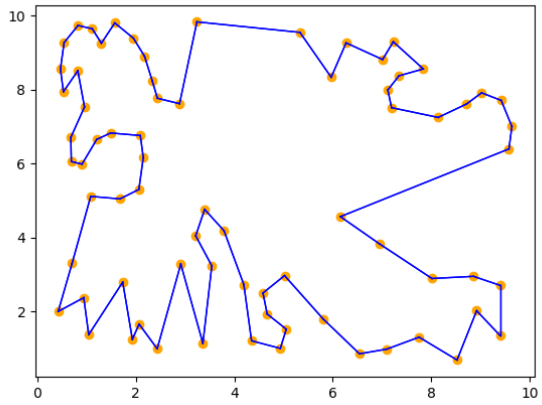
Uwagi:
Alternatywnie triangulacje Delaunaya można przeprowadzać między innymi za pomocą algorytmu typu dziel i zwyciężaj lub zamiatania. Dodatkowo triangulację z ograniczeniami można uzyskać triangulując, powstałą poprzez usunięcie krawędzi przecinających się z wymuszoną krawędzią lukę.

4.3 Porównanie metod

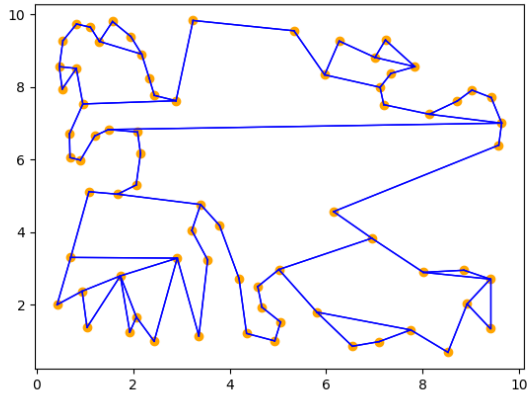
4.3.1 Zbiory testowe

Na potrzeby porównania algorytmów przygotowano następujące wielokąty testowe:

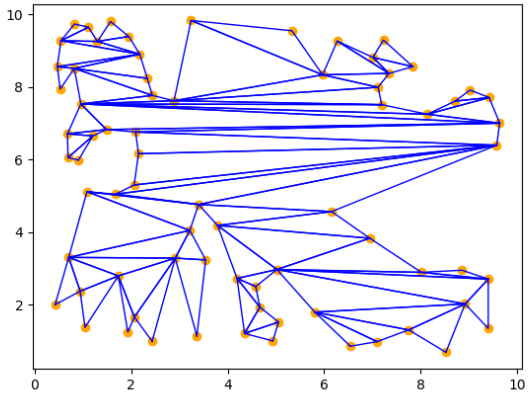
Wielokąt A



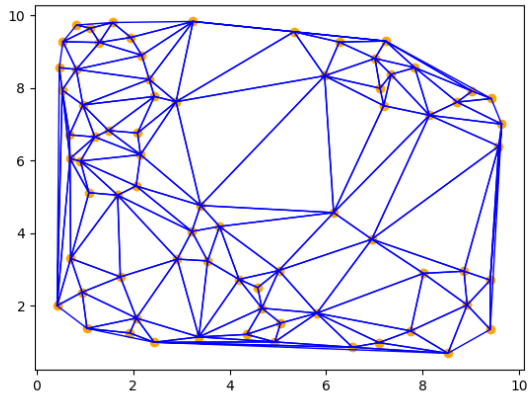
Rysunek 20: Wielokąt A



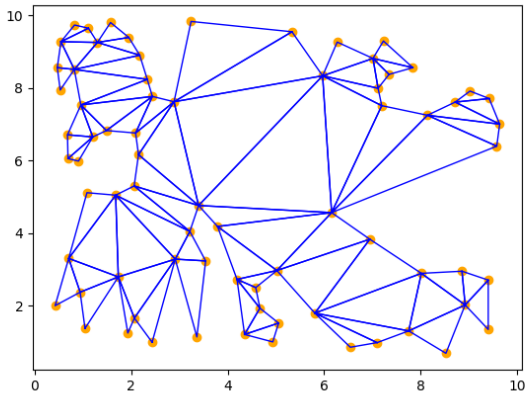
Rysunek 21: Wielokąt A po podziale na wielokąty monotoniczne



