Otoczka wypukła dla zbioru punktów w przestrzeni dwuwymiarowej

Temat 2, Grupa 2

Autorzy: Remigiusz Babiarz, Jakub Własiewicz

7 stycznia 2026

Spis treści

1. Wstęp

Ćwiczenie polegało na implementacji algorytmów wyznaczania otoczki wypukłej zaprezentowanych na wykładzie. Zaimplementowane zostały algorytmy:

- **Grahama**,
- **Jarvisa**,
- **przyrostowy**,
- **górnej i dolnej otoczki** - zamiennie w kodzie nazywany *monochain* (ponieważ buduje x--monotoniczne łańcuchy),
- **dziel i rządź**,
- **quickhull**,
- **Chana**.

Powyższe algorytmy następnie przetestowano na uprzednio przygotowanych danych testowych oraz porównano ich działanie. Przygotowano wizualizację działania każdego z algorytmów.

2. Szczegóły techniczne

2.1. Dane sprzętowe

- **Architektura procesora**: x86_64
- **System operacyjny**: Fedora Linux 43
- **Środowisko**: NeoVim/Visual Studio Code
- **Język i wersja interpretera**: Python 3.14
- **Procesor**: 12th Gen Intel Core i5-12450H × 12

2.2. Użyte biblioteki

W projekcie wykorzystano funkcjonalności zarówno z biblioteki standardowej języka, jak i bibliotek zewnętrznych. Poniżej znajduje się lista importowanych modułów wraz z opisem zastosowania:

- **itertools** - użyto do grupowania punktów względem współrzędnej,
- **time** - użyto do mierzenia czasu wykonywania algorytmów,
- **numpy** - użyto do ułatwienia zapisu danych, losowania punktów, tasowania zbiorów. Również wszystkie funkcje matematyczne zostały zaczerpnięte z tej biblioteki,
- **matplotlib** - umożliwia wizualizację działania algorytmów,
- **os** - użyto do obsługi systemu plików podczas zapisu i odczytu danych oraz animacji.

2.3. Struktura plików

Kod źródłowy projektu został podzielony na moduły.

Moduły realizujące rozwiązanie ćwiczenia znajdują się w katalogu **/src/**. Poniżej znajduje się lista modułów wraz z opisem każdego z nich:

- **tests.py** - zawiera generatory punktów losowych, oraz funkcję umożliwiającą przeprowadzenie analizy czasu działania zbioru algorytmów na zbiorach punktów generowanych przez zadany generator. Opisy poszczególnych generatorów znajdują się w sekcji,
- **drawing.py** - pozwala na wprowadzanie zbioru punktów przez użytkownika, zawiera funkcje umożliwiające wizualizację zbioru punktów oraz otoczki tego zbioru. Ponadto zawiera klasę *Visualization* pozwalającą na wizualizację kroków algorytmu. Każdy z algorytmów posiada wersję wykorzystującą tą klasę do prezentacji graficznej poszczególnych kroków.
- **__init__.py** - plik konieczny by katalog był rozpoznawany przez język Python jako moduł,
- **pozostałe pliki** - realizacje poszczególnych algorytmów wyznaczania otoczki wypukłej. W sekcji 3.3 przy opisie każdego z algorytmów znajduje się informacja, który plik zawiera jego kod źródłowy.

Ponadto plik **main.py**, znajdujący się poza katalogiem **/src/**, został przygotowany w celu realizacji warstwy użytkownika opisanej w następnej sekcji dokumentacji.

2.4. Warstwa użytkownika

Program należy uruchomić używając pliku **main.py**. Plik wykorzystuje funkcjonaności kodu źródłowego do prezentacji działania algorytmów wyznaczania otoczki wypukłej. Całość interfejsu użytkownika realizowana jest przez konsolę. Poprzez wybór różnych opcji użytkownik może:

- przeprowadzić analizę czasu działania algorytmów dla różnych zbiorów danych,
- wprowadzić własny zbiór punktów,
- wygenerować zbiór punktów z użyciem generatora z pliku **tests.py**,
- wczytać zbiór punktów z pliku,
- zapisać zbiór punktów do pliku,
- zapisać wynik działania wybranego algorytmu - otoczkę wypukłą zbioru punktów,
- wyświetlić wizualizację działania wybranego algorytmu w oknie *matplotlib*,
- zapisać wizualizację działania wybranego algorytmu do pliku *.gif*.

Jeżeli program jest uruchamiany z poziomu **main.py**:

- zbiory punktów zapisywane są w katalogu **/data/** (są również z niego wczytywane),
- otoczki wypukłe zapisywane są w katalogu **/hulls/**,
- wizualizacje działania algorytmów zapisywane są w katalogu **/gifs/**.

3. Realizacja ćwiczenia

3.1. Dane wejściowe

Przyjmujemy, że zbiorem danych wejściowych jest zbiór punktów na płaszczyźnie. Punkty są krotkami zawierającymi dwie liczby zmiennoprzecinkowe reprezentujące ich współrzędne. Przyjmujemy, że w zbiorze punktów nie ma żadnych duplikatów (punktów o tych samych współrzędnych). Mogą natomiast wystąpić pary punktów o tej samej jednej ze współrzędnych.

3.2. Oczekiwany wynik działania algorytmów

Każdy z zaimplementowanych algorytmów ma w założeniu wyznaczyć otoczkę wypukłą punktów z danych wejściowych. Za poprawną otoczkę przyjmuje się listę punktów taką, że:

- punkty z tej listy są wierzchołkami wielokąta następującymi po sobie w kolejności przeciwnej do ruchu wskazówek zegara,
- wielokąt ten jest otoczką wypukłą zbioru danych wejściowych.

3.3. Omówienie implementacji algorytmów

3.3.1. Elementy wspólne

3.3.1.1. Wyznacznik - *det*

Funkcja *det(a,b,c)* obliczająca wyznacznik macierzy:

$$\det(a,b,c) = \begin{vmatrix} x_a & y_a & 1 \\ x_b & y_b & 1 \\ x_c & y_c & 1 \end{vmatrix}$$

została wielokrotnie wykorzystana w zaimplementowanych algorytmach. Poprzez badanie znaku tego wyznacznika można określić orientację trzech następujących po sobie punktów.
Z racji na niedokładność obliczeń, za każdym razem, gdy badano znak wartości funkcji *det*, użyto wartości $\varepsilon = 10^{-12}$ jako tolerancji dla 0.

3.3.1.2. Sortowanie z usuwaniem punktów współliniowych - *x_sort*

Funkcja *x_sort* została zaimplementowana na potrzebę sortowania punktów względem ich współrzędnej ***x***, która to funkcjonalność znajduje zastosowanie w algorytmach: ***przyrostowym***, ***górnej i dolnej otoczki*** oraz ***dziel i rządź***. Funkcja poza sortowaniem punktów dodatkowo grupuje punkty o tej samej współrzędnej ***x***. Jeżeli w danej grupie występują co najmniej 3 punkty funkcja dodatkowo usuwa ze zbioru wynikowego wszystkie punkty poza tymi o najmniejszej i największej współrzędnej ***y***. Następnie łącząc grupy zwraca posortowaną listę. Złożoność takiego sortowania wynosi $O(nlog(n))$, gdzie n to liczba punktów.

3.3.2. Algorytm Grahama

Plik **graham.py**.

3.3.2.1. Przebieg algorytmu

Algorytm rozpoczyna działanie od przygotowania danych. Najpierw znajduje w zbiorze wejściowym punkt ***lowest_point*** o najniższej drugiej współrzędnej, lub - w przypadku remisu - punkt o najniższych współrzędnych. Następne sortuje pozostałe punkty na podstawie kątu jaki tworzy odcinek tworzony przez dany punkt oraz punkt ***lowest_point*** z osią ***OX***. Kąt ten jest obliczany z pomocą funkcji *atan2* z biblioteki *numpy*. W przypadku remisu punkty porządkowane są w kierunku rosnącej odległości od punktu ***lowest_point***. Ostatnim krokiem przygotowującym dane jest usunięcie z posortowanej listy punktów współliniowych - w przypadku trójki (***lowest_point***, ***p***, ***q***), gdzie punkt ***p*** leży na odcinku (***lowest_point***, ***q***), punkt ***p*** jest usuwany ze zbioru danych.

Następnie algorytm iteracyjnie wyznacza otoczkę wypukłą przygotowanego zbioru. Tworzy stos ***hull***, na którym umieszcza ***lowest_point***. Następnie, dla każdego kolejnego punktu ***p*** z posortowanej listy wykonuje:

- dopóki na stosie są co najmniej 2 punkty sprawdź relację dwóch ostatnich punktów na stosie z punktem ***p*** z użyciem funkcji *det*:

– dopóki *det(**przedostatni punkt**, **ostatni punkt**, **p**) <= 0* (skręt w prawo), usuń ostatni punkt ze stosu,
– w przeciwnym wypadku, dodaj punkt ***p*** na górę stosu.

