

DOKUMENTACJA PROJEKTU
Otoczka wypukła dla zbioru punktów w
przestrzeni dwuwymiarowej

Algorytmy geometryczne
Informatyka 3. sem., 1. st., AGH

Kacper Kafara
Łukasz Czarniecki

Spis treści

1	Informacje techniczne	6
1.1	Budowa programu	6
1.1.1	Moduł <i>lib</i>	6
1.1.2	Moduł <i>pure</i>	16
1.1.3	Moduł <i>vis</i>	17
1.2	Wymagania techniczne	17
1.3	Korzystanie z programu	17
1.3.1	Uruchomienie programu	17
2	Oznaczenia i definicje	17
3	Problem	18
4	Algorytmy	18
4.1	Algorytm Grahama	18
4.1.1	Opis działania	18
4.1.2	Szczegóły	19
4.1.3	Złożoność	19
4.1.4	Kod	19
4.2	Algorytm Jarvisa	20
4.2.1	Opis działania	20
4.2.2	Szczegóły	20
4.2.3	Złożoność	21
4.2.4	Kod	21
4.3	Algorytm górna-dolna	22
4.3.1	Opis działania	22
4.3.2	Złożoność	22
4.3.3	Kod	22
4.4	Algorytm przyrostowy	23
4.4.1	Opis działania	23
4.4.2	Szczegóły	24
4.4.3	Złożoność	24
4.4.4	Kod	24
4.5	Algorytm dziel i zwyciężaj	26
4.5.1	Opis działania	26
4.5.2	Szczegóły	26
4.5.3	Złożoność	27
4.5.4	Kod	27
4.6	Algorytm Chana	30
4.6.1	Opis działania	30
4.6.2	Szczegóły	31
4.6.3	Złożoność	31
4.6.4	Kod	32

4.7	Algorytm QuickHull	34
4.7.1	Opis działania	34
4.7.2	Szczegóły	35
4.7.3	Złożoność	35
4.7.4	Kod	35
5	Wydażność algorytmów	36
5.1	Algorytm Grahama	37
5.2	Porównanie z przewidywaniami teoretycznymi	37
5.3	Algorytm górna-dolna	38
5.4	Porównanie z przewidywaniami teoretycznymi	38
5.5	Algorytm Chana	40
5.6	Porównanie z przewidywaniami teoretycznymi	40
5.7	Algorytm QuickHull	40
5.8	Porównanie z przewidywaniami teoretycznymi	40
5.9	Algorytm dziel i zwyciężaj	43
5.10	Porównanie z przewidywaniami teoretycznymi	43
5.11	Algorytm przyrostowy	43
5.12	Porównanie z przewidywaniami teoretycznymi	44
5.13	Algorytm Jarvisa	44
5.14	Porównanie z przewidywaniami teoretycznymi	44
6	Porównanie algorytmów	46
6.1	Zbiór typu A	47
6.2	Zbiór typu B	47
6.3	Zbiór typu C	47
6.4	Zbiór typu D	47
7	Bibliografia	47

Spis rysunków

1	Zbiór typu A, algorytm Grahama	37
2	Zbiór typu B, algorytm Grahama	37
3	Zbiór typu C, algorytm Grahama	38
4	Zbiór typu D, algorytm Grahama	38
5	Zbiór typu A, algorytm górna-dolna	39
6	Zbiór typu B, algorytm górna-dolna	39
7	Zbiór typu C, algorytm górna-dolna	39
8	Zbiór typu D, algorytm górna-dolna	39
9	Zbiór typu A, algorytm Chana	41
10	Zbiór typu B, algorytm Chana	41
11	Zbiór typu C, algorytm Chana	41
12	Zbiór typu D, algorytm Chana	41

13	Zbiór typu A, algorytm QuickHull	42
14	Zbiór typu B, algorytm QuickHull	42
15	Zbiór typu C, algorytm QuickHull	42
16	Zbiór typu D, algorytm QuickHull	42
17	Zbiór typu A, algorytm dziel i zwyciężaj	43
18	Zbiór typu B, algorytm dziel i zwyciężaj	43
19	Zbiór typu C, algorytm dziel i zwyciężaj	44
20	Zbiór typu D, algorytm dziel i zwyciężaj	44
21	Zbiór typu A, algorytm przyrostowy	45
22	Zbiór typu B, algorytm przyrostowy	45
23	Zbiór typu C, algorytm przyrostowy	45
24	Zbiór typu D, algorytm przyrostowy	45
25	Zbiór typu A, algorytm Jarvisa	46
26	Zbiór typu B, algorytm Jarvisa	46
27	Zbiór typu C, algorytm Jarvisa	47
28	Zbiór typu D, algorytm Jarvisa	47
29	Zbiór typu A, zestawienie	48
30	Zbiór typu A, zestawienie bez Jarvisa i Chana	48
31	Zbiór typu B, zestawienie	48
32	Zbiór typu B, zestawienie bez Jarvisa	48
33	Zbiór typu C, zestawienie	49
34	Zbiór typu C, zestawienie bez Chana	49
35	Zbiór typu D, zestawienie	50
36	Zbiór typu D, zestawienie bez Chana	50

Spis tablic

1	Czas wykonania algorytmu Grahama w zależności od typu zbioru testowego oraz mocy zbioru punktów.	37
2	Czas wykonania algorytmu górna-dolna w zależności od typu zbioru testowego oraz mocy zbioru punktów.	38
3	Czas wykonania algorytmu Chana w zależności od typu zbioru testowego oraz mocy zbioru punktów.	40
4	Czas wykonania algorytmu QuickHull w zależności od typu zbioru testowego oraz mocy zbioru punktów.	41
5	Czas wykonania algorytmu dziel i zwyciężaj w zależności od typu zbioru testowego oraz mocy zbioru punktów.	43
6	Czas wykonania algorytmu przyrostowego w zależności od typu zbioru testowego oraz mocy zbioru punktów.	44
7	Czas wykonania algorytmu Jarvisa w zależności od typu zbioru testowego oraz mocy zbioru punktów.	46
8	Czasy wykonania poszczególnych algorytmów dla zbioru typu A przy różnych mocach zbiorów punktów.	47

9	Czasy wykonania poszczególnych algorytmów dla zbioru typu B przy różnych mocach zbiorów punktów.	48
10	Czasy wykonania poszczególnych algorytmów dla zbioru typu C przy różnych mocach zbiorów punktów.	49
11	Czasy wykonania poszczególnych algorytmów dla zbioru typu D przy różnych mocach zbiorów punktów.	49

1 Informacje techniczne

1.1 Budowa programu

Program złożony jest z następujących części:

- *lib* – moduł biblioteczny – zawiera zbiór pomocniczych funkcji i struktur danych wykorzystywanych przez algorytmy
- *pure* – moduł z algorytmami w *czystej postaci* tj. nie posiadające części wizualizacyjnej
- *vis* – moduł z algorytmami wraz z kodem odpowiadającym za wizualizację
- *kafara_czarniecki_program.ipynb* – główny plik programu, notebook z prezentacją algorytmów

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* – funkcje generujące zbiory punktów różnych typów
3. *sorting.py* – implementację iteracyjnej wersji algorytmu *QuickSort* wykorzystywaną w algorytmie Grahama
4. *stack.py* – klasę implementującą *stos*
5. *util.py* – szereg funkcji pomocniczych wykorzystywanych przez zaimplementowane algorytmy
6. *mytypes.py* – definicje typów stworzone w celu zwiększenia czytelności kodu.
7. *timemeasure.py* – funkcje do pomiaru czasu wykonania algorytmów
8. *tangent.py* – metoda wyznaczająca styczną (punkt styczności) do wielokąta, przechodzącą przez zadany punkt

Kod modułu *mytypes*

```
1 from typing import Tuple, List
2
3 Point = Tuple[float, float]
4
5 Segment = Tuple[Point, Point]
6
7 ListOfPoints = List[Point]
8
9 ListOfSegments = List[Segment]
```

Kod modułu *getrand*

```
1 import numpy as np
2 from typing import Tuple
3 from numpy import random
4
5 def dist(point_a, point_b):
6     return np.sqrt((point_a[0] - point_b[0])** 2 + (point_a[1] - point_b
7     [1])**2)
8
9 def rand_seg_point(point_a, point_b):
10     rng = np.random.default_rng()
11
12     point = np.empty(shape=2)
13
14     if point_a[0] == point_b[0]:
15         min_y = min(point_a[1], point_b[1])
16         max_y = max(point_a[1], point_b[1])
17         point[0] = point_a[0]
18         point[1] = rng.random() * (max_y - min_y) + min_y
19
20     else:
21         min_x = min(point_a[0], point_b[0])
22         max_x = max(point_a[0], point_b[0])
23
24         a = (point_b[1] - point_a[1]) / (point_b[0] - point_a[0])
25         b = (point_a[1] - a * point_a[0])
26
27         param_t = rng.random() * (max_x - min_x) + min_x
28         point[0] = param_t
29         point[1] = a * param_t + b
30
31     return point
32
33 def rand_rect_points(n: int, vertices):
34     rng = np.random.default_rng()
35     A, B, C, D = vertices[0], vertices[1], vertices[2], vertices[3]
36     len_AB = dist(A, B)
37     len_BC = dist(B, C)
38
39     points = np.empty(shape=(n,2))
```

```

40
41     for i in range(n):
42         side = rng.random(1) * 2 * (len_AB + len_BC)
43
44         if side < len_BC:
45             points[i] = rand_seg_point(A, D)
46         elif side < 2 * len_BC:
47             points[i] = rand_seg_point(B, C)
48         elif side < 2 * len_BC + len_AB:
49             points[i] = rand_seg_point(A, B)
50         else:
51             points[i] = rand_seg_point(C, D)
52
53     return points
54
55 def rand_in_range( low: float = 0,
56                  high: float = 0,
57                  data_type: str = 'float64') -> np.array:
58     rng = random.default_rng()
59
60     return np.array((rng.random(1) * (high - low) + low), dtype=data_type
61 )
62
63 def rand_arr(n: int,
64            low: float,
65            high: float,
66            data_type: str = 'float64') -> np.array:
67
68     rg = np.random.default_rng()
69
70     return np.array( (rg.random(n) * (high - low) + low), dtype=data_type
71 )
72
73 def rand_point2_set(n: int,
74                   low: float = 0,
75                   high: float = 1,
76                   data_type: str = 'float64') -> np.array:
77     return np.array(np.random.random((n, 2)) * (high - low) + low, dtype=
78 data_type)
79
80 def rand_circle_points( n: int = 10,
81                       x: float = 0,
82                       y: float = 0,
83                       r: float = 1,
84                       data_type: str = 'float64') -> np.array:
85     rng = np.random.default_rng()
86
87     circle = np.empty(n * 2, dtype=data_type).reshape(n, 2)
88
89     for i in range(n):
90         rn = rng.random() * 2 * np.pi
91         circle[i][0] = r * np.cos(rn) + x
92         circle[i][1] = r * np.sin(rn) + y

