

## Algorytmy Geometryczne

Projekt zaliczeniowy

# Lokalizacja punktu w przestrzeni dwuwymiarowej

Metoda Separatorów

**Autorzy:** Gabriel Wermański, Jacek Łoboda

**Grupa:** 6

**Data wykonania:** 08.01.2026

## 1 Środowisko obliczeniowe

Eksperymenty zostały przeprowadzone w następującej konfiguracji:

<b>Procesor:</b>	CPU 12th Gen Intel(R) Core(TM) i5-1235U, 1.30 GHz
<b>Pamięć RAM:</b>	16.0 GB, 3200 MT/s
<b>System operacyjny:</b>	Microsoft Windows 11 Home
<b>Środowisko programistyczne:</b>	Jupyter Notebook
<b>Język programowania:</b>	Python 3.13.5
<b>Biblioteki:</b>	numpy, matplotlib, functools, typing, math, bitalg
<b>Precyzja obliczeń:</b>	float64, epsilon = $10^{-24}$

## 2 Cel projektu

Celem projektu było zaimplementowanie algorytmu lokalizacji punktu w dowolnym podziale planarnym przy użyciu metody separatorów. Algorytm pozwala na efektywne określenie, w którym regionie podziału płaszczyzny znajduje się zadany punkt testowy.

Projekt obejmował:

- Implementację algorytmu wstępnego przetwarzania,
- Implementację algorytmu zapytań (lokalizacja punktu),
- Przygotowanie wizualizacji działania algorytmu,
- Obsługę dowolnych podziałów poligonowych z triangulacją,
- Przeprowadzenie testów na różnorodnych zbiorach danych,
- Analizę efektywności algorytmu.

## 3 Wstęp teoretyczny

### 3.1 Problem lokalizacji punktu

Dla zadanego podziału płaszczyzny na regiony  $R_0, R_1, \dots, R_n$  oraz punktu testowego  $p$ , problem lokalizacji punktu polega na znalezieniu regionu  $R_i$  zawierającego punkt  $p$ .

Naiwne rozwiązanie sprawdzające przynależność punktu do każdego regionu osobno ma złożoność  $O(n)$  dla pojedynczego zapytania. Metoda separatorów znajdzie tę przynależność w czasie logarytmicznym.

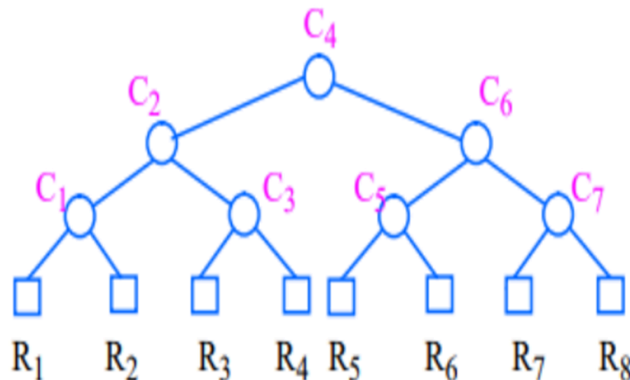
### 3.2 Metoda separatorów

Metoda separatorów wykorzystuje specjalne struktury geometryczne do dekompozycji płaszczyzny. Separator to ciąg odcinków, w którym każda prosta pozioma przecina łańcuch w co najwyżej jednym punkcie.

**Kluczowe właściwości:**

- Współrzędna  $y$  wierzchołków jest monotoniczna wzdłuż łańcucha
- Pozycję punktu względem łańcucha można sprawdzić w czasie  $O(\log n)$  używając wyszukiwania binarnego
- Płaszczyzna jest podzielona na regiony ograniczone przez pary łańcuchów
- Każdy łańcuch (separator) separuje regiony na te położone na lewo i na prawo od niego.

### 3.3 Drzewo wyszukiwania binarnego BST



Rys. 3.1 Zobrazowanie struktury drzewa

Algorytm wykorzystuje binarne drzewo wyszukiwania o budowie widocznej na Rys. 1.

- **Węzły wewnętrzne** zawierają łańcuchy monotoniczne separujące regiony
- **Liście** odpowiadają pozycjom między łańcuchami (regionom)
- Dla węzła z łańcuchem  $C_k$ : regiony po lewej stronie łańcucha znajdują się w lewym poddrzewie, regiony po prawej - w prawym poddrzewie
- Drzewo jest budowane rekurencyjnie przez wybór mediany z posortowanych łańcuchów

Każdy odcinek jest przechowywany w łańcuchu tylko raz, co zapewnia liniową złożoność pamięciową.

### 3.4 Złożoność algorytmu

Metoda separatorów osiąga następujące złożoności:

**Wstępne przetwarzanie:**  $O(n^2)$  gdzie  $n$  to liczba wierzchołków grafu planarnego

- Konstrukcja grafu skierowanego (DAG):  $O(n)$
- Sortowanie kątowe krawędzi:  $O(n \log n)$
- Obliczanie wag planarnych (dwa przejścia):  $O(n^2)$
- Generowanie łańcuchów:  $O(n)$
- Budowa drzewa wyszukiwania:  $O(n \log n)$
- Prekomputacja regionów:  $O(n^2)$

**Zapytanie:**  $O(\log^2 n)$

- Przeszukiwanie drzewa:  $O(\log n)$  poziomów
- Wyszukiwanie binarne segmentu na każdym poziomie:  $O(\log n)$
- Wyszukiwanie binarne odpowiedniego „bąbla” w regionie:  $O(\log K)$  gdzie  $K$  to liczba bąbli

**Pamięć:**  $O(n)$  - liniowa złożoność pamięciowa

## 4 Wstępne przetwarzanie - budowa struktury danych

### 4.1 Przebieg realizacji

1. Przygotowano procedury do interaktywnego wprowadzania wielokątów oraz ich zapisu i odczytu.
2. Zaimplementowano automatyczną triangulację wielokątów metodą Odcinania uszu.
3. Zaimplementowano konwersję podziału poligonowego na graf planarny z krawędziami skierowanymi.
4. Zaimplementowano algorytm obliczania wag planarnych metodą dwuprzebiegową.
5. Zaimplementowano generator łańcuchów monotonicznych.
6. Zaimplementowano budowę drzewa wyszukiwania binarnego.
7. Zaimplementowano prekomputację regionów dla szybkiego wyszukiwania.
8. Przygotowano wizualizacje zbiorów testowych i wyników działania algorytmów.
9. Przeprowadzono testy na różnorodnych zbiorach danych.

### 4.2 Implementacja struktur danych

#### 4.2.1 Klasa Vertex

Każdy wierzchołek grafu jest reprezentowany przez obiekt przechowujący:

- Współrzędne  $x$  i  $y$  wierzchołka
- Listę krawędzi wychodzących (skierowanych w górę) wraz z ich wagami
- Listę krawędzi wchodzących (od dołu) wraz z ich wagami
- Skumulowane wagi przepływu wchodzącego i wychodzącego

#### 4.2.2 Klasa MonotoneChain

Łańcuch monotoniczny przechowuje:

- Listę wierzchołków tworzącą ścieżkę łańcucha
- Listę segmentów (par kolejnych wierzchołków) dla szybkiego wyszukiwania
- Unikalny identyfikator łańcucha używany do mapowania regionów

Segmenty łańcucha są posortowane według współrzędnej  $y$ ,  $x$  co umożliwia efektywne wyszukiwanie binarne podczas zapytań.