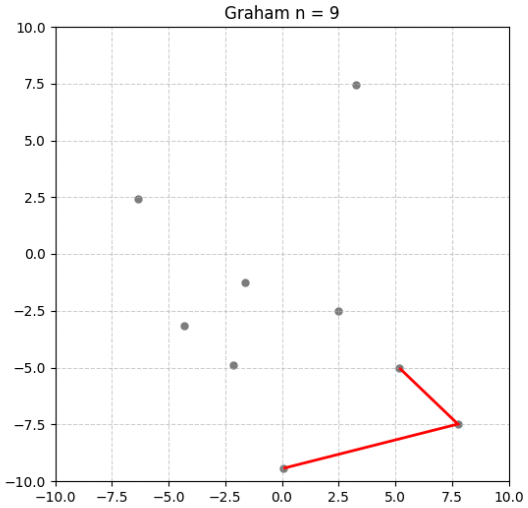
Po przetworzeniu wszystkich punków na stosie pozostaje lista wynikowa będąca otoczką wypukłą zbioru punktów z danych wejściowych.

3.3.2.2. Analiza złożoności obliczeniowej

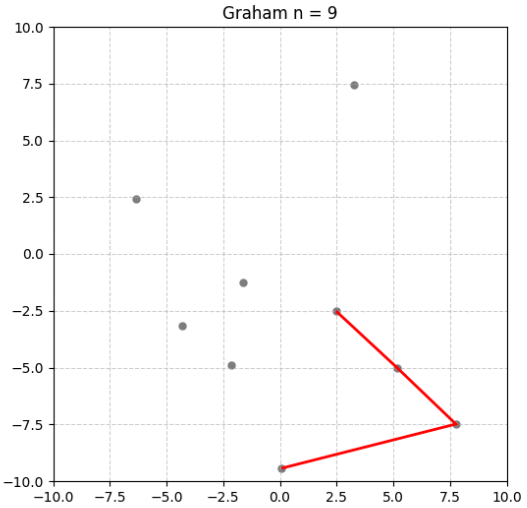
Cały algorytm ma złożoność obliczeniową $O(n \log(n))$, gdzie n to liczba punktów na płaszczyźnie. Najbardziej kosztownym etapem algorytmu jest sortowanie punktów - wykonuje się ono w czasie $O(n \log(n))$. Następny etap algorytmu jest iteracją po punktach i wymaga czasu $O(n)$. Algorytm Grahama jest bardzo uniwersalnym algorytmem o przewidywalnym czasie działania dla każdego zbioru danych.

3.3.2.3. Prezentacja działania

Dopóki warunek wypukłości nie jest naruszony algorytm dodaje punkty jeden po drugim (rys. 1, rys. 2).

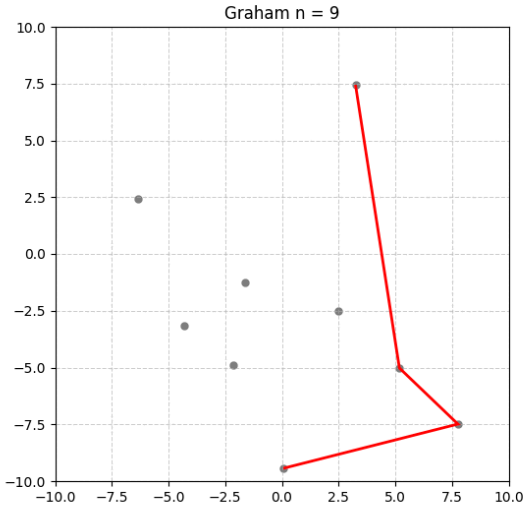


Rysunek 1: dodanie punktu

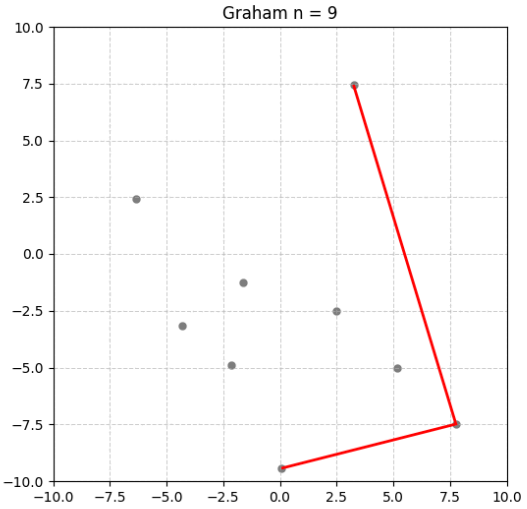


Rysunek 2: dodanie punktu

Kiedy kąt wewnętrzny okazuje się być rozwarty po dodaniu punktu algorytm usuwa dodane punkty aż warunek wypukłości będzie spełniony (rys. 3, rys. 4).



Rysunek 3: naruszenie warunku wypukłości



Rysunek 4: usunięcie punktu wewnętrznego

W ten sposób algorytm buduje całą otoczkę.

3.3.3. Algorytm Jarvisa

Plik **jarvis.py**.

3.3.3.1. Przebieg algorytmu

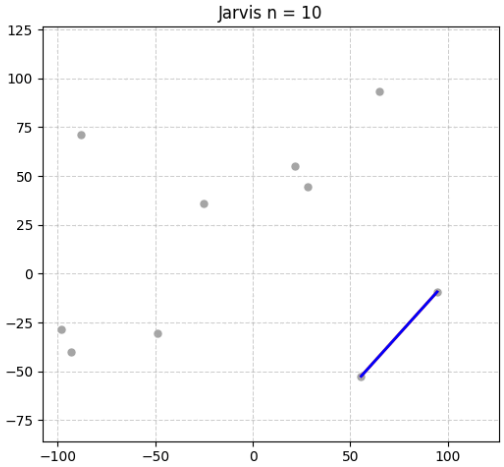
Algorytm jest inaczej nazywany algorytmem *owijania prezentu* (ang. *gift wrapping*). Podobnie jak algorytm Grahama (3.3.2) rozpoczyna działanie od znalezienia punktu ***lowest_point*** o najmniejszej pierwszej współrzędnej, lub - w przypadku remisu - o najmniejszych obu współrzędnych. Następnie algorytm znajduje następny punkt należący do otoczki z pomocą funkcji *det* - iteruje po wszystkich punktach znajdując taki punkt ***best***, dla którego wszystkie inne punkty leżą po lewej stronie odcinka (***last***, ***best***), gdzie ***last*** jest ostatnim znalezionym punktem należącym do otoczki - początkowo ***lowest_point***. Po przetworzniu wszystkich puntów punkt ***best*** staje się punktem ***last*** i jest dodawany do otoczki. Algorytm kroki te powtarza, aż znaleziony zostanie punkt startowy, co reprezentuje zamknięcie otoczki.

3.3.3.2. Analiza złożoności obliczeniowej

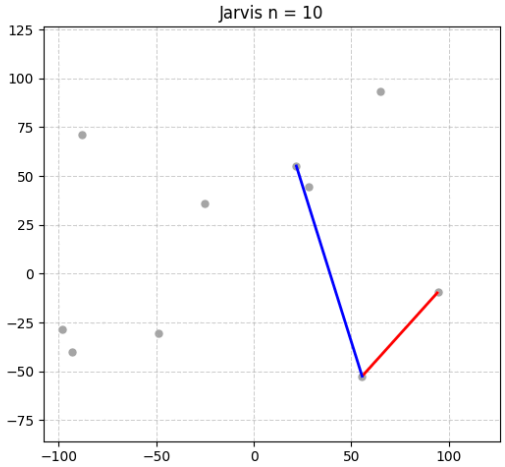
Algorytm Jarvisa ma złożoność $O(nk)$, gdzie n to liczba punktów na płaszczyźnie, oraz k to liczba punktów należących do otoczki. Wynika ona z prostego faktu znajdowania jednego punktu należącego do otoczki w każdym kroku algorytmu, który objemuje iteracje po wszystkich punktach ze zbioru wejściowego. Faktyczny czas działania algorytmu może być nieprzewidywalny i jest bardzo wrażliwy na różne dane wejściowe - w oczywisty sposób Jarvisa nie jest najlepszym wyborem do wyznaczania otoczek zbiorów punktów o potencjalnie wielu punktach należących do otoczki.

3.3.3.3. Prezentacja działania

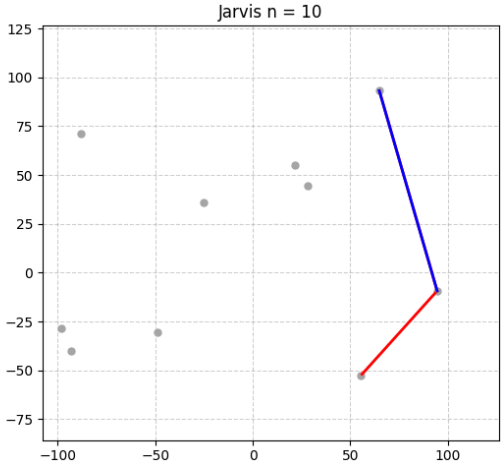
Na rysunkach 5-8 zaprezentowano działanie algorytmu.



