

Otoczka wypukła dla zbioru punktów w przestrzeni dwuwymiarowej

Dokumentacja projektu
Algorytmy geometryczne

K. Kafara
Ł. Czarniecki

Spis treści

1	Informacje techniczne	4
1.1	Budowa programu	4
1.1.1	Moduł <i>lib</i>	4
1.1.2	Moduł <i>pure</i>	4
1.1.3	Moduł <i>vis</i>	5
1.2	Wymagania techniczne	5
1.3	Korzystanie z programu	5
1.3.1	Uruchomienie programu	5
2	Oznaczenia i definicje	5
3	Problem	6
4	Algorytmy	6
4.1	Algorytm Grahama	6
4.1.1	Opis działania	6
4.1.2	Szczegóły	7
4.1.3	Złożoność	7
4.1.4	Kod	7
4.2	Algorytm Jarvisa	8
4.2.1	Opis działania	8
4.2.2	Szczegóły	8
4.2.3	Złożoność	9
4.2.4	Kod	9
4.3	Algorytm górna-dolna	10
4.3.1	Opis działania	10
4.3.2	Złożoność	10
4.3.3	Kod	10

4.4	Algorytm przyrostowy	11
4.4.1	Opis działania	11
4.4.2	Szczegóły	12
4.4.3	Złożoność	12
4.4.4	Kod	12
4.5	Algorytm dziel i zwyciężaj	13
4.5.1	Opis działania	13
4.5.2	Szczegóły	14
4.5.3	Złożoność	14
4.5.4	Kod	15
4.6	Algorytm Chana	17
4.6.1	Opis działania	17
4.6.2	Szczegóły	18
4.6.3	Złożoność	18
4.6.4	Kod	19
4.7	Algorytm QuickHull	20
4.7.1	Opis działania	20
4.7.2	Szczegóły	21
4.7.3	Złożoność	21
4.7.4	Kod	21
5	Wydażność algorytmów	22
5.1	Algorytm Grahama	23
5.2	Algorytm górna-dolna	23
5.3	Algorytm Chana	25
5.4	Algorytm QuickHull	25
5.5	Algorytm dziel i zwyciężaj	25
5.6	Algorytm przyrostowy	29
5.7	Algorytm Jarvisa	29
6	Porównanie algorytmów	29
7	Bibliografia	29

Spis rysunków

1	Zbiór typu A, algorytm Grahama	23
2	Zbiór typu B, algorytm Grahama	23
3	Zbiór typu C, algorytm Grahama	24
4	Zbiór typu D, algorytm Grahama	24
5	Zbiór typu A, algorytm górna-dolna	24
6	Zbiór typu B, algorytm górna-dolna	24
7	Zbiór typu C, algorytm górna-dolna	25
8	Zbiór typu D, algorytm górna-dolna	25

9	Zbiór typu A, algorytm Chana	26
10	Zbiór typu B, algorytm Chana	26
11	Zbiór typu C, algorytm Chana	26
12	Zbiór typu D, algorytm Chana	26
13	Zbiór typu A, algorytm QuickHull	27
14	Zbiór typu B, algorytm QuickHull	27
15	Zbiór typu C, algorytm QuickHull	27
16	Zbiór typu D, algorytm QuickHull	27
17	Zbiór typu A, algorytm dziel i zwyciężaj	28
18	Zbiór typu B, algorytm dziel i zwyciężaj	28
19	Zbiór typu C, algorytm dziel i zwyciężaj	28
20	Zbiór typu D, algorytm dziel i zwyciężaj	28
21	Zbiór typu A, algorytm przyrostowy	29
22	Zbiór typu B, algorytm przyrostowy	29
23	Zbiór typu C, algorytm przyrostowy	30
24	Zbiór typu D, algorytm przyrostowy	30
25	Zbiór typu A, algorytm Jarvisa	30
26	Zbiór typu B, algorytm Jarvisa	30

Spis tablic

1	Czas wykonania algorytmu Grahama w zależności od typu zbioru testowego oraz mocy zbioru punktów.	23
2	Czas wykonania algorytmu górna-dolna w zależności od typu zbioru testowego oraz mocy zbioru punktów.	24
3	Czas wykonania algorytmu Chana w zależności od typu zbioru testowego oraz mocy zbioru punktów.	25
4	Czas wykonania algorytmu QuickHull w zależności od typu zbioru testowego oraz mocy zbioru punktów.	26
5	Czas wykonania algorytmu dziel i zwyciężaj w zależności od typu zbioru testowego oraz mocy zbioru punktów.	27
6	Czas wykonania algorytmu przyrostowego w zależności od typu zbioru testowego oraz mocy zbioru punktów.	29
7	Czas wykonania algorytmu Jarvisa w zależności od typu zbioru testowego oraz mocy zbioru punktów.	30

1 Informacje techniczne

1.1 Budowa programu

Program złożony jest z następujących modułów:

- *lib* – biblioteczny – zawiera zbiór pomocniczych funkcji i struktur danych wykorzystywanych przez algorytmy.
- *pure* – algorytmy w *czystej postaci* tj. nie posiadające części wizualizacyjnej.
- *vis* – algorytmy wraz z kodem odpowiadającym za wizualizację

Poniżej przedstawiamy dokładny opis zawartości poszczególnych modułów.

1.1.1 Moduł *lib*

Moduł zawiera w sobie następujące podmoduły:

1. *geometric_tool_lab.py* – narzędzie graficzne dostarczone w ramach przedmiotu *Algorytmy geometryczne*
2. *getrand.py* – zawiera funkcje generujące zbiory punktów różnych typów
3. *sorting.py* – zawiera implementację iteracyjnej wersji algorytmu *QuickSort* wykorzystywaną m.in w algorytmie Grahama
4. *stack.py* – zawiera klasę implementującą *stos*
5. *util.py* – zawiera szereg funkcji pomocniczych wykorzystywanych przez zaimplementowane algorytmy
6. *mytypes.py* – zawiera definicje typów stworzone w celu zwiększenia czytelności kodu

1.1.2 Moduł *pure*

Moduł zawiera w sobie następujące podmoduły:

1. *divide_conq.py* – implementacja algorytmu dziel i zwyciężaj
2. *graham.py* – implementacja algorytmu Grahama
3. *increase.py* – implementacja algorytmu przyrostowego
4. *jarvis.py* – implementacja algorytmu Jarvisa
5. *lowerupper.py* – implementacja algorytmu "górną-dolną"

1.1.3 Moduł *vis*

Moduł zawiera w sobie następujące podmoduły:

1. *divide_conq_vis.py* – implementacja algorytmu dziel i zwyciężaj wraz z kodem tworzącym wizualizację
2. *graham_vis.py* – implementacja algorytmu Grahama wraz z kodem tworzącym wizualizację
3. *increase_vis.py* – implementacja algorytmu przyrostowego wraz z kodem tworzącym wizualizację
4. *jarvis_vis.py* – implementacja algorytmu Jarvisa wraz z kodem tworzącym wizualizację
5. *lowerupper_vis.py* – implementacja algorytmu "górna-dolna" wraz z kodem tworzącym wizualizację

1.2 Wymagania techniczne

1. Python 3.8.3 64-bit lub nowszy wraz z modułami:

- *matplotlib*
- *numpy*
- *json*
- *csv*
- *pprint*

2. Jupyter Notebook

1.3 Korzystanie z programu

1.3.1 Uruchomienie programu

W celu uruchomienia wizualizacji algorytmów należy uruchomić notebook (poprzez Jupyter Notebook) *program.ipynb*, oraz wykonywać kolejne komórki notatnika.

W celu uruchomienia pomiarów wydajności algorytmów należy przejść do sekcji *Pomiary czasu* wykonać pierwszą komórkę (z importami algorytmów) a następnie przeprowadzać testy i wyświetlać rezultaty w interesujących nas przypadkach.

2 Oznaczenia i definicje

Na potrzeby dalszych wywodów przyjmujemy w tym miejscu szereg oznaczeń i definicji:

*def. 1. **Zbiorem wypukłym*** nazwiemy dowolny podzbiór płaszczyzny taki, że dla każdych dwóch punktów do niego należących, odcinek je łączący również należy do tego zbioru.

*def. 2. **Otoczką wypukłą*** dowolnego zbioru punktów S płaszczyzny nazwiemy najmniejszy zbiór wypukły $CH(S)$ zawierający S .

