

Otoczka wypukła dla zbioru punktów w przestrzeni dwuwymiarowej

Dokumentacja projektu
Algorytmy geometryczne

K. Kafara
Ł. Czarniecki

Spis treści

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

4.4	Algorytm przyrostowy	9
4.4.1	Opis działania	9
4.4.2	Szczegóły	10
4.4.3	Złożoność	11
4.4.4	Kod	12
4.5	Algorytm dziel i zwyciężaj	14
4.5.1	Opis działania	14
4.5.2	Szczegóły	15
4.5.3	Złożoność	15
4.5.4	Kod	15
4.6	Algorytm Chana	18
4.6.1	Opis działania	18
4.6.2	Szczegóły	19
4.6.3	Złożoność	19
4.6.4	Kod	20
4.7	Algorytm QuickHull	21
4.7.1	Opis działania	21
4.7.2	Szczegóły	21
4.7.3	Złożoność	22
4.7.4	Kod	22

Spis rysunków

Spis tablic

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 "górną-dolną" wraz z kodem tworzącym wizualizację

1.2 Wymagania techniczne

1. Python 3.9.0 64-bit lub nowszy
2. Jupyter Notebook

1.3 Korzystanie z programu

1.3.1 Uruchomienie wizualizacji

W celu uruchomienia wizualizacji algorytmów należy uruchomić notebook (poprzez Jupyter Notebook) *program.ipynb*, a następnie zapoznać się z zamieszczoną tam instrukcją.

2 Oznaczenia i definicje

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

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. Usuwamy 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) Usuwamy 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. Znaleziony punkt 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 ostatniej 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 zbiory 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 uproszenie 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

Styczne wyznaczamy w czasie logarytmicznym względem liczby punktów należących do otoczki do której szukamy stycznych, wykonując poszukiwanie binarne elementów skrajnych (najmniejszego i największego) względem następującego porządku, określonego dla dowolnych dwóch punktów płaszczyzny P_1, P_2 , względem ustalonego punktu Q :

$P_1 >_Q P_2 \Leftrightarrow P_2$ znajduje się po lewej stronie odcinka skierowanego QP_1 .

W tak określonym porządku elementem największym będzie prawy punkt styczności P_{max} , ponieważ wszystkie inne punkty otoczki znajdują się na lewo od prostej (stycznej) QP_{max} . Podobnie lewym punktem styczności będzie element najmniejszy w zadanym porządku P_{min} .

Wyznaczanie P_{max} (P_{min} wyznaczamy w sposób zupełnie analogiczny):

Wprowadźmy najpierw potrzebne oznaczenia:

Q punkt zewnętrzny, przez który mają przechodzić styczne do otoczki.

Niech otoczka będzie dana w postaci ciągu punktów P_0, \dots, P_{k-1} ($P_k := P_0$) będących

współrzędnymi kolejnych wierzchołków w kolejności przeciwnej do ruchu wskazówek zegara.

Krawędź skierowaną otoczki e_i definiujemy następująco: $e_i := P_i P_{i+1}$.

Krawędź $e_i := P_i P_{i+1}$ jest skierowana w górę $\Leftrightarrow P_{i+1} >_Q P_i$.

Krawędź e_i jest skierowana w dół, jeżeli nie jest skierowana w górę.

1. Jeżeli P_0 jest większy od swoich obu sąsiadów, to go zwracamy jako element największy.
2. Definiujemy indeksy $l := 0, r := k$
3. Dopóki nie znajdziemy elementu największego:
 - (a) Wyznaczamy indeks środkowego elementu $m := \left\lfloor \frac{l+k}{2} \right\rfloor$
 - (b) Jeżeli P_m jest elementem największym:
 - Zwracamy P_m .
 - (c) Jeżeli e_l jest skierowana w górę:
 - i. Jeżeli e_m jest skierowana w dół
 - $r := m$
 - ii. W przeciwnym przypadku
 - A. Jeżeli $P_l >_Q P_m$:
 - $r := m$
 - B. W przeciwnym przypadku
 - $l := m$
 - (d) W przeciwnym przypadku:
 - i. Jeżeli e_m jest skierowana w górę:
 - $l := m$
 - ii. W przeciwnym przypadku:
 - A. Jeżeli $P_m >_Q P_l$:
 - $r := m$
 - B. W przeciwnym przypadku:
 - $l := m$