```



```
91
92     return circle
```

Kod modułu *sorting*

```
1  import os
2  from sys import path
3  path.append(os.path.dirname(os.path.realpath(__file__)) + "../")
4
5  from random import randint
6  from typing import Callable, List
7  from lib.stack import Stack
8
9
10 def partition_cmp(arr: List, p: int, q: int, cmp: Callable = lambda x, y:
11     x <= y) -> int:
12     i = p - 1
13
14     pivot_idx = randint(p, q)
15     pivot = arr[pivot_idx]
16
17     # zamieniamy miejscami pivot z ostatnim elementem
18     arr[pivot_idx], arr[q] = arr[q], arr[pivot_idx]
19
20     for j in range(p, q):
21         if cmp(arr[j], pivot):
22             i += 1
23             arr[i], arr[j] = arr[j], arr[i]
24
25     arr[i + 1], arr[q] = arr[q], arr[i + 1]
26
27     return i + 1
28
29 def qsort_iterative(arr: List, cmp: Callable = lambda x, y: x <= y):
30     stack = Stack(len(arr))
31
32     stack.push(0)
33     stack.push(len(arr) - 1)
34
35     while not stack.is_empty():
36         q = stack.top()
37         stack.pop()
38
39         p = stack.top()
40         stack.pop()
41
42         pivot_idx = partition_cmp(arr, p, q, cmp=cmp)
43
44         if pivot_idx - 1 > p:
45             stack.push(p)
46             stack.push(pivot_idx - 1)
47
48         if pivot_idx + 1 < q:
```

```

49         stack.push(pivot_idx + 1)
50         stack.push(q))

```

Kod modułu *stack*

```

1 class Stack:
2     def __init__(self, size = 0):
3         self.s = [] if size == 0 else [None] * size
4         self.size = size
5         self.itop = -1
6
7     def push(self, item):
8         if self.itop < self.size - 1:
9             self.itop += 1
10            self.s[self.itop] = item
11        else:
12            self.itop += 1
13            self.size += 1
14            self.s.append(item)
15
16    def pop(self):
17        if self.itop >= 0:
18            self.itop -= 1
19
20    def top(self):
21        if self.itop >= 0:
22            return self.s[self.itop]
23        else:
24            return None
25
26    def sec(self):
27        if self.itop > 0:
28            return self.s[self.itop - 1]
29        else:
30            return None
31
32    def is_empty(self):
33        return not (self.itop >= 0)
34
35    def getsize(self):
36        return self.itop+1
37
38    def __str__(self):
39        return str(self.s[:self.itop+1])

```

Kod modułu *util*

```

1 import json
2 from typing import Literal, Union
3 from lib.mytypes import *
4 from random import randint
5 import matplotlib.colors as mcolors
6 import math
7 import numpy as np

```

```

8 import csv
9
10 def det3x3( ux, uy, uz,
11            vx, vy, vz,
12            wx, wy, wz) -> float:
13     return (ux * vy * wz) + (uy * vz * wx) + (uz * vx * wy) - (wx * vy *
14         uz) - (wy * vz * ux) - (wz * vx * uy)
15
16 def orientation(p1: Point, p2: Point, p3: Point, eps: float = 1e-5) ->
    Literal[-1, 0, 1]:
17     det = det3x3(p1[0], p1[1], 1,
18                 p2[0], p2[1], 1,
19                 p3[0], p3[1], 1)
20     if det < -eps:
21         return -1
22     elif det < eps:
23         return 0
24     else:
25         return 1
26
27 def save_points_to_json(path: str, points: ListOfPoints, indent: int =
    None) -> None:
28     with open(path, 'w') as file:
29         file.write(json.dumps(points, indent = indent))
30
31
32 def load_points_from_json(path: str) -> None:
33     with open(path, 'r') as file:
34         return json.load(file)
35
36
37 def save_data_csv(path: str, data):
38     with open(path, 'w') as csvfile:
39         writer = csv.writer(csvfile, delimiter=',')
40         for row in data:
41             writer.writerow(row)
42
43
44 def index_of_min(points: ListOfPoints, cmp_idx = 0) -> Union[int, None]:
45     if len(points) < 1 or cmp_idx < 0 or cmp_idx > 1: return None
46
47     min_el = points[0]
48     min_idx = 0
49
50     for i in range(1, len(points)):
51         if points[i][cmp_idx] < min_el[cmp_idx] \
52             or \
53             points[i][cmp_idx] == min_el[cmp_idx] and points[i][1 -
54                 cmp_idx] < min_el[1 - cmp_idx]:
55
56             min_el = points[i]
57             min_idx = i

```

```

58
59     return min_idx
60
61
62 def index_of_max(points: ListOfPoints, cmp_idx = 0) -> Union[int, None]:
63     if len(points) < 1 or cmp_idx < 0 or cmp_idx > 1: return None
64
65     max_el = points[0]
66     max_idx = 0
67
68     for i in range(1, len(points)):
69         if points[i][cmp_idx] > max_el[cmp_idx] \
70            or \
71            points[i][cmp_idx] == max_el[cmp_idx] and points[i][1 -
cmp_idx] > max_el[1 - cmp_idx]:
72
73             max_el = points[i]
74             max_idx = i
75
76     return max_idx
77
78
79 def dist_sq(p1, p2):
80     return (p2[0] - p1[0]) ** 2 + (p2[1] - p1[1]) ** 2
81
82 def swap_points(p, q):
83     p[0], p[1], q[0], q[1] = q[0], q[1], p[0], p[1]
84
85
86
87 def divide(points, m): #dzieli zbior punktow na w miarae rowne podzbiory o
rozmiarze m lub m-1
88     n=len(points)
89     for i in range(1,n):#gwarantuje, ze w pierwszym zbiorze Qi pierwszy
element jest najnizszy, czyli nalezy do otoczki ostatecznej
90         if points[i][1] < points[0][1]:
91             buf=points[i]
92             points[i] = points[0]
93             points[0]=buf
94
95     k=math.ceil(n/m)
96     Q=[[ ] for i in range(k)]
97     i=0
98     while i<n:
99         for j in range(k):
100             if i==n:
101                 break
102             Q[j].append(points[i])
103             i+=1
104     if len(Q[0]) > m:
105         return None
106
107     return Q
108

```

```

109 def makeSheaf(Points): #laczy punkty w kolejnosci jakiej sa podane
110     Sheaf=[]
111     for i in range(len(Points)-1):
112         Sheaf.append([Points[i],Points[i+1]])
113     return Sheaf
114
115 def makeFullSheaf(Points): #laczy punkty w kolejnosci jakiej sa podane,
116     dodatkowo domyka cylk
117     Sheaf=[]
118     for i in range(len(Points)-1):
119         Sheaf.append([Points[i],Points[i+1]])
120     Sheaf.append([Points[len(Points)-1],Points[0]])
121     return Sheaf
122
123 def length(v):
124     return np.sqrt((v[1][0]-v[0][0])**2+(v[1][1]-v[0][1])**2)
125
126 def det(a,b,c):
127     return a[0]*b[1]-a[0]*c[1]-b[0]*a[1]+b[0]*c[1]+c[0]*a[1]-c[0]*b[1]
128
129 def tangent_r(p, Q, accur=10 ** (-7)): # Q-zbior punktow w formie
130     otoczki
131     ln = len(Q)
132
133     def tangetUtil(p, Q, l, r):
134         if r < 1: # zdarza sie tylko, gdy punkt jest wewnatrz otoczki
135             return None
136
137         mid = (l + r) // 2
138         if det(Q[0], Q[1], p) >= accur and det(Q[ln - 1], Q[0], p) >=
139         accur:
140             if (det(Q[0], p, Q[mid]) <= -accur) or (det(p, Q[mid], Q[(mid
141             + 1) % ln]) <= -accur and \
142                                     det(p, Q[mid], Q[(mid
143             - 1) % ln]) <= -accur) or \
144                 (det(p, Q[mid], Q[(mid + 1) % ln]) <= -accur and det(
145                 p, Q[mid], Q[(mid - 1) % ln]) > -accur):
146                 return tangetUtil(p, Q, mid + 1, r)
147
148         else:
149             if det(Q[0], p, Q[mid]) > -accur and \
150                 ((det(p, Q[mid], Q[(mid + 1) % ln]) <= -accur and det
151                 (p, Q[mid], Q[(mid - 1) % ln]) > -accur) or \
152                 (det(p, Q[mid], Q[(mid + 1) % ln]) <= -accur and det
153                 (p, Q[mid], Q[
154                 (mid - 1) % ln]) <= -accur)): # chyba nie
155                 potrzebne sprawdz na koncu
156                 return tangetUtil(p, Q, mid + 1, r)
157
158         if (det(p, Q[mid], Q[(mid + 1) % ln]) > -accur and det(p, Q[mid],
159         Q[(mid - 1) % ln]) > -accur) \
160             or (
161                 -accur < det(p, Q[mid], Q[(mid + 1) % ln]) < accur and (Q

```

```

153 [mid][0] <= p[0] <= Q[(mid + 1) % ln][0]) and \
154 (Q[mid][1] <= p[1] <= Q[(mid + 1) % ln][1])):
155     while -accur < det(p, Q[mid], Q[(mid + 1) % ln]) < accur and
156     length([Q[(mid + 1) % ln], p]) > length(
157         [Q[mid], p]):
158         mid = (mid + 1) % ln # jesli jest styczna wspol liniowa,
159         to bierzmy pod uwage punkt blizszy
160         return mid
161     else:
162         return tangetUtil(p, Q, l, mid - 1)
163     return tangetUtil(p, Q, 0, ln - 1)
164
165 def tangent_l(p, Q, accur=10 ** (-7)): # Q-zbior punktow w formie
166     otoczki
167     ln = len(Q)
168
169     def tangetUtil(p, Q, l, r):
170         if r < l: # zdarza sie tylko, gdy punkt jest wewnatrz otoczki
171             return None
172         mid = (l + r) // 2
173         if det(Q[0], Q[1], p) >= accur and det(Q[ln - 1], Q[0], p) >=
174             accur:
175             if (det(Q[0], p, Q[mid]) >= accur) or (det(p, Q[mid], Q[(mid
176                 + 1) % ln]) >= accur and \
177                 det(p, Q[mid], Q[(mid
178                 - 1) % ln]) >= accur) or \
179                 (det(p, Q[mid], Q[(mid + 1) % ln]) <= -accur and det(
180                 p, Q[mid], Q[(mid - 1) % ln]) > -accur):
181                 return tangetUtil(p, Q, l, mid-1)
182             else:
183                 if det(Q[ln - 1], Q[0], p) >= accur and det(Q[0], Q[1], p)
184                 <= - accur:
185                     return 0
186                 if det(Q[0], p, Q[mid]) < accur and \
187                     ((det(p, Q[mid], Q[(mid + 1) % ln]) >= accur and det(
188                     p, Q[mid], Q[(mid - 1) % ln]) < accur) or \
189                     (det(p, Q[mid], Q[(mid + 1) % ln]) >= accur and det(
190                     p, Q[mid], Q[
191                     (mid - 1) % ln]) >= accur)): # chyba nie
192                     potrzebne sprawdz na koncu
193                     if ln > 2:
194                         return tangetUtil(p, Q, l, mid-1)
195                 if (det(p, Q[mid], Q[(mid + 1) % ln]) < accur and det(p, Q[mid],
196                     Q[(mid - 1) % ln]) < accur) \
197                     or (
198                     -accur < det(p, Q[mid], Q[(mid + 1) % ln]) < accur and (Q
199                     [mid][0] <= p[0] <= Q[(mid + 1) % ln][0]) and \
200                     (Q[mid][1] <= p[1] <= Q[(mid + 1) % ln][1])):

