Algorytm Kirkpatricka Dokumentacja

Wiktor Warzecha

Łukasz Kwinta

Spis treści

[1 Dane techniczne 2](#_Toc155280904)

[2 Wymagane oprogramowanie 2](#_Toc155280905)

[3 Wizualizacja 3](#_Toc155280906)

[4 Typy prymitywne i struktury danych 3](#_Toc155280907)

[5 Moduł kirkpatrick\_point\_location 4](#_Toc155280908)

[6 Opis działania 6](#_Toc155280909)

[6.1 Wyznaczenie zewnętrznego trójkąta i wstępna triangulacja 6](#_Toc155280910)

[6.2 Otrzymanie zbioru wierzchołków niezależnych 8](#_Toc155280911)

[6.3 Usuwanie zbioru niezależnych wierzchołków z wielokąta 9](#_Toc155280912)

[6.4 Przetwarzanie 10](#_Toc155280913)

[6.5 Przeszukiwanie 10](#_Toc155280914)

[7 Moduł Kirkpatrick\_point\_Location\_With\_Visualization 11](#_Toc155280915)

[8 Przykład użycia 12](#_Toc155280916)

[9 Testy 13](#_Toc155280917)

[10 Bibliografia 16](#_Toc155280918)

# Dane techniczne

Procesor: 64 bitowy procesor

System operacyjny: Ubuntu 20.04 w środowisku WSL 2 na Windows 11 x64

Pamięć ram: 32 GB DDR4

Środowisko i język: Python 3.9 + Jupyter Notebook w środowisku Anaconda

Wykresy tworzono przy pomocy narzędzia przygotowanego przez KN Bit, do obliczeń numerycznych używano biblioteki numpy. Dane przechowywane były w zmiennych typu float – typ danych o rozmiarze 64 bitów, odpowiednik typu double w języku C.

# Wymagane oprogramowanie

Do uruchomienia projektu potrzebne są następujące pakiety środowiska python. Wszystkie dostępne są z repozytorium PIP.

* numpy –
  + używamy do obliczeń numerycznych oraz przekształceń tablic wynikowych
  + w wersji co najmniej 1.25.2
* matplotlib –
  + używamy do wizualizacji działania algorytmu
  + w wersji co najmniej 3.7.2
* notebook –
  + potrzebny do uruchomienia pliku ipynb – Jupyter Notebook
  + w wersji co najmniej 6.5.4
* SciPy –
  + używamy do triangulacji Delaunaya początkowego wielokąta, jest ona w tym momencie wygodna, ponieważ nie musimy osobno dodawać krawędzi otoczki wypukłej do triangulacji i triangulować obszaru pomiędzy zewnętrznym trójkątem i wielokątem, dostarczona triangulacja jest zrealizowana w czasie
  + w wersji co najmniej 1 .11.4
* planegeometry –
  + używamy do przechowywania grafu planarnego, udostępnia wygodny interfejs do listy sąsiedztwa, oraz prymitywne typy geometryczne (Punkt, Odcinek, Trójkąt) których użyliśmy w naszym algorytmie oraz funkcje interfejsu do owych typów
  + w wersji co najmniej 1.0.1
* mapbox\_earcut –
  + używamy do triangulacji dziur powstałych przez usunięcie zbioru niezależnych odcinków, triangulacja Delaunaya nie była wygodna do triangulacji dziur z powodu konieczności radzenia sobie z odcinkami dodanymi poza obszarem wielokąta. Biblioteka ta jest bindingiem do biblioteki mapbox zrealizowanej w języku C++.
  + w wersji co najmniej 1.0.1
* Tkinter –
  + Używany do zrealizowania interaktywnej prezentacji działania algorytmu
  + W wersji co najmniej 8.6.12

Kod naszego oprogramowania składa się z następujących plików - części:

* kirkpatrick-algorithm.ipynb – zawiera przykładowe wycinki kodu, z fragmentami implementacji, obrazujące poszczególne części z ilustracjami wyjaśniającymi fragmenty kodu
* kirkpatrick\_point\_location.point\_location.py – użytkowa biblioteka zawierająca końcową implementację algorytmu zebraną w jedną całość
* kirkpatrick\_point\_location\_visualization.point\_location\_visualization.py – biblioteka generująca obrazki z poszczególnymi krokami działania algorytmu
* kirkpatrick\_point\_location\_visualization.point\_location\_interactive\_visualization.py – biblioteka zawierająca aplikację z GUI w którym można zadać chmurę punktów oraz punkt do zlokalizowania
* kirkpatrick\_point\_location\_visualization.test.ipynb – plik w którym pokazano jak uruchomić graficzną aplikację z wizualizacją.

