ALGORYTMY GEOMETRYCZNE

Dokumentacja do algorytmów wyznaczających otoczki wypukłe

AUTORZY

PAWEŁ JAROSZ MIKOŁAJ WNĘK

Spis Treści

1 Wstęp

Algorytmy zostały zaimplementowane w języku **Python** 3.9.5 za pomocą dostępnych bibliotek. Kod źródłowy znajduje się w repozytorium na **GitHub**(tutaj). Użyte biblioteki open source to:

- `matplotlib` w wersji 3.4.3
- `numpy` w wersji 1.21.2

Działanie kodu zostało przetestowane na komputerze z specyfikacją:

- **Windows 10** - wersja 2H22
- **CPU** - i5-7600k 3.80Ghz
- **RAM** - 32GB 3000Hz

Paczka składa się z dwóch folderów, *algorithms* zawierający implementacje algorytmów wyznaczających otoczki wypukłe, oraz *utils* zawierającego funkcje pomocnicze takie jak wizualizacje oraz funkcje testujące wydajność. Funkcje zaimplementowaliśmy w sposób `nazwa` - zwraca otoczkę bez elementów wizualizacji korzystając z algorytmu nazwa, natomiast `nazwaVis` zwraca listę scen interpretowanych przez funkcje wizualizującą.

2 Dokumentacja

2.1 Algorytmy znajdujące otoczkę

2.1.1 chan(points)

Argumentem jest lista `points` będąca typu `[[p1.x, p1.y], [p2.x, p2.y], ...]`. Funkcja zwraca listę punktów stanowiących otoczkę wypukłą w postaci analogicznej do `points`. Jest to algorytm Chana działający w czasie $O(n \log k)$, gdzie $n = \text{len}(\texttt{points})$ oraz k - liczba punktów w otoczce.

2.1.2 chanVis(points)

Argumentem jest lista `points` będąca typu `[[p1.x, p1.y], [p2.x, p2.y], ...]`. Funkcja zwraca listę obiektów typu `Scene`, które pozwalają wizualizować działanie algorytmu. Ta implementacja jest znacznie wolniejsza niż jej odpowiednik bez wizualizacji.

2.1.3 divideconquer(points)

Argumentem jest lista `points` będąca typu `[[p1.x, p1.y], [p2.x, p2.y], ...]`. Funkcja zwraca listę punktów stanowiących otoczkę wypukłą w postaci analogicznej do `points`. Jest to algorytm dziel i podbijaj działający w czasie $O(n \log n)$, gdzie $n = \text{len}(\texttt{points})$.

2.1.4 divideconquerVis(points)

Argumentem jest lista `points` będąca typu `[[p1.x, p1.y], [p2.x, p2.y], ...]`. Funkcja zwraca listę obiektów typu `Scene`, które pozwalają wizualizować działanie algorytmu. Ta implementacja jest znacznie wolniejsza niż jej odpowiednik bez wizualizacji. Kolor czarny - obecne otoczki, kolor czerwony - lewa otoczka z pary, niebieski - prawa otoczka, żółty - dolna styczna i zielony - prawa.

2.1.5 grahams(points)

Argumentem jest lista `points` będąca typu `[[p1.x, p1.y], [p2.x, p2.y], ...]`. Funkcja zwraca listę punktów stanowiących otoczkę wypukłą w postaci analogicznej do `points`. Jest to algorytm Grahamsa działający w czasie $O(n \log n)$, gdzie $n = \text{len}(\texttt{points})$.

2.1.6 grahamsVis(points)

Argumentem jest lista `points` będąca typu `[[p1.x, p1.y], [p2.x, p2.y], ...]`. Funkcja zwraca listę obiektów typu `Scene`, które pozwalają wizualizować działanie algorytmu. Ta implementacja jest znacznie wolniejsza niż jej odpowiednik bez wizualizacji. Kolor czerwony - obecna otoczka.

2.1.7 increment(points)

Argumentem jest lista `points` będąca typu `[[p1.x, p1.y], [p2.x, p2.y], ...]`. Funkcja zwraca listę punktów stanowiących otoczkę wypukłą w postaci analogicznej do `points`. Jest to algorytm przyrostowy działający w czasie $O(n \log n)$, gdzie $n = \text{len}(\texttt{points})$.