```

```

193         while -accur < det(p, Q[mid], Q[(mid - 1) % ln]) < accur and
194         length([Q[(mid - 1) % ln], p]) > length(
195             [Q[mid], p]):
196             mid = (mid - 1) % ln # jesli jest styczna wspolliniowa,
197             to bierzmy pod uwage punkt blizszy
198             return mid
199         else:
200             return tangetUtil(p, Q, mid + 1, r)
201
202     return tangetUtil(p, Q, 0, ln - 1)
203
204
205 def compr(p,q,current,accur=10**(-6)):
206     if det(current,p,q)>accur:
207         return -1
208     elif det(current,p,q)<accur:
209         return 1
210     else:
211         return 0

```

Kod modułu *timemeasure*

```

1 import os
2 from sys import path
3 path.append(os.path.dirname(os.path.realpath(__file__)) + "../")
4
5 from timeit import default_timer as timer
6 from typing import Any, Callable, List
7 from pprint import pprint
8 import numpy as np
9
10 def get_exec_time(func: Callable, *args, points = []) -> float:
11     tstart = timer()
12     func(points, *args)
13     tstop = timer()
14     return (tstop - tstart)
15
16
17 def avg_exec_time(func: Callable, *args, points = [], times = 1) -> float:
18     :
19     total_exec_time = 0
20
21     for _ in range(times):
22         points_copy = points.copy()
23         total_exec_time += get_exec_time(func, *args, points =
24             points_copy)
25
26     return total_exec_time / times

```

Kod modułu *tangent*

```

1 def tangent(p, Q, accur=0): # Q-zbior punktow w formie otoczki

```

```

2     ln = len(Q)
3
4     def tangetUtil(p, Q, l, r):
5         if r < l: # zdarza sie tylko, gdy punkt jest wewnatrz otoczki
6             return None
7
8         mid = (l + r) // 2
9         if det(Q[0], Q[1], p) > 0 and det(Q[ln - 1], Q[0], p) > 0:
10             if (det(Q[0], p, Q[mid]) < 0 or (det(p, Q[mid], Q[(mid + 1)
11 % ln]) < 0 and \
12                                     det(p, Q[mid], Q[(mid - 1)
13 % ln]) < 0) or \
14                                     (det(p, Q[mid], Q[(mid + 1) % ln]) < 0 and det(p, Q[
15 mid], Q[(mid - 1) % ln]) >= 0):
16                 return tangetUtil(p, Q, mid + 1, r)
17
18         else:
19             if det(Q[0], p, Q[mid]) >= 0 and \
20                 ((det(p, Q[mid], Q[(mid + 1) % ln]) < 0 and det(p, Q[
21 mid], Q[(mid - 1) % ln]) >= 0) or \
22                 (det(p, Q[mid], Q[(mid + 1) % ln]) < 0 and det(p, Q[
23 mid], Q[
24                 (mid - 1) % ln]) < 0)): # chyba nie potrzebne
25                 sprawdz na koncu
26                 return tangetUtil(p, Q, mid + 1, r)
27
28             if det(p, Q[mid], Q[(mid + 1) % ln]) >= 0 and det(p, Q[mid], Q[(
29 mid - 1) % ln]) >= 0 \
30                 or (det(p, Q[mid], Q[(mid + 1) % ln]) == 0 and (Q[mid][0]
31 <= p[0] <= Q[(mid + 1) % ln][0] and \
32                     (Q[mid][1] <= p[1] <= Q[(mid + 1) % ln][1]))):
33
34                 while (det(p, Q[mid], Q[(mid + 1) % ln]) == 0):
35                     mid = (mid + 1) % ln # jesli jest styczna wspolliniowa,
36                     to bierzmy pod uwage punkt blizszy
37                 return mid
38
39             else:
40                 return tangetUtil(p, Q, l, mid - 1)
41
42     return tangetUtil(p, Q, 0, ln - 1)

```

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”

Kody algorytmów znajdują się w sekcji *Algorytmy*.

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 3.8.3 64-bit lub nowszy wraz z modułami:
 - *matplotlib*
 - *numpy*
2. Jupyter Notebook

1.3 Korzystanie z programu

1.3.1 Uruchomienie programu

W celu uruchomienia wizualizacji algorytmów należy uruchomić notebook (poprzez Jupyter Notebook) *kafara_czarniecki_program.ipynb*, oraz wykonywać kolejne komórki notatnika.

W celu uruchomienia pomiarów wydajności algorytmów należy przejść do sekcji *Pomiary czasu* wykonać pierwszą komórkę (z importami algorytmów) a następnie przeprowadzać testy i wyświetlać rezultaty w interesujących nas przypadkach.

2 Oznaczenia i definicje

Na potrzeby dalszych wywodów przyjmujemy w tym miejscu szereg oznaczeń i definicji:

*def. 1. **Zbiorem wypukłym*** nazwiemy dowolny podzbiór płaszczyzny taki, że dla każdych dwóch punktów do niego należących, odcinek je łączący również należy do tego zbioru.

*def. 2. **Otoczką wypukłą*** dowolnego zbioru punktów S płaszczyzny nazwiemy najmniejszy zbiór wypukły $CH(S)$ zawierający S .

Algorytmicznie otoczkę wypukłą dowolnego zbioru S punktów płaszczyzny reprezentujemy jako ciąg punktów (wierzchołków) $< v_1, v_2, \dots, v_n >$ wielokąta wypukłego, gdzie $\forall i \in \{1, \dots, n\}$ v_i jest poprzednikiem v_{i+1} (w kolejności wierzchołków przeciwnej do ruchu wskazówek zegara).

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. Usuwanie 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) Urywamy 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. 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[
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 ostatnie 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 zbioru 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 uproszczenie 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

Niech Q będzie punktem przez który ma przechodzić styczna.

1. W czasie liniowym znajdujemy punkt otoczki P_i o największej odciętej (jeżeli jest wiele, to wybieramy ten o najmniejszej rzędnej)
2. Dopóki P_{i+1} nie znajduje się po lewej stronie odcinka QP_i :
 - (a) Jeżeli Q, P_i, P_{i+1} są współliniowe oraz P_i leży dalej (lub w takiej samej odległości) Q niż P_{i+1} to przerwij działanie pętli.
 - (b) $P_i := P_{i+1}$
3. W całkowicie analogiczny sposób wyznaczamy dolną styczną.

4.4.3 Złożoność

Posortowanie punktów zajmuje $O(n \lg n)$. Wydaje główna pętla programu wykonuje się w czasie $O(n)$, ponieważ możemy usunąć maksymalnie $k - 3$ punkty (gdy k jest liczebnością zbioru punktów otoczki znaną w danej iteracji), ale zauważmy, że każdy z punktów usuwany jest co najwyżej raz. Wyszukanie stycznych w głównej pętli także zajmuje czas liniowy, więc główna pętla programu wykonuje się w czasie liniowym. Zatem złożoność algorytmu jest rzędu $O(n \lg n)$

4.4.4 Kod

```

1 def rltangent(polygon: ListOfPoints, point: Point):
2     n = len(polygon)
3
4     right = index_of_max(polygon, cmp_idx=0)
5
6     left = right
7
8     left_orient = orientation(point, polygon[left % n], polygon[(left-1)%
9     n])
10    while left_orient != -1:
11        if left_orient == 0 and dist_sq(point, polygon[left]) >= dist_sq(
12        point, polygon[(left-1)%n]):
13            break
14        left = (left-1) % n
15        left_orient = orientation(point, polygon[left % n], polygon[(left
16        -1)%n])

```



```

14
15     right_orient = orientation(point, polygon[right%n], polygon[(right+1)
16     %n])
17     while right_orient != 1:
18         if right_orient == 0 and dist_sq(point, polygon[right]) >=
19         dist_sq(point, polygon[(right+1)%n]):
20             break
21             right = (right+1)%n
22             right_orient = orientation(point, polygon[right%n], polygon[(
23             right+1)%n])
24
25     return left, right
26
27 def increase_with_sorting(point2_set: ListOfPoints) -> Union[ListOfPoints
28 , None]:
29     if len( point2_set ) < 3: return None
30
31     point2_set.sort(key = operator.itemgetter(0, 1))
32
33     convex_hull = point2_set[:3]
34
35     if orientation(convex_hull[0], convex_hull[1], convex_hull[2]) == -1:
36         convex_hull[1], convex_hull[2] = convex_hull[2], convex_hull[1]
37
38     for i in range(3, len( point2_set )):
39         rltang = rltangent(convex_hull, point2_set[i])
40         left_tangent_idx = rltang[0]
41         right_tangent_idx = rltang[1]
42
43         left_tangent_point = convex_hull[left_tangent_idx]
44         right_tangent_point = convex_hull[right_tangent_idx]
45
46         deletion_side: Literal[-1, 0, 1] = orientation(left_tangent_point
47         , right_tangent_point, point2_set[i])
48
49         if deletion_side != 0:
50             lrlnext_orient = orientation(left_tangent_point,
51             right_tangent_point, convex_hull[(left_tangent_idx + 1) % len(
52             convex_hull)])
53             if lrlnext_orient == deletion_side or lrlnext_orient == 0:
54                 step = 0
55             else:
56                 step = -1
57
58             left = (left_tangent_idx + 1) % len(convex_hull)
59
60             while convex_hull[left % len(convex_hull)] !=
61             right_tangent_point:
62                 convex_hull.pop(left % len(convex_hull))
63                 left = (left + step) % len(convex_hull)
64
65             convex_hull.insert(left % len(convex_hull), point2_set[i])

```

```

60         else:
61             convex_hull = [point2_set[left_tangent_idx], point2_set[i]]
62         return convex_hull

```

4.5 Algorytm dziel i zwyciężaj

4.5.1 Opis działania

Prócz zbioru punktów, dodatkową daną wejściową dla algorytmu jest stała k oznaczająca liczebność zbioru punktów, przy której przechodzimy w algorytmie rekurencyjnym do przypadku bazowego – wyznaczamy otoczkę innym, wybranym algorytmem.

Opisany algorytm jest algorytmem rekurencyjnym. Przed pierwszym wywołaniem rekurencyjnym należy zbiór punktów posortować rosnąco po odciętych (w przypadku równych, mniejszy jest punkty o mniejszej rzędnej).

Jest to standardowe zastosowanie metody *”dziel i zwyciężaj”*:

1. Dzielimy wyjściowy problem na mniejsze tak długo, aż znajdujemy się w przypadku któryi potrafimy rozwiązać elementarnie / w inny sposób.
2. Łączymy kolejne rozwiązania częściowe w całość.