# Wizualizacja

Do wizualizacji wykorzystaliśmy moduł visualizer, przygotowany przez KN Bit. Zawiera on nakładkę na bibliotekę matplotlib, ułatwiającą wizualizację działania algorytmów. Dodatkowo wprowadziliśmy interaktywne GUI pozwalające po kolei wyświetlać kroki algorytmu. W innych modułach korzystamy z funkcji zawartych w tym module do rysowania wielokątów.

# Typy prymitywne i struktury danych

W całym module korzystaliśmy z typów geometrycznych zdefiniowanych w bibliotece planegeometry. Korzystaliśmy z tej biblioteki przede wszystkim dlatego, że udostępnia ona wygodny interfejs na listę sąsiedztwa dla grafu planarnego. Strukturą danych na której operujemy jest PlanarMap. Jest to klasa która używa słownika – hashmapy – do przechowywania relacji pomiędzy punktami w grafie. Umożliwia to dostęp w czasie stałym do sąsiadów danego punktu. Obiekt ten posiada również implementację struktury danych DCEL, lecz metody z których korzystaliśmy nie korzystają z tej funkcjonalności.

Przy okazji skorzystaliśmy z implementacji prymitywnych obiektów geometrycznych dostarczonych przez tą bibliotekę

* Point – implementacja klasy punktu dostarczająca mechanizm porównywania punktów i ich porządkowania względem kierunku wskazówek zegara którą użyliśmy do uporządkowania sąsiadów punktu ze zbioru niezależnego bez użycia funkcji trygonometrycznych aby stworzyć zbiór trójkątów, punkty wykorzystujemy w bibliotece do opisu wielokątów
* Segment – struktura opisująca odcinek między punktami A i B. Posiada metodą umożliwiają sprawdzenie przecięcia dwóch odcinków, którą wykorzystaliśmy do sprawdzenia, które trójkąty nachodzą na siebie
* Triangle – struktura opisująca trójkąt, posiada metodę pozwalającą sprawdzić czy punkt jest wewnątrz trójkąta, obiekty tego typu są zwracane jako wynik lokalizacji punktu

# Moduł kirkpatrick\_point\_location

Jest to główny moduł naszego projektu. Zawiera on główną bibliotekę służącą do lokalizacji punktów. Biblioteka zawiera definicję klasy Kirkpatrick, która przechowuje obecny stan lokalizacji.

Klasa Kirkpatrick posiada następujące metody[[1]](#footnote-1):

