Dokumentacja

Wyznaczanie najmniejszego okręgu oraz prostokąta
o najmniejszym polu i obwodzie zawierającego chmurę punktów

autor: Jakub Frączek

Spis treści

Część techniczna

Opis programu

Program jest wykorzystywany do:
- Wyznaczania najmniejszego okręgu zawierającego chmurę punktów
- Wyznaczania prostokąta o najmniejszym polu/obwodzie zawierającego chmurę punktów

Dodatkowe funkcje programu
- Generowanie losowych punktów na płaszczyźnie
- Wizualizacja działania algorytmów

Na program składają się następujące pliki:

- smallestcircle.py
- convexhull.py
- mbr.py
- random_points_generator.py
- geometry.py
- viewer.py

Moduły smallestcircle, convexhull, mbr, geometry są tematem projektu i mogą być wykorzystywane samodzielnie, natomiast moduły: random_points_generator, veiwer.py są dodatkiem.

Wymagania techniczne

Do poprawnego działania programu zalecane jest skorzystanie z komputera z systemem operacyjnym Windows (10 lub 11) lub Linux (opartym na debianie) oraz procesora o mocy obliczeniowej porównywalnej lub większej od AMD Ryzen 5 5500U 2.1 GHZ.

Program został przetestowany na komputerze z systemem operacyjnym Linux Mint oraz procesorem AMD Ryzen 5 5500U 2.1 GHZ.

Wykorzystane moduły

W projekcie zostały wykorzystane poniższe moduły:

- matplotlib
- copy
- time
- bitalg.visualizer
- functools
- collections
- random
- math
- customtkinter
-

Opis klas i funkcji programu

Moduł Geometry
Definicja klas ułatwiających operacje na obiektach geometrycznych

Klasa Circle
Klasa wykorzystywana do reprezentacji okręgu

Metoda __init__(self, S, r)
Konstruktor przyjmujący dwa argumenty S - krotka zawierająca współrzędne środka okręgu oraz r - promień okręgu

Klasa Rectangle
Klasa wykorzystywana do reprezentacji prostokąta

Metoda __init__(self, A, B, C, D)
Konstruktor przyjmujący cztery argumenty, z których każdy jest krotką reprezentującą położenie wierzchołków prostokąta

Metoda getEdges(self)
Zwraca odcinki będące bokami prostokąta w postaci [((x1, y1), (x2, y2,)) , …]

Atrybut vertices
Lista przechowująca wierzchołki prostokąta

Klasa PointSet
Ułatwia operacje na zbiorze punktów

Metoda getEdges(points)
Zakłada, że punkty są posortowane. Zwraca odcinki tworzące krawędzie wielokąta w postaci[((x1, y1), (x2, y2,)) , …]

Moduł random_points_generator
Służy do generowania losowych danych w celu testowania algorytmów

Klasa random

Metoda generate_uniform_points(n, min_x, max_x, min_y, max_y)
Losowo generuje punkty na płaszczyźnie. Argumenty: n - ilość punktów do wygenerowania, min_x - minimalna współrzędna x-owa punktu, max_x - maksymalna współrzędna x - owa punktu, min_y - minimalna współrzędna y - owa punktu, max_y - maksymalna współrzędna y - owa punktu.

Metoda generate_circle_points(n, circle)
Losowo generuje punkty na okręgu, gdzie n - ilość punktów do wygenerowania, circle - Circle z modułu geometry, którego parametry wyznaczają okrąg na którym zostaną wygenerowane punkty.

Metoda statyczna generate_rectangle_points(n, rectangle)
Losowo generuje punkty na prostokącie zadanym przez rectangle, gdzie rectangle to Rectangle z modułu geometry

Moduł mbr
Skrót mbr oznacza minimum bounding rectangle

Klasa mbr

Metoda compare_area(a, b)
Argumenty a i b to są długości boków prostokąta (krótsza i dłuższa). Funkcja zwraca pole prostokąta z bokami a, b. Może być wykorzystywana jako komparator dla funkcji smallest_rectangle i smallest_rectangle_draw

Metoda compare_perimeter(a, b)
Argumenty a i b to są długości boków prostokąta (krótsza i dłuższa). Funkcja, może być wykorzystywana jako komparator dla funkcji smallest_rectanglei smallest_rectangle_draw

Metoda smallest_rectangle(hull_points, compare)
Argumenty to hull_points - punkty należące do otoczki wypukłej, compare - funkcja zwracająca wartość na podstawie dwóch argumentów (długość krótszego i dłuższego boku prostokąta). Funkcja zwraca obiekt Rectangle będący prostokątem o najmniejszej wartości zwracanej przez compare zawierającym chmurę punktów (Przykładowo dla compare_area funkcja zwraca prostokąt o najmniejszym polu zawierającym chmurę punktów)

Metoda smallest_rectangle_draw(hull_points, points, compare)
Argumenty to hull_points - punkty należące do otoczki wypukłej, compare - funkcja zwracająca wartość na podstawie dwóch argumentów (długość krótszego i dłuższego boku prostokąta) oraz points - wszystkie punkty. Funkcja wizualizuje działanie algorytmu krok po kroku. Zwraca obiekt Visualizer().