4.4.3 Złożoność

Posortowanie punktów zajmuje $O(n \lg n)$. Każde wykonanie pętli zajmuje czas logarytmiczny względem liczności otoczki znanej w danej iteracji. Zatem złożoność algorytmu jest rzędu $O(n \lg n)$

4.4.4 Kod

```
1 def right_tangent(polygon: ListOfPoints, point: Point) -> Union[int, None
2   ]:
3     n = len(polygon)
4     if n < 3: return None
5
6     if orientation(polygon[0], polygon[1], point) == 1 and orientation(
7     polygon[-1], polygon[0], point) == -1:
8         return 0
9
10    left = 0
11    right = n
12
13    while True:
14        mid = (left + right) // 2
15
16        is_mid_down: bool = (orientation(polygon[mid], polygon[(mid + 1)
17        % n], point) == 1)
18
19        if is_mid_down and orientation(polygon[mid - 1], polygon[mid],
20        point) == -1:
21            return mid % n
22
23        is_left_up: bool = (orientation(polygon[left % n], polygon[(left
24        + 1) % n], point) == -1)
25
26        if is_left_up:
27            if is_mid_down:
28                right = mid
29            else:
30                if orientation(point, polygon[left], polygon[mid]) == 1:
31                    right = mid
32                else:
33                    left = mid
34        else:
35            if not is_mid_down:
36                left = mid
37            else:
38                if orientation(point, polygon[mid % n], polygon[left % n
39                ]) == 1:
40                    right = mid
41                else:
42                    left = mid
43
44 def left_tangent(polygon: ListOfPoints, point: Point) -> Union[Point,
45   None]:
46     n = len(polygon)
47     if n < 3: return None
48
49     left = 0
```

```

46     right = n
47
48     if orientation(point, polygon[1], polygon[0]) == 1 and orientation(
49         point, polygon[-1], polygon[0]) == 1:
50         return 0
51
52     while True:
53         mid = (left + right) // 2
54
55         is_mid_down: bool = (orientation(point, polygon[mid % n], polygon
56             [(mid + 1) % n]) == 1)
57
58         if (not is_mid_down) and orientation(point, polygon[(mid - 1) % n
59             ], polygon[mid % n]) == 1:
60             return mid % n
61
62         is_left_down: bool = (orientation(point, polygon[left], polygon[(
63             left + 1) % n]) == 1)
64
65         if is_left_down:
66             if not is_mid_down:
67                 right = mid
68             else:
69                 if orientation(point, polygon[mid % n], polygon[left % n
70                     ]) == 1:
71                     right = mid
72                 else:
73                     left = mid
74
75         else:
76             if is_mid_down:
77                 left = mid
78             else:
79                 if orientation(point, polygon[left % n], polygon[mid % n
80                     ]) == 1:
81                     right = mid
82                 else:
83                     left = mid
84
85
86 def increase_with_sorting(point2_set: ListOfPoints) -> Union[ListOfPoints
87     , None]:
88     if len( point2_set ) < 3: return None
89
90     point2_set.sort(key = operator.itemgetter(0, 1))
91
92     convex_hull = point2_set[:3]
93
94     if orientation(convex_hull[0], convex_hull[1], convex_hull[2]) == -1:
95         convex_hull[1], convex_hull[2] = convex_hull[2], convex_hull[1]
96
97     for i in range(3, len( point2_set )):
98         left_tangent_idx = left_tangent(convex_hull, point2_set[i])
99         right_tangent_idx = right_tangent(convex_hull, point2_set[i])