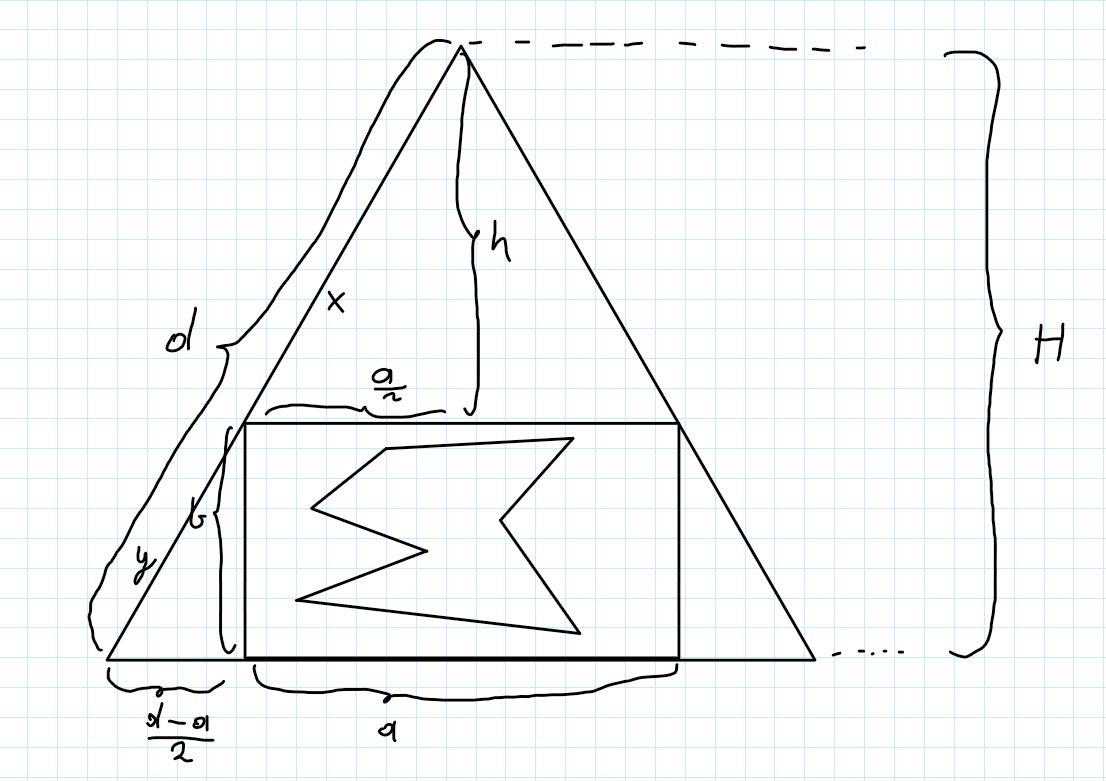
* \_\_init\_\_(self, polygon: List[tuple[float, float]])
  + Argumenty:
    - polygon – lista krotek ze współrzędnymi punktów wierzchołków wielokąta podanych w kierunku przeciwnym do wskazówek zegara
  + Opis działania:
    - Funkcja inicjalizuje pola z których później korzysta klasa na domyślne wartości. Do podanego wielokąta dodaje zewnętrzny trójkąt, oblicza triangulację Delaunaya oraz konwertuje otrzymane krawędzie do obiektu PlanarMap
  + Złożoność: – ze względu na triangulację Delaunaya
* preprocess(self)
  + Argumenty: brak
  + Opis działania:
    - Funkcja tworzy zbiór trójkątów, inicjalizuje drzewo przeszukiwań, a następnie tworzy je. Dopóki liczba wierzchołków w wielokącie (w reprezentacji poprzez PlanarMap) jest większa od 3 funkcja znajduje i usuwa zbiór niezależnych wierzchołków, a następnie przetwarza trójkąty powstałe przez triangulację powstałej dziury, tworząc graf relacji nakładania się na siebie trójkątów.
    - Funkcja wyrzuca wyjątek gdy wielokąt został już raz przetworzony
  + Złożoność: – ze względu na zmniejszającą się ilość wierzchołków w wielokącie i zmniejszający się stopień wierzchołków
* get\_triangles(self) -> List[Triangle]
  + Argumenty: brak
  + Zwracane wartości:
    - Lista obiektów typu Triangle z biblioteki planegeometry
  + Opis działania:
    - Funkcja zwraca listę trójkątów tworzącą najmniejszy podział wielokąta z zewnętrznym trójkątem (bezpośrednio po triangulacji Delaunaya)
* query(self, point: (float, float)) -> Triangle
  + Argumenty:
    - point - podwójna krotka typu float oznaczająca punkt który sprawdzamy
  + Zwracane wartości:
    - Obiekt typu Triangle z biblioteki planegeometry, jeden z listy trójkątów zwracanej przez get\_triangles(self) jest wynikiem przeszukiwania
  + Opis działania:
    - Funkcja przeszukuje drzewo wygenerowane przez funkcje przetwarzającą wielokąt. W każdym kroku funkcja sprawdza do którego z dzieci obecnego trójkąta należy sprawdzany punkt
    - Funkcja wyrzuca wyjątek w razie próby przeszukiwania w wielokącie który nie został jeszcze przetworzony
  + Złożoność: – ze względu na przeszukiwanie drzewa o wysokości
* query\_with\_show(self, (float, float))
  + Argumenty:
    - point - podwójna krotka typu float oznaczająca punkt który sprawdzamy
  + Opis działania:
    - Funkcja przeszukuje drzewo wygenerowane przez funkcje przetwarzającą wielokąt. W każdym kroku funkcja sprawdza do którego z dzieci obecnego trójkąta należy sprawdzany punkt. W wyniku przeszukiwania funkcja rysuje zbiór wszystkich trójkątów, zaznacza sprawdzany punkt oraz podświetla zlokalizowany trójkąt.
    - Funkcja wyrzuca wyjątek w razie próby przeszukiwania w wielokącie który nie został jeszcze przetworzony
  + Złożoność: – ze względu na przeszukiwanie drzewa o wysokości

# Opis działania

Wraz z biblioteką przygotowaliśmy Jupyter Notebook (kirkpatrick-algorithm.ipynb) jako demo prezentujące krok po kroku poszczególne elementy algorytmu, a na końcu pokazuje przykłady wykorzystania biblioteki wraz z testami które przeprowadziliśmy.

## Wyznaczenie zewnętrznego trójkąta i wstępna triangulacja

Aby wyznaczyć zewnętrzny trójkąt stosujemy metodę wyznaczenia prostokąta obejmującego zbiór punktów wielokąta, powiększeniu go, a następnie obliczenie wymiarów prostokąta stycznego do prostokąta.