#### 4.2.3 Klasa SearchTreeNode

Węzeł drzewa wyszukiwania zawiera:

- Referencję do segmentów łańcucha dla szybkiego dostępu
- Referencję do pełnego obiektu łańcucha
- Wskaźniki do lewego i prawego dziecka w drzewie
- Opcjonalny wskaźnik do rodzica

### ■ 4.3 Graf planarny jako DAG

Pierwszym krokiem wstępnego przetwarzania jest transformacja podziału planarnego w skierowany graf acykliczny:

#### **Sortowanie wierzchołków:**

1. Wszystkie wierzchołki są sortowane rosnąco według współrzędnej  $y$
2. Przy równych wartościach  $y$  sortuje się według  $x$

#### **Orientacja krawędzi:**

1. Dla każdej krawędzi wyznaczane są jej punkty końcowe
2. Krawędź jest zawsze skierowana od wierzchołka „niższego” do „wyższego”
3. Niższy oznacza mniejszą współrzędną  $y$ , lub przy równych  $y$  - mniejszą  $x$
4. Dzięki temu wszystkie krawędzie są skierowane „w górę”

#### **Listy sąsiedztwa:**

1. Dla każdego wierzchołka tworzona jest lista krawędzi wychodzących
2. Równocześnie dla każdego wierzchołka tworzona jest lista krawędzi wchodzących
3. Każda krawędź ma początkową wagę równą 1
4. Te wagi będą modyfikowane w kolejnym kroku

Taka reprezentacja zapewnia, że graf nie zawiera cykli i może być przetwarzany topologicznie od źródła (najniższego wierzchołka) do ujścia (najwyższego wierzchołka).

### ■ 4.4 Obliczanie wag planarnych

Wagi krawędzi określają liczbę łańcuchów monotonicznych przechodzących przez każdą krawędź. Algorytm wykorzystuje koncepcję przepływu w grafie planarnym.

#### **4.4.1 Sortowanie kątowe krawędzi**

Przed obliczeniem wag wykonywane jest sortowanie kątowe:

##### **Dla każdego wierzchołka:**

1. Pobierana jest pozycja wierzchołka jako punkt centralny
2. Wszystkie krawędzie wychodzące są sortowane przeciwnie do ruchu wskazówek zegara
3. Sortowanie wykorzystuje iloczyn wektorowy do określenia orientacji

##### **Iloczyn wektorowy:**

1. Dla trzech punktów  $o$ ,  $a$ ,  $b$  obliczany jest iloczyn wektorowy
2. Wzór:  $(a_x - o_x)(b_y - o_y) - (a_y - o_y)(b_x - o_x)$
3. Wartość dodatnia oznacza skręt w lewo (CCW)
4. Wartość ujemna oznacza skręt w prawo (CW)
5. Wartość bliska zeru oznacza współliniowość

#### **4.4.2 Przebieg w górę**

Algorytm przetwarza wierzchołki w kolejności topologicznej (od dołu do góry):

##### **Dla każdego wierzchołka $v$ :**

1. Obliczana jest suma wag krawędzi wchodzących
2. Obliczana jest liczba krawędzi wychodzących
3. Jeśli suma wag wchodzących jest większa niż liczba krawędzi wychodzących:
  - Nadmiar przepływu musi zostać przekazany dalej
  - Wybierana jest skrajnie lewa krawędź wychodząca

- Jej waga jest zwiększana o różnicę przepływów
- Ta krawędź będzie teraz „szerszym kanałem” dla przepływu

Jeśli do wierzchołka wpływa więcej łańcuchów niż mamy krawędzi do ich rozdzielenia, musimy niektóre łańcuchy „połączyć” w jeden szerszy strumień. Wybieramy skrajnie lewą krawędź, aby zachować spójność kierunku.

#### 4.4.3 Przebieg w dół

Algorytm przetwarza wierzchołki w odwrotnej kolejności topologicznej (od góry do dołu):

**Dla każdego wierzchołka  $v$ :**

1. Ponownie obliczana jest suma wag wchodzących
2. Ponownie obliczana jest suma wag wychodzących (po modyfikacjach z przebiegu w górę)
3. Jeśli suma wag wychodzących jest większa niż suma wag wchodzących:
  - Brakuje przepływu przychodzącego
  - Wybierana jest skrajnie lewa krawędź wchodząca
  - Jej waga jest zwiększana o różnicę przepływów
  - „Ściągamy” dodatkowy przepływ z góry

W przejściu w górę mogliśmy utworzyć sytuację, gdzie z wierzchołka wypływa więcej niż wpływa. Przejście w dół koryguje te niezgodności, „docinając” odpowiednie przepływy od góry.

Po zakończeniu obu przejść, dla każdego wierzchołka (poza źródłem i ujściem) suma wag wchodzących równa się sumie wag wychodzących. To jest warunek konieczny do prawidłowego wyodrębnienia łańcuchów.

## 4.5 Generowanie łańcuchów monotonicznych

**Inicjalizacja:**

1. Identyfikowany jest wierzchołek źródłowy (najniższy punkt)
2. Zliczana jest całkowita liczba łańcuchów (suma wag krawędzi wychodzących ze źródła)
3. Tworzona jest odpowiednia liczba pustych obiektów łańcuchów
4. Każdy łańcuch otrzymuje unikalny identyfikator

**Dla każdego łańcucha:**

1. Rozpoczyna się od wierzchołka źródłowego
2. Dodawany jest bieżący wierzchołek do ścieżki łańcucha
3. Sprawdzane są wszystkie krawędzie wychodzące z bieżącego wierzchołka
4. Wybierana jest skrajnie prawa krawędź o niezerowej wadze
5. Waga tej krawędzi jest zmniejszana o 1
6. Algorytm przechodzi do kolejnego wierzchołka
7. Proces kontynuowany jest aż do osiągnięcia ujścia (wierzchołka bez krawędzi wychodzących)

**Tworzenie segmentów:**

1. Po zebraniu wszystkich wierzchołków łańcucha
2. Tworzone są segmenty jako pary kolejnych wierzchołków
3. Każdy segment jest odcinkiem od punktu  $(x_i, y_i)$  do  $(x_{i+1}, y_{i+1})$
4. Segmenty są automatycznie posortowane według  $y$  dzięki topologicznej konstrukcji

Wybór skrajnie prawej krawędzi w każdym kroku zapewnia, że łańcuchy są generowane w spójnej kolejności kątowej, co jest istotne dla późniejszego separowania regionów.

## ■ 4.6 Konstrukcja drzewa wyszukiwania

Łańcuchy są organizowane w binarne drzewo wyszukiwania:

### **Podejście rekurencyjne:**

1. Wybierany jest łańcuch medianowy
2. Tworzony jest węzeł drzewa dla tego łańcucha
3. Rekurencyjnie budowane jest lewe poddrzewo z łańcuchów o mniejszych indeksach
4. Rekurencyjnie budowane jest prawe poddrzewo z łańcuchów o większych indeksach

### **Właściwości drzewa:**

1. Głębokość drzewa wynosi  $O(\log n)$  gdzie  $n$  to liczba łańcuchów
2. Każdy węzeł wewnętrzny reprezentuje separator między regionami
3. Liście reprezentują regiony między łańcuchami
4. Lewe poddrzewo zawiera łańcuchy po lewej stronie separatora
5. Prawe poddrzewo zawiera łańcuchy po prawej stronie separatora

Wybór mediany jako korzenia zapewnia, że drzewo jest zbalansowane - każde poddrzewo ma w przybliżeniu połowę łańcuchów, co gwarantuje logarytmiczną wysokość.