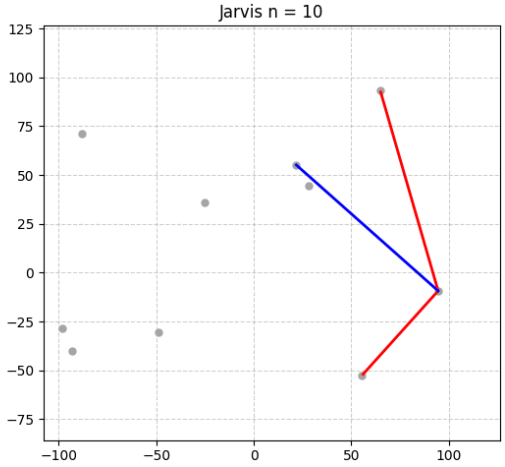
Rysunek 5: sprawdzanie punktów jeden po drugim



Rysunek 6: algorym pamięta najodleglejszy w sensie biegunowym punkt



Rysunek 7: wszystkie punkty zostały przetworzone



Rysunek 8: powtarzanie kroków dla nowego punktu otoczki

W ten sposób algorytm buduje całą otoczkę.

3.3.4. Algorytm przyrostowy

Plik **incremental.py**.

3.3.4.1. Przebieg algorytmu

Algorytm najpierw sortuje punkty z użyciem funkcji *x_sort*.

Algorytm tworzy pierwszą otoczkę na podstawie dwóch pierwszych punktów posortowanego zbioru. Następnie iteracyjnie dodaje każdy z pozostałych punktów do otoczki. Dołączanie punktu opiera się na znalezieniu stycznych do otoczki przechodzących przez ten punkt i usunięcie wszystkich punktów otoczki, które znajdowałyby się wewnątrz otoczki po dodaniu do niej rozważanego punktu.

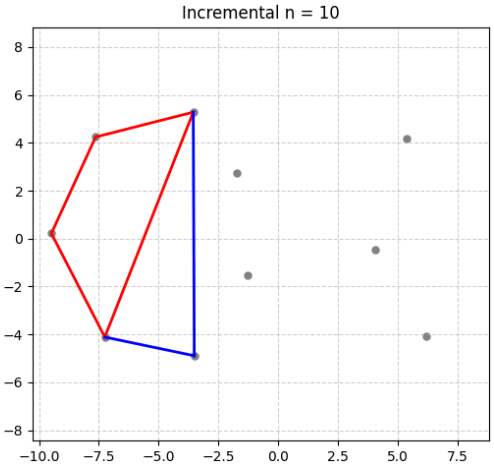
Styczne znajdowane są z pomocą funkcji *det*.

3.3.4.2. Analiza złożoności algorytmu

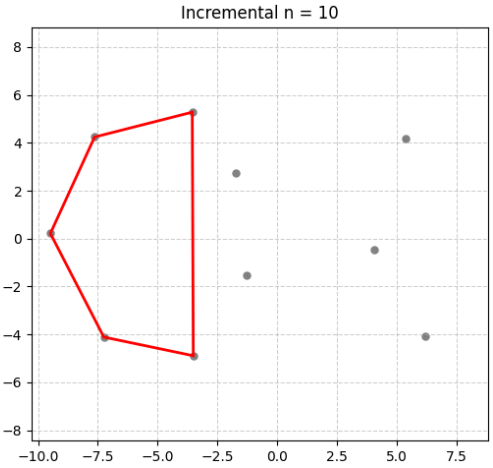
Algorytm ma złożoność $O(n \log(n))$, gdzie n to liczba punktów na płaszczyźnie. Samo sortowanie punktów zajmuje $O(n \log(n))$ czasu procesora. Podczas iteracyjnego dołączania punktów do otoczki każdy punkt jest dodawany do otoczki raz i maksymalnie raz z niej usuwany, złożoność tego kroku wynosi więc $O(n)$, a finalna złożoność algorytmu przyrostowego to faktycznie $O(n \log(n))$.

3.3.4.3. Prezentacja działania

Tak długo jak nie występują punkty wewnętrzne algorytm dodaje każdy punkt w kolejności sortowania (rys. 9, rys. 10).

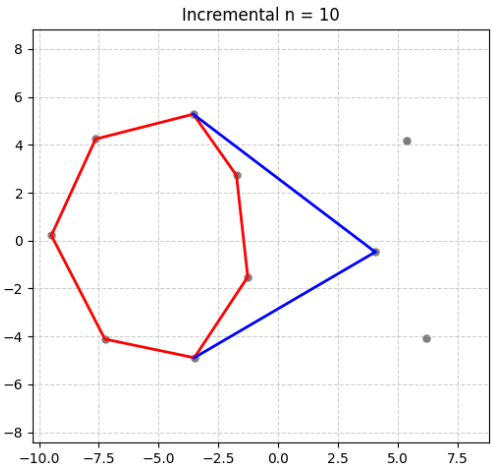


Rysunek 9: dodawanie punktu

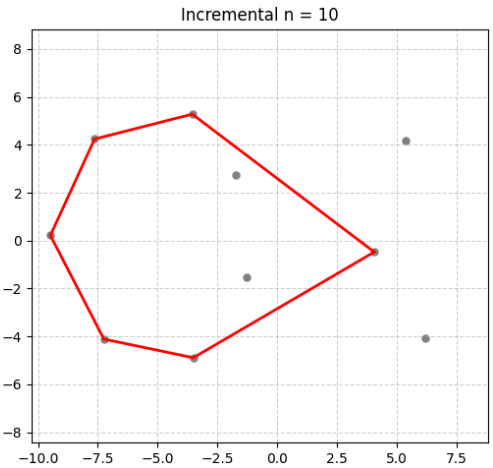


Rysunek 10: brak punktów wewnętrznych

W momencie wystąpienia punktów wewnętrznych algorytm usuwa wszystkie takie punkty (rys. 11, rys. 12).



Rysunek 11: dodawanie punktu



Rysunek 12: punkty wewnętrzne usunięte z otoczki

W ten sposób algorytm buduje całą otoczkę.

3.3.5. Algorytm górnej i dolnej otoczki

Plik **monochain.py**.

3.3.5.1. Przebieg algorytmu

Algorytm rozpoczyna pracę od posortowania zbioru punktów z użyciem funkcji *x_sort*.

Następnie algorytm iteracyjnie konstuuje górną otoczkę punktów. Początkowa górna otoczka składa się z dwóch pierwszych punktów w posortowanym zbiorze. Każdy kolejny punkt jest dodawany do górnej otoczki, po uprzednim usunięciu z niej wszystkich punktów, których obecność naruszyła by warunek wypukłości, który sprawdzany jest z użyciem funkcji *det*. Dolna otoczka wyznaczana jest analogicznie.

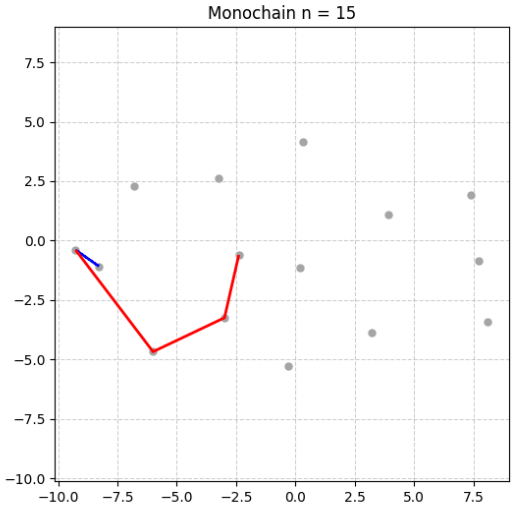
Ostatnim krokiem algorytmu jest połączenie górnej i dolnej otoczki z uwagą na warunek prawoskrętności otoczki.

3.3.5.2. Analiza złożoności obliczeniowej

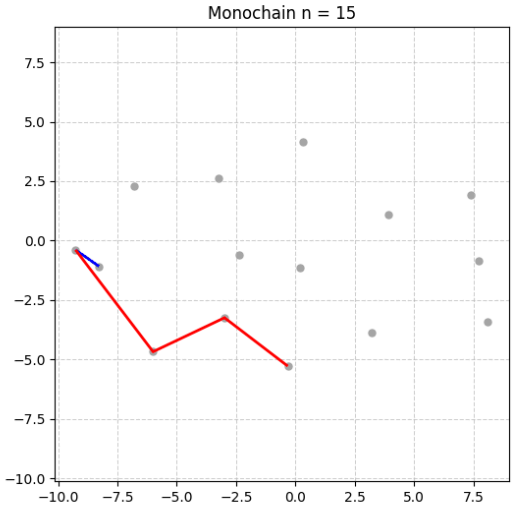
Sortowanie punktów wykonywane jest w czasie $O(n \log(n))$, gdzie n to liczba punktów na płaszczyźnie. Każdy punkt jest przetwarzany w iteracyjnej części algorytmu stałą liczbę razy. Finalna złożoność algorytmu górnej i dolnej otoczki to więc $O(n \log(n))$.

3.3.5.3. Prezentacja działania algorytmu

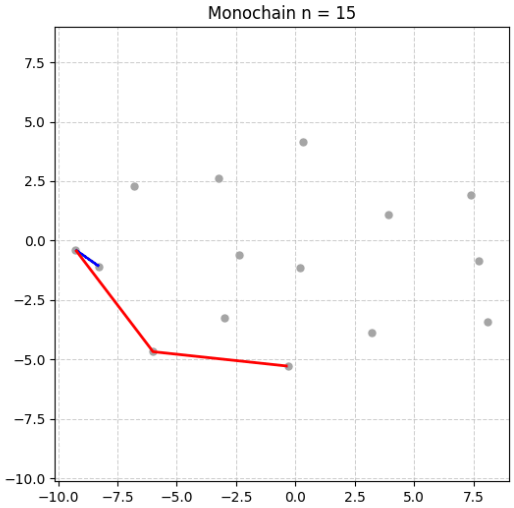
Na rysunkach 13, 14, 15 zaprezentowano kroki budowy dolnej otoczki. Górna otoczka jest budowana analogicznie i razem z dolną tworzy otoczkę wypukłą zbioru (rys. 16).