Popatrzmy na schemat działania:

1. Jeżeli liczebność rozważanego zbioru jest mniejsza bądź równa danej stałej k , to:
 - (a) Wyznaczamy otoczkę rozważanego zbioru punktów, za pomocą innej metody (np. innego algorytmu wyznaczania otoczki).
 - (b) Zwracamy tak uzyskaną otoczkę.
2. W przeciwnym przypadku:
 - (a) Wywołujemy się rekurencyjnie na zbiorze punktów o odciętych mniejszych od mediany.
 - (b) Wywołujemy się rekurencyjnie na zbiorze punktów o odciętych większych bądź równych medianie.
 - (c) Łączymy lewą i prawą otoczkę (pozyskane z wywołań rekurencyjnych) w jedną.
 - (d) Zwracamy tak uzyskaną otoczkę.

4.5.2 Szczegóły

- Do wyznaczania otoczki w przypadku podstawowym wykorzystany został algorytm Jarvisa, ponieważ dla $k \ll n$ ma on złożoność właściwie liniową.
- Sposób łączenia otoczek jest następujący:
 1. Wyznaczamy skrajny prawy punkt L lewej otoczki oraz skrajny lewy P punkt prawej otoczki.

2. Dopóki L i P nie tworzą górnej stycznej, ”wychodzimy w górę” naprzemiennie punktami L i P .
3. Analogicznie wyznaczamy dolną styczną.
4. Usuwamy punkty zawierające się we wnętrzu nowo utworzonej otoczki.

4.5.3 Złożoność

- Początkowe sortowanie: $O(n \lg n)$
- Rekurencja: $T(n) = 2T(\frac{n}{2}) + O(n) \implies T(n) = O(n \lg n)$
- Każde łączenie otoczek: $O(n)$

Algorytm dziel i zwyciężaj ma zatem złożoność $O(n \lg n)$

4.5.4 Kod

```

1 def merge_convex_hulls(left_convex_hull: ListOfPoints, right_convex_hull:
  ListOfPoints) -> List[Point]:
2     left_ch_size = len(left_convex_hull)
3     right_ch_size = len(right_convex_hull)
4
5     # znajdujemy prawy skrajny punkt lewej otoczki
6     left_ch_rightmost_idx = index_of_max(left_convex_hull, cmp_idx=0)
7     right_ch_leftmost_idx = index_of_min(right_convex_hull, cmp_idx=0)
8
9     left = left_convex_hull[left_ch_rightmost_idx]
10    right = right_convex_hull[right_ch_leftmost_idx]
11    left_idx = left_ch_rightmost_idx
12    right_idx = right_ch_leftmost_idx
13
14    left_flag, right_flag = True, True
15    while orientation(left, right, right_convex_hull[(right_idx - 1) %
  right_ch_size]) != -1 and right_flag \
16        or \
17        orientation(right, left, left_convex_hull[(left_idx + 1) %
  left_ch_size]) != 1 and left_flag:
18
19        left_flag, right_flag = False, False
20
21    # podnosimy punkt na prawej otoczce
22    left_right_orient = orientation(left, right, right_convex_hull[(
  right_idx - 1) % right_ch_size])
23    while left_right_orient != -1:
24        if left_right_orient == 0 and dist_sq(left, right) >= dist_sq
  (left, right_convex_hull[(right_idx - 1) % right_ch_size]):
25            right_flag = False
26            break
27
28    right_idx = (right_idx - 1) % right_ch_size
29    right = right_convex_hull[right_idx]

```

```

30         left_right_orient = orientation(left, right,
right_convex_hull[(right_idx - 1) % right_ch_size])
31     else:
32         right_flag = True
33
34     # podnosimy punkt na lewej otoczce
35     right_left_orient = orientation(right, left, left_convex_hull[(
left_idx + 1) % left_ch_size])
36     while right_left_orient != 1:
37         if right_left_orient == 0 and dist_sq(right, left) >= dist_sq
(right, left_convex_hull[(left_idx + 1) % left_ch_size]):
38             left_flag = False
39             break
40
41     left_idx = (left_idx + 1) % left_ch_size
42     left = left_convex_hull[left_idx]
43     right_left_orient = orientation(right, left, left_convex_hull
[(left_idx + 1) % left_ch_size])
44     else:
45         left_flag = True
46
47
48     upper_tangent_left_idx = left_idx
49     upper_tangent_right_idx = right_idx
50
51     # dolna styczna
52     left = left_convex_hull[left_ch_rightmost_idx]
53     right = right_convex_hull[right_ch_leftmost_idx]
54     left_idx = left_ch_rightmost_idx
55     right_idx = right_ch_leftmost_idx
56
57     left_flag, right_flag = True, True
58     while orientation(left, right, right_convex_hull[(right_idx + 1) %
right_ch_size]) != 1 and right_flag \
59         or \
60         orientation(right, left, left_convex_hull[(left_idx - 1) %
left_ch_size]) != -1 and left_flag:
61
62         left_flag, right_flag = False, False
63
64     # opuszczamy punkt na prawej otoczce
65     left_right_orient = orientation(left, right, right_convex_hull[(
right_idx + 1) % right_ch_size])
66     while left_right_orient != 1:
67         if left_right_orient == 0 and dist_sq(left, right) >= dist_sq
(left, right_convex_hull[(right_idx + 1) % right_ch_size]):
68             right_flag = False
69             break
70
71     right_idx = (right_idx + 1) % right_ch_size
72     right = right_convex_hull[right_idx]
73     left_right_orient = orientation(left, right,
right_convex_hull[(right_idx + 1) % right_ch_size])
74     else:

```

```

75         right_flag = True
76
77         # opuszczamy punkt na lewej otoczce
78         right_left_orient = orientation(right, left, left_convex_hull[(
left_idx - 1) % left_ch_size])
79         while right_left_orient != -1:
80             if right_left_orient == 0 and dist_sq(right, left) >= dist_sq
(right, left_convex_hull[(left_idx - 1) % left_ch_size]):
81                 left_flag = False
82                 break
83
84                 left_idx = (left_idx - 1) % left_ch_size
85                 left = left_convex_hull[left_idx]
86                 right_left_orient = orientation(right, left, left_convex_hull
[(left_idx - 1) % left_ch_size])
87             else:
88                 left_flag = True
89
90
91         lower_tangent_left_idx = left_idx
92         lower_tangent_right_idx = right_idx
93
94         merged_convex_hull = [ ]
95
96         while upper_tangent_left_idx != lower_tangent_left_idx:
97             merged_convex_hull.append(left_convex_hull[upper_tangent_left_idx
])
98             upper_tangent_left_idx = (upper_tangent_left_idx + 1) %
left_ch_size
99         else:
100             merged_convex_hull.append(left_convex_hull[lower_tangent_left_idx
])
101
102
103         while lower_tangent_right_idx != upper_tangent_right_idx:
104             merged_convex_hull.append(right_convex_hull[
lower_tangent_right_idx])
105             lower_tangent_right_idx = (lower_tangent_right_idx + 1) %
right_ch_size
106         else:
107             merged_convex_hull.append(right_convex_hull[
lower_tangent_right_idx])
108
109         return merged_convex_hull
110
111
112 def divide_conq(point2_set: List[Point], k: int = 2) -> Union[List[Point
], None]:
113     if len(point2_set) < 3 or k <= 0: return None
114
115
116     def divide_conq_rec(point2_set: List[Point]) -> List[Point]:
117         if len(point2_set) <= 2: return point2_set
118         elif len(point2_set) <= k: return jarvis(np.array(point2_set))

```

```

119
120     left_convex_hull = divide_conq_rec(point2_set[ : len(point2_set)
// 2])
121     right_convex_hull = divide_conq_rec(point2_set[len(point2_set) //
2 : ])
122
123     return merge_convex_hulls(left_convex_hull, right_convex_hull)
124
125     point2_set.sort(key = operator.itemgetter(0, 1))
126
127     return divide_conq_rec(point2_set)

```

4.6 Algorytm Chana

4.6.1 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 nadrzędną 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 popsuć złożoności dużą ilością powtórzeń głównej części algorytmu najlepiej za każdym razem parametr m podnosić do kwadratu. W przypadku gdy $m \geq n$ za m przyjmujemy n . Wtedy algorytm Chana sprowadza się do algorytmu Grahama.

4.6.2 Szczegóły

- Najniższy punkt zbioru wyznaczamy w czasie liniowym. W trakcie podziału zapewniamy, że znajdzie się on w otoczce C_0 w indeksie 0.
- Podział zapewnia, że w każdym podzbiorze będzie m , lub $m-1$ punktów.
- Znajdowanie stycznej wykonujemy w czasie $O(\log n)$. Wykorzystujemy do tego binary search, można tak zrobić, ponieważ otoczka zadana jest przeciwnie do ruchu wskazówek zegara. Najpierw obieramy wskaźniki l i r . Następnie obliczamy $mid = (l+r)/2$. Gdy obaj sąsiedzi mid a są po lewej stronie - mid jest szukany indeksem. Gdy mid znajduje się po prawej stronie odcinka pq , gdzie p , to pierwszy wierzchołek otoczki (pierwszy l), a q , to punkt do którego stycznej szukamy. Wtedy prawa styczna znajduje się w przedziale $mid+1, r$ i urachamiamy tą samą funkcję dla tych indeksów. Gdy mid jest współliniowy, bądź po lewej stronie z pq , to jeśli jego następnik ($mid+1$) jest po prawej stronie odcinka $|qmid|$, to również styczna się znajduje w przedziale $mid+1, r$. W pozostałych przypadkach znajduje się w przedziale $l, mid-1$.

4.6.3 Złożoność

Złożoność głównej części algorytmu.

- Na złożoność pierwszej części algorytmu składa się:
 - Podział zbioru punktów na podzbiory $O(n)$;
 - Wyznaczenie otoczek dla podzbiorów. Mamy $\lceil \frac{n}{m} \rceil$ podzbiorów rozmiaru m , dla każdego z nich wyznaczamy otoczkę algorytmem Grahama. Algorytm Grahama działa $O(n \log(n))$. Więc łącznie mamy $O(\lceil \frac{n}{m} \rceil \cdot m \log(m)) = O(n \log(m))$.

Łącznie dla pierwszej części mamy $O(n \log(m))$, gdzie m jest wybranym maksymalnym rozmiarem podzbiorów.