## ■ 4.7 Prekomputacja regionów

Aby przyspieszyć końcowy etap wyszukiwania, algorytm prekomputuje wszystkie regiony:

### **Dla każdej pary sąsiednich łańcuchów:**

1. Pobierane są wierzchołki obu łańcuchów
2. Inicjalizowane są indeksy przechodzące po obu listach wierzchołków
3. Śledzone są ostatnie wspólne wierzchołki w obu łańcuchach

### **Identyfikacja bąbli:**

1. Algorytm przechodzi równolegle po obu łańcuchach
2. Gdy znajduje wspólny wierzchołek (punkt, gdzie łańcuchy się spotykają):
  - Jeśli od ostatniego wspólnego punktu były inne wierzchołki
  - Tworzy bąbel - zamknięty region między łańcuchami
  - Zapisuje ścieżkę po lewym łańcuchu między punktami wspólnymi
  - Zapisuje ścieżkę po prawym łańcuchu między punktami wspólnymi
3. Gdy wierzchołki nie są wspólne, algorytm przechodzi do niższego z nich

### **Dla każdego bąbla zapisywane są:**

1. Zakres współrzędnych  $y$
2. Lista wszystkich krawędzi ograniczających bąbel
3. Krawędzie z obu ścieżek

### **Organizacja bąbli:**

1. Bąble dla każdej pary łańcuchów są sortowane według maksymalnej współrzędnej  $y$
2. To sortowanie umożliwia szybkie wyszukiwanie binarne podczas zapytań
3. Bąble są indeksowane parą identyfikatorów łańcuchów

## 5 Zapytanie - lokalizacja punktu

### 5.1 Określenie pozycji względem łańcucha

Dla punktu zapytania i łańcucha w węźle drzewa, algorytm wykonuje:

#### 5.1.1 Wyszukiwanie binarne segmentu

1. Inicjalizowane są wskaźniki na początek i koniec listy segmentów łańcucha
2. W pętli wykonywane jest klasyczne wyszukiwanie binarne:
  - Obliczany jest indeks środkowego segmentu
  - Pobierany jest segment jako para punktów
  - Wyznaczane są współrzędne  $y$  końców segmentu ( $y_{\min}$  i  $y_{\max}$ )
  - Jeśli  $y_{\min} \leq p_y \leq y_{\max}$ , znaleziono właściwy segment
  - Jeśli  $p_y < y_{\min}$ , szukanie kontynuowane w dolnej połowie
  - Jeśli  $p_y > y_{\max}$ , szukanie kontynuowane w górnej połowie
3. Jeśli nie znaleziono segmentu, punkt jest poza zakresem łańcucha

#### 5.1.2 Określenie orientacji punktu

Po znalezieniu odpowiedniego segmentu:

1. Obliczany jest iloczyn wektorowy dla punktów segmentu i punktu zapytania
2. Jeśli wartość bezwzględna iloczynu jest mniejsza niż epsilon:
  - Punkt może leżeć na segmencie
  - Sprawdzane jest, czy współrzędna  $x$  punktu mieści się między końcami segmentu
  - Jeśli tak, zwracane jest 0 (punkt na łańcuchu)
  - W przeciwnym razie punkt jest po jednej ze stron
3. Jeśli iloczyn jest ujemny, punkt jest po lewej stronie łańcucha
4. Jeśli iloczyn jest dodatni, punkt jest po prawej stronie łańcucha

Iloczyn wektorowy mierzy „skierowaną powierzchnię” trójkąta utworzonego przez segment i punkt. Jego znak wskazuje, po której stronie prostej wyznaczonej przez segment znajduje się punkt.

### 5.2 Przeszukiwanie drzewa

Algorytm rekursywnie schodzi w dół drzewa wyszukiwania:

#### 5.2.1 Przypadek bazowy - osiągnięcie liścia

1. Zwracana jest para łańcuchów: najbliższy z lewej i najbliższy z prawej
2. Te dwa łańcuchy definiują region, w którym znajduje się punkt
3. Para ta była śledzona podczas schodzenia po drzewie

#### 5.2.2 Określenie pozycji względem separatora

1. Dla bieżącego węzła wywoływana jest funkcja określania pozycji
2. Zwracana jest wartość  $-1$ ,  $0$  lub  $1$

#### 5.2.3 Przypadek: punkt na łańcuchu

1. Jeśli punkt leży dokładnie na łańcuchu separatora

2. Zwracana jest referencja do tego łańcucha
3. To oznacza, że punkt leży na granicy między regionami

#### 5.2.4 Przypadek: punkt po lewej

1. Punkt znajduje się po lewej stronie separatora
2. Algorytm kontynuuje w prawym poddrzewie (łańcuchy o wyższych indeksach)
3. Bieżący łańcuch staje się nowym najbliższym z lewej strony
4. Najbliższy z prawej jest przekazywany bez zmian

#### 5.2.5 Przypadek: punkt po prawej

1. Punkt znajduje się po prawej stronie separatora
2. Algorytm kontynuuje w lewym poddrzewie (łańcuchy o niższych indeksach)
3. Bieżący łańcuch staje się nowym najbliższym z prawej strony
4. Najbliższy z lewej jest przekazywany bez zmian

W każdym momencie rekursji para (`najbliższy_lewy`, `najbliższy_prawy`) zawiera łańcuchy, między którymi na pewno znajduje się punkt.

### 5.3 Wyznaczanie krawędzi regionu

Po zlokalizowaniu pary łańcuchów ograniczających region:

#### 5.3.1 Pobieranie prekomputowanych danych

1. Z mapy regionów pobierana jest lista bąbli dla pary łańcuchów
2. Klucz stanowi para identyfikatorów (`id_lewego`, `id_prawego`)
3. Jeśli klucz nie istnieje, próbowana jest odwrócona para
4. Każdy bąbel zawiera zakres  $y$  i listę krawędzi

#### 5.3.2 Wyszukiwanie binarne bąbla

1. Inicjalizowane są granice wyszukiwania (`low`, `high`)
2. Pobierana jest współrzędna  $y$  punktu zapytania
3. Wykonywane jest wyszukiwanie binarne po maksymalnych wartościach  $y$  bąbli:
  - Obliczany jest indeks środkowy
  - Jeśli maksymalne  $y$  bąbla jest mniejsze niż  $p_y$ , szukanie w górnej połowie
  - W przeciwnym razie szukanie w dolnej połowie
4. Algorytm znajduje pierwszy bąbel, którego maksymalne  $y$  jest  $\geq p_y$

#### 5.3.3 Weryfikacja przynależności

1. Dla znalezionej bąbla pobierany jest zakres ( $y_{\min}$ ,  $y_{\max}$ )
2. Sprawdzane jest czy  $y_{\min} \leq p_y \leq y_{\max}$
3. Jeśli tak, zwracana jest lista krawędzi tego bąbla
4. Jeśli nie, punkt znajduje się poza wszystkimi bąblami (zwracana jest pusta lista)

Dzięki prekomputacji i sortowaniu bąbli, wyszukiwanie wymaga tylko  $O(\log K)$  operacji, gdzie  $K$  to liczba bąbli między daną parą łańcuchów.

