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 *lib* . . . 3
1.1.2 Moduł *pure* . . . 3
1.1.3 Moduł *vis* . . . 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 . . . 7
4.3.1 Opis działania . . . 7
4.3.2 Szczegóły . . . 7
4.3.3 Złożoność . . . 7

4.3.4 Kod 7
4.4 Algorytm przyrostowy 7
4.4.1 Opis działania 7
4.4.2 Szczegóły 7
4.4.3 Złożoność 7
4.4.4 Kod 7
4.5 Algorytm dziel i zwyciężaj 8
4.5.1 Opis działania 8
4.5.2 Szczegóły 8
4.5.3 Złożoność 8
4.5.4 Kod 8
4.6 Algorytm Chana 8
4.6.1 Opis działania 8
4.6.2 Szczegóły 8
4.6.3 Złożoność 8
4.6.4 Kod 8

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 "górna-dolna" 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 opisania 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 leżacej na jednej prostej, poza najbardziej oddalonym od Q usuwamy w czasie liniowym w następujący sposób: Iterując przez posortowaną tablicę, zaczynająć 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 $[0, \ldots, j-1]$.

4.1.3 Złożoność

Operacją dominującą w algorytmie jest sortowanie – realizowane w czasie $O(n\,lgn)$. 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 $O(n\,lgn)$.

4.1.4 Kod

```python
def get_point_cmp(ref_point: Point, eps: float = 1e-7) -> Callable:
    def point_cmp(point1, point2):
        orient = orientation(ref_point, point1, point2, eps)

        if orient == -1:
            return False
        elif orient == 1:
            return True
        elif dist_sq(ref_point, point1) <= dist_sq(ref_point, point2):
            return True
        else:
            return False

    return point_cmp


def graham(points: ListOfPoints) -> ListOfPoints:
    istart = index_of_min(points, 1)

    points[istart], points[0] = points[0], points[istart]

    qsort_iterative(points, get_point_cmp(points[0]))

    i, new_size = 1, 1
    while i < len(points):
        while (i < len(points) - 1) \
        and \
        (orientation(points[0], points[i], points[i + 1], 1e-7) == 0):
            i += 1

        points[new_size] = points[i]
        new_size += 1
        i += 1

    s = Stack()
    s.push(points[0])
    s.push(points[1])
    s.push(points[2])

    for i in range(3, new_size, 1):
        while orientation(s.sec(), s.top(), points[i], 1e-7) != 1:
            s.pop()

```

```
        s.push(points[i])

    return s.s[:s.itop+1]


```

4.2 Algorytm Jarvisa

4.2.1 Opis działania

4.2.2 Szczegóły

4.2.3 Złożoność

4.2.4 Kod

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.