2.1.8 incrementVis(points)

Argumentem jest lista `points` będąca typu `[[p1.x, p1.y], [p2.x, p2.y], ...]`. Funkcja zwraca listę obiektów typu `Scene`, które pozwalają wizualizować działanie algorytmu. Ta implementacja jest znacznie wolniejsza niż jej odpowiednik bez wizualizacji. Kolor czarny - obecna otoczka, kolor czerwony i zielony proste łączące obecna otoczkę z rozpatrywanym punktem.

2.1.9 jarvis(points)

Argumentem jest lista `points` będąca typu `[[p1.x, p1.y], [p2.x, p2.y], ...]`. Funkcja zwraca listę punktów stanowiących otoczkę wypukłą w postaci analogicznej do `points`. Jest to algorytm Jarvisa działający w czasie $O(n^2)$, gdzie $n = \text{len}(\texttt{points})$.

2.1.10 jarvisVis(points)

Argumentem jest lista `points` będąca typu `[[p1.x, p1.y], [p2.x, p2.y], ...]`. Funkcja zwraca listę obiektów typu `Scene`, które pozwalają wizualizować działanie algorytmu. Ta implementacja jest znacznie wolniejsza niż jej odpowiednik bez wizualizacji. Kolor czerwony - obecna otoczka.

2.1.11 quickhull(points)

Argumentem jest lista `points` będąca typu `[[p1.x, p1.y], [p2.x, p2.y], ...]`. Funkcja zwraca listę punktów stanowiących otoczkę wypukłą w postaci analogicznej do `points`. Jest to algorytm rekurencyjny quickhull działający w czasie $O(n \log n)$, gdzie $n = \text{len}(\texttt{points})$.

2.1.12 quickhullVis(points)

Argumentem jest lista `points` będąca typu `[[p1.x, p1.y], [p2.x, p2.y], ...]`. Funkcja zwraca listę obiektów typu `Scene`, które pozwalają wizualizować działanie algorytmu. Ta implementacja jest znacznie wolniejsza niż jej odpowiednik bez wizualizacji. Kolor czerwony - obecna otoczka, kolor szary - poprzednio dodane przekątne.

2.1.13 upperlower(points)

Argumentem jest lista `points` będąca typu `[[p1.x, p1.y], [p2.x, p2.y], ...]`. Funkcja zwraca listę punktów stanowiących otoczkę wypukłą w postaci analogicznej do `points`. Jest to algorytm łączący otoczkę górną i dolną działający w czasie $\mathrm{O}(n \log n)$, gdzie $n = \mathrm{len}($`points`$)$.

2.1.14 upperlowerVis(points)

Argumentem jest lista `points` będąca typu `[[p1.x, p1.y], [p2.x, p2.y], ...]`. Funkcja zwraca listę obiektów typu `Scene`, które pozwalają wizualizować działanie algorytmu. Ta implementacja jest znacznie wolniejsza niż jej odpowiednik bez wizualizacji. Kolor czerwony - dolna otoczka, kolor zielony - górna otoczka.

2.2 Funkcje *utils*

2.2.1 helpers.py

Zawiera funkcje pomocnicze wykorzystywane przez większość algorytmów.

- `det(a,b,c)` - zwraca wyznacznik `(a[0]-c[0])*(b[1]-c[1])-(b[0]-c[0])*(a[1]-c[1])`, gdzie `a,b,c` to dwuelementowe listy zawierające współrzędne punktów.
- `orient(a,b,c)` - zwraca 1, -1 oraz 0 w zależności od znaku `det(a,b,c)`, analogicznie jak funkcja sgn(x). Zero zwracane jest jeżeli wyznacznik jest mniejszy niż zmienna globalna `EPS` na wartość bezwzględną.
- `lengthSquared(a,b)` - zwraca odległość miedzy punktami `a` i `b` (listy dwuelementowe) do kwadratu.
- `length(a,b)` - zwraca odległość miedzy punktami `a` i `b` (listy dwuelementowe).

2.2.2 gen.py

Zawiera funkcje generujące zadane zbiory punktów.

- `genUniformRectangle(xs,xe,ys,ye,s)` - zwraca zbiór s punktów w postaci listy list `[p.x, p.y]`. Punkty spełniają:

  $$x_s \leq p_x \leq x_e \ \wedge \ y_s \leq p_y \leq y_e.$$

- `genUniformCircle(x,y,r,s)` - zwraca zbiór s punktów w postaci listy list `[p.x, p.y]`. Punkty spełniają:

  $$(p_x - x)^2 + (p_y - y)^2 = r^2.$$

- `genUniformOnRectangle(xs,xe,ys,ye,s)` - zwraca zbiór s punktów w postaci listy list `[p.x, p.y]`. Punkty spełniają:

  $$[(p_x = x_s \vee p_x = x_e) \wedge (y_s \leq p_y \leq y_e)] \vee [(p_y = y_s \vee p_y = y_e) \wedge (x_s \leq p_x \leq x_e)]$$

- `genUniformOnSquare(side,s,sd)` - zwraca zbiór $2(s + sd)$ punktów w postaci listy list `[p.x, p.y]`. Punkty zawierają się na przekątnych kwadratu oraz bokach zawierających się w osiach OX i OY oraz długości boku `side`.