### 5.4 Główna funkcja lokalizacji

Kompletny proces lokalizacji punktu:

#### 5.4.1 Faza preprocessing (jednorazowa)

1. Budowa grafu skierowanego DAG z wierzchołków i krawędzi
2. Obliczenie wag planarnych metodą dwuprzebiegową
3. Wygenerowanie wszystkich łańcuchów monotonicznych
4. Prekomputacja regionów (bąbli) między łańcuchami
5. Konstrukcja zbalansowanego drzewa wyszukiwania

#### 5.4.2 Faza zapytania (dla każdego punktu)

1. Wywołanie rekursyjnego przeszukiwania drzewa z korzenia
2. Przekazanie pierwszego i ostatniego łańcucha jako początkowych granic
3. Otrzymanie wyniku - łańcucha lub pary łańcuchów

#### 5.4.3 Interpretacja wyniku

**Jeśli zwrócono pojedynczy łańcuch:**

1. Punkt leży dokładnie na łańcuchu separatorze
2. Przeszukiwane są segmenty łańcucha
3. Znajduje się segment zawierający punkt na odpowiedniej wysokości  $y$
4. Zwracany jest ten pojedynczy segment jako wynik

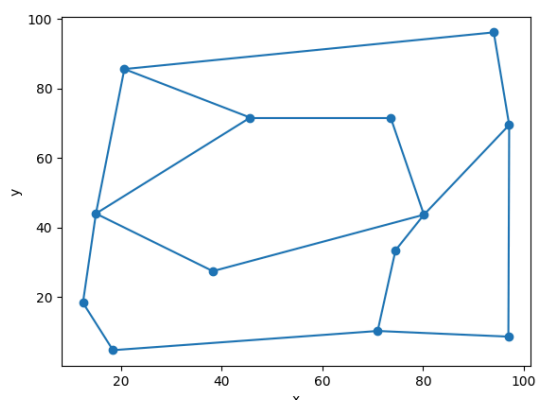
**Jeśli zwrócono parę łańcuchów:**

1. Punkt znajduje się w regionie między dwoma łańcuchami
2. Wywoływana jest funkcja wyszukiwania bąbla
3. Zwracana jest lista wszystkich krawędzi ograniczających region

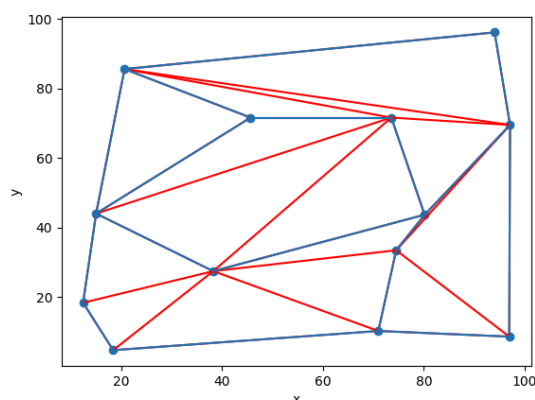
Cała operacja zapytania wykonuje  $O(\log n)$  porównań z łańcuchami i  $O(\log n)$  wyszukiwań binarnych na segmentach, co daje łączną złożoność  $O(\log^2 n)$ .

## 6 Obsługa dowolnych wielokątów

Metoda separatorów wymaga regionów monotonicznych. Dowolne wielokąty są automatycznie triangulowane.



Rys. 6.1 Obszar przed triangulacją



Rys. 6.2 Obszar po triangulacji

Na Rysunkach 6.1 i 6.2 przedstawiono efekt działania algorytmu triangulacji na przykładowym zbiorze testowym.

## ■ 6.1 Triangulacja metodą Ucinania Uszu

### 6.1.1 Definicja ucha

Ucho to trójkąt utworzony przez trzy kolejne wierzchołki wielokąta, taki że:

1. Trójkąt jest wypukły (zwrot przeciwny do ruchu wskazówek zegara)
2. Wewnątrz trójkąta nie znajduje się żaden inny wierzchołek wielokąta
3. Usunięcie środkowego wierzchołka nie zmienia kształtu pozostałej części wielokąta

### 6.1.2 Proces triangulacji

#### Inicjalizacja:

1. Kopiowana jest lista indeksów wierzchołków wielokąta
2. Tworzona jest pusta lista trójkątów wynikowych
3. Ustawiany jest licznik iteracji jako zabezpieczenie przed nieskończoną pętlą

#### Pętla główna (dopóki są więcej niż 2 wierzchołki):

1. Dla każdego wierzchołka sprawdzane jest, czy tworzy ucho:
  - Pobierany jest poprzedni, bieżący i następny wierzchołek
  - Obliczany jest iloczyn wektorowy dla określenia wypukłości
  - Jeśli trójkąt jest wypukły (iloczyn  $>$  epsilon):
    - Sprawdzane jest, czy żaden inny wierzchołek nie leży wewnątrz trójkąta
    - Test wewnętrzności wykorzystuje wielokrotne iloczyny wektorowe
    - Jeśli trójkąt jest pusty, zostaje zaakceptowany jako ucho
2. Gdy ucho zostanie znalezione:
  - Trójkąt dodawany jest do listy wynikowej
  - Środkowy wierzchołek usuwany jest z listy wierzchołków wielokąta
  - Proces rozpoczyna się od nowa z nową konfiguracją
3. Jeśli nie znaleziono ucha w całej iteracji:
  - Usuwany jest pierwszy wierzchołek

### 6.1.3 Test punktu w trójkącie

Sprawdzenie czy punkt  $P$  leży wewnątrz trójkąta  $ABC$ :

1. Obliczane są trzy iloczyny wektorowe:
  - $cp_1$  dla  $(A, B, P)$
  - $cp_2$  dla  $(B, C, P)$
  - $cp_3$  dla  $(C, A, P)$
2. Punkt leży wewnątrz jeśli wszystkie iloczyny mają ten sam znak
3. Równoważnie: punkt nie leży wewnątrz jeśli występują zarówno wartości dodatnie jak i ujemne

## ■ 6.2 Obsługa wielokątów wklęsłych

Algorytm radzi sobie z dowolnymi wielokątami, w tym wklęsłymi:

#### Ograniczenia:

1. Wielokąt musi być planarny
2. Wierzchołki muszą być podane w kolejności (CW lub CCW)

## 7 Narzędzie interaktywne

Zaimplementowano funkcję do interaktywnego rysowania podziału planarnego.

### 7.1 Interfejs użytkownika

**Sterowanie myszą:**

1. **Lewy przycisk myszy (LPM)** - dodaje punkt do bieżącego regionu
2. **Prawy przycisk myszy (PPM)** - zamyka bieżący region i rozpoczyna nowy
3. **Środkowy przycisk/scroll** - kończy rysowanie i zwraca dane

## 8 Generatory grafów testowych

Zaimplementowano trzy typy automatycznych generatorów struktur testowych.

### 8.1 Siatka trójkątów

Generator tworzy regularną siatkę trójkątów:

**Generowanie wierzchołków:**

1. Dla siatki  $w \times h$  tworzona jest  $(w + 1) \times (h + 1)$  wierzchołków
2. Wierzchołki umieszczane są w węzłach siatki jednostkowej
3. Wierzchołek o indeksach  $(x, y)$  ma współrzędne  $(x.0, y.0)$
4. Funkcja indeksująca:  $\text{indeks}(x, y) = y \cdot (w + 1) + x$

**Generowanie krawędzi:**

1. Dla każdego kwadratu w siatce:
  - Definiowane są cztery narożniki:  $u$ ,  $\text{right}$ ,  $\text{top}$ ,  $\text{top\_right}$
  - Dodawane są krawędzie prostokąta: poziome i pionowe
  - Dodawana jest przekątna łącząca  $u$  z  $\text{top\_right}$
2. Każdy kwadrat jest dzielony na dwa trójkąty
3. Wynikowa siatka zawiera  $2wh$  trójkątów

**Deduplicacja:**

1. Wszystkie krawędzie dodawane są do zbioru (set)
2. Każda krawędź normalizowana jest: mniejszy indeks pierwszy
3. Automatycznie eliminowane są duplikaty krawędzi wspólnych dla sąsiednich kwadratów
4. Zwracana jest lista unikalnych krawędzi

**Zastosowanie:** Siatka trójkątów jest idealnym testem dla algorytmu - zawiera wiele małych regionów monotonicznych i pozwala sprawdzić skalowalność.