Algorytmicznie otoczkę wypukłą dowolnego zbioru S punktów płaszczyzny reprezentujemy jako ciąg punktów (wierzchołków) $< v_1, v_2, \dots, v_n >$ wielokąta wypukłego, gdzie $\forall i \in \{1, \dots, n\}$ v_i jest poprzednikiem v_{i+1} (w kolejności wierzchołków przeciwnej do ruchu wskazówek zegara).

3 Problem

Wyznaczyć otoczkę wypukłą podanego zbioru punktów płaszczyzny dwuwymiarowej.

4 Algorytmy

4.1 Algorytm Grahama

W celu opisanego sposobu działania algorytmu Grahama, definiujemy następującą relację \preceq_Q określoną dla dowolnych dwóch punktów płaszczyzny P_1, P_2 względem wybranego i ustalonego punktu odniesienia Q .

$$P_1 \preceq_Q P_2 \Leftrightarrow (\angle(P_1, Q, OX) < \angle(P_2, Q, OX)) \vee (\angle(P_1, Q, OX) = \angle(P_2, Q, OX) \wedge d(P_1, Q) \leq d(P_2, Q))$$

gdzie $d(P, Q)$ oznacza odległość od siebie dwóch dowolnych punktów płaszczyzny.

Tak zdefiniowana relacja jest liniowym porządkiem (zwrotna, antysymetryczna, przechodnia i spójna).

4.1.1 Opis działania

1. Wyznaczamy najniższy punkt Q wyjściowego zbioru (jeżeli jest wiele o tej samej rzędnej – bierzemy ten o najmniejszej odciętej).
2. Ustawiamy go jako pierwszy element zbioru.
3. Sortujemy pozostałe punkty względem relacji \preceq_Q .
4. Usuwanie wszystkie, poza najbardziej oddalonym od Q , punkty leżące na półprostej QP , dla każdego P
5. Kładziemy pierwsze 3 punkty zbioru na stos S .
6. Iterujemy kolejno po punktach z posortowanego zbioru nie będących na stosie: Niech bieżącym punktem będzie P :

- (a) Dopóki P nie jest po lewej stronie $S_{n-1}S_n$ wykonujemy (b)
- (b) Uswamy punkt ze stosu.
- (c) Dodajemy P na stos.

7. Zwracamy zawartość stosu.

4.1.2 Szczegóły

- Najniższy punkt wyjściowego zbioru (punkt 1) wyznaczamy w czasie liniowym, iterując po kolejnych punktach zbioru.
- Wszystkie punkty leżące na jednej prostej, poza najbardziej oddalonym od Q usuwamy w czasie liniowym w następujący sposób: Iterując przez posortowaną tablicę, zaczynając od indeksu $i := 1$, zapamiętujemy ostatni indeks na który wstawialiśmy j (na początku $j := 1$). Jeżeli Q, P_i, P_{i+1} są współliniowe to $i := i + 1$. Jeżeli nie są współliniowe to P_i wpisujemy na pozycję j , a następnie $j := j + 1$. Następnie, w dalszej części algorytmu posługujemy się częścią tablicy $[0, \dots, j - 1]$.

4.1.3 Złożoność

Operacją dominującą w algorytmie jest sortowanie – realizowane w czasie $O(n \lg n)$. Wybór punktu najniższego, redukcja punktów współliniowych oraz iterowanie (punkt 6, zauważmy, że każdy punkt zbioru wyjściowego jest obsługiwany co najwyżej 2 razy – gdy jest dodawany do otoczki i gdy jest ewentualnie usuwany) są realizowane w czasie $O(n)$. Algorytm Grahama ma zatem złożoność $O(n \lg n)$.

4.1.4 Kod

```

1 def get_point_cmp(ref_point: Point, eps: float = 1e-7) -> Callable:
2     def point_cmp(point1, point2):
3         orient = orientation(ref_point, point1, point2, eps)
4
5         if orient == -1:
6             return False
7         elif orient == 1:
8             return True
9         elif dist_sq(ref_point, point1) <= dist_sq(ref_point, point2):
10            return True
11        else:
12            return False
13
14    return point_cmp
15
16
17 def graham(points: ListOfPoints) -> ListOfPoints:
18     istart = index_of_min(points, 1)
19
20     points[istart], points[0] = points[0], points[istart]
21

```

```

22     qsort_iterative(points, get_point_cmp(points[0]))
23
24     i, new_size = 1, 1
25     while i < len(points):
26         while (i < len(points) - 1) \
27             and \
28                 (orientation(points[0], points[i], points[i + 1], 1e-7) == 0):
29             i += 1
30
31         points[new_size] = points[i]
32         new_size += 1
33         i += 1
34
35     s = Stack()
36     s.push(points[0])
37     s.push(points[1])
38     s.push(points[2])
39
40     for i in range(3, new_size, 1):
41         while orientation(s.sec(), s.top(), points[i], 1e-7) != 1:
42             s.pop()
43
44         s.push(points[i])
45
46     return s.s[:s.itop+1]
47

```

4.2 Algorytm Jarvisa

4.2.1 Opis działania

1. Wyznaczamy najniższy punkt Q wyjściowego zbioru (jeżeli jest wiele o tej samej rzędnej – bierzemy ten o najmniejszej odciętej).
2. Dodajemy Q do zbioru punktów otoczki.
3. Przeglądamy punkty zbioru w poszukiwaniu takiego, który wraz z ostatnim punktem otoczki tworzy najmniejszy kąt skierowany względem ostatniej znanej krawędzi otoczki. Dla pierwszego szukanego punktu, kąt namierzamy względem poziomu.
4. Znalezione punkty dodajemy do zbioru punktów otoczki, jeżeli jest różny od Q .
5. Powtarzamy punkty 3 i 4 tak długo aż znalezionym punktem nie będzie Q .
6. Zwracamy listę punktów otoczki.

4.2.2 Szczegóły

- Najniższy punkt wyjściowego zbioru (punkt 1) wyznaczamy w czasie liniowym, iterując po kolejnych punktach zbioru.
- W celu wyznaczenia punktu wyspecyfikowanego w punkcie 3. nie obliczamy wartości odpowiedniego kąta. Zamiast tego, równoważnie, wyznaczamy punkt P , który

wraz z ostatnim znanym punktem otoczki P_0 tworzy wektor $\vec{P_0P}$ dla którego wszystkie pozostałe punkty zbioru są po lewej stronie. Robimy to w czasie liniowym korzystając z znanych własności wyznacznika.

4.2.3 Złożoność

Zauważmy, że jeżeli otoczka jest k - elementowa, to główna pętla algorytmu (punkty 3–4) wykonuje się k -razy. Każdy krok pętli (znalezienie odpowiedniego punktu P) zajmuje czas liniowy. Pozostałe operacje w algorytmie zajmują co najwyżej czas liniowy. Zatem algorytm Jarvisa ma złożoność $O(nk)$.

4.2.4 Kod

```

1 def jarvis(points: ListOfPoints) -> ListOfPoints:
2     EPS = 1e-8
3
4     convex_hull = []
5
6     start_idx = index_of_min(points, 1)
7
8     convex_hull.append(start_idx)
9
10    rand_idx = 0 if start_idx != 0 else 1
11
12    prev = start_idx
13
14    while True:
15        imax = rand_idx
16
17        for i in range(len(points)):
18            if i != prev and i != imax:
19                orient = orientation(
20                    points[prev],
21                    points[imax],
22                    points[i],
23                    EPS
24                )
25                if orient == -1:
26                    imax = i
27
28                elif orient == 0 and \
29                    (dist_sq(points[prev], points[imax]) < dist_sq(points[
prev], points[i])):
30                    imax = i
31
32            if imax == start_idx:
33                break;
34
35        convex_hull.append(imax)
36
37        prev = imax
38

```

```
39 return points[convex_hull]
40
```

W ostatniej linii algorytmu, korzystamy z możliwości biblioteki *numpy*.