2.2.3 benchmark.py

Zawiera funkcje testującą wydajność:

- `benchmark(hullAlgorithm,timeOfBenchmark,generator,*args)` -
  Funkcja przyjmuje algorytm tworzący otoczkę, czas po jakim chcemy przeprowadzić benchmark w sekundach oraz argumenty generatora. Dzięki temu unikniemy sytuacji zadania za dużego zbioru danych. Wyniki zostają uśrednione oraz wypisane na ekran.

2.2.4 viz.py

Głównie składa się z udostępnionego narzędzia graficznego. Ponadto zawiera naszeptujące funkcje wizualizujące:

- `plotPoints(points)` - Funkcja generująca graficzna reprezentacje zbioru.
- `plotHull(points,hull)` - Funkcja generująca graficzna reprezentacje otoczki bez wizualizacji. Przyjmuje argumenty `points`, oraz `hull`. `hull` musi być podane zgodnie lub przeciwnie do wskazówek zegara aby wygenerować spójny wykres.
- `visHull(hullAlgorithmVis,points)` - Funkcja generująca wizualizacje działania algorytmu. Przekazana **musi** zostać funkcja z sufixem *Vis*, aby wygenerowanie wizualizacji było możliwe.

2.2.5 save.py

Plik zawiera dwie funkcje, które pozwalają na zapisywanie listy do pliku.

- `saveList(points,filename)` - Funkcja zapisująca listę `points` w pliku o nazwie (lub ścieżce) `filename`. Funkcja sama tworzy plik jeżeli takowy nie istnieje oraz nadpisuje już istniejący w przeciwnym wypadku.
- `readList(filename)` - Funkcja odczytująca listę z pliku zapisanego przy pomocy `saveList` oraz zwraca listę odczytanych punktów.

2.2.6 convexhulls.py

Jest to główny plik importujący funkcje, który pozwala na użycie nagłówka:

```python
from convexhulls import *
```

aby skorzystać z zaimplementowanych przez nas algorytmów.

3 Użycie

3.1 Generowanie punktów

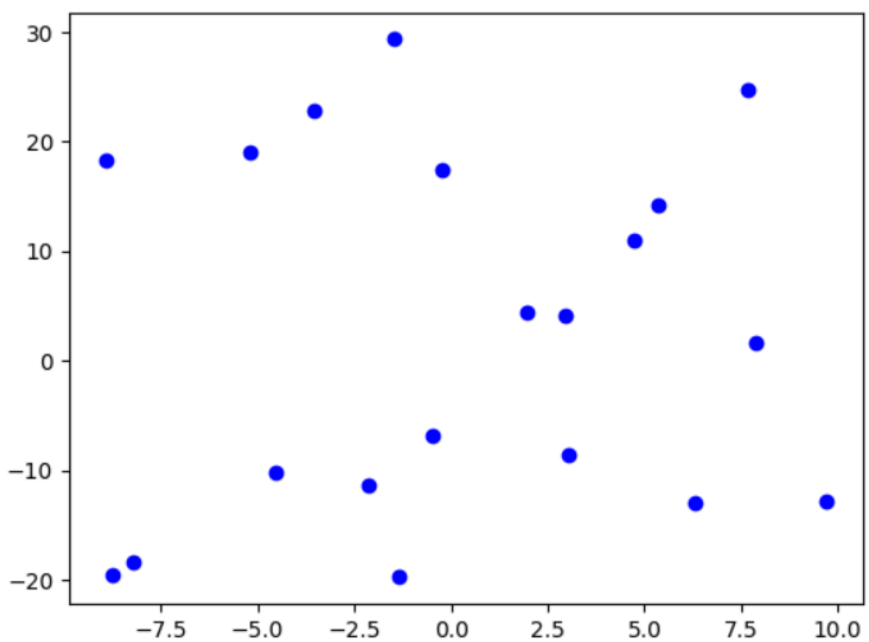
Przykład generowania 4 zbiorów punktów:

```
points1 = genUniformRectangle(-10, 10, -20, 30, 20)
points2 = genUniformCircle(10, 10, 10, 100)
points3 = genUniformOnRectangle(-10, -10, 20, 30, 100)
points4 = genUniformOnSquare(10, 20, 20)
```

Wizualizacje zbioru możemy dokonać poprzez:

```
plotPoints(points1)
```

co otwiera okno `matplotlib` z wizualizacją jak w *Wizualizacji 1*.