Rysunek 22: Efekt końcowy triangulacji wielokąta A poprzez podział na wielokąty monotoniczne



Rysunek 23: Triangulacja Delaunaya wielokąta A bez ograniczeń



Rysunek 24: Efekt końcowy triangulacji - po dodaniu ograniczeń i usunięciu zewnętrznych trójkątów

Wielokąt A znajduje się na Rysunku 20. Na Rysunkach 21. i 22. widać kolejne etapy algorytmu podziału na wielokąty monotoniczne. Algorytm najpierw dzieli na wielokąty monotoniczne, a następnie trianguluje każdy

wielokąt osobno.

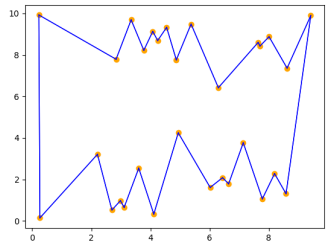
Na Rysunkach 23. i 24. widać kolejne etapy działania algorytmu triangulacji Delaunaya. Najpierw przeprowadzana jest zwykła triangulacja Delaunaya, a następnie dodawane są ograniczenia i usuwane krawędzie zewnętrzne.

Obie metody działają inaczej i tworzą różną triangulację. Ocena rezultatów triangulacji dla wielokąta A i pozostałych znajduje się w Tabeli 2.

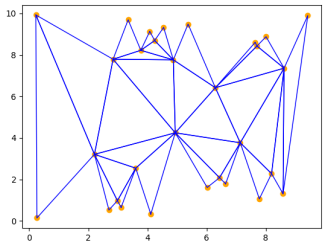
Kryterium oceny triangulacji Jako kryterium oceny triangulacji wybrano **średnią z minimalnych kątów dla każdego powstałego trójkąta**. Wybór ten uzasadnić można następująco:

1. **Stabilność numeryczna:**
   - Wąskie kąty (bliskie 0°) mogą prowadzić do niestabilności numerycznych w obliczeniach, szczególnie w zastosowaniach takich jak metoda elementów skończonych (FEM).
2. **Unikanie degeneracji:**
   - Niska wartość minimalnego kąta wskazuje na obecność „długich i wąskich" trójkątów, które są niepożądane w wielu zastosowaniach geometrycznych i numerycznych.
   - Maksymalizowanie minimalnego kąta prowadzi do bardziej równomiernych kształtów trójkątów.
3. **Sprawiedliwa ocena całej triangulacji:**
   - Średnia minimalnych kątów uwzględnia jakość wszystkich trójkątów w triangulacji, a nie tylko jednego najgorszego. Zapewnia to bardziej globalną ocenę.