- Na złożoność drugiej części algorytmu składa się:
 - 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 liczebnością podzbioru. Złożoność algorytmu dla próbowania algorytmu z kolejnymi m postaciami 2^{2^m} dla $m \geq 1$, to: w takim razie złożoność można opisać wzorem $\sum_{t=1}^{\lceil \log \log k \rceil} O(n \log(2^{2^t})) = O(n * 2^{1+\lceil \log \log k \rceil}) = O(n \log k)$

4.6.4 Kod

```

1 def divide(points, m):
2     n = len(points)
3     for i in range(1,
4         n): # gwarantuje, ze w pierwszym zbiorze Qi pierwszy
5         element jest najnizszy, czyli nalezy do otoczki ostatecznej
6         if points[i][1] < points[0][1]:
7             buf = points[i]
8             points[i] = points[0]
9             points[0] = buf
10
11     k = math.ceil(n / m)
12     Q = [[] for i in range(k)]
13     i = 0
14     while i < n:
15         for j in range(k):
16             if i == n:
17                 break
18             Q[j].append(points[i])
19             i += 1
20     if len(Q[0]) > m:
21         return None
22     return Q
23
24
25 def compr(p, q, current,
26     accur=10 ** (-6)): # jezeli p jest po prawej odcinka [current
27     ,q] - jest 'wiekszy', to zwracamy 1
28     if det(current, p, q) > accur:
29         return -1
30     elif det(current, p, q) < accur:
31         return 1
32     else:
33         return 0
34
35 def nextvert(C, curr): # dla danego punktu wspolzednymi z Q[i][j] jesli
36     jest to punkt nalezacy do finalnej otoczki, to
37     # zwraca nastepny punkt nalezacy do finalnej otoczki zadanego w
38     takich samych wspolzednych Q[nxt[0]][nxt[1]]
39     i, j = curr
40     nxt = (i, (j + 1) % len(C[i]))
41     for k in range(len(C)):
42         t = tangent(C[i][j], C[k])
43         if t != None and k != i and compr(C[nxt[0]][nxt[1]], C[k][t], C[i]
44             ][j]) > 0 and (k, t) != (curr):
45             nxt = (k, t)

```



```

42
43     return nxt
44
45 def chanUtil(points, m):
46     Q = divide(points, m)
47     C = []
48     for i in range(len(Q)):
49         C.append(Graham(Q[i]))
50
51     curr = (0, 0)
52     ans = []
53     i = 0
54     while i < n:
55         for j in range(k):
56             if i == n:
57                 break
58             Q[j].append(points[i])
59             i += 1
60     if len(Q[0]) > m:
61         return None
62
63     return Q
64
65     def nextvert(C, curr, accur=10**(-7)): # dla danego punktu
wspolzednymi z Q[i][j] jesli jest to punkt nalezacy do finalnej
otoczki, to
66         # zwraca nastepny punkt nalezacy do finalnej otoczki zadanego w
takich samych wspolzednych Q[nxt[0]][nxt[1]]
67         i, j = curr
68         nxt = (i, (j + 1) % len(C[i]))
69         for k in range(len(C)):
70             if k == i:
71                 continue
72
73             t = tangent_r(C[i][j], C[k])
74             if t == None:
75                 print("zle")
76                 continue
77
78             if det(C[i][j], C[nxt[0]][nxt[1]], C[k][t]) < accur and (k, t) != (
curr):
79                 if det(C[i][j], C[nxt[0]][nxt[1]], C[k][t]) > -accur:
80                     if length([C[k][t], C[i][j]]) <= length([C[nxt[0]][nxt
1]], C[i][j]]):
81                         continue
82                     nxt = (k, t)
83
84         return nxt
85
86
87 def chanUtil(points, m):
88     Q = divide(points, m)
89     C = []
90     for i in range(len(Q)):

```

```

91         C.append(gham(Q[i]))
92
93     curr = (0, 0)
94     ans = []
95     i = 0
96
97     while i < m:
98         ans.append(C[curr[0]][curr[1]])
99
100        nxt = nextvert(C, curr)
101        if nxt == (0, 0):
102            return ans
103        curr = nxt
104        i += 1
105
106    return None
107
108
109    def chan(points, visual=False):
110        plot = None
111        n = len(points)
112        m = 4
113        hoax = None
114
115
116        return hoax
117
118    return hoax

```

4.7 Algorytm QuickHull

4.7.1 Opis działania

Algorytm QuickHull polega na rekurencyjnym wyznaczaniu kolejnych punktów otoczki.

1. Algorytm rozpoczynamy od wyznaczenie 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.

4.7.2 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.

4.7.3 Złożoność

Pesymistyczna złożoność algorytmu to $O(n^2)$ - gdy wszystkie punkty zbioru znajdują się w otoczce. Jendakże w średnim przypadku złożoność wynosi $O(n \log n)$

4.7.4 Kod

```

1  from copy import deepcopy
2  from lib.det import *
3
4  def furthest(a, b, considering):
5      n = len(considering)
6      i = 0
7      ans = None
8      while i < n:
9          if det(a, b, considering[i]) < 0: # rozważany wierzchołek
jest po prawej stronie ab
10             if ans == None or det(a, b, considering[i]) < det(a, b,
11                                                         ans):
12                 # |det(a,b,c)| = 1/2|ab|*h, gdzie h jest wysokoscia z c na ab
13                 ans = considering[i]
14                 i += 1
15             return ans
16
17 def insideTriangle(a, b, c, i):
18     accur=10**(-7)

```

```

19         if det(a, b, i) > -accur and det(b, c, i) > -accur and det(c, a,
20            i) > -accur:
21             return True
22         return False
23
24     def removeInner(a, b, c, considering):
25         new=[]
26         for i in considering:
27             if not insideTriangle(a, b, c, i):
28                 new.append(i)
29             considering.clear()
30             considering+=new
31
32     def quickHullUtil(a, b, considering):
33         if len(considering) == 0:
34             return []
35
36         c = furthest(a, b, considering)
37         if c == None:
38             return []
39         considering.remove(c)
40
41         removeInner(a, c, b, considering)
42         return quickHullUtil(a, c, considering) +[c]+ quickHullUtil(c, b,
43            considering)
44
45     def quickHull(points):
46         a = min(points, key=lambda x: x)
47         b = max(points, key=lambda x: x)
48
49         considering = deepcopy(points)
50
51         considering.remove(a)
52         considering.remove(b)

```

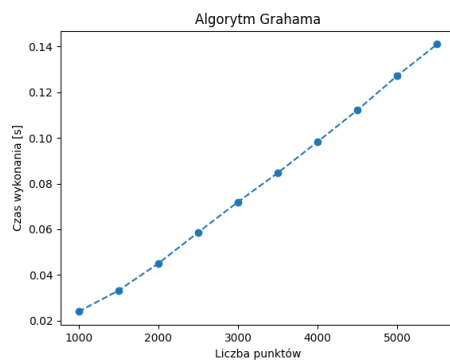
5 Wydajność algorytmów

Testy prowadzone były na następujących zbiorach:

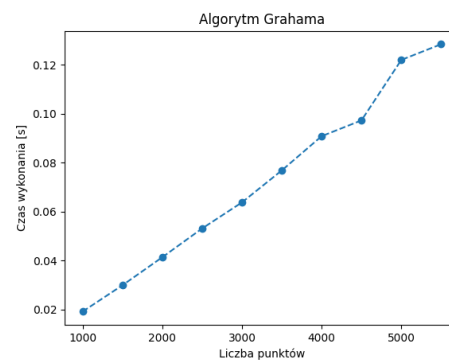
- typ A - losowo rozłożone punkty płaszczyzny o określonych zakresach współrzędnych
- typ B - losowo rozłożone punkty leżące na okręgu o zadanych parametrach
- typ C - losowo rozłożone punkty leżące na bokach prostokąta o zadanych parametrach
- typ D - losowo rozłożone punkty leżące na 2. przekątnych oraz 2. bokach kwadratu, umiejscowionego tak, że dwa boki pokrywają się z osiami układu

Tablica 1: Czas wykonania algorytmu Grahama w zależności od typu zbioru testowego oraz mocy zbioru punktów.

	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Typ zbioru	Czas wykonania [s]									
A	0.0239	0.0331	0.045	0.0586	0.072	0.0847	0.0983	0.1122	0.1273	0.141
B	0.0191	0.0299	0.0413	0.0531	0.0637	0.0768	0.0908	0.0973	0.122	0.1284
C	0.069	0.1115	0.1392	0.1878	0.2349	0.2736	0.3153	0.351	0.3988	0.4601
D	0.4256	0.6523	0.924	1.2255	1.4233	1.8752	1.9285	2.3207	2.5692	2.88



Rysunek 1: Zbiór typu A, algorytm Grahama



Rysunek 2: Zbiór typu B, algorytm Grahama

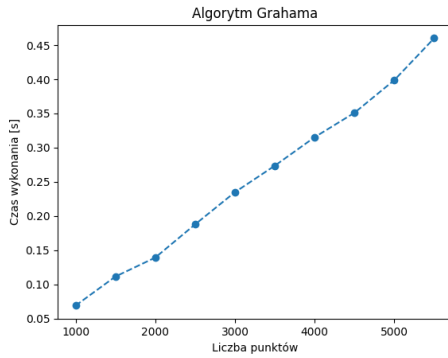
Charakterystyki zbiorów zostały dobrane w taki sposób aby odzwierciedlać możliwie różne zachowanie się poszczególnych algorytmów w zależności od ilości przypadków zdegenerowanych (liczebność współliniowych otoczek), ilości potrzebnych do wykonania porównań i operacji. Bardziej szczegółowe omówienie charakterystyki algorytmów dla poszczególnych zbiorów znajduje się w kolejnych sekcjach.

5.1 Algorytm Grahama

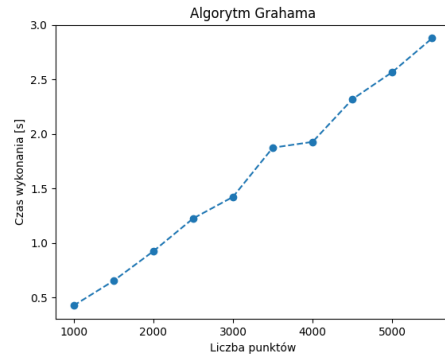
W tabeli 1 przedstawiamy czasy uzyskane przez algorytm Grahama dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 1, 2, 3, 4 widzimy ilustrację danych z tabeli 1.

5.2 Porównanie z przewidywaniami teoretycznymi

Kształt dla każdego z typu zbiorów dla algorytmu Grahama przypomina wykres liniowy. Zgoda się to z przewidywaniami teoretycznymi dla tegoż algorytmu - ma on złożoność $O(n \log n)$. Wykres o takiej złożoności tak bardzo przypomina wykres liniowy. Spośród algorytmów o tej złożoności Graham zdaje się być najwolniejszy. Ilość wierzchołków w otoczce nie ma znaczenia dla każdego rozkładu graham wykazuje się podobnym czasem.



Rysunek 3: Zbiór typu C, algorytm Grahama



Rysunek 4: Zbiór typu D, algorytm Grahama

Tablica 2: Czas wykonania algorytmu górna-dolna w zależności od typu zbioru testowego oraz mocy zbioru punktów.

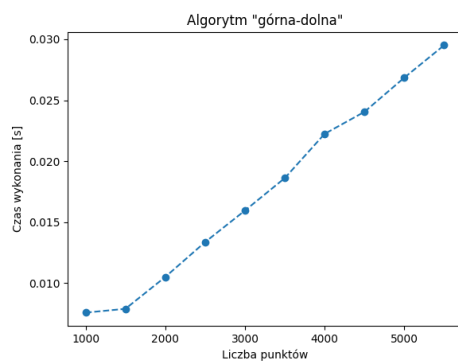
	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Typ zbioru	Czas wykonania [s]									
A	0.0076	0.0079	0.0105	0.0134	0.016	0.0186	0.0222	0.0241	0.0269	0.0295
B	0.004	0.0064	0.0086	0.0111	0.0132	0.0164	0.018	0.0203	0.0229	0.0254
C	0.0155	0.0241	0.0321	0.0388	0.0486	0.0549	0.0651	0.0711	0.0784	0.084
D	0.0635	0.0927	0.1261	0.1566	0.1875	0.2217	0.259	0.2897	0.316	0.3495

5.3 Algorytm górna-dolna

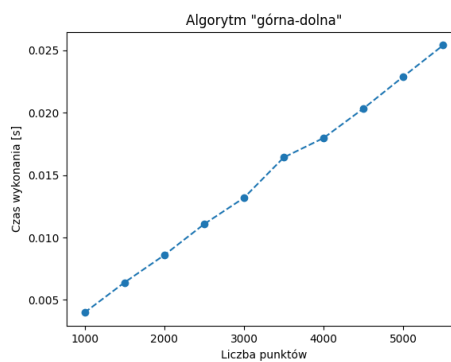
W tabeli 2 przedstawiamy czasy uzyskane przez algorytm górna-dolna dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 5, 6, 7, 8 widzimy ilustrację danych z tabeli 2.