Wizualizacja 1: Zbiór punktów.

3.2 Wyznaczanie otoczki

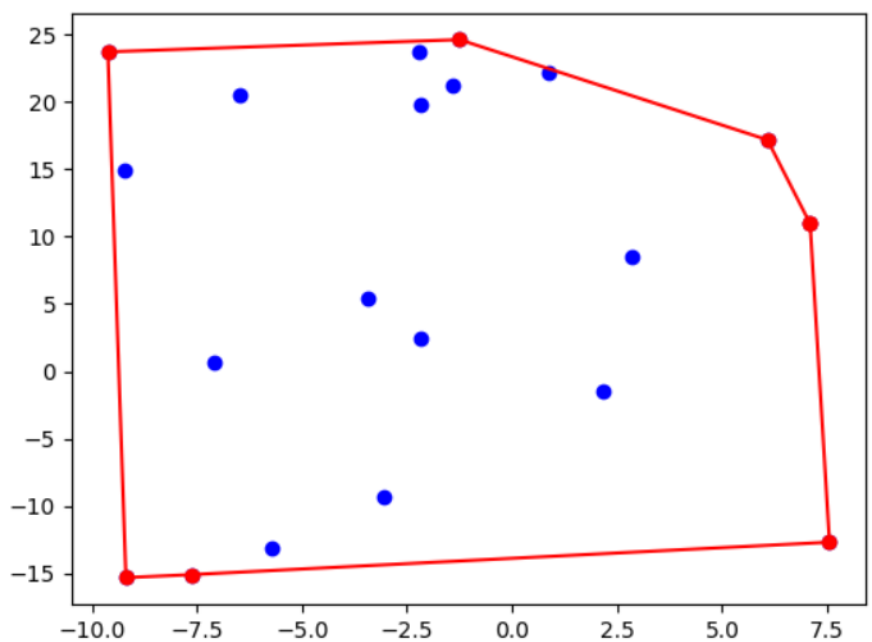
Punkty tworzące otoczkę możemy znaleźć za pomocą (w tym przypadku korzystając z metody `increment`):

```
points = genUniformRectangle(-10, 10, -20, 30, 20)
hull = increment(points)
```

Statyczną wizualizacje wywołujemy za pomocą:

```
points = genUniformRectangle(-10, 10, -20, 30, 20)
hull = increment(points)
plotHull(points, hull)
```

co otwiera okno `matplotlib` z wizualizacją jak w *Wizualizacji 2.*



Wizualizacja 2: Otoczka wypukła.

Ponadto funkcja wypisuje ilość punktów zawartych w otoczce.

```
Convex hull contains 7 points.
```

3.3 Testy wydajności

Tester posiada argument określający czas przeprowadzanej symulacji, tak, aby w danym czasie przeprowadzić największą możliwą ilość testów oraz uśrednić otrzymane wyniki (dokładny opis **tutaj**). Użycie:

```
benchmark(increment, 2, genUniformRectangle, -10, 10, -10, 10, 10000)
```

Test będzie się wykonywał przez co najmniej 2sec. Funkcja wyświetla:

```
TESTS FINISHED. ------------------------------------------------
Executed 24 tests.
Algorithm used: increment
Generator used: genUniformRectangle
Sample size: 10000
Average time per test: 0.07890247305234273s
```

3.4 Zapisywanie do pliku

Korzystając z funkcji z pliku save.py możemy przy jednym uruchomieniu programu zapisać dany zbiór w pliku o nazwie *points1*:

```
points1 = genUniformRectangle(-10, 10, -20, 30, 20)
saveList(points1, "points1")
```

Natomiast przy kolejnym odczytać go z wcześniej utworzonego pliku:

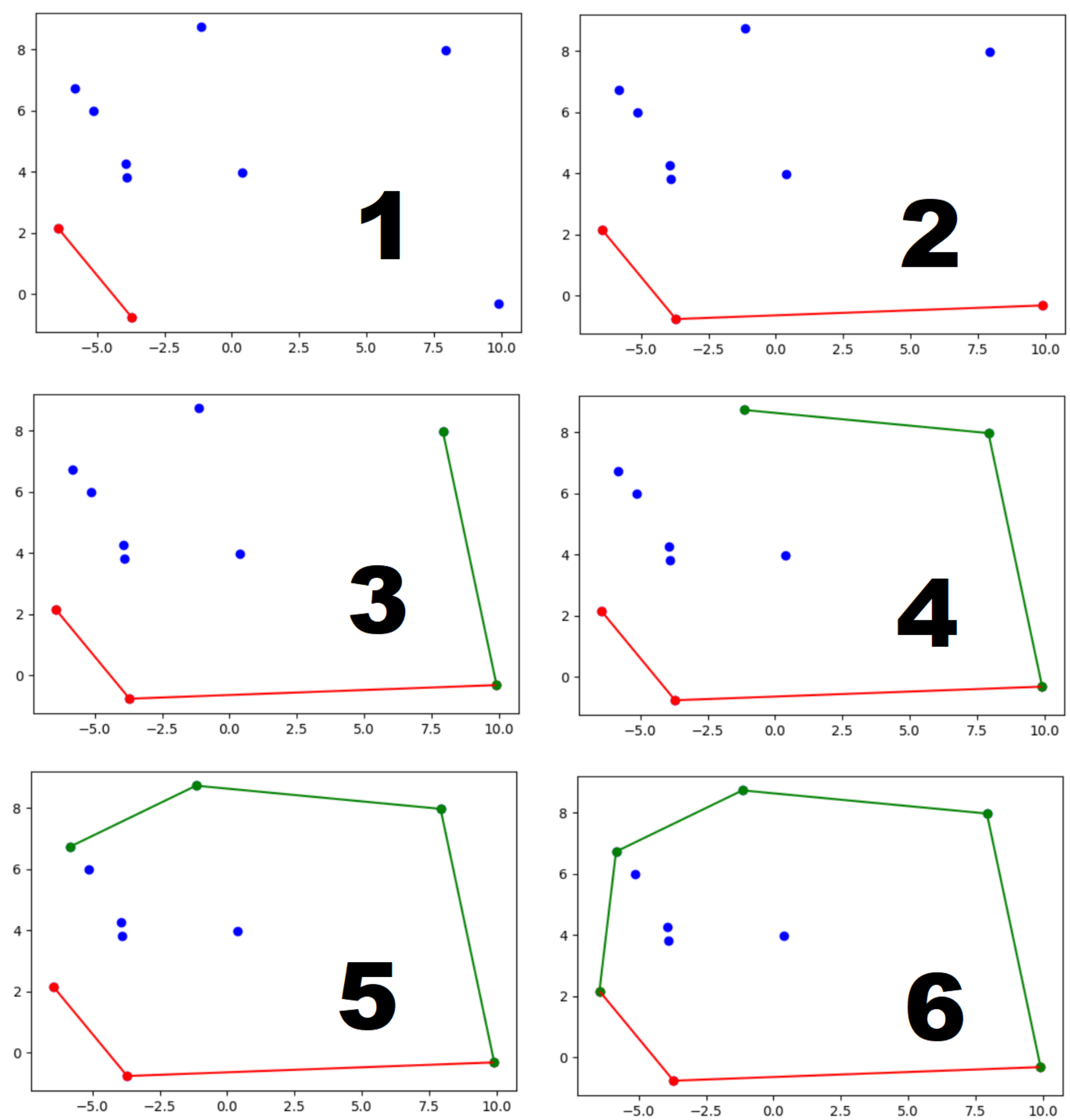
```
points1 = readList("points1")
```

3.5 Wizualizacja

Działanie danego algorytmu możemy wizualizować za pomocą:

```
points = genUniformRectangle(-10, 10, -20, 30, 20)
visHull(incrementVis, points)
```

co otwiera okno `matplotlib` z wizualizacją, której klatki zmieniają się przy klikaniu 'następny', 'poprzedni'. Poniżej zawarte są wybrane klatki z wizualizacji algorytmy `upperlowerVis` dla zbioru wygenerowanego przy użyciu `genUniformRectangle(-10, 10, -10, 10, 10)`.



Wizualizacja 3: Otoczka górna i dolna.

Punkty nienależące do otoczki zaznaczone są kolorem niebieski, natomiast pozostałymi kolorami zaznaczone są różne elementy wyznaczanych otoczek.