Rysunek 1 Szkic z oznaczeniami, dodawanego zewnętrznego trójkąta

Niech:

* a, b – wymiary uzyskanego prostokąta
* d – długość boku dodanego trójkąta równobocznego
* H – wysokość trójkąta równobocznego

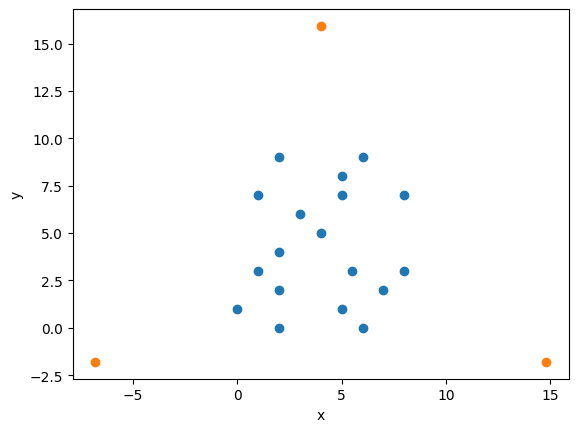
Z twierdzenia pitagorasa dla małego trójkąta prostokątnego otrzymujemy

Z wymiarów trójkąta 30,60,90 wiemy, że , a więc , więc podstawiając:

Rozwiązując równanie kwadratowe otrzymujemy:

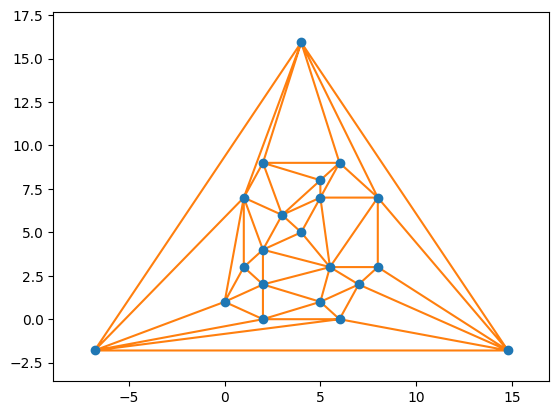
Wybieramy większy pierwiastek, czyli ten ze znakiem +, a więc długość boku trójkąta równobocznego który nas interesuje jest równa:

Dzięki temu otrzymujemy wierzchołki trójkąta o następujących współrzędnych:



Rysunek 2 Rysunek pokazuje dodane punkty do przykładowego wielokąta

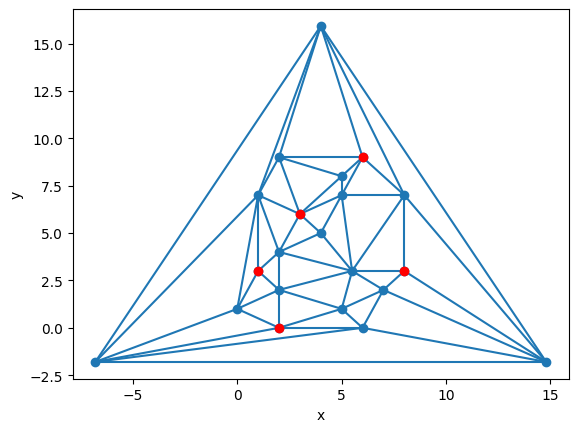
Do wstępnej triangulacji używamy algorytmu triangulacji Delaunaya. Poniżej wykres dla powyższego przykładu.



Rysunek 3 Przykładowa triangulacja Delaunaya

## Otrzymanie zbioru wierzchołków niezależnych

Do otrzymywania zbioru niezależnych wierzchołków używamy algorytmu zachłannego. Wybieramy jeden z wierzchołków w liście sąsiedztwa, który nie jest wierzchołkiem zewnętrznego trójkąta. Oznaczam sąsiadów tego wierzchołka jako odwiedzonych, dodajemy go do zbioru wierzchołków niezależnych, a następnie przechodzimy do dowolnego wierzchołka który jeszcze nie został odwiedzony i powtarzamy powyższe kroki ponownie.