Moduł smallest_circle

Klasa Graham

Metoda __distance_between_two_points(A, B)
Zwraca dystans między punktami A i B na płaszczyźnie

Metoda __center(A, B, C)
Zwraca środek okręgu, na którego brzegu leżą punkty A, B, C.

Metoda __construct_circle(self, R)
Argument R jest tablicą zawierającą od 0 do 3 punktów. W zależności od ilości punktów
metoda zwraca okrąg, do którego brzegu należą te punkty

Metoda __in_circle(self, circle, point, eps)
Zwraca True jeśli punkt point znajduje się w okręgu circle z dokładnością eps lub False w przeciwnym wypadku.

Metoda __r_welzl_algorithm(self, P, R, last)
Argumenty P - lista punktów, R - lista punktów leżących na brzegu okręgu, last - ilość elementów z P branych aktualnie pod uwagę. Zwraca obiekt klasy Circle opisujący najmniejszy okrąg zawierający wszystkie punkty z P

Metoda welzl_algorithm(self, points)
Metoda pomocnicza wywołująca __r_welzl_algorithm(points, [], len(points)). Argument points zawiera zbiór punktów. Zwraca obiekt klasy Circle opisujący najmniejszy okrąg zawierający punkty z points.

Metoda __r_welzl_algorithm_draw(self, P, R, last, vis)
Służy do wizualizacji działania algorytmu. Zwraca obiekt klasy Visualizer.

Metoda welzl_algorithm_draw(self, points)
Metoda pomocnicza wywołująca __r_welzl_algorithm_draw. Wywołuje __r_welzl_algorithm(points, [], len(points, Visualizer())

Moduł convexhull

Klasa Graham

Metoda det(a, b, c)
Zwraca wyznacznik macierzy 2x2

Metoda distance(P, A, B)
Jeśli A jest bliżej P zwraca 1, jeśli B jest bliżej P to zwraca -1

Metoda cmp(P, A, B, eps)
Liczy orientację punktu P względem odcinka PA. Zwraca 1 jeśli B jest po prawej PA, -1 jeśli B jest po lewej, a w przypadku, gdy B leży na PA, jest jest wywoływana metoda distance i zwracana jej wartość.

Metoda orient(P, A, B, eps)
Liczy orientację punktu P względem odcinka PA. Zwraca 1 jeśli B jest po prawej PA, -1 jeśli B jest po lewej i 0, gdy B leży na PA.

Metoda graham_algoithm(X)
Przyjmuje zbiór punktów X, zwraca zbiór punktów tworzący otoczkę wypukłą

Metoda graham_algorithm_draw(X)
Służy do wizualizacji działania algorytmu Grahama. Przyjmuje zbiór punktów X, zwraca obiekt Visualizer.

Część użytkowa

Instalacja programu

1. Pobrać kod źródłowy ze strony: https://github.com/JakubFr4czek/Algorytmy-Geometryczne.
2. Następnie pobrać kod źródłowy biblioteki bitalg ze strony: https://github.com/aghbit/Algorytmy-Geometryczne/tree/master/bitalg/visualizer.
3. Skopiować folder visualizer do folderu Algorytmy-Geometryczne
4. Zainstalować moduły: matplotlib, customtkinter

Przykład instalacji modułów na systemie operacyjnym linux (opartym na debianie):

```
sudo apt-get update
sudo pip3 install matplotlib customtkinter
```

Sposób korzystania z programu

1. Losowe generowanie punktów na płaszczyźnie

```
from random_points_generator import Random

points = Random.generate_uniform_points(n = 50, min_x=-10, max_x=10, min_y=-10, max_y=10)

print(points)
```

2. Losowe generowanie punktów na okręgu

```
from random_points_generator import Random
from geometry import Circle

center = (0, 0)
radius = 10

my_circle = Circle(S = center, r = radius)

points = Random.generate_circle_points(n = 50, circle=my_circle)

print(points)
```