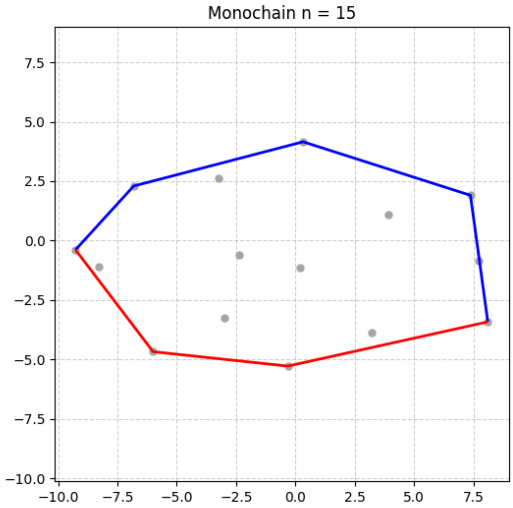
Rysunek 13: budowanie dolnej otoczki



Rysunek 14: naruszenie warunku wypukłości



Rysunek 15: usunięcie punktów wewnętrznych



Rysunek 16: otoczka wypukła będąca sumą otoczki górnej i dolnej

3.3.6. Algorytm dziel i rządź

Plik **divide_and_conquer.py**.

3.3.6.1. Przebieg algorytmu

Algorytm opiera się na utworzeniu zbioru otoczek, które w sumie obejmują cały zbiór punktów wejściowych, oraz na późniejszym łączeniu ich w czasie stałym do momentu otrzymania jednej otoczki obejmującej cały zbiór.

Pierwszym krokiem jest posortowanie punktów z użyciem funkcji *x_sort*. Następnie tak posortowana lista jest dzielona na części. Każda z tych części jest listą kolejnych ***k*** punktów należących do posortowanej listy wejściowej, gdzie ***k*** jest małą stałą będącą parametrem algorytmu. Następnie dla każdego z tych podzbiorów punktów wyznaczana jest otoczka wypukła z użyciem algorytmu **Grahama**.

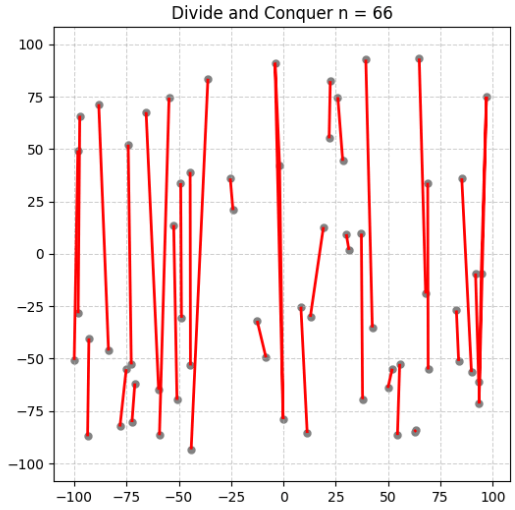
Tak powstałe sąsiednie otoczki są łączone poprzez znajdowanie stycznych z użyciem funkcji *det*. Łączenie to jest powtarzane do momentu otrzymania jednej otoczki będącej sumą wszystkich otoczek.

3.3.6.2. Analiza złożoności obliczeniowej

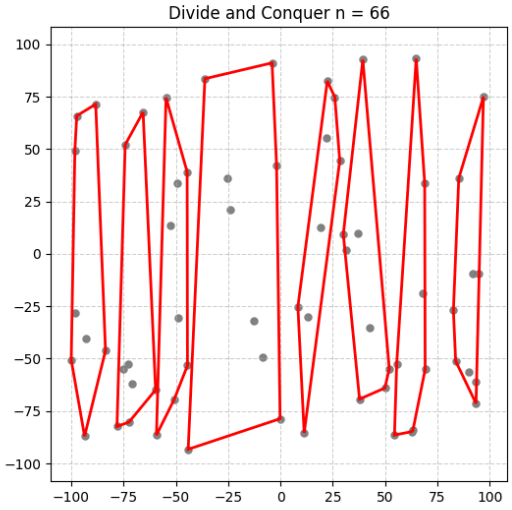
Samo sortowanie zbioru wejściowego wykonuje się w czasie $O(n\log(n))$. Wyznaczanie otoczek podzbiorów dla małej stałej k zajmuje stały czas $O(k)$. Ponieważ ten krok powtarzany jest dla n/k otoczek zajmuje on w sumie $O\left(k \times \frac{n}{k}\right) = O(n)$ czasu procesora. Łączenie 2 otoczek zajmuje stały czas, a samych otoczek do połączenia jest $\frac{n}{k}$. Czas poświęcany na ten krok wynosi więc $O\left(\frac{n}{k}\log\frac{n}{k}\right) = O(n\log n)$. Finalna złożoność algorytmu wynosi więc $O(n\log(n))$.

3.3.6.3. Prezentacja działania algorytmu

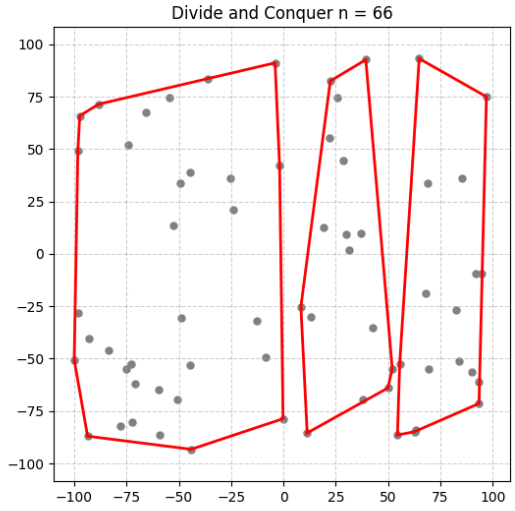
Na rysunkach 17-20 zaprezentowano wybrane kroki algorytmu, na których widać, jak otoczki łączą się. Połączenie wszystkich otoczek jest otoczką wypukłą zbioru.



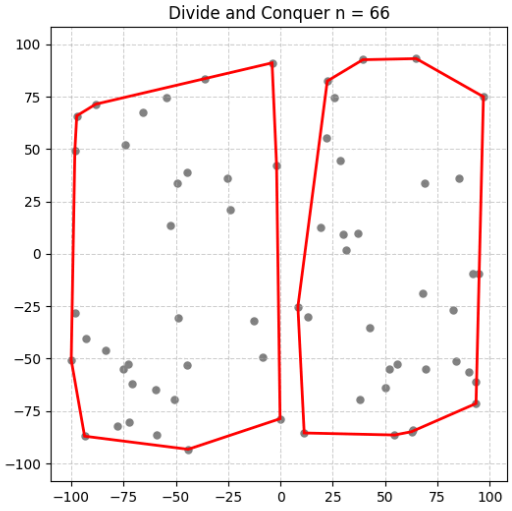
Rysunek 17: początkowy stan dla k=2



Rysunek 18: połączone otoczki



Rysunek 19: trzeci od końca krok algorytmu



Rysunek 20: przedostatni krok algorytmu

3.3.7. Algorytm Quickhull

Plik **quickhull.py**

3.3.7.1. Przebieg algorytmu

Algorytm rozpoczyna pracę od znalezienia 4 punktów o skrajnych współrzędnych:

- maksymalnej współrzędnej ***x***,
- minimalnej współrzędnej ***x***,
- maksymalnej współrzędnej ***y***,
- minimalnej współrzędnej ***y***.

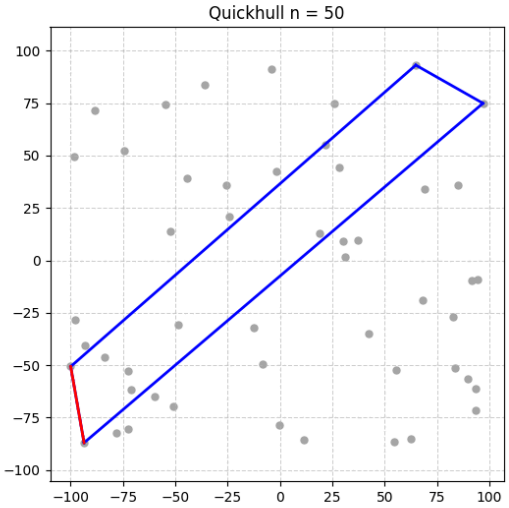
Punkty te definiują 4 odcinki, które tworzą wielokąt. Wielokąt ten określamy otoczką, którą będziemy rekurencyjnie rozszerzać. Wszystkie punkty wewnątrz otoczki są usuwane. Pozostałe punkty przetwarzane są rekurencyjnie z użyciem metody *dziel i rządź* poprzez wywoływanie funkcji ***rec_hull***, która działa następująco:
Dla danego odcinka znajdowany jest punkt znajdujący się najdalej od niego i będący na zewnątrz wielokąta tworzonego przez aktualne punkty otoczki. Punkt ten wraz z końcami rozpatrywanego odcinka definuje 2 kolejne odcinki, które dodawane są do otoczki. Na tych nowych odcinkach rekurencyjnie wywoływana jest funckja ***rec_hull*** tak długo jak istnieją punkty poza otoczką.