4.3 Algorytm górna-dolna

4.3.1 Opis działania

1. Sortujemy punkty rosnąco po odciętych (w przypadku równych, mniejszy jest punkt o mniejszej rzędnej).
2. Pierwsze dwa punkty z posortowanego zbioru wpisujemy do zbioru punktów otoczki górnej oraz dolnej.
3. Iterujemy po zbiorze punktów zaczynając od $i = 2$ (trzeciego punktu), niech P będzie bieżącym punktem:
 - (a) Dopóki górna (dolna) otoczka ma co najmniej 2 punkty i P nie znajduje się po prawej (lewej) stronie odcinka skierowanego utworzonego przez ostatnie dwa punkty otoczki (ostatni jest końcem odcinka), wykonujemy (b):
 - (b) Usuwamy ostatni punkt z otoczki górnej (dolnej).
 - (c) Dodajemy P do punktów otoczki górnej (dolnej).
4. Odwracamy kolejność wierzchołków w otoczce dolnej.
5. Łączymy zbioru punktów otoczki górnej oraz dolnej.
6. Zwracamy złączony zbiór punktów otoczki.

4.3.2 Złożoność

Dominującą operacją w algorytmie jest sortowanie realizowane w czasie $O(n \lg n)$. Każdy krok pętli (dla wyznaczania otoczki górnej oraz dolnej) zajmuje czas stały. Zauważmy, że podobnie do algorytmu Grahama każdy z punktów jest rozważany co najwyżej dwukrotnie – w momencie dodania do otoczki i przy ewentualnym usunięciu ze zbioru punktów otoczki. Pozostałe operacje realizowane są w czasie liniowym. Zatem algorytm "górna-dolna" ma złożoność $O(n \lg n)$.

4.3.3 Kod

```
1 def lower_upper(point2_set: ListOfPoints) -> ListOfPoints:
2     if len(point2_set) < 3: return None
3
4     point2_set.sort(key = operator.itemgetter(0, 1))
5
6     upper_ch = [ point2_set[0], point2_set[1] ]
7     lower_ch = [ point2_set[0], point2_set[1] ]
8
```

```

9 for i in range( 2, len(point2_set) ):
10     while len(upper_ch) > 1 and orientation(upper_ch[-2], upper_ch[-1],
11         point2_set[i]) != -1:
12         upper_ch.pop()
13
14     upper_ch.append(point2_set[i])
15
16 for i in range(2, len(point2_set) ):
17     while len(lower_ch) > 1 and orientation(lower_ch[-2], lower_ch[-1],
18         point2_set[i]) != 1:
19         lower_ch.pop()
20
21     lower_ch.append(point2_set[i])
22
23 lower_ch.reverse()
24 upper_ch.extend(lower_ch)
25
26 return upper_ch

```

4.4 Algorytm przyrostowy

4.4.1 Opis działania

Ogólne sformułowanie algorytmu ma postać:

1. Dodajemy pierwsze 3 punkty do zbioru punktów otoczki.
2. Iterujemy po pozostałych punktach. Niech P będzie punktem bieżącym:
 - (a) Jeżeli P nie należy do wnętrza obecnie znanej otoczki wykonujemy (b) oraz (c).
 - (b) Znajdujemy styczne do obecnie znanej otoczki poprowadzone przez punkt P .
 - (c) Aktualizujemy otoczkę.
3. Zwracamy punkty otoczki.

Możemy go jednak sformułować inaczej, co pozwoli na uproszczenie implementacji, przy zachowaniu takiego samego rzędu złożoności.

1. Sortujemy punkty rosnąco po odciętych (w przypadku równych, mniejszy jest punkt o mniejszej rzędnej).
2. Dodajemy pierwsze 3 punkty do zbioru punktów otoczki, w takiej kolejności, aby były podane w kolejności odwrotnej do ruchu wskazówek zegara.
3. Iterujemy po pozostałych punktach. Niech P będzie punktem bieżącym:
 - (a) Znajdujemy styczne do obecnie znanej otoczki poprowadzone przez punkt P .
 - (b) Aktualizujemy otoczkę.
4. Zwracamy punkty otoczki.

Dzięki wstępnemu posortowaniu punktów, omijamy konieczność testowania należenia P do otoczki znanej w danym kroku algorytmu, ponieważ biorąc kolejny punkt mamy gwarancję, że nie należy on do wcześniej znanej otoczki.

4.4.2 Szczegóły

Wyznaczanie stycznych

4.4.3 Złożoność

Posortowanie punktów zajmuje $O(n \lg n)$. Wydaje główna pętla programu wykonuje się w czasie $O(n)$, ponieważ możemy usunąć maksymalnie $k - 3$ punkty (gdzie k jest liczebnością zbioru punktów otoczki znaną w danej iteracji), ale zauważmy, że każdy z punktów usuwany jest co najwyżej raz. Wyszukanie stycznych w głównej pętli także zajmuje czas liniowy, więc główna pętla programu wykonuje się w czasie liniowym. Zatem złożoność algorytmu jest rzędu $O(n \lg n)$

4.4.4 Kod

```
1 def rltangent(polygon: ListOfPoints, point: Point):
2     n = len(polygon)
3
4     right = index_of_max(polygon, cmp_idx=0)
5
6     left = right
7
8     left_orient = orientation(point, polygon[left % n], polygon[(left-1)%
9     n])
10    while left_orient != -1:
11        if left_orient == 0 and dist_sq(point, polygon[left]) >= dist_sq(
12        point, polygon[(left-1)%n]):
13            break
14            left = (left-1) % n
15            left_orient = orientation(point, polygon[left % n], polygon[(left
16            -1)%n])
17
18    right_orient = orientation(point, polygon[right%n], polygon[(right+1)
19    %n])
20    while right_orient != 1:
21        if right_orient == 0 and dist_sq(point, polygon[right]) >=
22        dist_sq(point, polygon[(right+1)%n]):
23            break
24            right = (right+1)%n
25            right_orient = orientation(point, polygon[right%n], polygon[(
26            right+1)%n])
27
28    return left, right
```

```

26 def increase_with_sorting(point2_set: ListOfPoints) -> Union[ListOfPoints
    , None]:
27     if len( point2_set ) < 3: return None
28
29     point2_set.sort(key = operator.itemgetter(0, 1))
30
31     convex_hull = point2_set[:3]
32
33     if orientation(convex_hull[0], convex_hull[1], convex_hull[2]) == -1:
34         convex_hull[1], convex_hull[2] = convex_hull[2], convex_hull[1]
35
36     for i in range(3, len( point2_set )):
37         rltang = rltangent(convex_hull, point2_set[i])
38         left_tangent_idx = rltang[0]
39         right_tangent_idx = rltang[1]
40
41         left_tangent_point = convex_hull[left_tangent_idx]
42         right_tangent_point = convex_hull[right_tangent_idx]
43
44         deletion_side: Literal[-1, 0, 1] = orientation(left_tangent_point
    , right_tangent_point, point2_set[i])
45
46         if deletion_side != 0:
47             lrlnext_orient = orientation(left_tangent_point,
    right_tangent_point, convex_hull[(left_tangent_idx + 1) % len(
    convex_hull)])
48             if lrlnext_orient == deletion_side or lrlnext_orient == 0:
49                 step = 0
50             else:
51                 step = -1
52
53             left = (left_tangent_idx + 1) % len(convex_hull)
54
55             while convex_hull[left % len(convex_hull)] !=
    right_tangent_point:
56                 convex_hull.pop(left % len(convex_hull))
57                 left = (left + step) % len(convex_hull)
58
59             convex_hull.insert(left % len(convex_hull), point2_set[i])
60         else:
61             convex_hull = [point2_set[left_tangent_idx], point2_set[i]]
62     return convex_hull

```

4.5 Algorytm dziel i zwyciężaj

4.5.1 Opis działania

Prócz zbioru punktów, dodatkową daną wejściową dla algorytmu jest stała k oznaczająca liczebność zbioru punktów, przy której przechodzimy w algorytmie rekurencyjnym do przypadku bazowego – wyznaczamy otoczkę innym, wybranym algorytmem.

Opisany algorytm jest algorytmem rekurencyjnym. Przed pierwszym wywołaniem rekurencyjnym należy zbiór punktów posortować rosnąco po odciętych (w przypadku równych, mniejszy jest punkty o mniejszej rzędnej).

Jest to standardowe zastosowanie metody "dziel i zwyciężaj":

1. Dzielimy wyjściowy problem na mniejsze tak długo, aż znajdujemy się w przypadku którymi potrafimy rozwiązać elementarnie / w inny sposób.
2. Łączymy kolejne rozwiązania częściowe w całość.