3. Losowe generowanie punktów na prostokącie

```
from random_points_generator import Random
from geometry import Rectangle
```

```
A = (-10, 10)
B = (10, 10)
C = (10, -10)
D = (-10, -10)

my_rectangle = Rectangle(A, B, C, D)

points = Random.generate_rectangle_points(n = 100, rectangle=my_rectangle)

print(points)
```

4. Wyliczenie najmniejszego okręgu zawierającego chmurę punktów

```
from random_points_generator import Random
from smallestcircle import Welzl

points = Random.generate_uniform_points(n = 50)

circle = Welzl.welzl_algorithm(points)

print(circle.S, circle.r)
```

5. Wyliczenie prostokąta o najmniejszym obwodzie zawierającego chmurę punktów

```
from random_points_generator import Random
from mbr import Mbr

points = Random.generate_uniform_points(n = 50)

min_area_rectangle, area = Mbr.smallest_rectangle(points, Mbr.compare_area)

print(area)
print(min_area_rectangle.vertices)
```

6. Wyliczenie prostokąta o najmniejszym polu zawierającego chmurę punktów

```
from random_points_generator import Random
from mbr import Mbr

points = Random.generate_uniform_points(n = 50)

min_perimeter_rectangle, perimeter = Mbr.smallest_rectangle(points, Mbr.compare_perimeter)

print(perimeter)
print(min_area_rectangle.vertices)
```

7. Wizualizacja działania algorytmów

Należy w terminalu (konsoli) uruchomić plik viewer.py za pomocą interpretera pythona.

Przykład uruchomienia na linux'ie:

```
python3 viewer.py
```

Następnie należy postępować, zgodnie z instrukcjami wyświetlanymi na ekranie

Sprawozdanie

1. Wstęp

Projekt polegał na stworzenie programu wyznaczającego:
- minimalny okrąg zawierający chmurę punktów
- prostokąt o najmniejszym polu zawierający chmurę punktów
- prostokąt o najmniejszym obwodzie zawierający chmurę punktów

1.1 Dane techniczne

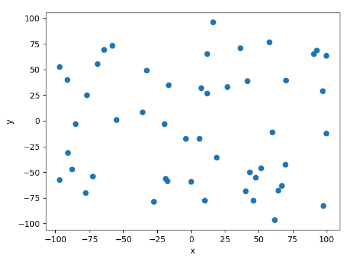
Program został napisany i przetestowany na komputerze z systemem operacyjnym Linux Mint z procesorem AMD Ryzen 5 5500U 2.1 GHZ. Wykorzystane technologie to Python3 wraz z modułami:
- matplotlib
- copy
- time
- bitalg.visualizer
- functools
- collections
- random
- math
- customtkinter

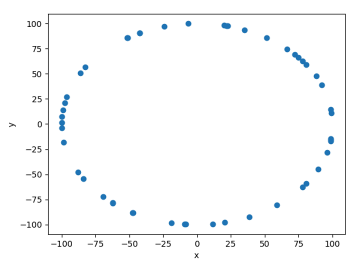
oraz Jupyter Notebook.

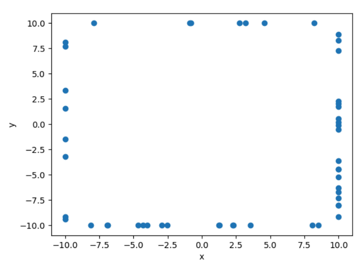
2. Generowania przykładowych danych

Przykładowe zbiory punktów na płaszczyźnie zostały wygenerowane za pomocą napisanego przeze mnie modułu random_points_generator, moduł obejmuje generowanie punktów:
- losowo na płaszczyźnie
- losowo na okręgu
- losowo na prostokącie

Na wykresach poniżej, wizualizacje przykładowych danych, wygenerowane za pomocą modułu