3.3.7.2. Analiza złożoności obliczeniowej

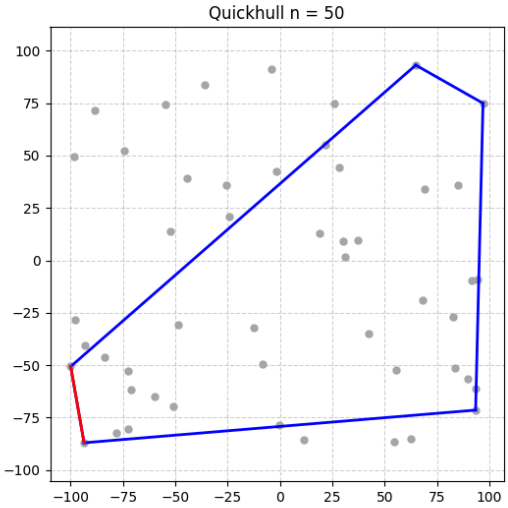
Każde rekurencyjne wywołanie fukcji ***rec_hull*** iteruje się po zbiorze punktów, którego rozmiar jest proporcjonalny do rozmiaru zbioru wejściowego. Ilość takich wywołań w pełni zależy od charakterystyki zbioru wejściowego. W pesymistycznym przypadku w każdym wywołaniu ***rec_hull*** jedyny usuwany punkt jest tym najbardziej odległym od rozpatrywanego, a pozostałe punkty leżą w pełni na jednej gałęzi rekurecji - wtedy następuje efektywnie identyczne w sensie czasu pracy wywołanie funkcji ***rec_hull*** i złożoność algorytmu wynosi $O(n^2)$. Realistycznie jednak, zakładając względnie równomierne rozłożenie punktów, przy każdym „powiększaniu” otoczki przez funkcję ***rec_hull*** punkty należące do obszaru proporcjonalnego do długości odcinka są usuwane, a pozostałe punkty dzielone są między dwie gałęzie rekurecji. Zamortyzowana złożoność obliczeniowa wynosi więc *O(nlog(n))*.

3.3.7.3. Prezentacja działania algorytmu

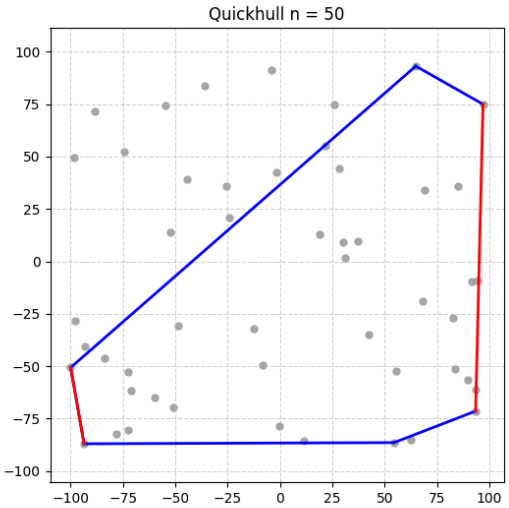
Na rysunkach 21-24 zaprezentowano wybranekroki algorytmu quickhull. Na czerwono zostały oznaczone odcinki, które na danym etapie algorytmu zostały przetworzone i na pewno należą do otoczki.



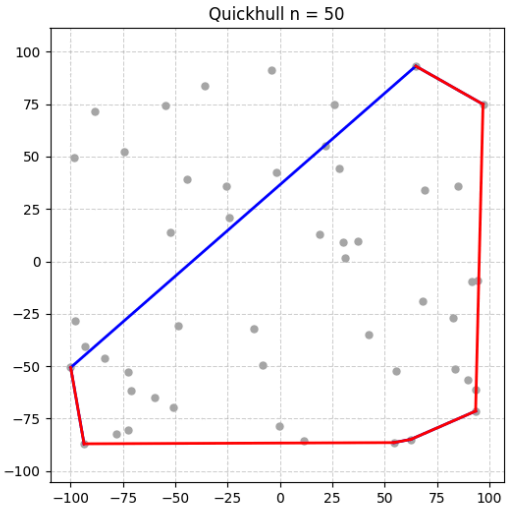
Rysunek 21: rozpoczęcie rekurencji



Rysunek 22: rozbicie odcinka



Rysunek 23: jedna z gałęzi zakończyła rekurencję



Rysunek 24: stan otoczki na moment przetworzenia 3 z 4 odcinków startowych

Przetworzenie ostatniego odcinka (rys. 24 - niebieski odcinek) skutkuje wyznaczeniem otoczki wypukłej.

3.3.8. Algorytm Chana

Plik **chan.py**.

3.3.8.1. Przebieg algorytmu

Algorytm zakłada, że znamy rozmiar otoczki m, więc algorytm będzie zwiększał zmienną t począwszy od zera, przypisując $m := \min\left(2^{2^t}, n\right)$ i wywołując funkcję ***step_chan***, aż do znalezienia otoczki. Funkcja ***step_chan*** przyjmuje zbiór punktów oraz rozmiar otoczki m. Następnie dzieli zbiór na $\lceil \frac{n}{m} \rceil$ grup o rozmiarze m i wyznacza dla każdej otoczkę algorytmem Grahama. Algorytm rozpoczyna od punktu o największej współrzędnej $\boldsymbol{x}$ i stara się znaleźć punkt maksymalizujący kąt tj. taki, że po lewej stronie prostej od poprzedniego punktu do szukanego znajdują się wszystkie inne punkty zbioru, tak jak w algorytmie Jarvisa, tylko zamiast sprawdzać wszystkie punkty, algorytm znajduje punkty przecinane przez prawe styczne otoczek. Jeżeli nie uda się znaleźć otoczki o rozmiarze m, funkcja zwraca *None*, a główna pętla zwiększa wartość zmiennej t.

3.3.8.2. Analiza złożoności obliczeniowej

Funkcja ***step_chan*** dzieli zbiór na $\frac{n}{m}$ grup rozmiaru m i wyznacza otoczkę każdej z nich, co posiada złożoność $O(m \log m) \times \frac{n}{m} = O(n \log m)$. Następnie wykonywana jest pierwsza pętla $O(m)$ razy, druga pętla $\frac{n}{m}$ razy, a szukanie stycznej dla otoczek zajmuje $O(\log m)$ używając funkcji ***rtangent***, co razem daje $O(n \log m)$ - końcową złożoność funkcji. W funkcji ***chan***, algorytm zwiększy t maksymalnie $O(\log \log h)$ razy. Ponieważ $m := \min\left(n, 2^{2^t}\right)$, to ***step_chan***, ma złożoność $O\left(n \log 2^{2^t}\right) = O\left(n \cdot 2^t\right)$. Całkowita złożoność to:

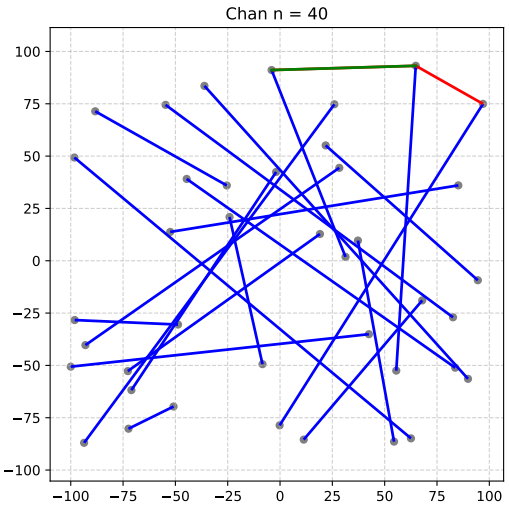
$$\sum_{t=0}^{\log \log h} O\left(n \cdot 2^t\right)$$

Jesteśmy w stanie ograniczyć sumę od góry, aby dojść do końcowej złożoności:

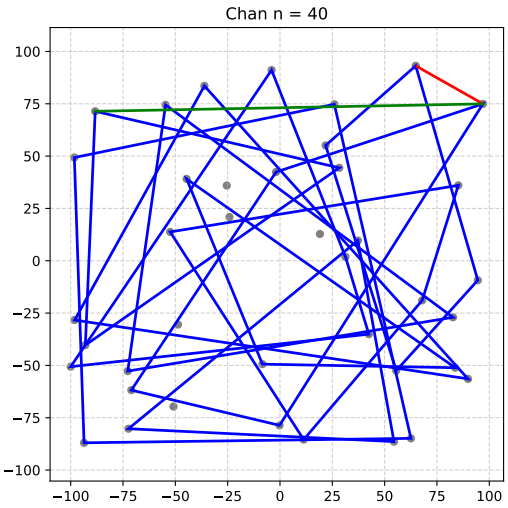
$$\sum_{t=0}^{\log \log h} O\left(n \cdot 2^t\right) = O(n) \sum_{t=0}^{\log \log h} O\left(2^t\right) \leq O(n) \cdot O\left(2^{\log \log h + 1}\right) = O(n \log h)$$

3.3.8.3. Prezentacja działania algorytmu

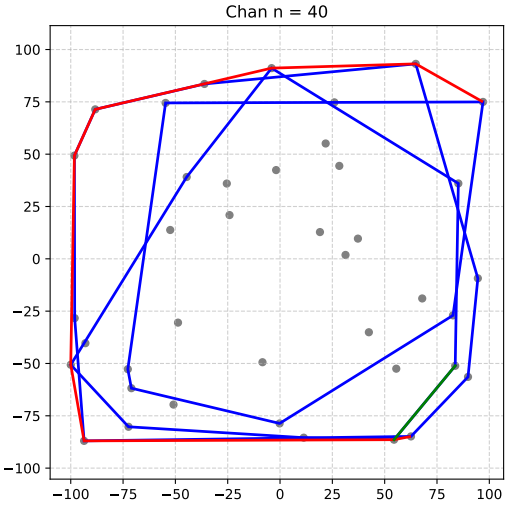
Na rysunkach 25 do 28 przedstawione są trzy wywołania funkcji ***step_chan***, w trzeciej udało się wyznaczyć otoczkę zanim m zrównało się z n.