4 Sprawozdanie

4.1 Wstęp

Celem projektu było zaimplementowanie 7 algorytmów wyznaczających otoczki wypukłe wraz z wizualizacjami. Zaimplementowaliśmy algorytmy **Jarvis'a, Grahams'a, Chan'a, przyrostowy, dziel i podbijaj, quickhull oraz górna i dolna otoczka**. Dodatkowo utworzyliśmy pliki zawierające funkcje pomocnicze, które pozwalają na wizualizacje, zapisywanie oraz dodawanie punktów przez użytkownika. Nasz moduł posiada także możliwość sprawdzenia szybkości działania algorytmu na zadanych zbiorach danych.

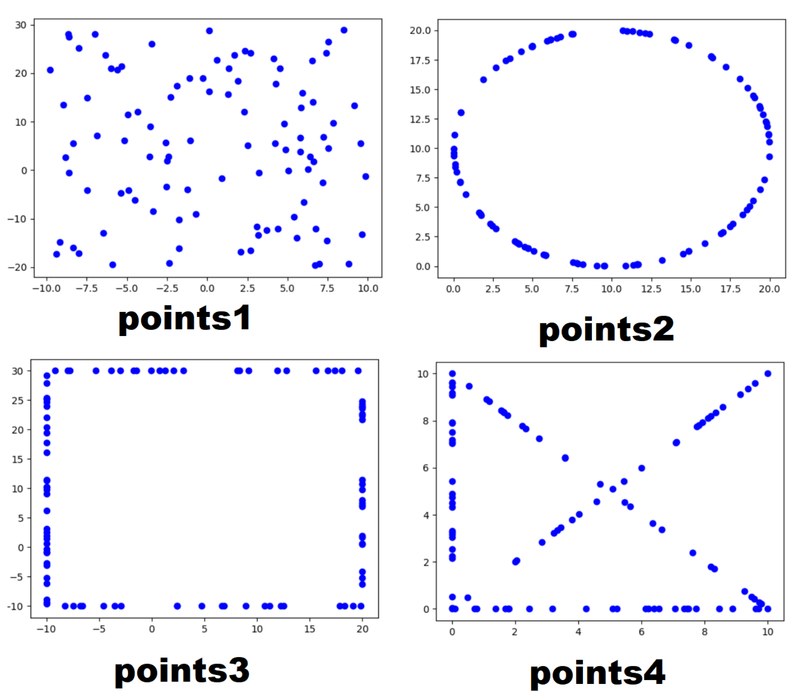
4.2 Generowanie punktów

Dokładny opis funkcji generujących zbiory punktów zawarty jest (**tutaj**) w dokumentacji. Poniżej zawarte są przykładowe zbiory, które pokazują charakterystykę generowanych zbiorów. Kod generujący zbiory wraz z wizualizacjami:

```
points1 = genUniformRectangle(-10, 10, -20, 30, 100)
points2 = genUniformCircle(10, 10, 10, 100)
points3 = genUniformOnRectangle(-10, -10, 20, 30, 100)
points4 = genUniformOnSquare(10, 25, 25)

plotPoints(points1)
plotPoints(points2)
plotPoints(points3)
plotPoints(points4)
```

Wizualizacje:



Wizualizacja 4: Zbiory punktów.

4.3 Opis implementacji

4.3.1 Jarvis

Złożoność pesymistyczna: O(n^2).
Algorytm `jarvis()` opiera się na funkcji `reduce` z modułu `functools`. Dzięki niej możemy znaleźć element minimalny z listy pod względem przekazanego komparatora, w tym przypadku będzie to kąt pomiędzy osią OX, a prostą przechodzącą przez punkt ostatnio dodany i punkt rozpatrzany. Algorytm trwa, aż punkt który powinien zostać dodany jako kolejny, jest pierwszym punktem otoczki. Każdy taki krok posiada złożoność O(n), co dla otoczek pesymistycznych (o liczebności n) będzie skutkować złożonością O(n^2), natomiast w przypadku, gdzie punktów będzie k, O(nk).

4.3.2 Grahams

Złożoność pesymistyczna: O($n \log n$).
Algorytm `grahams()` opiera się na wpierw posortowaniu punktów po kącie w czasie O($n \log n$), który jest tworzony miedzy osią OX oraz prostą przechodzącą przez punkt startowy(taki który posiada najmniejszą współrzędną y-ową oraz w drugiej kolejności x-ową) i rozpatrzany punkt. Możemy to osiągnąć za pomocą funkcji `cmp_to_key` z modułu `functools` oraz użyć funkcji `points.sort()` z odpowiednim komparatorem. Następnie dodajemy pierwsze dwa punkty do otoczki oraz przetwarzamy wszystkie pozostałe dodając je do otoczki oraz sprawdzając czy nie utworzyliśmy kąta wklęsłego(patrząc od wewnątrz zbioru) przy pomocy wyznacznika. W takim wypadku punkt usuwany jest z otoczki. Jako, że liniowo przechodzimy po otoczce, to krok ten zajmuje tylko O(n), co skutkuję złożonością całkowitą O($n \log n$) + O(n) = O($n \log n$).