### 8.2 Siatka czworokątów

Generator tworzy siatkę prostokątów z opcjonalnym pochyleniem:

**Generowanie wierzchołków z pochyleniem:**

1. Dla każdego rzędu  $y$  obliczane jest przesunięcie:  $\text{shift} = y \cdot \text{skew}$

2. Wierzchołek  $(x, y)$  ma współrzędne  $(x + \text{shift}, y)$
3. Parametr skew=0 daje prostą siatkę ortogonalną
4. Parametr skew>0 daje pochyloną siatkę równoległoboków
5. Parametr skew może być ujemny dla pochylenia w drugą stronę

#### **Generowanie krawędzi poziomych:**

1. Dla każdego rzędu  $y$  od 0 do  $h$
2. Dla każdego  $x$  od 0 do  $w - 1$
3. Dodawana jest krawędź między  $(x, y)$  i  $(x + 1, y)$
4. To tworzy  $(w + 1) \cdot h$  krawędzi poziomych

#### **Generowanie krawędzi pionowych:**

1. Dla każdej kolumny  $x$  od 0 do  $w$
2. Dla każdego  $y$  od 0 do  $h - 1$
3. Dodawana jest krawędź między  $(x, y)$  i  $(x, y + 1)$
4. To tworzy  $w \cdot (h + 1)$  krawędzi pionowych

#### **Właściwości:**

1. Bez przekątnych - czyste czworokąty
2. Regiony nie są monotoniczne - wymagają triangulacji
3. Pochylenie testuje odporność algorytmu na nieregularne kształty

### **8.3 Triangulacja Delaunaya (manualna)**

Generator tworzy triangulację Delaunaya dla losowych punktów:

#### **Generowanie punktów:**

1. Dodawane są 4 punkty narożne obszaru:  $(0, 0)$ ,  $(w, 0)$ ,  $(0, h)$ ,  $(w, h)$
2. Generowane jest  $n$  losowych punktów wewnątrz obszaru
3. Każdy losowy punkt:  $x \in [0.1, w - 0.1]$ ,  $y \in [0.1, h - 0.1]$
4. Margines 0.1 zapobiega punktom na krawędziach

#### **Algorytm triangulacji:**

1. Dla każdej trójki punktów  $(i, j, k)$ :
  - Obliczane jest położenie centrum okręgu opisanego
  - Wykorzystywane są wzory oparte na wyznacznikach
  - Obliczany jest promień okręgu opisanego
2. Sprawdzane jest kryterium Delaunaya:
  - Dla każdego innego punktu  $m$
  - Obliczana jest odległość od centrum okręgu
  - Jeśli jakikolwiek punkt leży wewnątrz okręgu (odległość < promień)
  - Trójkąt jest odrzucany
3. Trójkąty spełniające kryterium są akceptowane:
  - Dodawane są trzy krawędzie trójkąta
  - Krawędzie są normalizowane i deduplikowane

**Kryterium Delaunaya:** Okrąg opisany na trójkącie nie zawiera żadnych innych punktów - zapewnia to „optymalność” triangulacji (maksymalizacja minimalnego kąta).

#### **Zabezpieczenie (fallback):**

1. Jeśli algorytm nie znajdzie żadnych trójkątów (bardzo rzadkie)

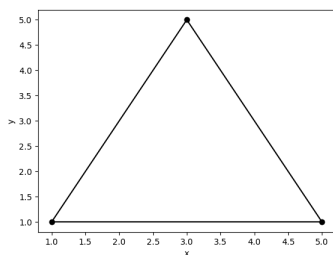
2. Tworzona jest prosta ścieżka łącząca wszystkie punkty sekwencyjnie
3. Ostatni punkt łączony jest z pierwszym

**Zastosowanie:** Triangulacja Delaunaya testuje algorytm na nieregularnych, realistycznych podziałach płaszczyzny.

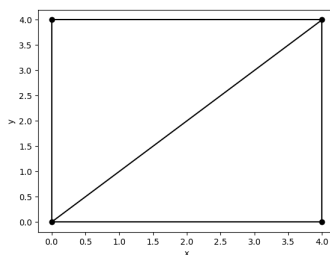
## 9 Testy

### 9.1 Zestawy danych testowych

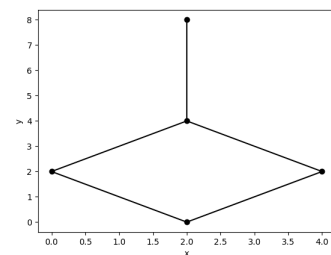
Przygotowano różnorodne zbiory testowe:



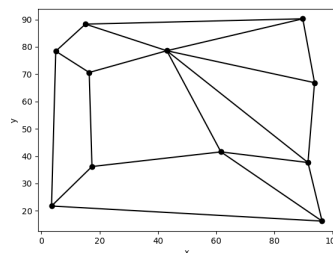
Rys. 9.1 Prosty Trójkąt



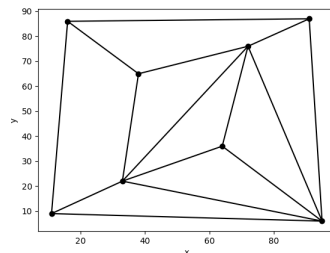
Rys. 9.2 Kwadrat z Przekątną



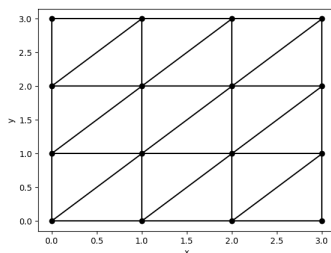
Rys. 9.3 Dwa Diamenty



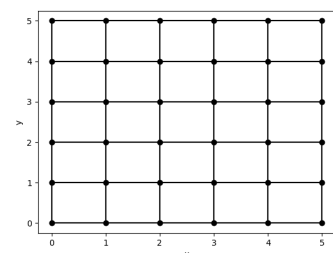
Rys. 9.4 Wielokąt nieregularny (Zestaw B)