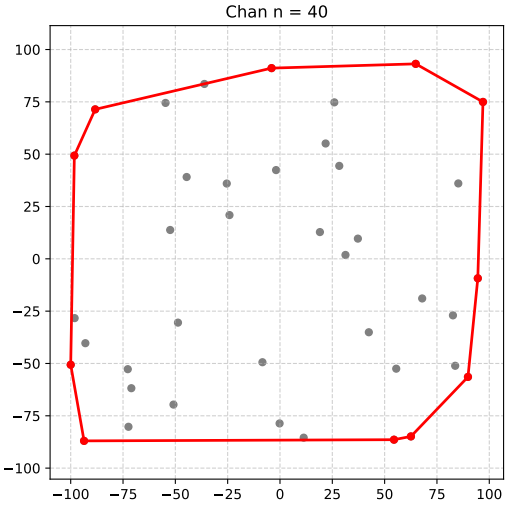
Rysunek 25: pierwsza próba, $m = 2$



Rysunek 26: druga próba, $m = 4$



Rysunek 27: trzecia próba, $m = 8$



Rysunek 28: wyznaczona otoczka

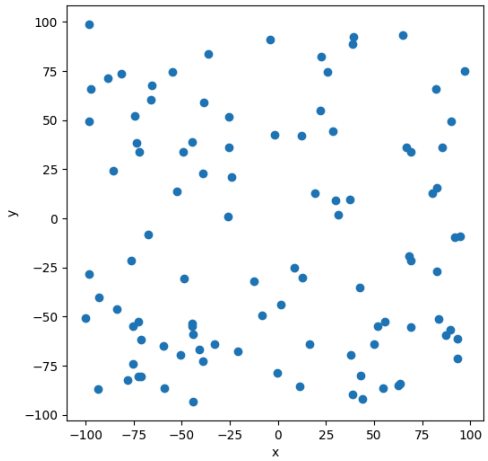
3.4. Przygotowane generatory zbiorów testowych

Plik **tests.py**.

3.4.1. generate_uniform_points

Generuje zbiór **n** losowych punktów leżących w obszarze [**left**,**right**] × [**left**,**right**], gdzie **left**, **right** oraz **n** to parametry generatora.

Poniżej, na rysunku 29, znajduje się wizualizacja przykładowego zbioru punktów wygenerowana z użyciem generatora **generate_uniform_points** i parametrów **n** = 100, **left** = −100, **right** = 100.

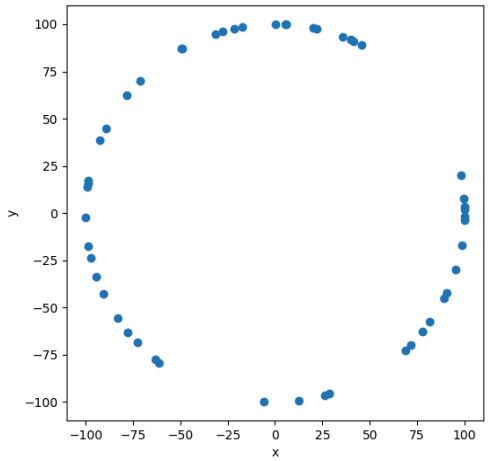


Rysunek 29: przykładowy zbiór punktów

3.4.2. generate_circle_points

Generuje zbiór **n** losowych punktów leżących na kole o środku w punkcie **O** oraz promieniu **R**, gdzie **O**, **R** oraz **n** to parametry generatora.

Poniżej, na rysunku 30, znajduje się wizualizacja przykładowego zbioru punktów wygenerowana z użyciem generatora **generate_circle_points** i parametrów **n** = 50, **O** = (0,0), **R** = 100.

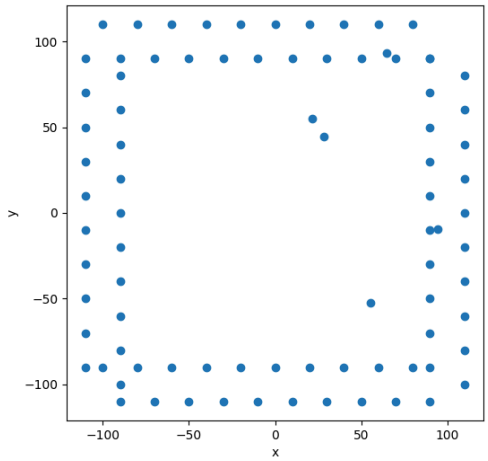


Rysunek 30: przykładowy zbiór punktów

3.4.3. generate_zigzag_points

Generuje zbiór **n** losowych punktów leżących na prostokącie o środku w początku układu współrzędnych, oraz o długościach boków **width** oraz **height**, które są parametrami generatora. Punkty dodatkowo otoczone są naprzemienną obramówką punktów, tak jak widać na rysunku 31. Szerokość obramówki definiuje parametr **amplitude**, a częstość punktów na niej parametr **period**.

Poniżej, na rysunku 31, znajduje się wizualizacja przykładowego zbioru punktów wygenerowana z użyciem generatora **generate_zigzag_points** i parametrów **n** = 5, **width** = 200, **height** = 200, **amplitude** = 10, **period** = 10.

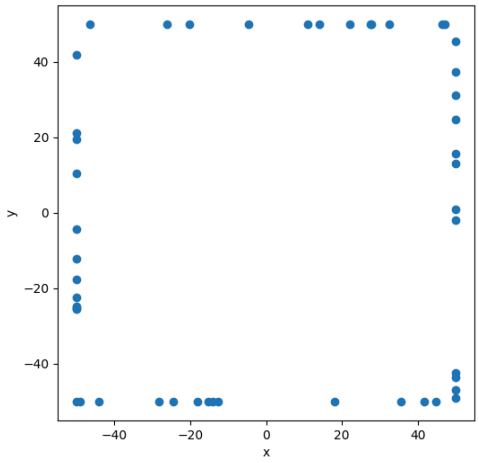


Rysunek 31: przykładowy zbiór punktów

3.4.4. generate_square_points

Generuje zbiór **n** losowych punktów leżących na kwadracie o środku w początku układu współrzędnych i o boku długości **a**, gdzie **a** to parametr generatora.

Poniżej, na rysunku 32, znajduje się wizualizacja przykładowego zbioru punktów wygenerowana z użyciem generatora **generate_square_points** i parametrów **n** = 50, **a** = 100.

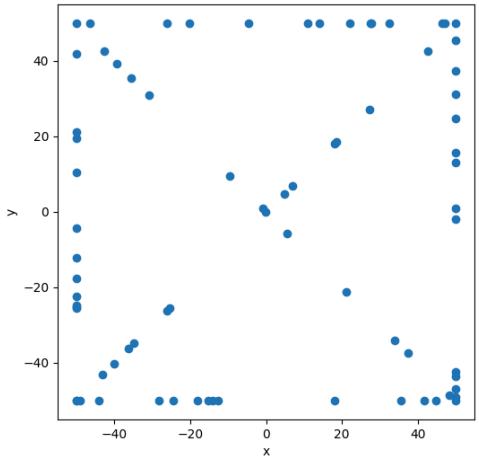


Rysunek 32: przykładowy zbiór punktów

3.4.5. generate_x_square_points

Generuje zbiór **n** losowych punktów leżących na kwadracie o środku w początku układu współrzędnych i o boku długości **a**, lub na jego przekątnych gdzie, **a** to parametr generatora. Ponadto generator dodaje do zbioru wynikowego 4 punkty będące wierzchołkami kwadratu.

Poniżej, na rysunku 33, znajduje się wizualizacja przykładowego zbioru punktów wygenerowana z użyciem generatora **generate_x_square_points** i parametrów **n** = 50, **a** = 100.