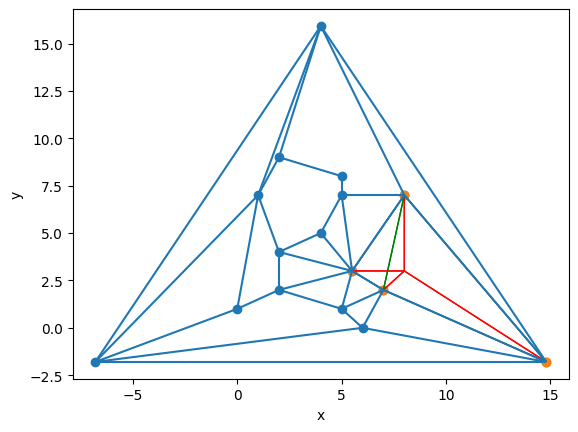


Rysunek 4 Rysunek pokazujący przykładowy wyznaczony zbiór niezależny

## Usuwanie zbioru niezależnych wierzchołków z wielokąta

Aby usunąć zbiór niezależnych punktów, dla każdego punktu ze zbioru usuwamy połączenia z listy sąsiedztwa. W czasie usuwania połączeń zapisujemy docelowe punkty aby uzyskać wielokąt tworzący dziurę. Sortujemy te punkty aby były podane w kolejności przeciwnej do wskazówek zegara. Następnie z posortowanych punktów, tworzymy zbiór trójkątów które zostały usunięte. Są one wykorzystywane do stworzenia drzewa przeszukiwań – sprawdzamy które trójkąty nowe – po ponownej triangulacji dziury – nachodzą na trójkąty poprzednie.

Na Rysunku 5 przedstawiono wielokąt po usunięciu zbioru niezależnego z rysunku 4. Jedną z dziur striangulowano. Na czerwono zaznaczono usunięte krawędzie z tejże dziury, a na zielono krawędź dodaną po ponownej triangulacji.



Rysunek 5 Przykład ilustrujący usunięcie niezależnego zbioru i ponowną triangulację jednej z dziur

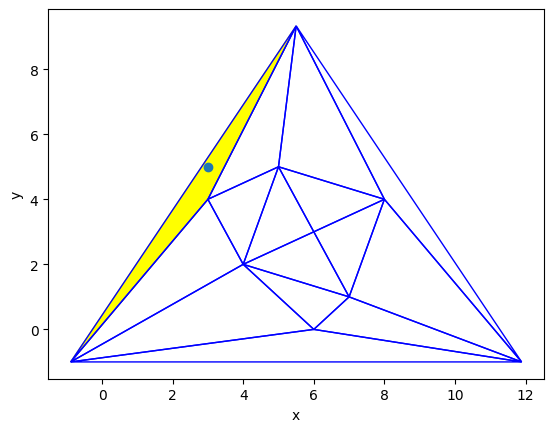
## Przetwarzanie

Przetwarzanie polega na zainicjalizowaniu grafu przeszukiwania, a następnie wykonaniu poniższych kroków na dopóki liczba wierzchołków w wielokącie jest większa od 3.

1. Znalezienie zbioru niezależnego
2. Usunięcie zbioru niezależnego
3. Zmniejszenie licznika wierzchołków o liczbę usuniętych wierzchołków
4. Następnie każdą powstałą dziurę triangulujemy
5. Dla każdego nowego trójkąta sprawdzamy z którym ze starych trójkątów on się przecina, i dodajemy tą relację do grafu przeszukiwania

## Przeszukiwanie

Przeszukiwanie realizujemy poprzez sprawdzanie w którym z dzieci obecnego trójkąta znajduje się szukany punkt. Jeśli dotrzemy do trójkąta który nie ma dzieci, to oznacza, że znaleźliśmy szukany trójkąt. Zaczynamy od dodanego zewnętrznego trójkąta.



Rysunek 6 Przykład wywołania funkcji Kirkpatrick.query\_with\_show((3,5))

# Moduł Kirkpatrick\_point\_Location\_With\_Visualization

Wizualizacja została zrealizowana przy pomocy dwóch klas.  
Pierwsza klasa KirkpatrickVisualization zawiera funkcje klasy Kirkpatrick oraz dodatkowo

* show\_query(self)
  + Opis działania:
    - Funkcja pokazuje na kolejnych wykresach, kolejne kroki wyszukiwania punktu w strukturze Hierarchii.
* show\_prep(self)
  + Opis działania:
    - Funkcja pokazuje na kolejnych wykresach, kolejne kroki preprocessingu.

Druga klasa to KirkpatrickInteractiveVisualization, jest on realizacją interfejsu graficznego pozwalającego na ręczne przechodzenie przez kolejne kroki, oraz interaktywne zadanie wielokąta i punktu.

Zawiera następujące metody dostępne dla użytkownika:

* \_\_init\_\_(self, polygon = None, point = None):
  + Argumenty:
    - w przypadku, gdy chcemy uruchomić wizualizację na gotowych danych należy podać je jako argumenty. Odpowiednio polygon - lista krotek ze współrzędnymi punktów wierzchołków wielokąta podanych w kierunku przeciwnym do wskazówek zegara oraz point – krotka zawierająca współrzędne punktu który chcemy zlokalizować.
  + Opis działania:
    - Funkcja uruchamia okienko z możliwością startu - klinięcie na przycisk start.  
      Następnie jeżeli nie podaliśmy argumentów otwiera okienkow w którym należy zadać wielokąt - przy pomocy klikania lewego przycisku myszy oraz wyszukiwany punkt za pomocą kliknięcia prawego przycisku myszy.  
      W późniejszych części przy pomocy przycisków umożliwia intuicyjne zmienianie wyświetlanych kroków.