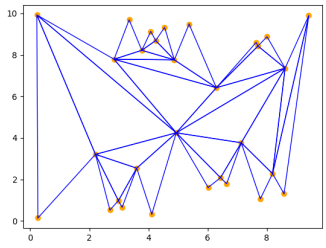
Wielokąt B



Rysunek 25: Graficzna reprezentacja wielokąta B

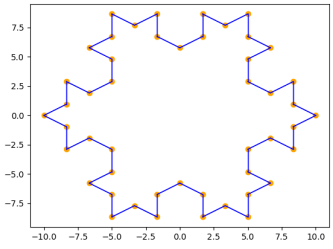


Rysunek 26: Wizualny efekt triangulacji Delaunaya

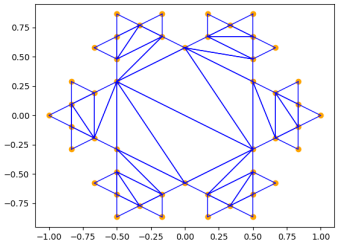


Rysunek 27: Wizualny efekt triangulacji przez podział

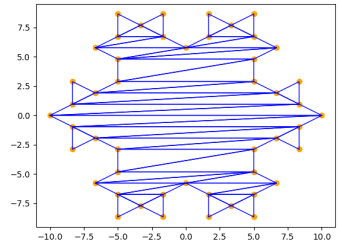
Wielokąt C



Rysunek 28: Graficzna reprezentacja wielokąta C



Rysunek 29: Wizualny efekt triangulacji Delaunaya



Rysunek 30: Wizualny efekt triangulacji przez podział

4.4 Porównanie jakości triangulacji

Tabela 2: Średnie minimalne kąty dla triangulacji różnych wielokątów

Wielokąt	Delaunay (°)	Podział na monotoniczne (°)
A	31.02	21.81
B	24.19	21.30
C	38.90	22.98

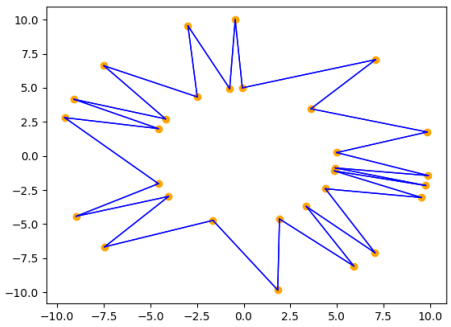
Z Tabeli 2 wynika, że w każdym przypadku trójkąty uzyskane w wyniku triangulacji są bardziej jakościowe. Największa różnica występuje dla wielokąta C (Rysunki 28–30). W przypadku, gdy punkty są gęsto upakowane wzdłuż osi OY, triangulacja poprzez podział na wielokąty monotoniczne prowadzi do powstawania wąskich i długich trójkątów.

Dla wielokąta B różnica miary jakości jest niewielka, co zgadza się z intuicją wynikającą z Rysunków 27 i 26 (triangulacje są bardzo podobne).

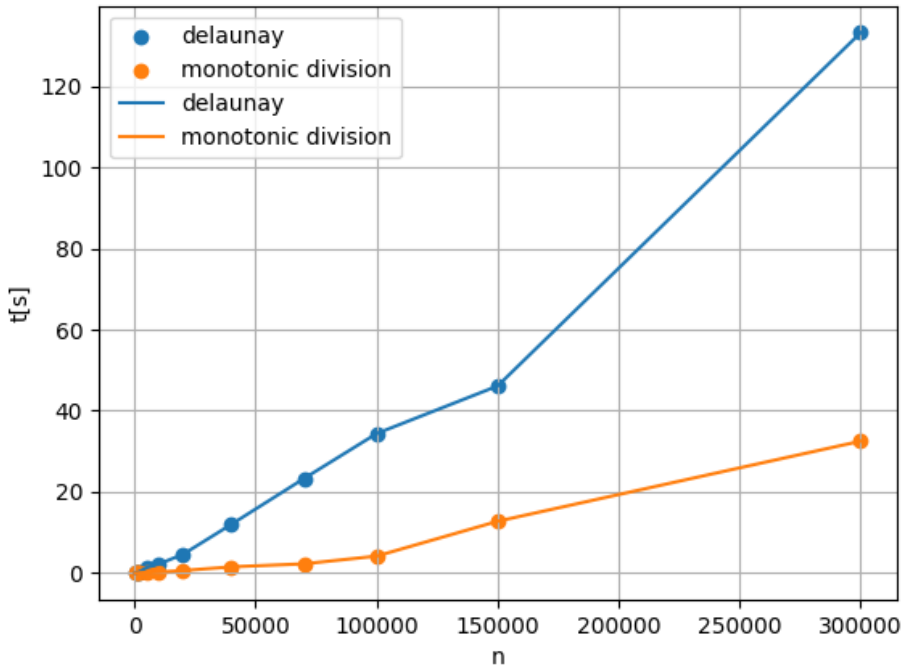
Algorytm Delaunaya tworzy lepsze trójkąty. Czasem jednak, różnica jest niewielka. Wówczas należy rozważyć użycie algorytmu nieco szybszego - triangulacji poprzez podział na wielokąty monotoniczne.