5.4 Porównanie z przewidywaniami teoretycznymi

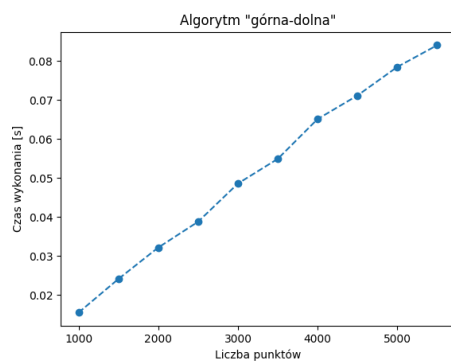
Kształt dla każdego z typu zbiorów dla algorytmu górna-dolna przypomina wykres liniowy. Jest bardzo szybki (szybsze są tylko quickhull i chan) na tle Grahama wykres złożoności tego algorytmu przypomina raczej stałą. Jednak, gdy wyizolować sam wykres algorytmu górna-dół, to widzimy, że także i on przypomina wykres liniowy. Zgadza się to z przewidywaniami teoretycznymi dla tegoż algorytmu - ma on złożoność $O(n \log n)$. Wykres o takiej złożoności tak bardzo przypomina wykres liniowy. Ilość wierzchołków w otoczce nie ma znaczenia, czas wykonania algorytmu górna-dół zależy tylko od mocy zbioru punktów, dla którego otoczki szukamy.



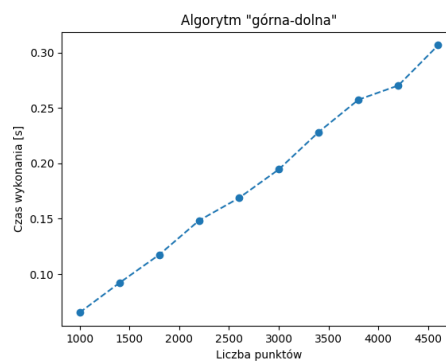
Rysunek 5: Zbiór typu A, algorytm górna-dolna



Rysunek 6: Zbiór typu B, algorytm górna-dolna



Rysunek 7: Zbiór typu C, algorytm górna-dolna



Rysunek 8: Zbiór typu D, algorytm górna-dolna

Tablica 3: Czas wykonania algorytmu Chana w zależności od typu zbioru testowego oraz mocy zbioru punktów.

	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Typ zbioru	Czas wykonania [s]									
A	0.1055	0.1544	0.3575	0.2408	0.5449	0.6155	0.7104	0.7904	0.8621	0.9826
B	0.3714	0.5483	0.6962	0.8893	1.0799	1.2537	1.415	1.827	1.7746	1.9095
C	0.2457	0.5615	0.727	0.9188	1.1007	1.3044	1.5073	1.6875	1.88	2.1234
D	0.6322	0.9372	1.2306	2.1543	2.8556	6.0844	2.4157	3.943	3.094	4.7075

5.5 Algorytm Chana

W tabeli 3 przedstawiamy czasy uzyskane przez algorytm Chana dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 9, 10, 11, 12 widzimy ilustrację danych z tabeli 3.

5.6 Porównanie z przewidywaniami teoretycznymi

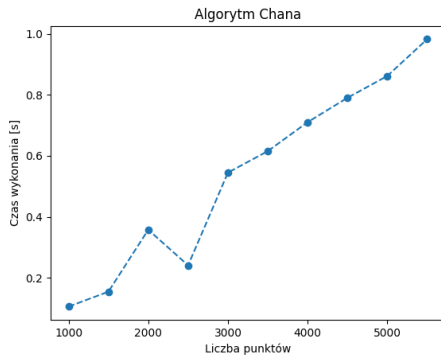
Chan spośród wszystkich rozważanych algorytmów wypadł najgorzej względem reszty. Teoretyczna złożoność chana wynosi $O(n \log k)$, gdzie k jest ilością wierzchołków w otocze. Patrząc na wykresy czasu dla Chana ilość wierzchołków w otocze faktycznie wpływa na szybkość jego działania dla chmury punktów działał bardzo nieregularnie, natomiast dla pozostałych rozkładów, o w stałej ilości wierzchołków w otocze względem wierzchołków zbioru w algorytmie chana dominowała natura liniowa. Wygląda na to, że przewidywania teoretyczne po części sprawdzają się dla chana, jego wykres ma kształt w przybliżeniu liniowy, a dla okręgu, gdzie jest najwięcej wierzchołków w otocze jest najbardziej stromy. Jednakże poza tym, że ma najlepszą złożoność asymptotyczną, to niestety ta implementacja ma największy współczynnik stały w porównaniu z innymi algorytmami i dlatego wypada najgorzej. Jedyne wyniki porównywalne z innymi uzyskał w zbiorze typu D, gdzie ma najlepsze warunki - są tylko 4 wierzchołki w otocze.

5.7 Algorytm QuickHull

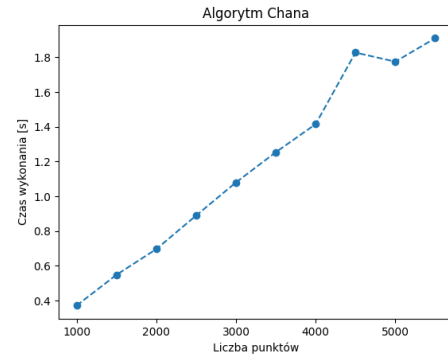
W tabeli 4 przedstawiamy czasy uzyskane przez algorytm QuickHull dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 13, 14, 15, 16 widzimy ilustrację danych z tabeli 4.

5.8 Porównanie z przewidywaniami teoretycznymi

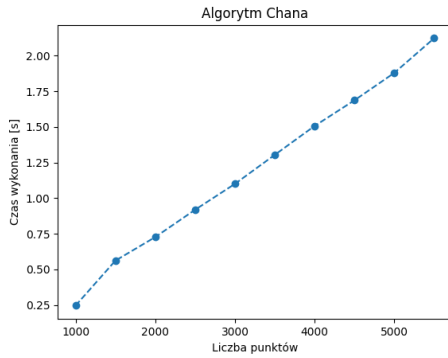
Algorytm QuickHull z wszystkich wypadła najlepiej. Na tle innych wykresów wygląda jakby miał asymptotyczność stałą. Po zobaczeniu jednak jego wykresu osobno widać, że jest to wykres w przybliżeniu liniowy (z wyglądu), co się zgadza z przewidywaniami teoretycznymi. Ma on jednak najmniejszy stały współczynnik względem pozostałych algorytmów, co odpowiada jego zdecydowanie większej prostocie względem innych. ///



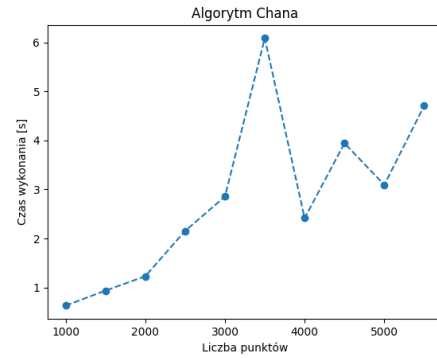
Rysunek 9: Zbiór typu A, algorytm Chana



Rysunek 10: Zbiór typu B, algorytm Chana



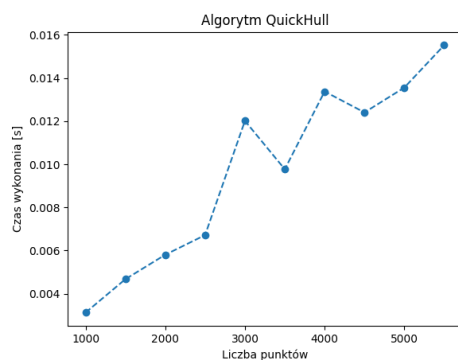
Rysunek 11: Zbiór typu C, algorytm Chana



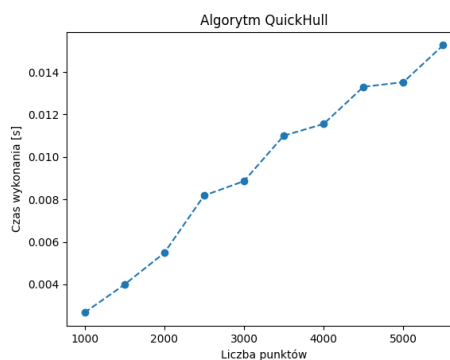
Rysunek 12: Zbiór typu D, algorytm Chana

Tablica 4: Czas wykonania algorytmu QuickHull w zależności od typu zbioru testowego oraz mocy zbioru punktów.

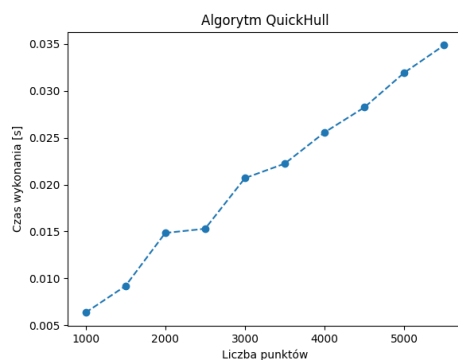
	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Typ zbioru	Czas wykonania [s]									
A	0.0031	0.0047	0.0058	0.0067	0.012	0.0098	0.0134	0.0124	0.0136	0.0155
B	0.0027	0.004	0.0055	0.0082	0.0089	0.011	0.0116	0.0133	0.0135	0.0153
C	0.0064	0.0092	0.0148	0.0153	0.0207	0.0222	0.0256	0.0282	0.0319	0.0348
D	0.0287	0.0426	0.0551	0.0702	0.0847	0.0986	0.1143	0.1294	0.1411	0.1562



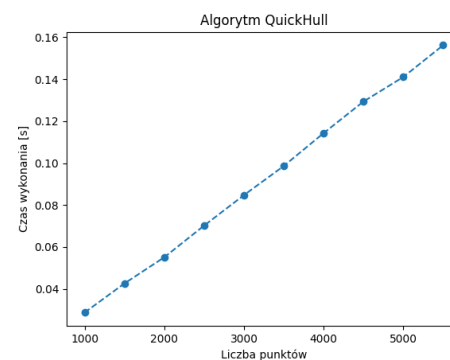
Rysunek 13: Zbiór typu A, algorytm QuickHull



Rysunek 14: Zbiór typu B, algorytm QuickHull



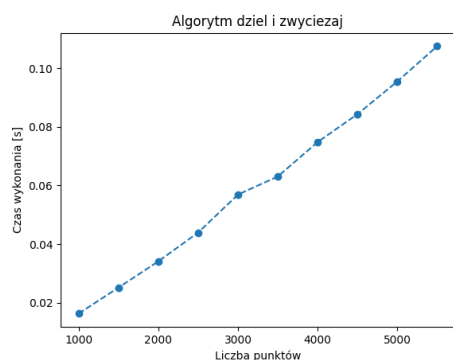
Rysunek 15: Zbiór typu C, algorytm QuickHull



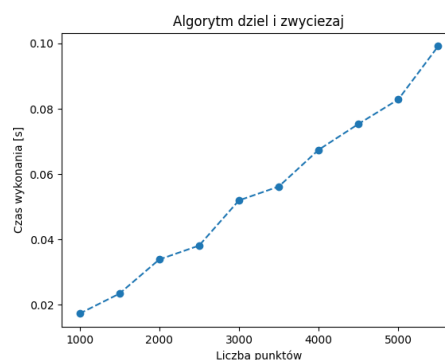
Rysunek 16: Zbiór typu D, algorytm QuickHull

Tablica 5: Czas wykonania algorytmu dziel i zwyciężaj w zależności od typu zbioru testowego oraz mocy zbioru punktów.