4.3.3 Quikchull

Złożoność pesymistyczna: O($n \log n$).
Zaimplementowaliśmy rekurencyjną wersje algorytmu `quickhull()`. Zaczynamy poprzez wybranie punktów skrajnych pod względem współrzędnej x-owej. Następnie pozostałe punkty dzielimy na dwa zbiory zależnie, od położenia względem prostej przechodzącej, przez znalezione wcześniej punkty. Zbiory wraz z punktami przekazujemy do rekurencyjnej funkcji. Znajduje ona punkt położony najdalej od prostej przechodzącej przez przekazane dwa punkty oraz wyznacza kolejne dwa zbiory znajdujące się na zewnątrz trójkąta łączącego trzy punkty. Rekurencja wykonuje się, aż nie zostanie przekazany zbiór pusty, wtedy podział nie jest wykonywany. Jako, że zawsze dokonujemy dzielenia na 2 dopełniające się problemy, to złożoność tego algorytmu wynosi O($n \log n$).

4.3.4 Dolna-Górna

Złożoność pesymistyczna: O($n \log n$).
Algorytm `upperlower()` opiera się na znalezieniu otoczki ograniczającej zbiór od góry, od doły oraz połączeniu ich. Na początku sortujemy punkty w pierwszej kolejności po współrzędnych x-owych, a w drugiej po y-owych. Następnie przechodząc po wszystkich punktach, analogicznie do algorytmu Grahamsa, dodajemy punkty, które nie tworzą kątów wklęsłych patrząc od wewnątrz zbioru. To samo robimy przechodząc od tyłu po zbiorze punktów, co pozwala wyznaczyć górną otoczkę. Na koniec wystarczy połączyć obie otoczki.

4.3.5 Przyrostowy

Algorytm tworzy kolejkę z posortowanymi wg x malejąco punktami. Następnie inicjalizuję listę reprezentującą otoczkę wyjmując 2 pierwsze punkty z kolejki. Następnie , w pętli - aż do opróżnienia kolejki, powtarzane są następujące czynności:

- Wyjmij pierwszy punkt z kolejki.
- Znajdź dla niego górny i dolny punkt styczności na otoczce (binary search).
- Zamień na liście otoczki łańcuch [dolny punkt styczności + 1] - [górny punkt styczności - 1], przeciwnie z ruchem wskazówek zegara, na punkt wyjęty z kolejki

4.3.6 Chan

Złożoność pesymistyczna: O($n \log h$).
Algorytm zakłada znajomość liczby punktów h na otoczce. Punkty dzielone są na podzbiory o liczebności maksymalnie h. Dla każdego takiego podzbioru algorytmem grahama wyznaczana jest jego otoczka. Na koniec zaczynając od punktu skrajnego stosujemy algorytm analogiczny do Jarvisa, z tym ze możemy rozpatrywać po jednym punkcie z każdej wyznaczonej otoczki (punkcie styczności). W praktyce raczej nie znamy h wiec algorytm wywołujemy dla 'zgadniętego' $h = 2^{2^t}$ gdzie t=1,2...

4.3.7 Dziel i podbijaj

Złożoność pesymistyczna: O($n \log n$).
Algorytm sortuje punkty wg współrzędnej x punktów. Jak w typowym algorytmie dziel i zwyciężaj, problem znalezienia otoczki wypukłej całego zbioru dzielony jest na podproblemy w następujący sposób: zbiór punktów dzielimy kolejno na połowy wg współrzędnej x rekurencyjnie, aż otrzymamy zbiory o liczbie punktów mniejszej od 8. Dla nich algorytmem brute force wyznaczana jest otoczka wypukła. Następnie każde dwie otoczki, będące otoczkami "połówek" tego samego zbioru, scalamy w jedną, poprzez znalezienie dolnej i górnej stycznej do obu zbiorów i odrzucenie łańcuchów leżących wewnętrznie między punktami styczności danej otoczki. Proces powtarzany jest aż finalnie otoczki dwóch połówek wyjściowego zbioru zostaną scalone w poszukiwaną otoczkę.