4.5 Wydajność

Zbiory testowe użyte do testowania wydajności algorytmów generowane były w pliku `generate_sun_like_figure.py`. Tworzone były na zasadzie losowego generowania punktów na dwóch okręgach o określonych promieniach i odpowiednim łączeniu ich. Przykład takiej figury znajduje się na Rysunku 31. Z wykresu wydajności można odczytać, że triangulacja za pomocą podziału na wielokąty monotoniczne jest od około 5 do 10 razy szybsza. Również można zauważyć, że faktycznie oba algorytmy działają w czasie zbliżonym do $O(nlogn)$, wzrost czasu jest mniej więcej liniowo proporcjonalny do n.



Rysunek 31: Generowany losowo zbiór dla małego n



Rysunek 32: Wykresy czasu triangulacji zbioru testowego: losowa gwiazdka o n punktach

4.6 Podsumowanie

W ramach niniejszego sprawozdania przeanalizowano dwie metody triangulacji wielokąta prostego: triangulację Delaunaya oraz triangulację przez dekompozycję do wielokątów monotonicznych. Celem było zbadanie wydajności oraz jakości obu algorytmów. Wnioski z przeprowadzonych testów jednoznacznie wskazują na różnice w charakterystyce obu podejść, co pozwala na sformułowanie konkretnych rekomendacji dotyczących ich użycia w zależności od priorytetów projektowych.

Triangulacja Delaunaya, generuje mniej wąskich, długich trójkątów, które powodują problemy w wielu zastosowaniach. Jednakże wysoka jakość wynikowej triangulacji odbywa się kosztem czasu obliczeń. Nasze testy wykazały, że czas wykonania algorytmu Delaunaya jest wyraźnie dłuższy niż w przypadku metody podziału na wielokąty monotoniczne (Rysunek 32), co może stanowić istotne ograniczenie przy dużych zbiorach danych.

Z kolei metoda dekompozycji wielokątów na części monotoniczne oferuje znacznie lepszą wydajność obliczeniową. Dzięki podziałowi wielokąta na prostsze komponenty możliwa jest szybka triangulacja każdego z nich w czasie liniowym względem liczby wierzchołków. Nasze eksperymenty wykazały, że metoda ta jest od 5 do 10 razy szybsza niż triangulacja Delaunaya, co czyni ją bardziej praktycznym rozwiązaniem w sytuacjach, gdzie kluczowym czynnikiem jest czas obliczeń. Niemniej jednak, w porównaniu z triangulacją Delaunaya, siatki generowane przez tę metodę mogą zawierać mniej optymalne trójkąty (Tabela 2).

W praktyce wybór odpowiedniej metody triangulacji powinien być uzależniony od specyfiki problemu. Jeśli priorytetem jest uzyskanie możliwie najlepszej jakości siatki, a czas obliczeń nie stanowi kluczowego ograniczenia, to triangulacja Delaunaya będzie zdecydowanie lepszym wyborem. Z drugiej strony, jeśli kluczowe jest szybkie przetwarzanie danych i generowanie siatek trójkątów, szczególnie w aplikacjach wymagających skalowalności, metoda podziału na wielokąty monotoniczne okazuje się bardziej efektywna.

4.7 Źródła

1. Computational Geometry: Algorithms and Applications by Mark de Berg, Otfried Cheong Third Edition, Marc van Kreveld, Mark Overmars

2. Wykłady Algorytmy Geometryczne dr. Barbary Głut

3. Triangulations and Applications by Øyvind Hjelle, Morten Dæhlen
4. https://cp-algorithms.com/geometry/planar.html
5. https://ianthehenry.com/posts/delaunay/
6. A Fast Algorithm For Generatin Constrained Delaunay Triangulations by S. W. Sloan
7. Polygon partitioning algorithms presentation
8. https://tchayen.com/constrained-delaunay-triangulation-from-a-paper