Popatrzmy na schemat działania:

1. Jeżeli liczebność rozważanego zbioru jest mniejsza bądź równa danej stałej k , to:
 - (a) Wyznaczamy otoczkę rozważanego zbioru punktów, za pomocą innej metody (np. innego algorytmu wyznaczania otoczki).
 - (b) Zwracamy tak uzyskaną otoczkę.
2. W przeciwnym przypadku:
 - (a) Wywołujemy się rekurencyjnie na zbiorze punktów o odciętych mniejszych od mediany.
 - (b) Wywołujemy się rekurencyjnie na zbiorze punktów o odciętych większych bądź równych medianie.
 - (c) Łączymy lewą i prawą otoczkę (pozyskane z wywołań rekurencyjnych) w jedną.
 - (d) Zwracamy tak uzyskaną otoczkę.

4.5.2 Szczegóły

- Do wyznaczania otoczki w przypadku podstawowym wykorzystany został algorytm Jarvisa, ponieważ dla $k \ll n$ ma on złożoność właściwie liniową.
- Sposób łączenia otoczek jest następujący:
 1. Wyznaczamy skrajny prawy punkt L lewej otoczki oraz skrajny lewy P punkt prawej otoczki.
 2. Dopóki L i P nie tworzą górnej stycznej, "wychodzimy w górę" naprzemiennie punktami L i P .
 3. Analogicznie wyznaczamy dolną styczną.
 4. Usuwamy punkty zawierające się we wnętrzu nowo utworzonej otoczki.

4.5.3 Złożoność

- Początkowe sortowanie: $O(n \lg n)$
- Rekurencja: $T(n) = 2T(\frac{n}{2}) + O(n) \implies T(n) = O(n \lg n)$
- Każde łączenie otoczek: $O(n)$

Algorytm dziel i zwyciężaj ma zatem złożoność $O(n \lg n)$

4.5.4 Kod

```
1 def merge_convex_hulls(left_convex_hull: ListOfPoints, right_convex_hull:
   ListOfPoints) -> List[Point]:
2     left_ch_size = len(left_convex_hull)
3     right_ch_size = len(right_convex_hull)
4
5     # znajdujemy prawy skrajny punkt lewej otoczki
6     left_ch_rightmost_idx = index_of_max(left_convex_hull, cmp_idx=0)
7     right_ch_leftmost_idx = index_of_min(right_convex_hull, cmp_idx=0)
8
9     left = left_convex_hull[left_ch_rightmost_idx]
10    right = right_convex_hull[right_ch_leftmost_idx]
11    left_idx = left_ch_rightmost_idx
12    right_idx = right_ch_leftmost_idx
13
14    left_flag, right_flag = True, True
15    while orientation(left, right, right_convex_hull[(right_idx - 1) %
   right_ch_size]) != -1 and right_flag\
16        or \
17        orientation(right, left, left_convex_hull[(left_idx + 1) %
   left_ch_size]) != 1 and left_flag:
18
19        left_flag, right_flag = False, False
20
21    # podnosimy punkt na prawej otoczce
22    left_right_orient = orientation(left, right, right_convex_hull[(
   right_idx - 1) % right_ch_size])
23    while left_right_orient != -1:
24        if left_right_orient == 0 and dist_sq(left, right) >= dist_sq
   (left, right_convex_hull[(right_idx - 1) % right_ch_size]):
25            right_flag = False
26            break
27
28        right_idx = (right_idx - 1) % right_ch_size
29        right = right_convex_hull[right_idx]
30        left_right_orient = orientation(left, right,
   right_convex_hull[(right_idx - 1) % right_ch_size])
31    else:
32        right_flag = True
33
34    # podnosimy punkt na lewej otoczce
35    right_left_orient = orientation(right, left, left_convex_hull[(
   left_idx + 1) % left_ch_size])
36    while right_left_orient != 1:
37        if right_left_orient == 0 and dist_sq(right, left) >= dist_sq
   (right, left_convex_hull[(left_idx + 1) % left_ch_size]):
38            left_flag = False
39            break
40
41        left_idx = (left_idx + 1) % left_ch_size
42        left = left_convex_hull[left_idx]
43        right_left_orient = orientation(right, left, left_convex_hull
   [(left_idx + 1) % left_ch_size])
```

```

44         else:
45             left_flag = True
46
47
48         upper_tangent_left_idx = left_idx
49         upper_tangent_right_idx = right_idx
50
51         # dolna styczna
52         left = left_convex_hull[left_ch_rightmost_idx]
53         right = right_convex_hull[right_ch_leftmost_idx]
54         left_idx = left_ch_rightmost_idx
55         right_idx = right_ch_leftmost_idx
56
57         left_flag, right_flag = True, True
58         while orientation(left, right, right_convex_hull[(right_idx + 1) %
59             right_ch_size]) != 1 and right_flag \
60             or \
61             orientation(right, left, left_convex_hull[(left_idx - 1) %
62             left_ch_size]) != -1 and left_flag:
63
64             left_flag, right_flag = False, False
65
66             # opuszczamy punkt na prawej otoczce
67             left_right_orient = orientation(left, right, right_convex_hull[(
68             right_idx + 1) % right_ch_size])
69             while left_right_orient != 1:
70                 if left_right_orient == 0 and dist_sq(left, right) >= dist_sq
71                 (left, right_convex_hull[(right_idx + 1) % right_ch_size]):
72                     right_flag = False
73                     break
74
75                 right_idx = (right_idx + 1) % right_ch_size
76                 right = right_convex_hull[right_idx]
77                 left_right_orient = orientation(left, right,
78                 right_convex_hull[(right_idx + 1) % right_ch_size])
79             else:
80                 right_flag = True
81
82             # opuszczamy punkt na lewej otoczce
83             right_left_orient = orientation(right, left, left_convex_hull[(
84             left_idx - 1) % left_ch_size])
85             while right_left_orient != -1:
86                 if right_left_orient == 0 and dist_sq(right, left) >= dist_sq
87                 (right, left_convex_hull[(left_idx - 1) % left_ch_size]):
88                     left_flag = False
89                     break
90
91                 left_idx = (left_idx - 1) % left_ch_size
92                 left = left_convex_hull[left_idx]
93                 right_left_orient = orientation(right, left, left_convex_hull
94                 [(left_idx - 1) % left_ch_size])
95             else:
96                 left_flag = True

```



```

90     lower_tangent_left_idx = left_idx
91     lower_tangent_right_idx = right_idx
92
93     merged_convex_hull = [ ]
94
95     while upper_tangent_left_idx != lower_tangent_left_idx:
96         merged_convex_hull.append(left_convex_hull[upper_tangent_left_idx
97 ])
98         upper_tangent_left_idx = (upper_tangent_left_idx + 1) %
99 left_ch_size
100     else:
101         merged_convex_hull.append(left_convex_hull[lower_tangent_left_idx
102 ])
103
104     while lower_tangent_right_idx != upper_tangent_right_idx:
105         merged_convex_hull.append(right_convex_hull[
106 lower_tangent_right_idx])
107         lower_tangent_right_idx = (lower_tangent_right_idx + 1) %
108 right_ch_size
109     else:
110         merged_convex_hull.append(right_convex_hull[
111 lower_tangent_right_idx])
112
113     return merged_convex_hull
114
115 def divide_conq(point2_set: List[Point], k: int = 2) -> Union[List[Point
116 ], None]:
117     if len(point2_set) < 3 or k <= 0: return None
118
119     def divide_conq_rec(point2_set: List[Point]) -> List[Point]:
120         if len(point2_set) <= 2: return point2_set
121         elif len(point2_set) <= k: return jarvis(np.array(point2_set))
122
123         left_convex_hull = divide_conq_rec(point2_set[ : len(point2_set)
124 // 2])
125         right_convex_hull = divide_conq_rec(point2_set[len(point2_set) //
126 2 : ])
127
128         return merge_convex_hulls(left_convex_hull, right_convex_hull)
129
130     point2_set.sort(key = operator.itemgetter(0, 1))
131
132     return divide_conq_rec(point2_set)

```

4.6 Algorytm Chana

4.6.1 Opis działania

Główna część algorytmu Chana składa się z dwóch części:

1. Pierwsza, która składa się na :

- Podział zbioru punktów Q na podzbiory Q_i o w miarę równych ilościach punktów w nich zawartych, z czego żaden nie zawiera więcej niż dane m .
 - Wyznaczenie otoczek C_i
2. Druga polega na wykonaniu algorytmu na wzór Jarvisa, tylko na otoczkach. Dokładniej mówiąc:
- Startujemy z najniższy wierzchołkiem z całego zbioru Q i dodajemy go do finalnej otoczki jako pierwszy wierzchołek.
 - Dla każdego punktu należącego do otoczki, możemy znaleźć jego następnego sąsiada w otoczce idąc w kolejności przeciwnej do ruchu wskazówek zegara. Aby to zrobić należy wybrać spośród zbioru punktów utworzonego z: punktów tworzących prawą styczną z otoczkami C_i dla rozważanego wierzchołka, kolejnego punktu podotoczki do której dany punkt należy taki wierzchołek, że wszystkie inne wierzchołki z tego zbioru są na lewo od niego.
 - W ten sposób wyznaczamy kolejne wierzchołki otoczki, dopóki następnym wierzchołkiem otoczki nie jest jej pierwszy punkt. Wtedy otoczka jest pełna i kończymy algorytm.
3. Zwracamy punkty otoczki.