4.4 Test wydajności

Specyfikacja komputera na którym przeprowadzona zostały testy znajduje się **tutaj**. Każdy algorytm został przetestowany za pomocą funkcji:

```
benchmark(algName, 5, genName, *args)
```

gdzie `algName` - nazwa testowanego algorytmu, 5 - czas w sekundach, dla którego przeprowadzona była symulacja, `genName` - nazwa funkcji generującej zbiór wraz z `args` argumentami. Poniżej w tabeli zawarliśmy wyniki testów.

Metoda \ Liczność	10^2	10^3	10^4	10^5	10^6
`chan()`	1ms	15ms	160ms	1.6s	17s
`jarvis()`	0.6ms	9ms	130ms	1.6s	18s
`grahams()`	0.3ms	5ms	65ms	0.88s	11s
`quickhull()`	0.2ms	2ms	17ms	0.18s	1.8s
`divideconquer()`	1ms	12ms	120ms	1.5s	13s
`upperlower()`	0.1ms	1ms	17ms	0.24s	3.13s
`increment()`	0.4ms	5ms	73ms	1.69s	272s

Tabela 1: Czasy dla generatora `genUniformRectangle()`.

Metoda \ Liczność	10^2	10^3	10^4	10^5	10^6
`chan()`	3ms	54ms	860ms	23s	-
`jarvis()`	5ms	550ms	50s	-	-
`grahams()`	0.3ms	5ms	65ms	0.83s	11s
`quickhull()`	0.5ms	9ms	12ms	1.5s	20s
`divideconquer()`	1ms	10ms	125ms	1.6s	-
`upperlower()`	0.1ms	1ms	15ms	0.23s	2.9s
`increment()`	1ms	110ms	10s	-	-

Tabela 2: Czasy dla generatora `genUniformCircle()`.

Metoda \ Liczność	10^2	10^3	10^4	10^5	10^6
`chan()`	x	x	x	x	x
`jarvis()`	0.5ms	5ms	50ms	0.51s	5.2s
`grahams()`	0.3ms	5ms	81ms	1.07s	14s
`quickhull()`	0.2ms	2ms	27ms	0.27s	2.7s
`divideconquer()`	2ms	11ms	120ms	1.7s	14s
`upperlower()`	0.1ms	1ms	19ms	0.25s	3.1s
`increment()`	0.3ms	3ms	37ms	0.91s	158s

Tabela 3: Czasy dla generatora `genUniformOnRectangle()`.

Metoda \ Liczność	10^2	10^3	10^4	10^5	10^6
`chan()`	0.4ms	4ms	37ms	0.37s	3.7s
`jarvis()`	0.3ms	3ms	31ms	0.28s	2.8s
`grahams()`	0.4ms	7ms	100ms	1.3s	17s
`quickhull()`	0.1ms	1ms	14ms	0.14s	1.4s
`divideconquer()`	2ms	11ms	110ms	1.6s	14s
`upperlower()`	0.1ms	1ms	16ms	0.21s	2.87s
`increment()`	0.2ms	2ms	33ms	0.81s	-

Tabela 4: Czasy dla generatora `genUniformOnSquare()`.

5 Wnioski

Jak widzimy, najlepsze czasy osiąga odpowiednio:

- `genUniformRectangle()` - `quickhull()`
- `genUniformCircle()` - `upperlower()`
- `genUniformOnRectangle()` - `quickhull()`
- `genUniformOnSquare()` - `quickhull()`

Algorytmy takie jak `jarvis()`, `grahams()`, `upperlower()`, `quickhull()` były stosunkowo proste do zaimplementowania ze względu na proste oraz mało złożone operacje. Ich prędkość w dużej mierze zależy od stosunkowo małej ilości nowych struktur, oraz braku manipulacji na fragemntach list. Przeciwnie do nich `increment()`, `chan()` i `divideconquer()` charakteryzował dosyć duży poziom trudności implementacji. Są to algorytmu opierające się na manipulacji na listach, co przekłada się na dłuższy czas działania i obniżoną wydajność. Ponadto problemy sprawiał *binary search* potrzebny do implementacji `increment()` oraz `chan()`, dla punktów współliniowych. Dodatkowym kłopotem był relatywny brak przykładowych (i działających!) implementacji w innych językach dla algorytmów z bardziej skomplikowanych. Z tego powodu nie udało nam się zaimplementować algorytmu chana, tak aby działał dla `genUniformOnRectangle()`.