2.1. Interaktywne zadawanie punktów na płaszczyźnie

Przetestowane zostały także dane zadawane za pomocą myszki

3. Minimalny okrąg zawierający chmurę punktów

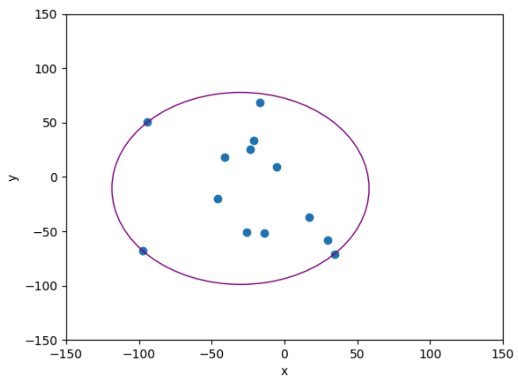
W celu wyznaczenia najmniejszego okręgu zawierającego chmurę punktów skorzystałem z algorytmu Emmerich'a Welzl'a. Oczekiwana złożoność algorytmu to O(n). Algorytm składa się z następujących kroków:

```
algorithm welzl is
        input: Finite sets P and R of points in the plane |R| ≤ 3.
        output: Minimal disk enclosing P with R on the boundary.

        if P is empty or |R| = 3 then
        return trivial(R)
        choose p in P (randomly and uniformly)
        D := welzl(P − {p}, R)
        if p is in D then
        return D

        return welzl(P − {p}, R ∪ {p})
```

Innymi słowami wybieramy losowo pewien punkt p z wejściowego zbioru P, a następnie sprawdzamy, czy należy on do najmniejszego okręgu zawierającego wszystkie punkty ze zbioru P-{p}. Jeśli tak, to ten okrąg jest tym szukanym, w przeciwnym wypadku należy on do brzegu szukanego okręgu.

3.1. Efekt działania algorytmu

Wizualizacja algorytmu dla przykładowych danych jest pokazany na wykresie 1.



Wykres 1. Wynik działania algorytmu Welzl'a

3.2. Porównanie czasów działania algorytmu

W tabeli 2. przedstawiono porównanie czasów działania algorytmu Welzl'a dla różnej ilości punktów w zbiorze.

Ilość punktów	Czas wykonania [s]
100	0,002
1000	0,008
10000	0,136
100000	0,537
1000000	9,769

Tabela 1. Czasy działania algorytmu Welzl'a

4. Prostokąt o najmniejszym polu/obwodzie zawierający chmurę punktów

Do rozwiązania problemu została wykorzystana bardzo ciekawa implementacja algorytmu Rotating Calipers zaproponowanego po raz pierwszy przez Godfrieda Toussaint'a. Inspiracja do implementacji algorytmu została zaczerpnięta z dyskusji znalezionej tutaj.

Bardzo ważną rolę w tej implementacji odgrywa wyznaczenie otoczki wypukłej. W tym celu korzystam z zaimplementowanego na zajęciach Algorytmu Grahama, który wyznacza otoczkę w czasie O(n*log(n)). W zasadzie jest to główne ograniczenie algorytmu Rotating Calipers, który nie wliczając w niego złożoności otoczki jest wykonywany w O(n).

Na początku należy zauważyć, że każdy prostokąt o najmniejszym polu/obwodzie będzie równoległy do przynajmniej jednej ściany otoczki wypukłej. Zamysł algorytmu Rotating Calipers polega na stworzeniu najmniejszego prostokąta zawierającego wszystkie punkty ze zbioru, którego jeden z boków jest równoległy do lrawędzi otoczki liniowej, a następnie "obracaniu go" zgodnie lub przeciwnie do ruchu wskazówek zegara, tak aby jeden z boków prostokąta cały czas był równoległy do którejś z krawędzi otoczki wypukłej. Należy następnie zmierzyć odpowiednio pole/obwód każdego z powstałych prostokątów i wybrać ten, który nas najbardziej interesuje.

Algorytm Rotating Calipers

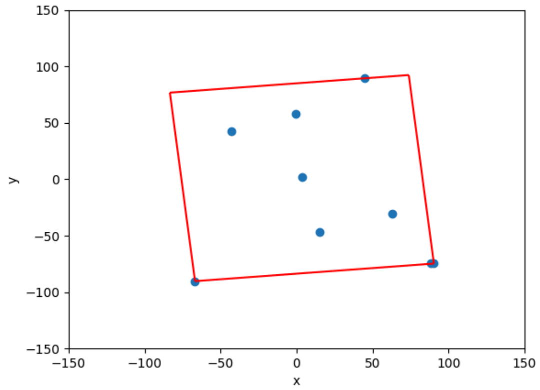
input: lista punktów P,
output: cztery punkty będące wierzchołkami prostokąta