# Przykład użycia

# dołączenie zależności dla interaktywnej wizualizacji

from kirkpatrick\_algorithm.kirkpatrick\_point\_location.

point\_location\_\_interactive\_visualization

import KirkpatrickInteractiveVisualization

# uruchomienie aplikacji z wizualizacją

KirkpatrickInteractiveVisualization()

# dołączenie zależności

from kirkpatrick\_algorithm.kirkpatrick\_point\_location.point\_location

import Kirkpatrick

# deklaracja współrzędnych wielokąta

polygon = [(5,5), (3,4), (6,3), (4,2), (6,0), (7,1), (8,4)]

# stworzenie obiektu klasy Kirkpatrick

kirkpatrick = Kirkpatrick(polygon)

# uruchomienie przetwarzania

kirkpatrick.preprocess()

# lokalizacja punktu o współrzędnych (3, 5)

found\_triangle = kirkpatrick.query((3, 5))

# uzyskanie listy wszystkich trójkątów

all\_triangles = kirkpatrick.get\_triangles()

# lokalizacja punktu o współrzędnych (3, 5) z wizualizacją

kirkpatrick.query\_with\_show((3, 5))

# Testy

# dołączenie zależności dla wizualizacji

from kirkpatrick\_algorithm.kirkpatrick\_point\_location.

point\_location\_visualization import KirkpatrickVisualization

# deklaracja współrzędnych wielokąta

polygon = [(5,5), (3,4), (6,3), (4,2), (6,0), (7,1), (8,4)]

# stworzenie obiektu klasy KirkpatrickVisualization

kirkpatrick\_vis = KirkpatrickVisualization(polygon)

# uruchomienie przetwarzania

kirkpatrick\_vis.preprocess()

# lokalizacja punktu o współrzędnych (5, 4)

\_ = kirkpatrick\_vis.query((5, 4))

# wyświetlenie obrazków kolejnych kroków przetwarzania

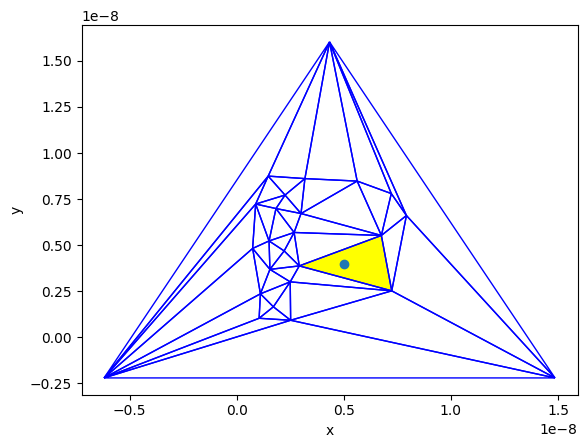
kirkpatrick\_vis.show\_prep()

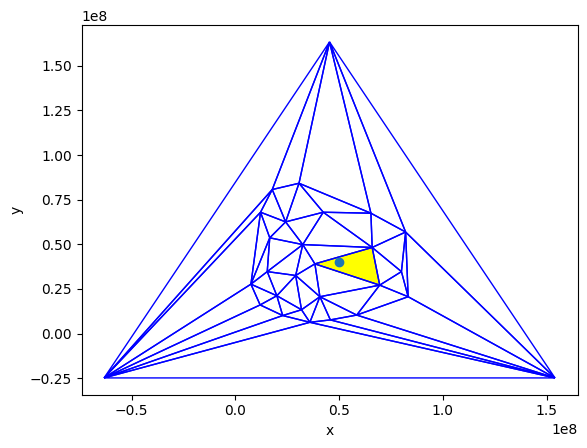
# wyświetlenie obrazków kolejnych kroków przeszukiwania

kirkpatrick\_vis.show\_query()

Wykonaliśmy testy jakościowe dla liczb rzędu 10e-8 oraz 10e8 aby zweryfikować działanie programu. W obu przypadkach algorytm dał prawidłowy wynik.