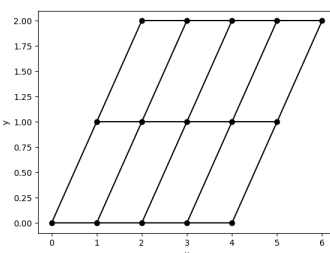
Rys. 9.5 Złożony Wielokąt



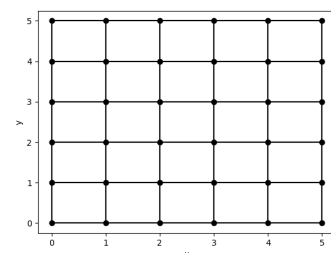
Rys. 9.6 Siatka trójkątów 3x3



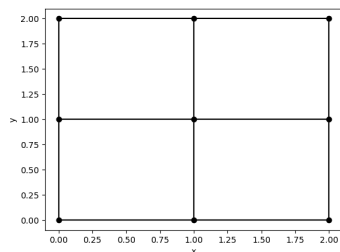
Rys. 9.7 Siatka trójkątów 5x5



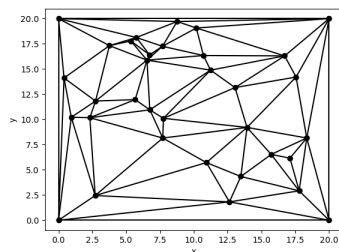
Rys. 9.8 Siatka czworokątów 4x2



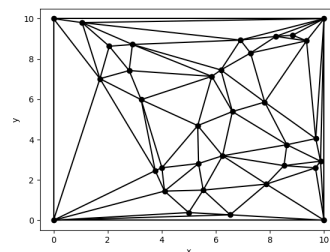
Rys. 9.9 Siatka czworokątów 5x5



Rys. 9.10 Siatka czworokątów  
2×2



Rys. 9.11 Delaunay 20×20



Rys. 9.12 Delaunay 10×10

### 9.1.1 Proste przypadki geometryczne

#### Prosty Trójkąt (Rys. 9.1):

- 3 wierzchołki tworzące trójkąt
- 3 krawędzie
- Test najbardziej podstawowego przypadku
- 4 punkty testowe (wewnątrz i na zewnątrz)

#### Kwadrat z Przekątną (Rys. 9.2):

- 4 wierzchołki kwadratu
- 5 krawędzi (4 boki + 1 przekątna)
- Podział na 2 regiony trójkątne
- 3 punkty testowe w różnych regionach

#### Dwa Diamenty (Rys. 9.3):

- 5 wierzchołków tworzących dwa romby połączone wierzchołkiem
- 5 krawędzi łączących diamenty pionowo
- Test spójności między regionami
- 3 punkty testowe

### 9.1.2 Złożone wielokąty ręczne

#### Zestawy A, B, C, D, E (Rys. 9.4 - Zestaw B):

- Od 6 do 12 wierzchołków
- Nieregularne wielokąty o różnych kształtach
- Kombinacje regionów wypukłych i wklęsłych
- Różna liczba punktów testowych (1-4)
- Współrzędne zmiennoprzecinkowe z dużą precyzją

#### Złożony Wielokąt (Rys. 9.5):

- 8 wierzchołków
- 15 krawędzi po triangulacji
- Wielokąt wklęsły wymagający pełnej triangulacji
- 5 punktów testowych w strategicznych lokalizacjach

### 9.1.3 Siatki regularne

#### Siatka trójkątów 3×3 (Rys. 9.6):

- 16 wierzchołków ( $4 \times 4$  węzłów)
- 24 krawędzie
- 18 trójkątów
- 5 punktów testowych

**Siatka trójkątów  $5 \times 5$  (Rys. 9.7):**

- 36 wierzchołków ( $6 \times 6$  węzłów)
- 85 krawędzi
- 50 trójkątów
- 7 punktów testowych

**Siatka czworokątów  $4 \times 2$ , skew=1 (Rys. 9.8):**

- 15 wierzchołków
- 26 krawędzi
- Pochylenie o 1 jednostkę na rząd
- 4 punkty testowe

**Siatka czworokątów  $5 \times 5$ , skew=0 (Rys. 9.9):**

- 36 wierzchołków
- 60 krawędzi (bez przekątnych)
- Wymaga triangulacji
- 6 punktów testowych

**Siatka czworokątów  $2 \times 2$ , skew=0 (Rys. 9.10):**

- 9 wierzchołków
- 12 krawędzi
- Najmniejszy test czworokątów
- 4 punkty testowe

**9.1.4 Triangulacje Delaunaya****Delaunay  $20 \times 20$ , 30 punktów (Rys. 9.11):**

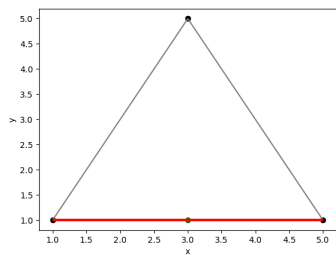
- 34 wierzchołki (4 narożne + 30 losowych)
- Zmienna liczba krawędzi (zależna od losowania)
- Nieregularna triangulacja
- 7 punktów testowych

**Delaunay  $10 \times 10$ , 30 punktów (Rys. 9.12):**

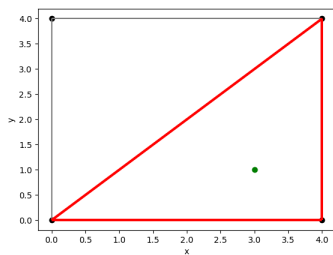
- 34 wierzchołki w mniejszym obszarze
- Gęstsza triangulacja
- Test zbieżności punktów
- 7 punktów testowych

**■ 9.2 Wyniki testów poprawności**

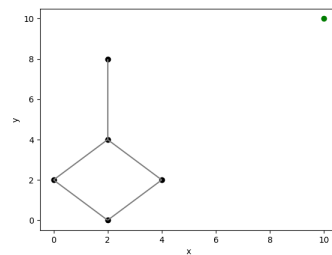
Algorytm został przetestowany na wszystkich zbiorach. **Wszystkie testy zakończyły się sukcesem.**