```

```

93     left_tangent_point = convex_hull[left_tangent_idx]
94     right_tangent_point = convex_hull[right_tangent_idx]
95
96     deletion_side: Literal[-1, 1] = orientation(left_tangent_point,
97     right_tangent_point, point2_set[i])
98
99     if orientation(left_tangent_point, right_tangent_point,
100     convex_hull[(left_tangent_idx + 1) % len(convex_hull)]) ==
101     deletion_side:
102         step = 0
103     else:
104         step = -1
105
106     left = (left_tangent_idx + 1) % len(convex_hull)
107
108     while convex_hull[left] != right_tangent_point:
109         convex_hull.pop(left)
110         left = (left + step) % len(convex_hull)
111
112     convex_hull.insert(left, point2_set[i])
113
114     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 któryi 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. Usuujemy punkty zawierające się we wnętrzu nowo utworzonej otoczki.

4.5.3 Złożoność

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[(
23     right_idx - 1) % right_ch_size])
24     while left_right_orient != -1:
25         if left_right_orient == 0 and dist_sq(left, right) >= dist_sq
26         (left, right_convex_hull[(right_idx - 1) % right_ch_size]):
27             right_flag = False
28             break
29
30     right_idx = (right_idx - 1) % right_ch_size
31     right = right_convex_hull[right_idx]
32     left_right_orient = orientation(left, right,
33     right_convex_hull[(right_idx - 1) % right_ch_size])
34     else:
35         right_flag = True
36
37     # podnosimy punkt na lewej otoczce
38     right_left_orient = orientation(right, left, left_convex_hull[(
39     left_idx + 1) % left_ch_size])
40     while right_left_orient != 1:
41         if right_left_orient == 0 and dist_sq(right, left) >= dist_sq
42         (right, left_convex_hull[(left_idx + 1) % left_ch_size]):
43             left_flag = False
44             break
45
46     left_idx = (left_idx + 1) % left_ch_size
47     left = left_convex_hull[left_idx]
48     right_left_orient = orientation(right, left, left_convex_hull
49     [(left_idx + 1) % left_ch_size])
50     else:
51         left_flag = True
52
53     upper_tangent_left_idx = left_idx
54     upper_tangent_right_idx = right_idx
55
56     # dolna styczna
57     left = left_convex_hull[left_ch_rightmost_idx]
58     right = right_convex_hull[right_ch_leftmost_idx]
59     left_idx = left_ch_rightmost_idx
60     right_idx = right_ch_leftmost_idx
61
62     left_flag, right_flag = True, True
63     while orientation(left, right, right_convex_hull[(right_idx + 1) %
64     right_ch_size]) != 1 and right_flag \
65         or \
66         orientation(right, left, left_convex_hull[(left_idx - 1) %
67     left_ch_size]) != -1 and left_flag:
68
69         left_flag, right_flag = False, False
70
71     # opuszczamy punkt na prawej otoczce
72     left_right_orient = orientation(left, right, right_convex_hull[(
73     right_idx + 1) % right_ch_size])
74     while left_right_orient != 1:

```



```

67         if left_right_orient == 0 and dist_sq(left, right) >= dist_sq
        (left, right_convex_hull[(right_idx + 1) % right_ch_size]):
68             right_flag = False
69             break
70
71             right_idx = (right_idx + 1) % right_ch_size
72             right = right_convex_hull[right_idx]
73             left_right_orient = orientation(left, right,
right_convex_hull[(right_idx + 1) % right_ch_size])
74         else:
75             right_flag = True
76
77             # opuszczamy punkt na lewej otoczce
78             right_left_orient = orientation(right, left, left_convex_hull[(
left_idx - 1) % left_ch_size])
79             while right_left_orient != -1:
80                 if right_left_orient == 0 and dist_sq(right, left) >= dist_sq
(right, left_convex_hull[(left_idx - 1) % left_ch_size]):
81                     left_flag = False
82                     break
83
84                     left_idx = (left_idx - 1) % left_ch_size
85                     left = left_convex_hull[left_idx]
86                     right_left_orient = orientation(right, left, left_convex_hull
[(left_idx - 1) % left_ch_size])
87             else:
88                 left_flag = True
89
90
91         lower_tangent_left_idx = left_idx
92         lower_tangent_right_idx = right_idx
93
94         merged_convex_hull = [ ]
95
96         while upper_tangent_left_idx != lower_tangent_left_idx:
97             merged_convex_hull.append(left_convex_hull[upper_tangent_left_idx
])
98             upper_tangent_left_idx = (upper_tangent_left_idx + 1) %
left_ch_size
99         else:
100             merged_convex_hull.append(left_convex_hull[lower_tangent_left_idx
])
101
102
103         while lower_tangent_right_idx != upper_tangent_right_idx:
104             merged_convex_hull.append(right_convex_hull[
lower_tangent_right_idx])
105             lower_tangent_right_idx = (lower_tangent_right_idx + 1) %
right_ch_size
106         else:
107             merged_convex_hull.append(right_convex_hull[
lower_tangent_right_idx])
108
109         return merged_convex_hull

```

```

110
111
112 def divide_conq(point2_set: List[Point], k: int = 2) -> Union[List[Point], None]:
113     if len(point2_set) < 3 or k <= 0: return None
114
115
116     def divide_conq_rec(point2_set: List[Point]) -> List[Point]:
117         if len(point2_set) <= 2: return point2_set
118         elif len(point2_set) <= k: return jarvis(np.array(point2_set))
119
120         left_convex_hull = divide_conq_rec(point2_set[: len(point2_set)
121 // 2])
122         right_convex_hull = divide_conq_rec(point2_set[len(point2_set) //
123 2 : ])
124
125         return merge_convex_hulls(left_convex_hull, right_convex_hull)
126
127     point2_set.sort(key = operator.itemgetter(0, 1))
128
129     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 nadzowaną 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 $\lceil n/m \rceil$ podzbiorów rozmiaru m , dla każdego z nich wyznaczamy otoczkę algorytmem Grahama. Algorytm Grahama działa $O(n \log(n))$. Więc łącznie mamy $O(\lceil n/m \rceil * m \log(m)) = O(n \log(m))$.

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 postaci 2^{2^m} dla $m \geq 1$, to: w takim razie złożoność można opisać wzorem $\sum_{t=1}^{\lceil \log \log k \rceil} 2^{-n} =$

1

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)): # jezeli p jest po prawej odcinka [current
7     ,q] - jest 'wiekszy', 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 wspolzednymi z Q[i][j] jesli
16     jest to punkt nalezacy do finalnej otoczki, to
17     # zwraca nastepny punkt nalezacy do finalnej otoczki zadanego w
18     takich samych wspolzednych 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
28
29 def chanUtil(points, m):
30     Q = divide(points, m)
31     C = []
32     for i in range(len(Q)):
33         C.append(Graham(Q[i]))
34
35     curr = (0, 0)
36     ans = []
37     i = 0
38     while i < m:
39         ans.append(C[curr[0]][curr[1]])
40         if nextvert(C, curr) == (0, 0):
41             return ans
42         curr = nextvert(C, curr)
43         i += 1
44
45     return None
46
47 def chan(points):
48     n = len(points)
49     m = 4
50     hoax = None
51     while hoax == None:
52         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 otoczce. 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 wierzchołek
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:
22          return True
23      return False
24
25  def removeInner(a, b, c, considering):
26      new=[]
27      for i in considering:
28          if not insideTriangle(a, b, c, i):
29              new.append(i)
30      considering.clear()
31      considering+=new
32
33  def quickHullUtil(a, b, considering):
34      if len(considering) == 0:
35          return []
36
37      c = furthest(a, b, considering)
38      if c == None:
39          return []
40      considering.remove(c)
41
42      removeInner(a, c, b, considering)
43      return quickHullUtil(a, c, considering) + [c] + quickHullUtil(c, b,
44          considering)
```

```
43
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)
```