

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	14
4.5.3	Złożoność	14
4.5.4	Kod	14
4.6	Algorytm dziel i zwyciężaj	14
4.6.1	Opis działania	14
4.6.2	Szczegóły	14
4.6.3	Złożoność	14
4.6.4	Kod	14
4.7	Algorytm Chana	14
4.7.1	Opis działania	14
4.7.2	Szczegóły	15
4.7.3	Złożoność	15
4.7.4	Kod	16
4.8	Algorytm QuickHull	17
4.8.1	Opis działania	17
4.8.2	Szczegóły	18
4.8.3	Złożoność	18
4.8.4	Kod	19

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 "górna-dolna"

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) 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. 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

1. Sortujemy punkty rosnąco po odciętych (w przypadku równych, mniejszy jest punkt o mniejszej rzędnej).

4.5.2 Szczegóły

4.5.3 Złożoność

4.5.4 Kod

4.6 Algorytm dziel i zwyciężaj

4.6.1 Opis działania

4.6.2 Szczegóły

4.6.3 Złożoność

4.6.4 Kod

4.7 Algorytm Chana

4.7.1 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 popsóc 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.

4.7.2 Szczegóły

4.7.3 Złożoność

Złożoność głównej części algorytmu.

- Złożoność pierwszej części składa się na :
 - Podział zbioru punktów na podzbiory $O(n)$;
 - Wyznaczenie otoczek dla podzbiorów. Mamy $\text{ceil}(n/m)$ 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 postaci 2^{2^m} dla $m \geq 1$, to: Iteracja zako

4.7.4 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 współzedydami 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)):

```



```

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

```

4.8 Algorytm QuickHull

4.8.1 Opis działania

Algorytm QuickHull polega na rekurencyjnym wyznaczaniu kolejnych punktów otoczki.

1. Algorytm rozpoczynamy od wyznaczenia 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 należy 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.

4.8.2 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.

4.8.3 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 z pomocą algorytmu wyszukiwania binarnego 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 $k=m$ (gdy k, m przerywamy algorytm, więc złożoność pozostaje ta sama), to ostatecznie mamy złożoność dla drugiej części rzędu : $O(k \cdot \text{ceil}(n/m) \cdot \log(m)) = O(n \log(m))$ 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: Iteracja zako

4.8.4 Kod

```
1 def furthest(a, b, considering):
2     n = len(considering)
3     i = 0
4     ans = None
5     while i < n:
6         if det(a, b, considering[i]) < 0: # rozwazany wierzcholek jest
7             # po prawej stronie ab
8             if ans == None or det(a, b, considering[i]) < det(a, b,
9                                                         ans): # |
10                 det(a,b,c)| = 1/2|ab|*h, gdzie h jest wysokoscia z c na ab
11                 ans = considering[i]
12                 i += 1
13     return ans
14
15 def insideTriangle(a, b, c, i):
16     if det(a, b, i) > 0 and det(b, c, i) > 0 and det(c, a, i) > 0:
17         return True
18     return False
19
20 def removeInner(a, b, c, considering):
21     new=[]
22     for i in considering:
23         if not insideTriangle(a, b, c, i):
24             new.append(i)
25     considering.clear()
26     considering+=new
27
28 def quickHullUtil(a, b, considering):
29     if len(considering) == 0:
30         return []
31
32     c = furthest(a, b, considering)
33     if c == None:
34         return []
35     considering.remove(c)
36
37     removeInner(a, c, b, considering)
38     return quickHullUtil(a, c, considering) + [c] + quickHullUtil(c, b,
39     considering)
40
41 def quickHull(points):
42     a = min(points, key=lambda x: x[0])
43     b = max(points, key=lambda x: x[0])
44
45     considering = deepcopy(points)
46
47     considering.remove(a)
48     considering.remove(b)
```