	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Typ zbioru	Czas wykonania [s]									
A	0.0163	0.0251	0.0341	0.0439	0.0569	0.0631	0.0748	0.0844	0.0955	0.1076
B	0.0173	0.0234	0.0339	0.0381	0.052	0.0562	0.0675	0.0754	0.0829	0.0992
C	0.0441	0.0683	0.0877	0.1189	0.1516	0.1863	0.2067	0.2437	0.27	0.3132
B	0.1634	0.2474	0.3518	0.4703	0.5694	0.6705	0.7849	0.8928	1.0458	1.1565



Rysunek 17: Zbiór typu A, algorytm dziel i zwyciężaj



Rysunek 18: Zbiór typu B, algorytm dziel i zwyciężaj

5.9 Algorytm dziel i zwyciężaj

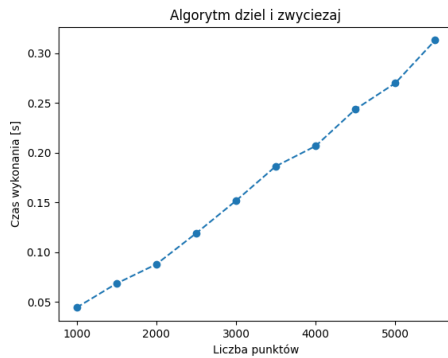
W tabeli 5 przedstawiamy czasy uzyskane przez algorytm Dziel i zwyciężaj dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 17, 18, 19, 20 widzimy ilustrację danych z tabeli 5.

5.10 Porównanie z przewidywaniami teoretycznymi

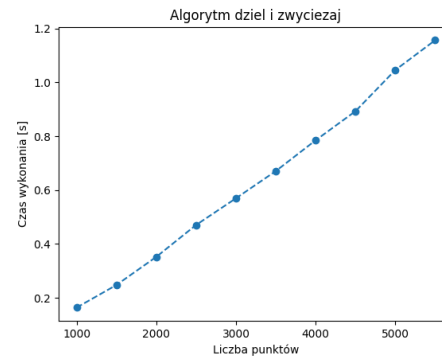
Algorytm dziel i zwyciężaj wypadł na drugim miejscu zaraz po grahamie. Jego wykres odpowiada przewidywaniom teoretycznym, jest to wykres w przybliżeniu liniowy bardzo powoli skłaniający się ku górze. Niezależnie od rozkładu punktów algorytm ten działa w zależności od ich rozmiarów.

5.11 Algorytm przyrostowy

W tabeli 6 przedstawiamy czasy uzyskane przez algorytm przyrostowy i zwyciężaj dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 21, 22, 23, 24 widzimy ilustrację danych z tabeli 6.



Rysunek 19: Zbiór typu C, algorytm dziel i zwyciężaj



Rysunek 20: Zbiór typu D, algorytm dziel i zwyciężaj

Tablica 6: Czas wykonania algorytmu przyrostowego w zależności od typu zbioru testowego oraz mocy zbioru punktów.

	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Typ zbioru	Czas wykonania [s]									
A	0.0125	0.0194	0.0251	0.0305	0.0377	0.0431	0.0497	0.0584	0.0621	0.0705
B	0.0864	0.1788	0.2978	0.4428	0.6359	0.7745	0.9738	1.1858	1.3526	1.5613
C	0.0315	0.0475	0.063	0.0793	0.0997	0.1136	0.1292	0.1449	0.1555	0.1742
D	0.1373	0.1992	0.2715	0.326	0.4165	0.4639	0.5195	0.5937	0.6773	0.7446

5.12 Porównanie z przewidywaniami teoretycznymi

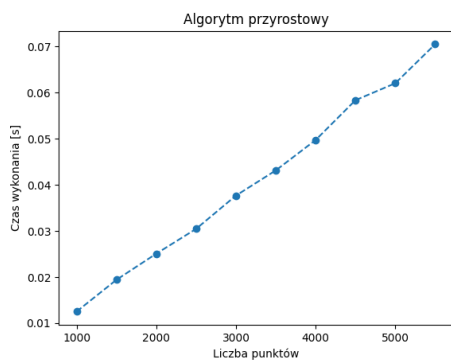
Algorytm przyrostowy wypadł odrobinę lepiej od dziel i zwyciężaj. Jego wykres odpowiada przewidywaniom teoretycznym, jest to wykres w przybliżeniu liniowy bardzo powoli skłaniający się ku górze. Niezależnie od rozkładu punktów algorytm ten działa w zależności od ich rozmiarów.

5.13 Algorytm Jarvisa

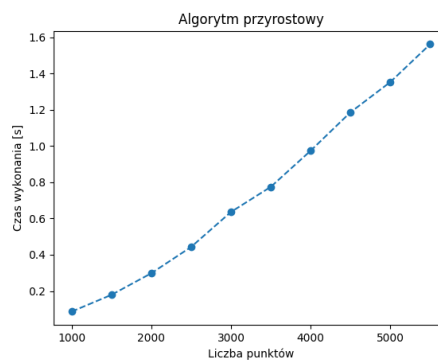
W tabeli 7 przedstawiamy czasy uzyskane przez algorytm Jarvisa i zwyciężaj dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 25, 26, 27, 28 widzimy ilustrację danych z tabeli 7.

5.14 Porównanie z przewidywaniami teoretycznymi

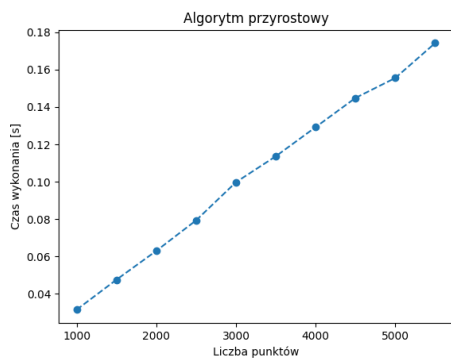
Algorytm Jarvisa zachowywał się zgodnie z przewidywaniami teoretycznymi. Jego złożoność wynosi $O(kn)$, gdzie k to liczba wierzchołków w otocze. Dla chmury punktów, gdzie liczba wierzchołków w otocze jest losowa zachowywał się nieregularnie. Dla pozostałych zbiorów: dla rozkładu kołowego ukwadratował się i miał zdecydowanie najgorsze



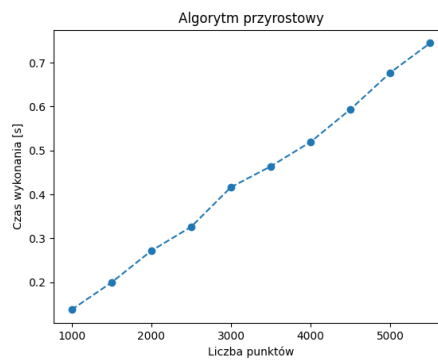
Rysunek 21: Zbiór typu A, algorytm przyrostowy



Rysunek 22: Zbiór typu B, algorytm przyrostowy



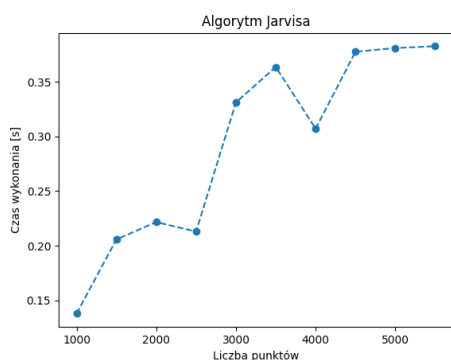
Rysunek 23: Zbiór typu C, algorytm przyrostowy



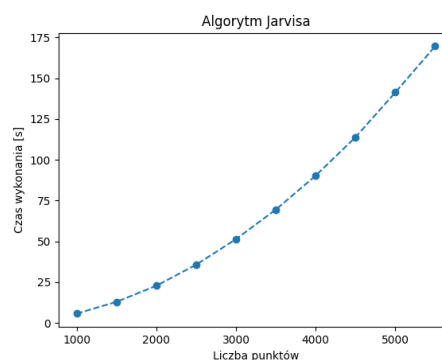
Rysunek 24: Zbiór typu D, algorytm przyrostowy

Tablica 7: Czas wykonania algorytmu Jarvisa w zależności od typu zbioru testowego oraz mocy zbioru punktów.

	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Typ danych	Czas wykonania [s]									
A	0.1379	0.2058	0.2217	0.213	0.3311	0.3635	0.307	0.3775	0.3808	0.3825
B	5.684	12.882	22.737	35.648	51.305	69.425	90.227	113.80	141.27	169.58
C	0.0529	0.077	0.1046	0.1306	0.1543	0.184	0.2086	0.2304	0.2571	0.2858
D	0.1094	0.1572	0.2167	0.2611	0.3129	0.3711	0.4218	0.4774	0.5255	0.5886



Rysunek 25: Zbiór typu A, algorytm Jarvisa



Rysunek 26: Zbiór typu B, algorytm Jarvisa

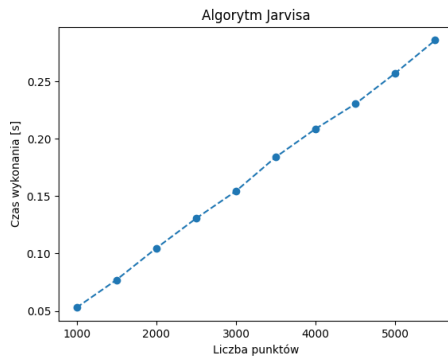
wyniki. Dla zbioru o rozmiarze 5500 wykonywał się niemal 3 min. Jednakże dla pozostałych zbiorów, jego złożoność stanowiła przewagę mógł śmiało konkurować z pozostałymi algorytmami $O(n \log n)$ i odznaczał się podobnymi czasami.