Jednakże nadzędną istotą powyższego algorytmu jest to, że wykona się on w drugiej części w co najwyżej m krokach (dany rozmiar podzbioru). Inaczej mówiąc m musi być większe bądź równe (w idealnym przypadku) liczbie punktów należących do otoczki k . Jeśli wykonamy m kroków i wciąż nie mamy otoczki, to przerywamy algorytm. I próbujemy z większym m . Aby nie popsuć złożoności dużą ilością powtórzeń głównej części algorytmu najlepiej za każdym razem parametr m podnosić do kwadratu. W przypadku gdy $m_k = n$ po prostu za m przyjmujemy n . Wtedy algorytm chana sprowadza się do algorytmu Grahama.

4.6.2 Szczegóły

4.6.3 Złożoność

Złożoność głównej części algorytmu.

- Złożoność pierwszej części składa się na :
 - Podział zbioru punktów na podzbiory $O(n)$;
 - Wyznaczenie otoczek dla podzbiorów. Mamy $\lceil \frac{n}{m} \rceil$ podzbiorów rozmiaru m , dla każdego z nich wyznaczamy otoczkę algorytmem Grahama. Algorytm Grahama działa $O(n \log(n))$. Więc łącznie mamy $O(\text{ceil}(n/m) * m \log(m)) = O(n \log(m))$.

Łącznie dla pierwszej części mamy $O(n \log(m))$, gdzie m jest wybranym maksymalnym rozmiarem podzbiorów.

- Złożoność drugiej części składa się na :
 - Wyznaczenie następnego punktu dla każdego punktu z otoczki głównej o rozmiarze k
 - Wyznaczenie następnego punktu składa się na wyznaczenie dla każdej z m podotoczek stycznej do tej podotoczki. Styczną wyznaczamy binary searchem w czasie $O(\log(m))$ (otoczka C_i ma co najwyżej m wierzchołków). Otoczek jest $\lceil (n/m) \rceil$, a zatem czas wyznaczenia kolejnego wierzchołka otoczki to $O(\lceil (n/m) \rceil \log(m))$.

Zakładając, że liczba wierzchołków otoczki $k \leq m$ (gdy $k > m$ przerywamy algorytm, więc złożoność pozostaje ta sama), to ostatecznie mamy złożoność dla drugiej części rzędu : $O(k \lceil (n/m) \rceil \log(m)) = O(n \log(m))$ w idealnym przypadku $O(n \log(k))$

Cała złożoność głównej części algorytmu, to $O(n \log(m))$, gdzie m jest wybraną wielkością podzbioru. Złożoność algorytmu dla próbowania algorytmu z kolejnymi m postaciami 2^{2^m} dla $m \geq 1$, to: w takim razie złożoność można opisać wzorem $\sum_{t=1}^{\lceil \log \log k \rceil} O(n \log(2^{2^t})) = O(n * 2^{1+\lceil \log \log k \rceil}) = O(n \log k)$

4.6.4 Kod

```

1 from divide import *
2 from det import *
3 from tangentBothsides import *
4
5 def compr(p, q, current,
6         accur=10 ** (-6)): # jeżeli p jest po prawej odcinka [current
7     ,q] - jest 'większy', to zwracamy 1
8     if det(current, p, q) > accur:
9         return -1
10    elif det(current, p, q) < accur:
11        return 1
12    else:
13        return 0
14
15 def nextvert(C, curr): # dla danego punktu współzedydami z Q[i][j] jeśli
16     jest to punkt należący do finalnej otoczki, to
17     # zwraca następny punkt należący do finalnej otoczki zadanego w
18     takich samych współzedydnych Q[nxt[0]][nxt[1]]
19     i, j = curr
20     nxt = (i, (j + 1) % len(C[i]))
21     for k in range(len(C)):
22         t = tangent(C[i][j], C[k])
23         if t != None and k != i and compr(C[nxt[0]][nxt[1]], C[k][t], C[i]
24         ][j]) > 0 and (k, t) != (curr):
25             nxt = (k, t)
26
27     return nxt

```

```

25 def chanUtil(points, m):
26     Q = divide(points, m)
27     C = []
28     for i in range(len(Q)):
29         C.append(Graham(Q[i]))
30
31     curr = (0, 0)
32     ans = []
33     i = 0
34     while i < m:
35         ans.append(C[curr[0]][curr[1]])
36         if nextvert(C, curr) == (0, 0):
37             return ans
38         curr = nextvert(C, curr)
39         i += 1
40
41     return None
42
43 def chan(points):
44     n = len(points)
45     m = 4
46     hoax = None
47     while hoax == None:
48         hoax = chanUtil(points, m)
49         m = min(n, m * m)
50
51     return hoax

```

4.7 Algorytm QuickHull

4.7.1 Opis działania

Algorytm QuickHull polega na rekurencyjnym wyznaczaniu kolejnych punktów otoczki.

1. Algorytm rozpoczynamy od wyznaczenie dwóch punktów skrajnych a, b - tj. o najmniejszej i największej współrzędnej x -owej.
2. Następnie uruchamiamy funkcję rekurencyjnego znajdowania łuku należącego do otoczki między danymi punktami należącymi do tej otoczki p, q na prawo od odcinka $\overline{p, q}$. Otoczką jest suma punktów a , wyniku działania funkcji rekurencyjnej dla odcinka $\overline{a, b}$, b oraz wyniku działania funkcji rekurencyjnej dla $\overline{b, a}$.
3. Funkcja rekurencyjnego wyznaczenia łuku należącego do otoczki między punktami p i q polega na :
 - Wyznaczeniu najbardziej oddalonego punktu na prawo od $\overline{p, q}$ jeśli są punkty po prawej.
 - Jeśli nie ma takich punktów, to takiego łuku nie ma i zwracamy pustą tablicę.
 - W przeciwnym przypadku p, q należą do otoczki, to wyznaczony punkt skrajny r musi należeć do otoczki.
 - Skoro p, k, r należą do otoczki, to wszystkie wierzchołki wewnątrz trójkąta pkr napewno do najmniej nie należą - usuwamy je.

- Szukany łuk, to suma działania tej samej funkcji dla punktów p,r, punktu r , oraz wyniku tej funkcji dla punktów r,q w zadanej kolejności.
- Na koniec zwracamy wyznaczony w ten sposób łuk.

4.7.2 Szczegóły

- Rozpatrywane punkty p,q,r zawsze są podane w kolejności przeciwnej do ruchu wskazówek zegara. Aby usunąć punkty wewnątrz takich trójkątów należy dla każdego punktu z rozważanych sprawdzić, należy do danego trójkąta.
- Sprawdzenie, czy dany punkt należy do trójkąta pqr wykonujemy poprzez sprawdzenie, czy dla każdego z odcinków pr, rq, qp dany punkt znajduje się na lewo od tego odcinka, bądź jest z nim współliniowy.
- Porównywanie odległości punktów r znajdujących się na prawo od odcinka pq wykonujemy za pomocą wyznacznika. Jest on wprostproporcjonalny do pola trójkąta rozpiętego na wektorach pq,pr. Ponieważ odcinek pq ma stałą długość dla każdego r, to wyznacznik ten jest wprostproporcjonalny do wysokości tego trójkąta opuszczonej na bok pq - odległości punktów.