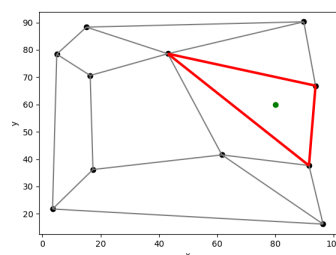
Rys. 9.13 Prosty Trójkąt



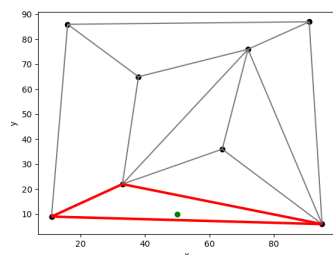
Rys. 9.14 Kwadrat z Przekątną



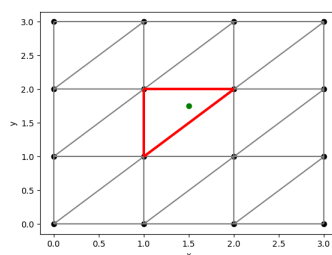
Rys. 9.15 Dwa Diamenty



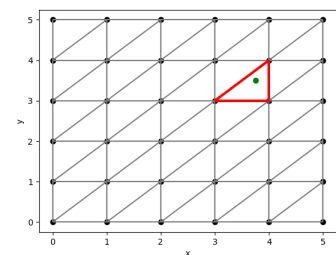
Rys. 9.16 Zestaw B



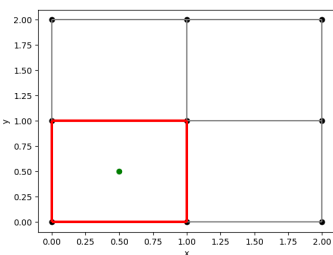
Rys. 9.17 Złożony Wielokąt



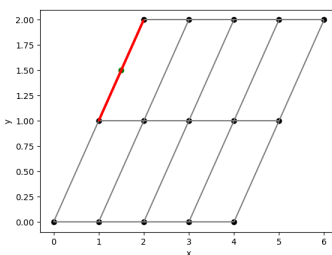
Rys. 9.18 Siatka Trójkątów 3x3



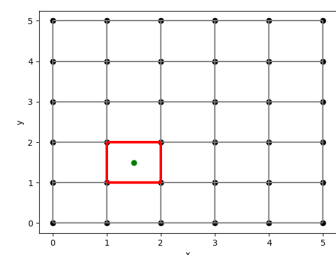
Rys. 9.19 Siatka Trójkątów 5x5



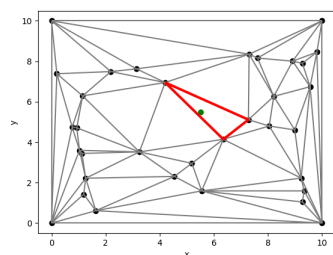
Rys. 9.20 Siatka Czworokątów 2x2



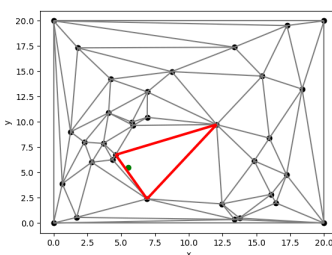
Rys. 9.21 Siatka Czworokątów 4x2



Rys. 9.22 Siatka Czworokątów 5x5



Rys. 9.23 Delaunay 10x10



Rys. 9.24 Delaunay 20x20

## 9.2.1 Proste przypadki

### Prosty Trójkąt (Rys. 9.13):

- Punkt (3, 2): znaleziono 3 krawędzie regionu trójkątnego
- Punkt (0, 0): poza regionem, 0 krawędzi
- Punkt (3, 1): na krawędzi, zwrócono 1 segment
- Punkt (0, 3): poza regionem

**Kwadrat z Przekątną (Rys. 9.14):**

- Punkt (1, 3): lewy trójkąt, 3 krawędzie
- Punkt (3, 1): prawy trójkąt, 3 krawędzie
- Punkt (5, 5): poza kwadratem

**Dwa Diamenty (Rys. 9.15):**

- Punkt (2, 2): dolny diament, 4 krawędzie
- Punkt (2, 6): na łączącym odcinku między diamentami
- Punkt (10, 10): poza strukturą

**9.2.2 Złożone wielokąty**

Dla wszystkich zestawów A-E (Rys. 9.16 - Zestaw B):

- Preprocessing zakończony bez błędów
- Wszystkie punkty testowe poprawnie zlokalizowane
- Zwrócone krawędzie tworzą spójne regiony
- Punkty na granicach poprawnie identyfikowane

**Złożony Wielokąt (8 wierzchołków) (Rys. 9.17):**

- Triangulacja utworzyła spójną siatkę
- 5 punktów testowych rozłożonych strategicznie
- Wszystkie lokalizacje poprawne

**9.2.3 Siatki regularne****Siatki trójkątów (Rys. 9.18, 9.19):**

- Preprocessing dla siatki  $3 \times 3$  (Rys. 9.18):  $< 0.01s$
- Preprocessing dla siatki  $5 \times 5$  (Rys. 9.19):  $< 0.02s$
- Wszystkie punkty testowe zlokalizowane
- Punkty na granicach trójkątów poprawnie obsłużone

**Siatki czworokątów (Rys. 9.20 - 9.22):**

- Automatyczna triangulacja przed procesowaniem
- Siatka  $4 \times 2$  ze skew (Rys. 9.21): geometria pochylona poprawnie obsłużona
- Siatka  $5 \times 5$  (Rys. 9.22): wszystkie 25 czworokątów zdekomponowane
- Punkty poza siatką zwracają pustą listę

**9.2.4 Triangulacje Delaunaya****Delaunay  $20 \times 20$  (Rys. 9.24):**

- 34 punkty wygenerowane poprawnie
- Triangulacja spełnia kryterium Delaunaya
- 7 punktów testowych:
  - Punkty wewnątrz trójkątów: 3 krawędzie
  - Punkty na krawędziach: 1 segment
  - Punkt (3, 0) na dolnej krawędzi: poprawnie

**Delaunay  $10 \times 10$  (Rys. 9.23):**

- Gęstsza triangulacja w mniejszym obszarze
- Wszystkie punkty testowe poprawnie zlokalizowane
- Brak problemów z precyzją numeryczną

### 9.2.5 Podsumowanie poprawności

Wszystkie 55 punktów testowych zostało poprawnie zlokalizowanych. Algorytm prawidłowo obsługuje:

- Punkty wewnątrz regionów
- Punkty na krawędziach i wierzchołkach
- Punkty poza podziałem
- Wielokąty wypukłe i wklęsłe
- Regularne i nieregularne podziały

## ■ 9.3 Analiza efektywności

Przeprowadzono pomiary czasu wykonania dla różnych rozmiarów grafów.

### 9.3.1 Metodologia testowania

#### Generowanie danych:

- Testowane rozmiary siatek:  
5x5, 10x10, ... , 50x50,  
75x75, 100x100, ... , 200x200,  
250x250, 300x300,... ,600x600
- Dla każdego rozmiaru generowane dwie konfiguracje:
  - Siatka trójkątów
  - Siatka czworokątów

## 10 Analiza wydajności

Przeprowadzono testy wydajnościowe algorytmu w celu weryfikacji teoretycznej złożoności obliczeniowej. Badanie podzielono na dwie części:

1. Pomiar czasu budowy struktury .
2. Pomiar czasu odpowiedzi na zapytanie o lokalizację punktu .

### 10.1 Tabele czasów

Typ Grafu	Liczba V	Liczba E	Czas Budowy [s]	Czas Zapytania [s]
TriGrid	36	85	0.00155616	0.00001878
TriGrid	121	320	0.01117849	0.00003171
TriGrid	256	705	0.01570678	0.00002425
TriGrid	441	1240	0.01908970	0.00003710
TriGrid	676	1925	0.03036237	0.00002061
TriGrid	961	2760	0.02800941	0.00001533
TriGrid	1681	4880	0.04599452	0.00001868
TriGrid	2601	7600	0.08189201	0.00002888
TriGrid	5776	17025	0.15848351	0.00001952
TriGrid	10201	30200	1.40723681	0.00004333
TriGrid	15876	47125	1.24264145	0.00004642
TriGrid	22801	67800	2.16456079	0.00004630
TriGrid	30976	92225	3.81330061	0.00004255
TriGrid	40401	120400	5.31679988	0.00005027
TriGrid	63001	188000	4.89468575	0.00003100
TriGrid	90601	270600	6.42702532	0.00005277
TriGrid	123201	368200	10.76700997	0.00005888
TriGrid	160801	480800	13.47085428	0.00004321
TriGrid	203401	608400	18.69723105	0.00007097
TriGrid	251001	751000	23.11309409	0.00005366
TriGrid	361201	1081200	36.60034084	0.00006928

wyniki czasów dla TriGrid.