6 Porównanie algorytmów

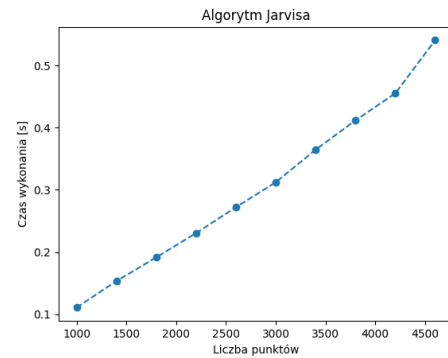
Ze względu na różną charakterystykę zbiorów punktów i zachowania algorytmów, porównania dzielimy na 4 sekcje, każda odpowiadająca wybranemu typowi zbioru punktów testowych.

Dla uproszczenia zapisu w tabelach wprowadzamy oznaczenia dla poszczególnych algorytmów:

- *GR* – algorytm Grahama
- *LU* – algorytm górna dolna
- *CH* – algorytm Chana
- *QH* – algorytm QuickHull



Rysunek 27: Zbiór typu C, algorytm Jarvisa



Rysunek 28: Zbiór typu D, algorytm Jarvisa

Tablica 8: Czasy wykonania poszczególnych algorytmów dla zbioru typu A przy różnych mocach zbiorów punktów.

	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Algorytm	Czas wykonania [s]									
<i>GR</i>	0.0239	0.0331	0.045	0.0586	0.072	0.0847	0.0983	0.1122	0.1273	0.141
<i>LU</i>	0.0076	0.0079	0.0105	0.0134	0.016	0.0186	0.0222	0.0241	0.0269	0.0295
<i>CH</i>	0.1055	0.1544	0.3575	0.2408	0.5449	0.6155	0.7104	0.7904	0.8621	0.9826
<i>QH</i>	0.0031	0.0047	0.0058	0.0067	0.012	0.0098	0.0134	0.0124	0.0136	0.0155
<i>DC</i>	0.0163	0.0251	0.0341	0.0439	0.0569	0.0631	0.0748	0.0844	0.0955	0.1076
<i>IN</i>	0.0125	0.0194	0.0251	0.0305	0.0377	0.0431	0.0497	0.0584	0.0621	0.0705
<i>JR</i>	0.1379	0.2058	0.2217	0.213	0.3311	0.3635	0.307	0.3775	0.3808	0.3825

- *DC* – algorytm dziel i zwyciężaj
- *IN* – algorytm przyrostowy
- *JR* – algorytm Jarvisa

6.1 Zbiór typu A

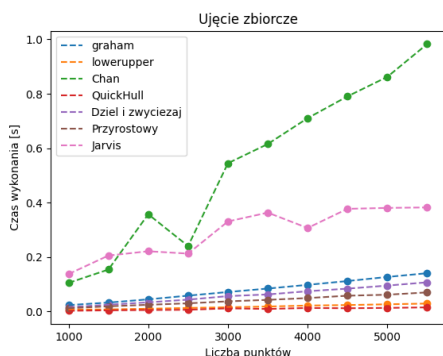
6.2 Zbiór typu B

6.3 Zbiór typu C

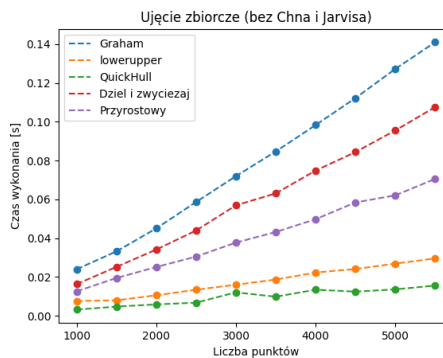
6.4 Zbiór typu D

7 Bibliografia

1. Wykład z przedmiotu *Algorytmy Geometryczne, Informatyka 3. sem., 1. st. AGH*, Barbara Głut



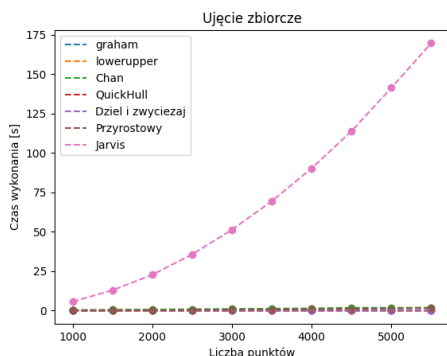
Rysunek 29: Zbiór typu A, zestawienie



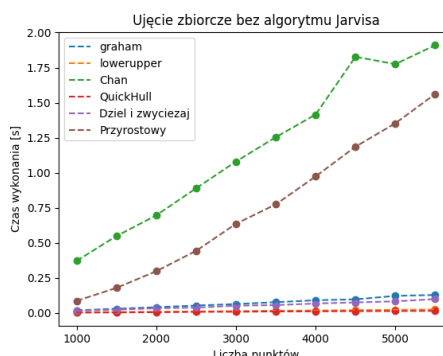
Rysunek 30: Zbiór typu A, zestawienie bez Jarvisa i Chana

Tablica 9: Czasy wykonania poszczególnych algorytmów dla zbioru typu B przy różnych mocach zbiorów punktów.

Algorytm	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
<i>GR</i>	0.0191	0.0299	0.0413	0.0531	0.0637	0.0768	0.0908	0.0973	0.122	0.1284
<i>LU</i>	0.004	0.0064	0.0086	0.0111	0.0132	0.0164	0.018	0.0203	0.0229	0.0254
<i>CH</i>	0.3714	0.5483	0.6962	0.8893	1.0799	1.2537	1.415	1.827	1.7746	1.9095
<i>QH</i>	0.0027	0.004	0.0055	0.0082	0.0089	0.011	0.0116	0.0133	0.0135	0.0153
<i>DC</i>	0.0173	0.0234	0.0339	0.0381	0.052	0.0562	0.0675	0.0754	0.0829	0.0992
<i>IN</i>	0.0864	0.1788	0.2978	0.4428	0.6359	0.7745	0.9738	1.1858	1.3526	1.5613
<i>JR</i>	5.68	12.88	22.73	35.64	51.30	69.42	90.22	113.80	141.27	169.58



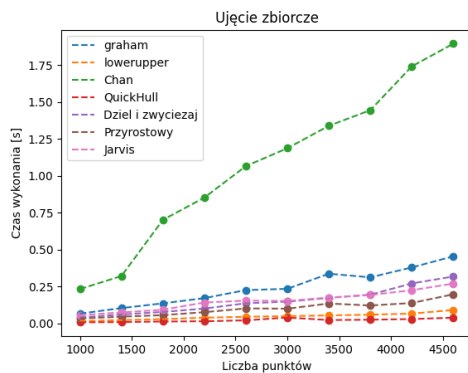
Rysunek 31: Zbiór typu B, zestawienie



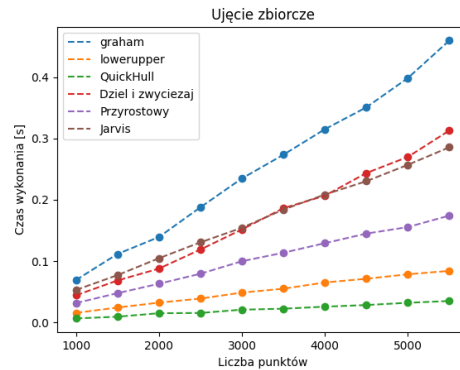
Rysunek 32: Zbiór typu B, zestawienie bez Jarvisa

Tablica 10: Czasy wykonania poszczególnych algorytmów dla zbioru typu C przy różnych mocach zbiorów punktów.

Algorytm	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Algorytm	Czas wykonania [s]									
<i>GR</i>	0.069	0.1115	0.1392	0.1878	0.2349	0.2736	0.3153	0.351	0.3988	0.4601
<i>LU</i>	0.0155	0.0241	0.0321	0.0388	0.0486	0.0549	0.0651	0.0711	0.0784	0.084
<i>CH</i>	0.2457	0.5615	0.727	0.9188	1.1007	1.3044	1.5073	1.6875	1.88	2.1234
<i>QH</i>	0.0064	0.0092	0.0148	0.0153	0.0207	0.0222	0.0256	0.0282	0.0319	0.0348
<i>DC</i>	0.0441	0.0683	0.0877	0.1189	0.1516	0.1863	0.2067	0.2437	0.27	0.3132
<i>IN</i>	0.0315	0.0475	0.063	0.0793	0.0997	0.1136	0.1292	0.1449	0.1555	0.1742
<i>JR</i>	0.0529	0.077	0.1046	0.1306	0.1543	0.184	0.2086	0.2304	0.2571	0.2858



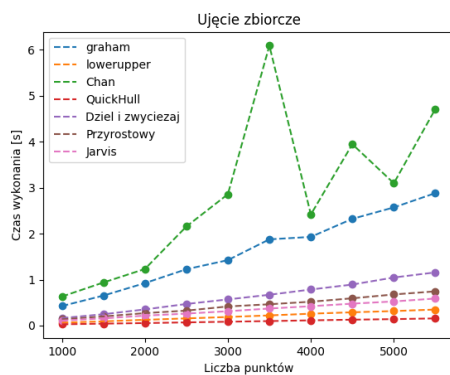
Rysunek 33: Zbiór typu C, zestawienie



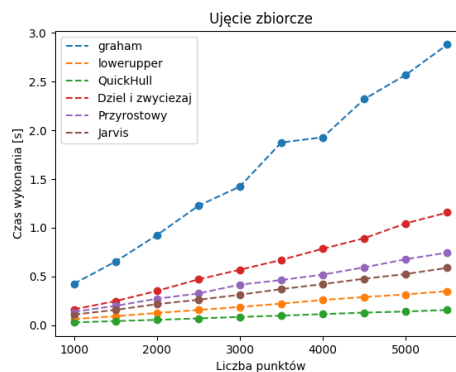
Rysunek 34: Zbiór typu C, zestawienie bez Chana

Tablica 11: Czasy wykonania poszczególnych algorytmów dla zbioru typu D przy różnych mocach zbiorów punktów.

Algorytm	Liczba punktów									
	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500
Algorytm	Czas wykonania [s]									
<i>GR</i>	0.4256	0.6523	0.924	1.2255	1.4233	1.8752	1.9285	2.3207	2.5692	2.88
<i>LU</i>	0.0635	0.0927	0.1261	0.1566	0.1875	0.2217	0.259	0.2897	0.316	0.3495
<i>CH</i>	0.6322	0.9372	1.2306	2.1543	2.8556	6.0844	2.4157	3.943	3.094	4.7075
<i>QH</i>	0.0287	0.0426	0.0551	0.0702	0.0847	0.0986	0.1143	0.1294	0.1411	0.1562
<i>DC</i>	0.1634	0.2474	0.3518	0.4703	0.5694	0.6705	0.7849	0.8928	1.0458	1.1565
<i>IN</i>	0.1373	0.1992	0.2715	0.326	0.4165	0.4639	0.5195	0.5937	0.6773	0.7446
<i>JR</i>	0.1094	0.1572	0.2167	0.2611	0.3129	0.3711	0.4218	0.4774	0.5255	0.5886



Rysunek 35: Zbiór typu D, zestawienie



Rysunek 36: Zbiór typu D, zestawienie bez Chana

2. *Computational Geometry – Algorithms and Applications*, Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars

3. <https://jeffe.cs.illinois.edu/teaching/compgeom/notes/01-convexhull.pdf>