4.7.3 Złożoność

Pesymistyczna złożoność algorytmu to $O(n^2)$ - gdy wszystkie punkty zbioru znajdują się w otocze. Jendakże w średnim przypadku złożoność wynosi $O(n \log n)$

4.7.4 Kod

```

1  from copy import deepcopy
2  from lib.det import *
3
4  def furthest(a, b, considering):
5      n = len(considering)
6      i = 0
7      ans = None
8      while i < n:
9          if det(a, b, considering[i]) < 0: # rozważany wierzcholek
10             jest po prawej stronie ab
11             if ans == None or det(a, b, considering[i]) < det(a, b,
12                                                         ans):
13                 # |det(a,b,c)| = 1/2|ab|*h, gdzie h jest wysokoscia z c na ab
14                 ans = considering[i]
15                 i += 1
16             return ans
17
18  def insideTriangle(a, b, c, i):
19      accur=10**(-7)
20      if det(a, b, i) > -accur and det(b, c, i) > -accur and det(c, a,
21      i) > -accur:

```

```

20         return True
21     return False
22
23
24     def removeInner(a, b, c, considering):
25         new=[]
26         for i in considering:
27             if not insideTriangle(a, b, c, i):
28                 new.append(i)
29         considering.clear()
30         considering+=new
31
32     def quickHullUtil(a, b, considering):
33         if len(considering) == 0:
34             return []
35
36         c = furthest(a, b, considering)
37         if c == None:
38             return []
39         considering.remove(c)
40
41         removeInner(a, c, b, considering)
42         return quickHullUtil(a, c, considering) +[c]+ quickHullUtil(c, b,
43             considering)
44
45     def quickHull(points):
46         a = min(points, key=lambda x: x)
47         b = max(points, key=lambda x: x)
48
49         considering = deepcopy(points)
50
51         considering.remove(a)
52         considering.remove(b)

```

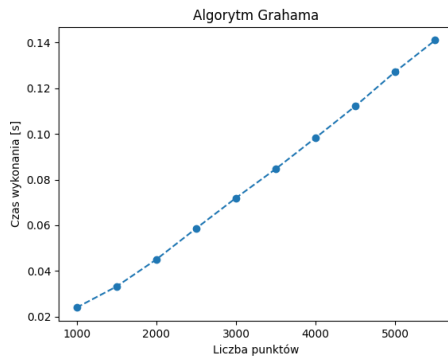
5 Wydajność algorytmów

Testy prowadzone były na następujących zbiorach:

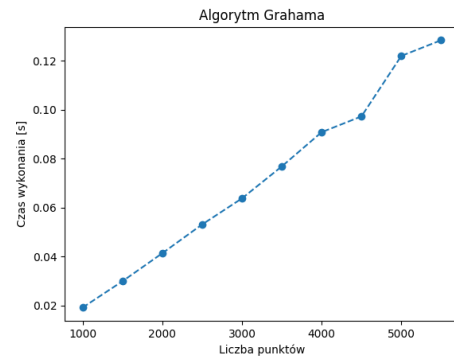
- Testy odbywały się na 4-rech typach zbiorów:
 - typ A - losowo rozłożone punkty płaszczyzny o określonych zakresach współrzędnych
 - typ B - losowo rozłożone punkty leżące na okręgu o zadanych parametrach
 - typ C - losowo rozłożone punkty leżące na bokach prostokąta o zadanych parametrach
 - typ D - losowo rozłożone punkty leżące na 2 bokach kwadratu, umiejscowionego tak, że te dwa boki pokrywają się z osiami układu oraz na jego przekątnych

Tablica 1: Czas wykonania algorytmu Grahama w zależności od typu zbioru testowego oraz mocy zbioru punktów.

	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Typ zbioru	Czas wykonania [s]									
A	0.0239	0.0331	0.045	0.0586	0.072	0.0847	0.0983	0.1122	0.1273	0.141
B	0.0191	0.0299	0.0413	0.0531	0.0637	0.0768	0.0908	0.0973	0.122	0.1284
C	0.069	0.1115	0.1392	0.1878	0.2349	0.2736	0.3153	0.351	0.3988	0.4601
D	0.4256	0.6523	0.924	1.2255	1.4233	1.8752	1.9285	2.3207	2.5692	2.88



Rysunek 1: Zbiór typu A, algorytm Grahama



Rysunek 2: Zbiór typu B, algorytm Grahama

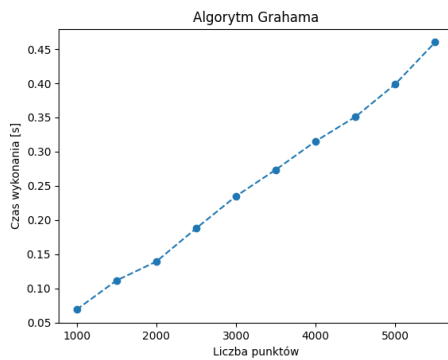
Charakterystyki zbiorów zostały dobrane w taki sposób aby odzwierciedlać możliwie różne zachowanie się poszczególnych algorytmów w zależności od ilości przypadków zdegenerowanych (liczebność współliniowych otoczek), ilości potrzebnych do wykonania porównań i operacji. Bardziej szczegółowe omówienie charakterystyki algorytmów dla poszczególnych zbiorów znajduje się w kolejnych sekcjach.

5.1 Algorytm Grahama

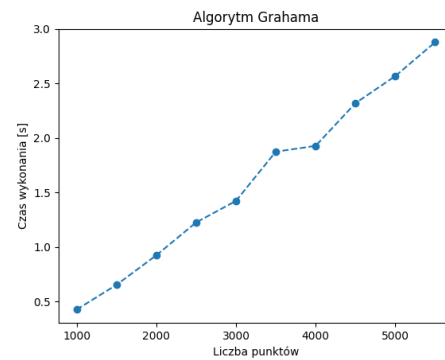
W tabeli 1 przedstawiamy czasy uzyskane przez algorytm Grahama dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 1, 2, 3, 4 widzimy ilustrację danych z tabeli 1.

5.2 Algorytm górna-dolna

W tabeli 2 przedstawiamy czasy uzyskane przez algorytm górna-dolna dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 5, 6, 7, 8 widzimy ilustrację danych z tabeli 2.



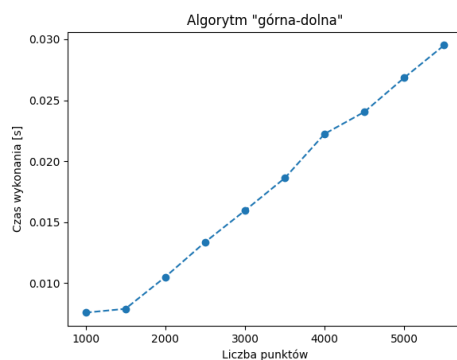
Rysunek 3: Zbiór typu C, algorytm Grahama



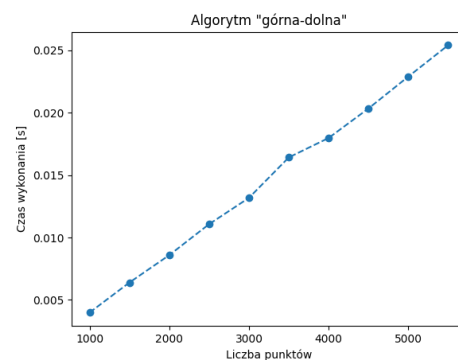
Rysunek 4: Zbiór typu D, algorytm Grahama

Tablica 2: Czas wykonania algorytmu górna-dolna w zależności od typu zbioru testowego oraz mocy zbioru punktów.