Typ Grafu	Liczba V	Liczba E	Czas Budowy [s]	Czas Zapytania [s]
QuadGrid	36	60	0.00764632	0.00000672
QuadGrid	121	220	0.00173616	0.00000946
QuadGrid	256	480	0.00629115	0.00002983
QuadGrid	441	840	0.00862384	0.00001852
QuadGrid	676	1300	0.63888717	0.00001592
QuadGrid	961	1860	0.02332520	0.00001843
QuadGrid	1681	3280	0.03195739	0.00004286
QuadGrid	2601	5100	0.06849694	0.00001905
QuadGrid	5776	11400	0.17012072	0.00002618
QuadGrid	10201	20200	0.34410954	0.00005446
QuadGrid	15876	31500	0.65701222	0.00003090
QuadGrid	22801	45300	0.65329552	0.00003996
QuadGrid	30976	61600	5.59486055	0.00008170
QuadGrid	40401	80400	2.09832454	0.00006799
QuadGrid	63001	125500	3.63943124	0.00003441
QuadGrid	90601	180600	5.25921130	0.00008075
QuadGrid	123201	245700	7.48945045	0.00004716
QuadGrid	160801	320800	7.59110141	0.00006276
QuadGrid	203401	405900	14.73398328	0.00007258
QuadGrid	251001	501000	13.17585278	0.00004716
QuadGrid	361201	721200	25.15765715	0.00010015

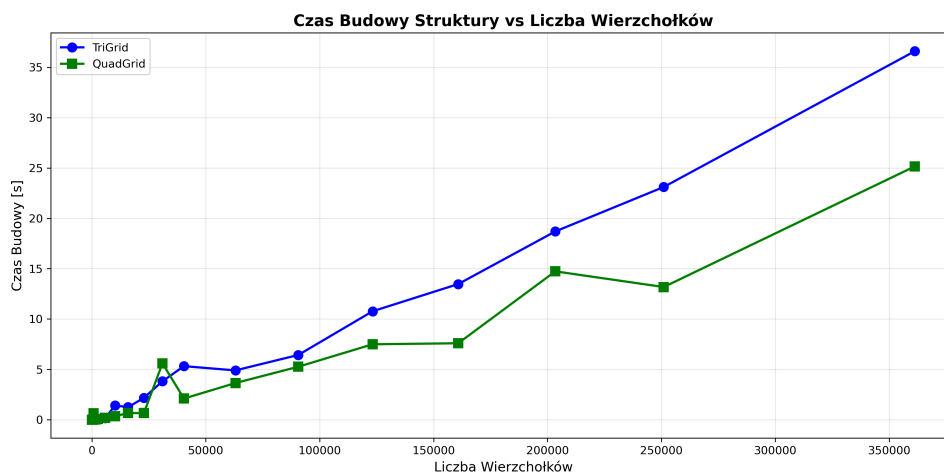
wyniki czasów dla QuadGrid.

Jako dane testowe wykorzystano generowane proceduralnie siatki trójkątne (TriGrid) oraz czworokątne (QuadGrid) o zmiennej liczbie wierzchołków  $V$  w zakresie od 36 do 361 201.

## 10.2 Czas budowy struktury

Zgodnie z przewidywaniami teoretycznymi zawartymi we wstępie, czas budowy struktury rośnie nieliniowo. Implementacja algorytmu wyznaczania wag planarnych oraz prekomputacji regionów posiada złożoność  $O(n^2)$ , co jest widoczne przy dużych zbiorach danych.

Dla największego badanego zbioru (ponad 360 tysięcy wierzchołków i milion krawędzi), czas budowy wynosi ok. 36 sekund, co jest wartością akceptowalną dla procesu wykonywanego jednorazowo (offline).



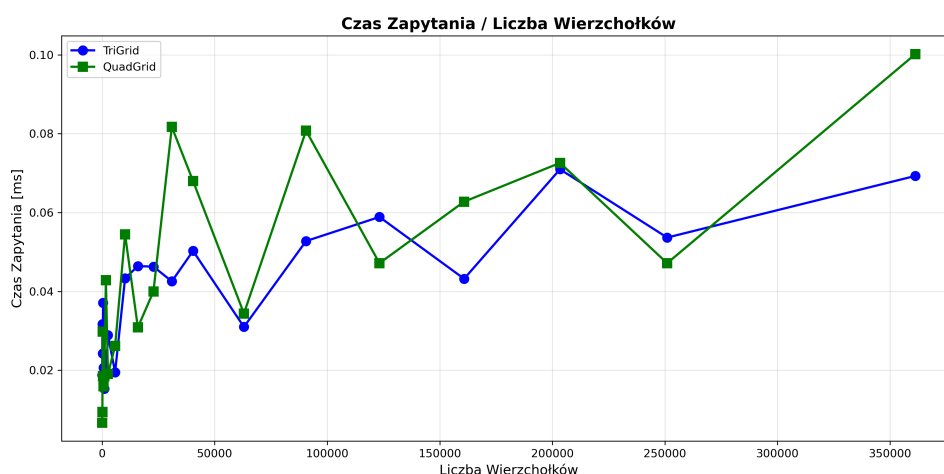
Wykres 10.1 Zestawienie zależności czasu przygotowywania danych od liczby wierzchołków w zbiorze dla dwóch wzorów

### 10.3 Czas zapytania

Kluczowym aspektem metody separatorów jest szybkość lokalizacji punktu po zbudowaniu struktury. Poniższa tabela prezentuje średni czas zapytania (uśredniony z 100 losowych punktów).

**Wnioski z analizy czasu zapytania:** Czas zapytania jest ekstremalnie krótki i stabilny. Dla grafu o wielkości 360 tysięcy wierzchołków, zlokalizowanie punktu zajmuje średnio mniej niż 0.0001 sekundy.

Wzrost czasu zapytania w funkcji liczby wierzchołków jest bardzo powolny, co potwierdza logarytmiczną złożoność teoretyczną algorytmu  $O(\log^2 n)$ . Nawet przy drastycznym zwiększeniu liczby wierzchołków (z 100 do 360 000), czas zapytania rośnie zaledwie ok. 2 krotnie. Potwierdza to wysoką skuteczność metody separatorów w zastosowaniach czasu rzeczywistego.



Wykres 10.2 Zestawienie zależności czasu zapytania od liczby wierzchołków w zbiorze dla dwóch wzorów

## 11 Wnioski i podsumowanie

Zrealizowany projekt pozwolił na szczegółowe zapoznanie się z problemem lokalizacji punktu w podziałach planarnych przy użyciu metody separatorów. Na podstawie przeprowadzonych testów i implementacji sformułowano następujące wnioski:

- **Efektywność zapytań:** Algorytm potwierdził swoją teoretyczną złożoność  $O(\log^2 n)$ . Wyszukiwanie punktu w gęstych strukturach odbywało się w czasie subiektywnie natychmiastowym.
- **Istotność triangulacji:** Implementacja wykazała, że metoda separatorów jest ściśle uzależniona od monotoniczności regionów. Zastosowanie algorytmu „Ucinania Uszu” jako etapu preprocessingu pozwoliło na obsługę dowolnych wielokątów, co znacząco rozszerzyło uniwersalność narzędzia kosztem jednorazowego zwiększenia złożoności przygotowania danych.
- **Stabilność numeryczna:** Zastosowanie precyzji `float64` oraz parametru `epsilon` ( $10^{-24}$ ) okazało się kluczowe przy obliczaniu iloczynów wektorowych. Pozwoliło to na poprawną klasyfikację punktów leżących bezpośrednio na krawędziach separatorów.

Podsumowując, metoda separatorów stanowi doskonały kompromis między wydajnością zapytania a złożonością pamięciową  $O(n)$ .

## 12 Bibliografia

- [1] **R. Tamassia, Brown University, (1993)** <https://cs.brown.edu/courses/cs252/misc/resources/lectures/pdf/notes05.pdf>
- [2] **D. T. Lee, F. P. Preparata, University of Illinois (1976)** <https://dl.acm.org/doi/epdf/10.1145/800113.803653>
- [3] **H. Edelsbrunner, L. J. Guibas, J. Stolfi, (1986)** [https://graphics.stanford.edu/courses/cs268-11-spring/notes/opt\\_point\\_loc.pdf](https://graphics.stanford.edu/courses/cs268-11-spring/notes/opt_point_loc.pdf)