Rysunek 33: przykładowy zbiór punktów

3.5. Testy algorytmów na przygotowanych zbiorach

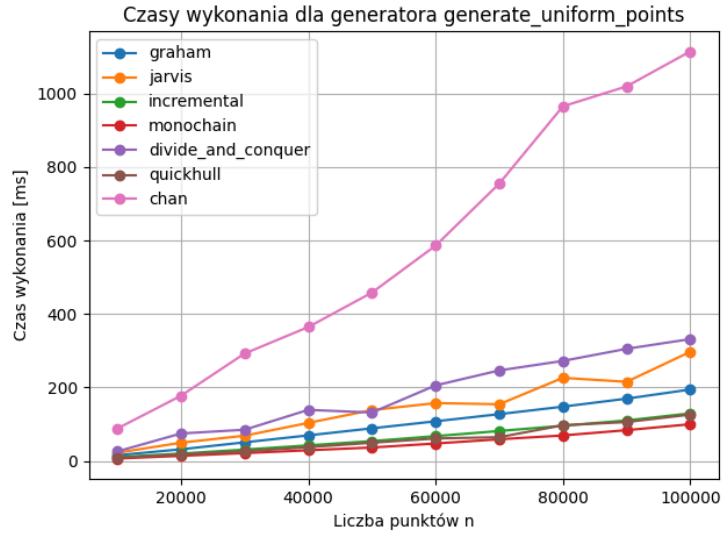
Z użyciem pliku **main.py** przeprowadzono analizę czasu działania algorytmów dla zbiorów punktów o różnych licznościach wygenerowanych przez generatory opisane w sekcji 3.4. Wyniki i wnioski dla każdego z generatorów znajdują się poniżej.

3.5.1. Wyniki dla generatora generate_uniform_points

Generator **generate_uniform_points** jest najprostszym z generatorów, analiza czasów działania algorytmów na zbiorach wygenerowanych przez niego może dać dobre pojęcie o prędkości każdego z algorytmów dla zbiorów danych o nieznanej charakterystyce (nie da się jednoznacznie stwierdzić ile punktów należy do otoczki, ani ile występuje punktów współliniowych).

Liczba punktów	Graham [ms]	Jarvis [ms]	Przyrostowy [ms]	Górnej i dolnej otoczki [ms]	Dziel i rządź [ms]	Quickhull [ms]	Chan [ms]
20000	33.34	53.28	22.6	15.65	74.28	18.53	176.74
40000	67.39	95.16	44.4	29.1	113.4	35.45	379.74
60000	106.15	160.03	71.36	46.06	205.38	58.97	548.89
80000	146.23	231.49	105.18	70.33	269.23	86.72	811.12
100000	193.13	266.97	136.79	94.12	331.78	121.46	1045.29

Tabela 1: czasy działania poszczególnych algorytmów na zbiorach punktów wygenerowanych przez **generate_uniform_points**



Rysunek 34: wykres na podstawie danych z tabeli 1

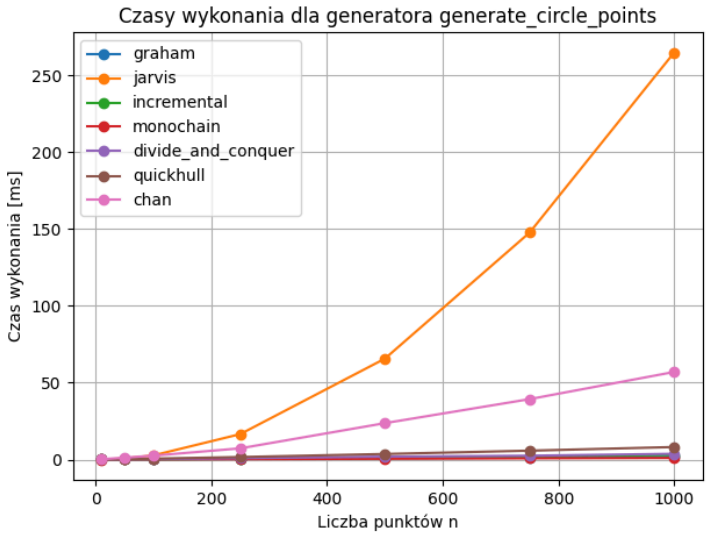
Czasy działania algorytmów zostały przedstawione w tabeli 1. Najszybszy okazał się algorytm **górnej i dolnej otoczki** (monochain). Zdecydowanie najwolnieszy był algorytm **Chana**, pomimo jego teoretycznie największej asymptotycznej prędkości. Wynika to najprawdopodobnej z faktu, że stała nie brana pod uwagę w złożoności $O(n \log k)$ jest bardzo wysoka i dopiero dla znacznie liczniejszych zbiorów danych rożnica byłaby znacząca. Jednakże jak widać na rysunku 30 dla tego generatora liczba punktów musiałaby być znacznie zbyt duża, by algorytm okazał się szybszy na sprzęcie, na którym był testowany.

3.5.2. Wyniki dla generatora generate_circle_points

Każdy punkt należący do zbioru wygenerowanego przez generator **generate_circle_points** jest należy do otoczki wypukłej tego zbioru. Przez to algorytmy **Jarvisa** oraz **Chana** powinny działać w teorii dłużej na takich zbiorach. Przeprowadzono stosowne testy i wyniki przedstawiono w tabeli 2.

Liczba punktów	Graham [ms]	Jarvis [ms]	Przyrostowy [ms]	Górna i dolna [ms]	Dziel i rządź [ms]	Quickhull [ms]	Chan [ms]
10	0.03	0.04	0.03	0.02	0.07	0.04	0.21
50	0.09	0.68	0.08	0.05	0.17	0.23	1.22
100	0.17	2.7	0.17	0.11	0.32	0.57	2.52
250	0.44	16.54	0.44	0.26	0.89	1.62	7.28
500	0.86	65.65	1.07	0.53	1.88	3.62	23.68
750	1.3	147.64	1.95	0.8	2.49	5.77	39.3
1000	1.76	264.87	3.03	1.03	3.74	8.14	57

Tabela 2: czasy działania poszczególnych algorytmów na zbiorach punktów wygenerowanych przez **generate_circle_points**



Rysunek 35: wykres na podstawie danych z tabeli 2

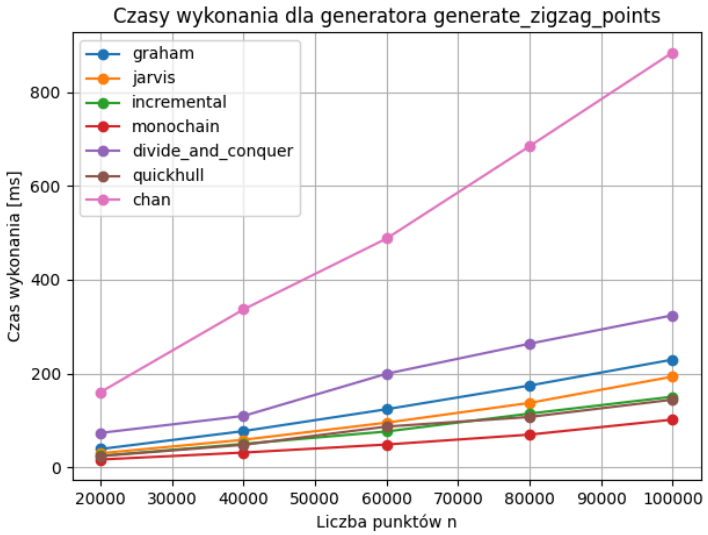
Z danych z tabeli 2, jak i z wykresu na rysunku 35 jasno wynika pogorszenie się złożoności algorytmów **Jarvisa** oraz **Chana**. Algorytm **Jarvisa** zdegradował do złożoności $O(n^2)$ i nawet na zbiorach o małej liczności jego czas pracy był bardzo długi. Podobnie algorytm **Chana**, którego złożoność zdegradowała do $O(n \log n)$ okazał się asymptotycznie wolniejszy niż w przypadku zbiorów wygenerowanych przez **generate_uniform_points**.

3.5.3. Wyniki dla generatora generate_zigzag_points

Ze względu na obramówkę każdy zbiór punktów wygenerowanych przez **generate_zigzag_points** ma dokładnie 8 punktów należących do swojej otoczki wypukłej. Dzięki temu algorytmy Jarvisa oraz Chana powinny osiągnąć znacznie lepsze czasy działania niż dla zbiorów nie cechujących się własnością stałej liczby punktów otoczki. Wyniki stosownych testów zaprezentowano w tabeli 3 oraz na wykresie widocznym na rysunku 36.