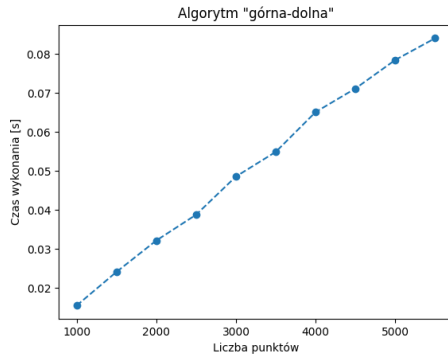
	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Typ zbioru	Czas wykonania [s]									
A	0.0076	0.0079	0.0105	0.0134	0.016	0.0186	0.0222	0.0241	0.0269	0.0295
B	0.004	0.0064	0.0086	0.0111	0.0132	0.0164	0.018	0.0203	0.0229	0.0254
C	0.0155	0.0241	0.0321	0.0388	0.0486	0.0549	0.0651	0.0711	0.0784	0.084
D	0.0635	0.0927	0.1261	0.1566	0.1875	0.2217	0.259	0.2897	0.316	0.3495



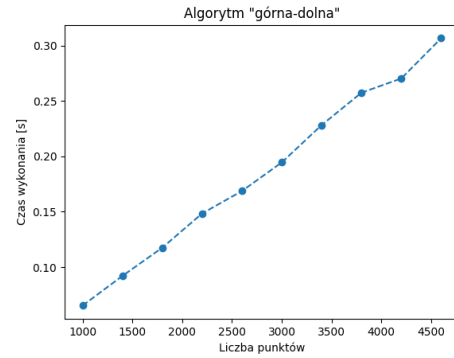
Rysunek 5: Zbiór typu A, algorytm górna-dolna



Rysunek 6: Zbiór typu B, algorytm górna-dolna



Rysunek 7: Zbiór typu C, algorytm górna-dolna



Rysunek 8: Zbiór typu D, algorytm górna-dolna

Tablica 3: Czas wykonania algorytmu Chana w zależności od typu zbioru testowego oraz mocy zbioru punktów.

	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Typ zbioru	Czas wykonania [s]									
A	0.1055	0.1544	0.3575	0.2408	0.5449	0.6155	0.7104	0.7904	0.8621	0.9826
B	0.3714	0.5483	0.6962	0.8893	1.0799	1.2537	1.415	1.827	1.7746	1.9095
C	0.2457	0.5615	0.727	0.9188	1.1007	1.3044	1.5073	1.6875	1.88	2.1234
D	0.6322	0.9372	1.2306	2.1543	2.8556	6.0844	2.4157	3.943	3.094	4.7075

5.3 Algorytm Chana

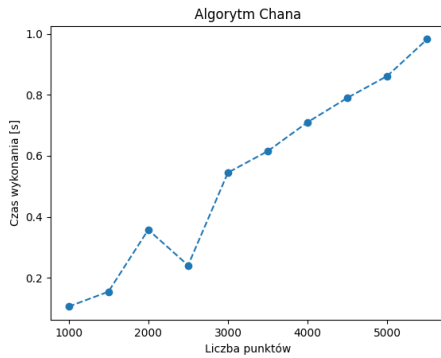
W tabeli 3 przedstawiamy czasy uzyskane przez algorytm Chana dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 9, 10, 11, 12 widzimy ilustrację danych z tabeli 3.

5.4 Algorytm QuickHull

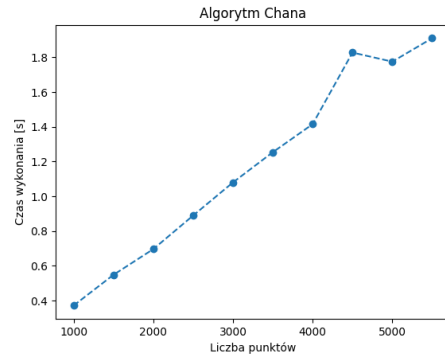
W tabeli 4 przedstawiamy czasy uzyskane przez algorytm QuickHull dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 13, 14, 15, 16 widzimy ilustrację danych z tabeli 4.

5.5 Algorytm dziel i zwyciężaj

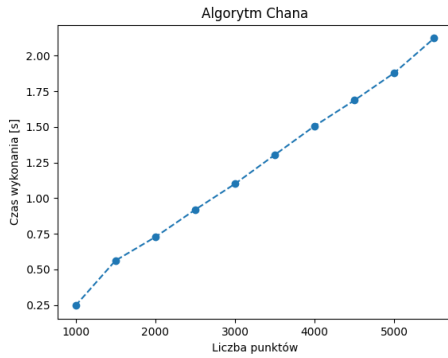
W tabeli 5 przedstawiamy czasy uzyskane przez algorytm Dziel i zwyciężaj dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 17, 18, 19, 20 widzimy ilustrację danych z tabeli 5.



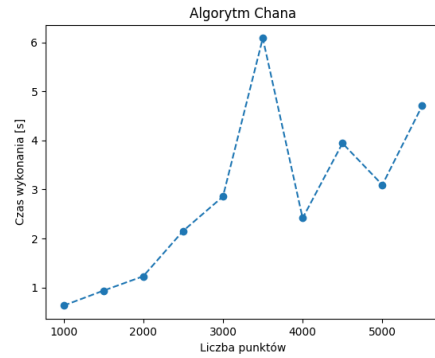
Rysunek 9: Zbiór typu A, algorytm Chana



Rysunek 10: Zbiór typu B, algorytm Chana



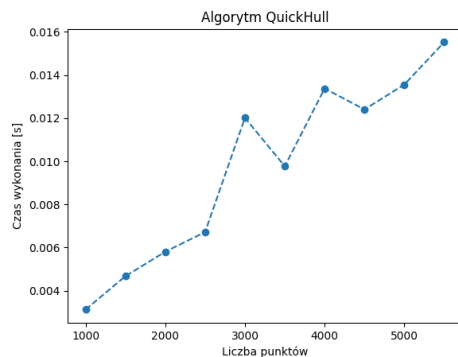
Rysunek 11: Zbiór typu C, algorytm Chana



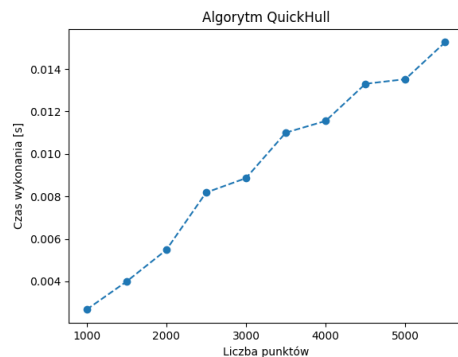
Rysunek 12: Zbiór typu D, algorytm Chana

Tablica 4: Czas wykonania algorytmu QuickHull w zależności od typu zbioru testowego oraz mocy zbioru punktów.

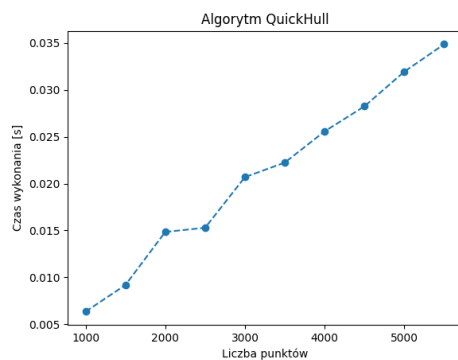
	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Typ zbioru	Czas wykonania [s]									
A	0.0031	0.0047	0.0058	0.0067	0.012	0.0098	0.0134	0.0124	0.0136	0.0155
B	0.0027	0.004	0.0055	0.0082	0.0089	0.011	0.0116	0.0133	0.0135	0.0153
C	0.0064	0.0092	0.0148	0.0153	0.0207	0.0222	0.0256	0.0282	0.0319	0.0348
D	0.0287	0.0426	0.0551	0.0702	0.0847	0.0986	0.1143	0.1294	0.1411	0.1562



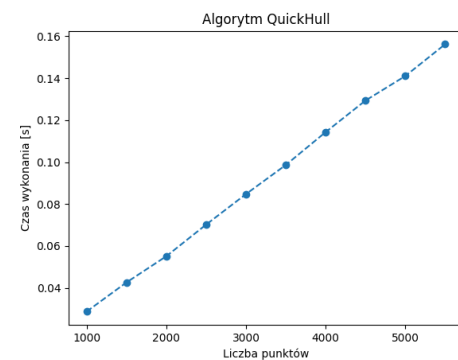
Rysunek 13: Zbiór typu A, algorytm QuickHull



Rysunek 14: Zbiór typu B, algorytm QuickHull



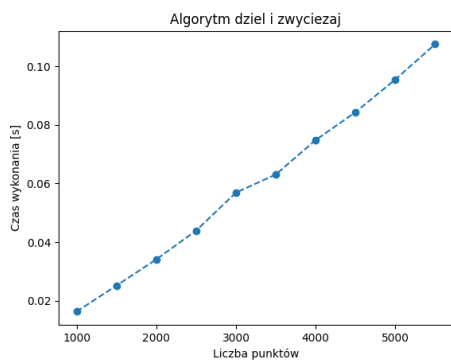
Rysunek 15: Zbiór typu C, algorytm QuickHull



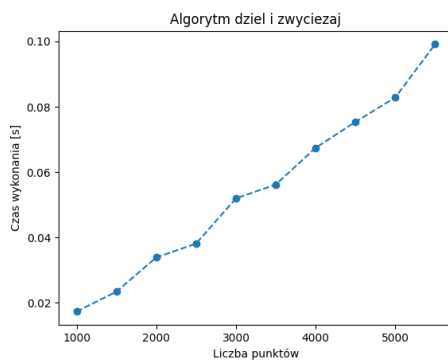
Rysunek 16: Zbiór typu D, algorytm QuickHull

Tablica 5: Czas wykonania algorytmu dziel i zwyciężaj w zależności od typu zbioru testowego oraz mocy zbioru punktów.