1. Wyznaczam listę "edges" krawędzi otoczki wypukłej za pomocą Algorytmu Grahama
2. Wyznaczam listę "dir_vec" wektorów kierunkowych dla każdego elementu "edges"
3. Wyznaczam listę "norm_vec" zawierającą znormalizowane wektory z "dir_ver"
4. Wyznaczam listę "perp_vec" wektorów prostopadłych do wektorów z "norm_vec"
5. Dla każdej krawędzi z "edges" wyznaczam najbardziej wysunięty punkt na lewo/prawo względem niego, otrzymując listy ekstremów "minX" oraz "maxX" dla każdej krawędzi. Tzn. Dla i-tego punktu otoczki wypukłej wymnażam jego współrzędne przez wektor z "norm_vec" i dodaje do siebie.
6. Dla każdej krawędzi z "edges" wyznaczam najbardziej wysunięty punkt w górę/dół względem niego, otrzymując listy ekstremów "minY" oraz "maxY" dla każdej krawędzi. Tzn. Dla i-tego punktu otoczki wypukłej wymnażam jego współrzędne przez wektor z "perp_vec" i dodaje do siebie.

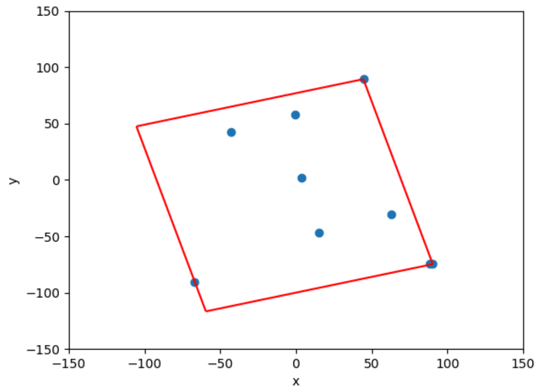
7. W efekcie dla każdej krawędzi otoczki mam najbardziej wysunięty punkt na prawo/lewo/górę/dół, zatem mogę z nich utworzyć prostokąt. Sprawdzam każdy taki prostokąt minimalizując jego pole/obwód oraz wybierając ten najbardziej mnie interesuje..
8. Jako, że ekstrema były wyznaczane względem krawędzi otoczki, otrzymane punkty należy jeszcze odpowiednio przetransformować. Tworzę wynikowe punkty w następujący sposób: Jeśli A to otrzymany punkt, a B to szukany, to: B.x = A.x * norm_vec.x + A.y * perp_vec.x, B.y = A.x * norm_vec.y + A.y * perp_vec.y

4.1. Efekt działania algorytmu

Poniżej przedstawiono efekt działania algorytmu dla przykładowych danych na wykresie 2 i wykresie 3.



Wykres 2. Prostokąt o najmniejszym obwodzie



Wykres 3. Prostokąt o najmniejszym polu

4.2. Porównanie czasów działania algorytmu

Algorytm jest najszybszy dla punktów na prostokącie, z tego powodu, że otoczka liniowa składa się tylko z 4 krawędzi, to samo dzieje się w przypadku losowych punktów, gdyż generowane są one z przedziału $(a, b)^2$ i przy dużej ich ilości otoczka wypukła też jest prostokątem. Najbardziej pesymistycznym przypadkiem są punkty na okręgu, gdyż wtedy otoczka wypukła składa się z największej liczby krawędzi i jest najwięcej potencjalnych prostokątów do przetestowania. Warto, także zwrócić uwagę, że w przypadku prostokąta znaczną część czasu wykonania zajmuje wywołanie Algorytmu Grahama. Przykładowo w ostatnim wierszu tabeli czas wykonania samego algorytmu Rotating Calipers bez wyznaczania otoczki to czas rzędu ułamków sekundy, natomiast w przypadku okręgu jest zupełnie na odwrót.

Typ punktów	Ilość punktów	Czas wykonania [s]
Losowo	100	0,001
	1000	0,001
	10000	0,140
	100000	1,668
	1000000	19,915
Na okręgu	100	0,004
	1000	0,313

	10000	33,570
	100000	-
	1000000	-
Na prostokącie	100	0,001
	1000	0,010
	10000	0,135
	100000	1,698
	1000000	22,024

Tabela 2. Czasy wykonania algorytmu Rotating Calipers

5. Wnioski

Oba algorytmy działają poprawnie i są szybkie. Jedynym problematycznym zbiorem dla algorytmu Rotating Calipers okazały się punkty zadane na okręgu, Warto zwrócić uwagę na kreatywną implementację Rotating Calipers, gdyż prawie każda udostępniona w Internecie implementacja wykorzystywała kosztowne funkcje trygonometryczne, a tutaj udało się tego uniknąć. Algorytm Rotating Calipers jest bardzo uniwersalny, inne jego implementacje pozwalają na wyznaczenie m. in. triangulacji wielokąta.