Liczba punktów	Graham [ms]	Jarvis [ms]	Przyrostowy [ms]	Górna i dolna [ms]	Dziel i rządź [ms]	Quickhull [ms]	Chan [ms]
20000	38.56	29.53	24.05	16.19	72.96	23.48	160.24
40000	76.84	58.33	49.54	31	109.25	47.56	336.87
60000	123.54	94.85	76.01	48.19	199.31	86.49	488.16
80000	174.17	137.08	114.23	69.36	263.54	107.07	685.34
100000	229.51	193.2	150.31	101.41	324.14	144	885.59

Tabela 3: czasy działania poszczególnych algorytmów na zbiorach punktów wygenerowanych przez **generate_zigzag_points**



Rysunek 36: wykres na podstawie danych z tabeli 3

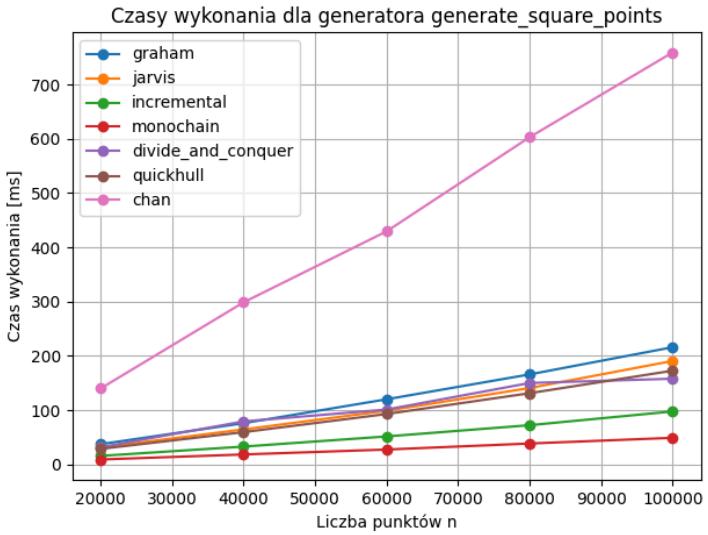
Wyniki testów widoczne w tabeli 3 potwierdzają hipotezę o korzystności danych o małej liczbie punktów otoczki dla algorytmów **Jarvisa** oraz **Chana**. Porównując je z wynikami dla generatora **generate_uniform_points**, gdzie liczba punktów w otoczce nie jest stała i może rosnąć można zauważyć, że wszystkie algorytmy poza wymienionymi powyżej osiągają podobne czasy działania (czasami nieznacznie większe, ze względu na dodanie punktów obramówki), natomiast algorytmy **Chana** oraz **Jarvisa** radzą sobie szybciej.

3.5.4. Wyniki dla generatora generate_square_points

Otoczka każdego zbioru punktów wygenerowanego przez **generate_square_points** ma 4 punkty (wykluczając przypadki zdegradowane np. gdy żaden punkt nie znajduje się na którymś boku kwadratu). Ponownie spodziewamy się poprawy czasu działania algorytmów **Chana** i **Jarvisa**. Wyniki testów znajdują się w tabeli 4.

Liczba punktów	Graham [ms]	Jarvis [ms]	Przyrostowy [ms]	Górna i dolna [ms]	Dziel i rządź [ms]	Quickhull [ms]	Chan [ms]
20000	37.36	31.3	15.72	9.1	31.75	28.67	139.93
40000	75.99	64.53	32.89	18.38	79.29	59.51	298.72
60000	120.05	98.9	51.59	27.46	101.1	92.93	429.49
80000	165.86	140.75	72.3	38.54	150.06	131.26	602.92
100000	215.95	190.67	97.63	49.03	157.94	172.72	759.16

Tabela 4: czasy działania poszczególnych algorytmów na zbiorach punktów wygenerowanych przez **generate_square_points**



Rysunek 37: wykres na podstawie danych z tabeli 4

Poza ponownym potwierdzeniem tezy o liniowej złożoności algorytmów **Jarvisa** oraz **Chana** dla zbiorów o stałej liczbie punktów otoczki, z tabeli 4 można wyciągnąć parę ciekawych wniosków:

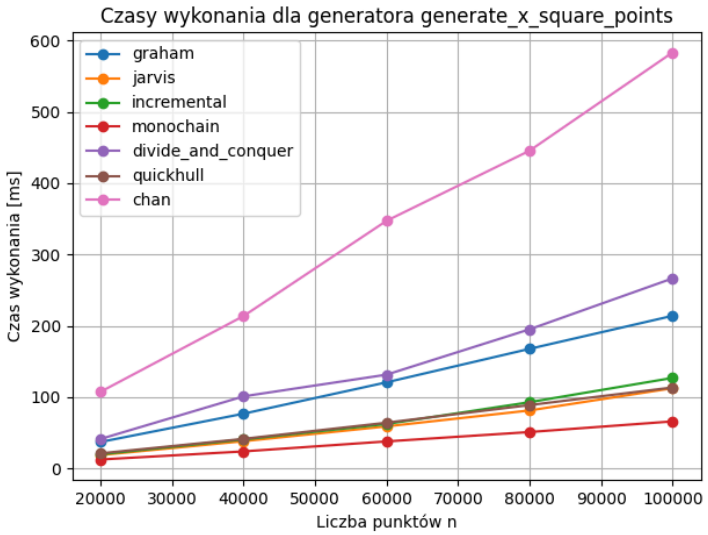
- w przybliżeniu prostokątny kształt i stały rozmiar (4 punkty) podotoczek tworzonych podczas działania algorytmu **dziel i rządź** pozwolił na szybkie ich łączenie i algorytm osiągnął czas w przybliżeniu dwukrotnie mniejszy niż dla zbiorów tworzonych przez **generate_uniform_points** i **generate_zigzag_points**,
- algorytm **górnej i dolnej otoczki**, dzięki charakterystyce danych względnie rzadko musi usuwać punkty podczas budowania otoczki. Podobnie jak **dziel i rządź** działał około 2 razy szybciej niż dla generatora **generate_uniform_points**.

3.5.5. Wyniki dla generatora generate_x_square_points

Generator **generate_x_square_points** różni się od **generate_square_points** głównie przez to że, otoczka każdego wygenerowanego przez niego zbioru punktów ma dokładnie 4 punkty. Wyniki testów na tym generatorze przedstawiono w tabeli 5 oraz na wykresie widocznym na rysunku 38.

Liczba punktów	Graham [ms]	Jarvis [ms]	Przyrostowy [ms]	Górna i dolna [ms]	Dziel i rządź [ms]	Quickhull [ms]	Chan [ms]
20000	38.24	19.06	19.9	12.33	41.83	20.23	105.38
40000	77.58	38.48	40.69	23.96	101.14	42.01	214.73
60000	120.98	60.35	62.85	36.22	130.84	63.74	343.87
80000	168.15	87.75	88.43	50.92	195.57	88.32	443.88
100000	269.45	114.11	117.44	67.63	267.27	117.77	579.92

Tabela 5: czasy działania poszczególnych algorytmów na zbiorach punktów wygenerowanych przez **generate_x_square_points**



Rysunek 38: wykres na podstawie danych z tabeli 5

Co wynika z tabeli 5, algorytm **Jarvisa** cechuje się liniowością dla stałej liczby punktów otoczki - w porównaniu do generatora **generate_square_points** (gdzie otoczki mają 8 punktów) wypadł około 2 razy szybciej. Podobnie algorytm **Chana** osiągnął znacznie mniejszy czas. Zestawiając to z faktem, że liczba punktów w zbiorze nie zmieniła się dowodzi to, że teoretyczna złożoność algorytmu Chana $O(n \log k)$ znajduje odzwierciedlenie w rzeczywistości, a algorytm został poprawnie zaimplementowany. Podotoczki występujące w algorytmie **dziel i rządź** nie miały już takiego samego, korzystnego kształtu jak w przypadku generatora **generate_square_points** i odbiło się to znacząco na jego czasie działania pomimo takiej samej liczby punktów. Podobnie algorytm **górnej i dolnej otoczki** ponownie osiągnął czas pracy bliski temu dla generatora **generate_uniform_points**.

3.5.6. Wnioski

Różnice w asymptotycznej złożoności algorytmów Chana i Jarvisa względem pozostałych algorytmów zostały jasno nakreślone podczas wykonywania testów. Na podstawie podobnych kształtów wykresów czasu pracy pozostałych algorytmów (rys. 34-38) można wnioskować, że ich złożoność jest taka sama i wynosi $O(n \log n)$.

Sama bezwzględna wartość czasu działania dla wszystkich zbiorów testowych była największa dla algorytmu **Chana**, choć jego teoretyczna złożoność obliczeniowa została potwierdzona poprzez porównanie wyników testów na zbiorze 4 oraz 5.

Najszybsze na ogół okazały się algorytmy: **górnej i dolnej otoczki**, **przyrostowy** oraz **quickhull**. W szczególności algorytmy **Grahama** oraz **quickhull** wykazały się dużą niewrażliwością na charakterystykę danych wejściowych i dla podobnej liczby losowych punktów zawsze działały w podobym czasie.

4. Podsumowanie

Kod przygotowany na potrzeby wykonania ćwiczenia pozwolił na analizę i porównanie działania algorytmów wyznaczania otoczki wypukłej zbioru punktów na płaszczyźnie. Przygotowana funkcjonalność tworzenia wizualizacji w jasny i przejrzysty sposób przedstawia metodykę działania każdego z algorytmów, a konsekwencje wynikające z różnic między nimi ujawniły się podczas wykonywania testów opisanych w sekcji 3.5.

W szczególności potwierdzone zostały teoretycznie złożoności algorytmów **Jarvisa** i **Chana**, które odbiegają od złożoności pozostałych algorytmów.

Wyniki testów oraz wizualizacje potwierdzają poprawność zaimplementowanych algorytmów.

5. Bibliografia

- slajdy z wykładu,
- materiały do laboratoriów,
- https://en.wikipedia.org/wiki/Convex_hull_algorithms#Akl%E2%80%93Toussaint_heuristic (7.01.2026),
- https://en.wikipedia.org/wiki/Gift_wrapping_algorithm (7.01.2026),
- https://en.wikipedia.org/wiki/Graham_scan (7.01.2026),
- https://en.wikipedia.org/wiki/Quickhull (7.01.2026),
- https://gist.github.com/tixxit/252229 (7.01.2026),
- https://www.youtube.com/watch?v=O_K5_whYhoU (7.01.2026),
- https://www.youtube.com/watch?v=NH6WbP3lDac (7.01.2026).