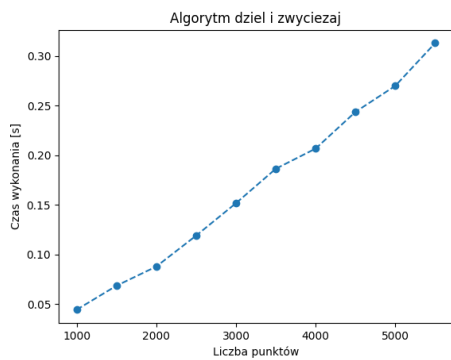
Typ zbioru	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Czas wykonania [s]										
A	0.0163	0.0251	0.0341	0.0439	0.0569	0.0631	0.0748	0.0844	0.0955	0.1076
B	0.0173	0.0234	0.0339	0.0381	0.052	0.0562	0.0675	0.0754	0.0829	0.0992
C	0.0441	0.0683	0.0877	0.1189	0.1516	0.1863	0.2067	0.2437	0.27	0.3132
B	0.1634	0.2474	0.3518	0.4703	0.5694	0.6705	0.7849	0.8928	1.0458	1.1565



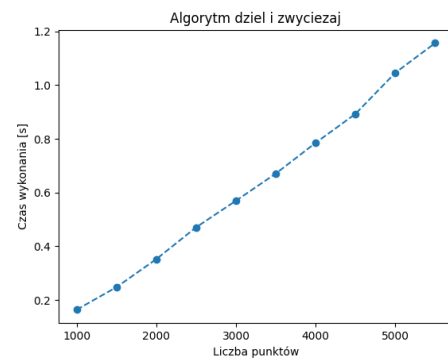
Rysunek 17: Zbiór typu A, algorytm dziel i zwyciężaj



Rysunek 18: Zbiór typu B, algorytm dziel i zwyciężaj



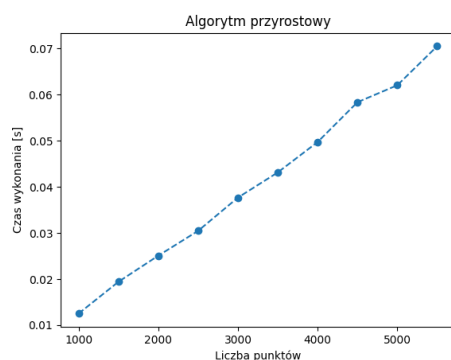
Rysunek 19: Zbiór typu C, algorytm dziel i zwyciężaj



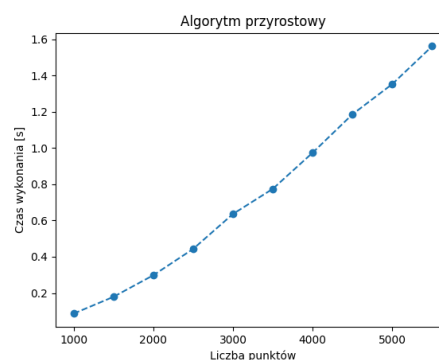
Rysunek 20: Zbiór typu D, algorytm dziel i zwyciężaj

Tablica 6: Czas wykonania algorytmu przyrostowego w zależności od typu zbioru testowego oraz mocy zbioru punktów.

	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Typ zbioru	Czas wykonania [s]									
A	0.0125	0.0194	0.0251	0.0305	0.0377	0.0431	0.0497	0.0584	0.0621	0.0705
B	0.0864	0.1788	0.2978	0.4428	0.6359	0.7745	0.9738	1.1858	1.3526	1.5613
C	0.0315	0.0475	0.063	0.0793	0.0997	0.1136	0.1292	0.1449	0.1555	0.1742
D	0.1373	0.1992	0.2715	0.326	0.4165	0.4639	0.5195	0.5937	0.6773	0.7446



Rysunek 21: Zbiór typu A, algorytm przyrostowy



Rysunek 22: Zbiór typu B, algorytm przyrostowy

5.6 Algorytm przyrostowy

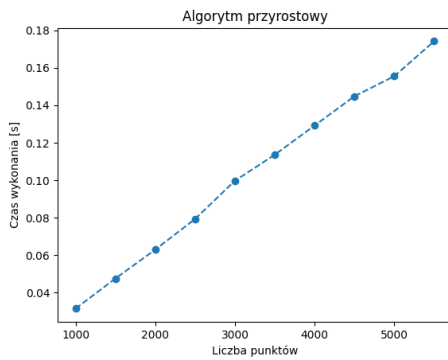
W tabeli 6 przedstawiamy czasy uzyskane przez algorytm przyrostowy i zwyciężaj dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 21, 22, 23, 24 widzimy ilustrację danych z tabeli 6.

5.7 Algorytm Jarvisa

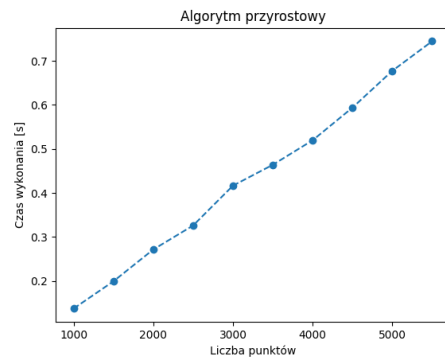
6 Porównanie algorytmów

7 Bibliografia

1. Wykład z przedmiotu *Algorytmy Geometryczne, Informatyka 3. sem., 1. st. AGH*, Barbara Głut
2. *Computational Geometry – Algorithms and Applications*, Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars
3. <https://jeffe.cs.illinois.edu/teaching/compgeom/notes/01-convexhull.pdf>



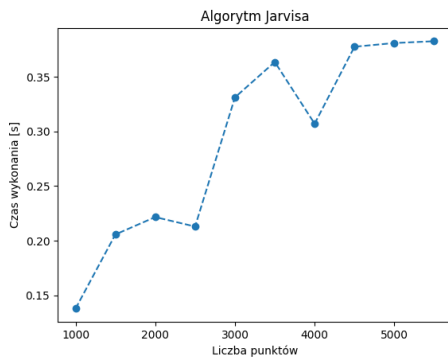
Rysunek 23: Zbiór typu C, algorytm przyrostowy



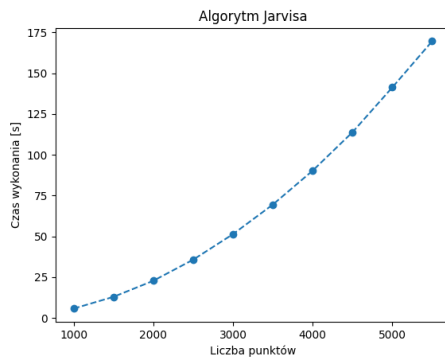
Rysunek 24: Zbiór typu D, algorytm przyrostowy

Tablica 7: Czas wykonania algorytmu Jarvisa w zależności od typu zbioru testowego oraz mocy zbioru punktów.

Typ danych	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	
A	0.1379	0.2058	0.2217	0.213	0.3311	0.3635	0.307	0.3775	0.3808	
B	5.6845	12.8821	22.7307	35.6488	51.3054	69.4251	90.2278	113.8051	141.2752	1
C	0.0529	0.077	0.1046	0.1306	0.1543	0.184	0.2086	0.2304	0.2571	
D	0.1094	0.1572	0.2167	0.2611	0.3129	0.3711	0.4218	0.4774	0.5255	



Rysunek 25: Zbiór typu A, algorytm Jarvisa



Rysunek 26: Zbiór typu B, algorytm Jarvisa

”

”””