

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	6
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	9
4.3.1	Opis działania	9
4.3.2	Szczegóły	9
4.3.3	Złożoność	9

4.3.4	Kod	9
4.4	Algorytm przyrostowy	9
4.4.1	Opis działania	9
4.4.2	Szczegóły	9
4.4.3	Złożoność	9
4.4.4	Kod	9
4.5	Algorytm dziel i zwyciężaj	10
4.5.1	Opis działania	10
4.5.2	Szczegóły	10
4.5.3	Złożoność	10
4.5.4	Kod	10
4.6	Algorytm Chana	10
4.6.1	Opis działania	10
4.6.2	Szczegóły	10
4.6.3	Złożoność	10
4.6.4	Kod	10

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 .

$$P_1 \preceq_Q P_2 \Leftrightarrow (\angle(P_1, Q, OX) < \angle(P_2, Q, OX)) \vee (\angle(P_1, Q, OX) = \angle(P_2, Q, OX) \wedge d(P_1, Q) < d(P_2, Q)) \quad (1)$$

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 współlinowych 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[
30 prev], points[i])):
31                 imax = i
32
33             if imax == start_idx:
34                 break;
35
36         convex_hull.append(imax)
37
38     prev = imax
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

4.3.2 Szczegóły

4.3.3 Złożoność

4.3.4 Kod

4.4 Algorytm przyrostowy

4.4.1 Opis działania

4.4.2 Szczegóły

4.4.3 Złożoność

4.4.4 Kod

4.5 Algorytm dziel i zwyciężaj

4.5.1 Opis działania

4.5.2 Szczegóły

4.5.3 Złożoność

4.5.4 Kod

4.6 Algorytm Chana

4.6.1 Opis działania

cokolwiek [Che89]

4.6.2 Szczegóły

4.6.3 Złożoność

4.6.4 Kod

Bibliografia

[Che89] Otfried Cheong. *Computational Geometry, Algorithms and Applications*. 1989.