Poniżej wyniki przeprowadzonych dwóch wybranych z przeprowadzonych testów:





Oprócz tego przeprowadziliśmy także testy wydajności algorytmu. Jako, że jako wejście algorytmu podajemy chmurę punktów to możemy wygenerować zbiory testowe losowo i zmierzyć czas działania. Nasza biblioteka rozdziela triangulację (wykonuje się w konstruktorze biblioteki) i samo przetwarzanie (funkcja preprocess).

def timeit\_preprocess\_query(n):

point\_cloud = generate\_uniform\_points(0, 10\*\*8, n)

kirkpatrick = Kirkpatrick(point\_cloud)

start\_preprocess = time.process\_time()

kirkpatrick.preprocess()

end\_preprocess = time.process\_time()

start\_query = time.process\_time()

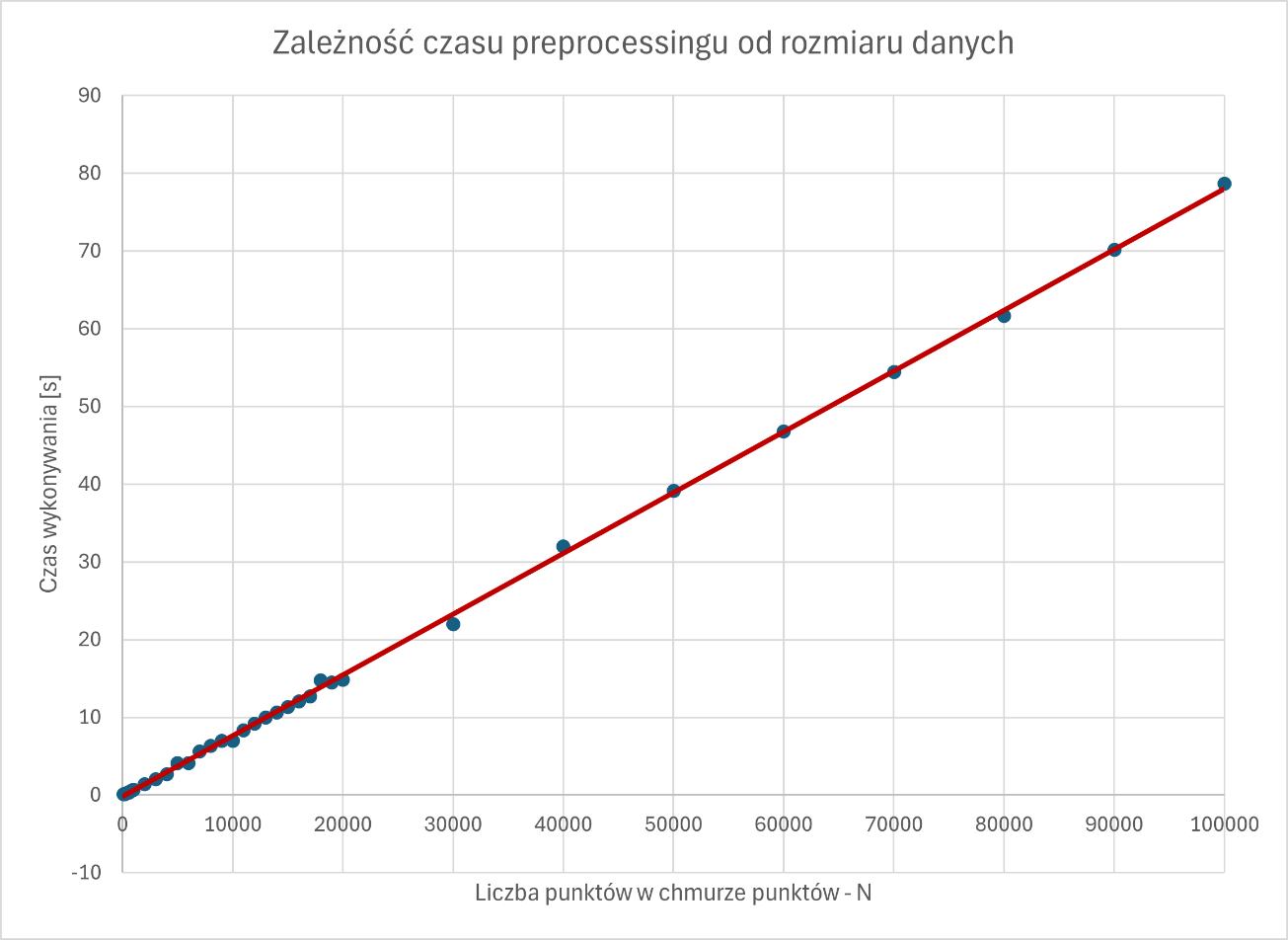
\_ = kirkpatrick.query((0.5\*10\*\*8, 0.5\*10\*\*8))

end\_query = time.process\_time()

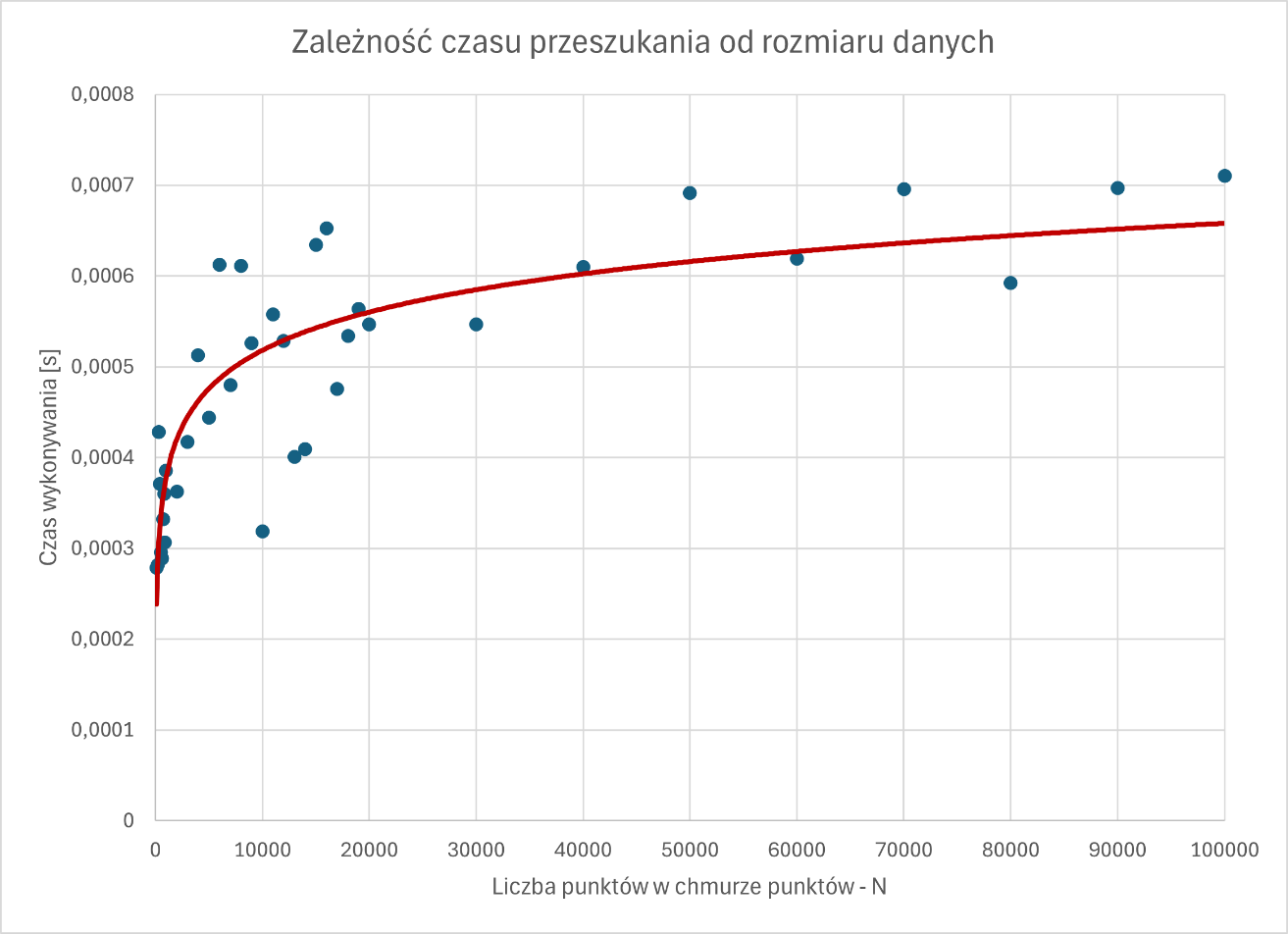
print(f"{n}\t{end\_preprocess - start\_preprocess}\t{end\_query-

start\_query}".replace('.', ','))

Przeprowadziliśmy testy dla zakresu n od 100 do 100000. Poniżej przedstawiamy wykresy uzyskanych wyników czasowych.



Rysunek 7 Zależność czasu preprocessingu od rozmiaru danych



Rysunek 8 Zależność czasu przeszukiwania od rozmiaru danych

Po analizie wykresów można stwierdzić, że obie części algorytmu działają z oczekiwaną złożonością teoretyczną. Czas przetwarzania (nie uwzględniając triangulacji Delaunaya) działa w czasie , a przeszukiwanie w czasie . W głównym notebooku dodaliśmy kod generujący dokładne dane testowe. Wykres wygenerowaliśmy programem Excel.

# Bibliografia

<https://ics.uci.edu/~goodrich/teach/geom/notes/Kirkpatrick.pdf>

https://github.com/rkaneriya/point-location

1. W tym miejscu pomijam metody i pola prywatne oznaczone w kodzie prefiksem „\_\_”, gdyż nie są one przeznaczone do ogólnego użytku. W opisie działania przedstawię funkcjonalność tychże metod [↑](#footnote-ref-1)