Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Wojciech Chrobak

Nr albumu: 1137045

Technika zamiatania płaszczyzny

Praca licencjacka na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2019

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia Podpis kierującego pracą

Chcę serdecznie podziękować Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za każdą cenną poradę, liczne wskazówki i poświęcony czas, które pomogły mi dokończyć tę pracę.

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów z geometrii obliczeniowej, które wykorzystują technikę zamiatania płaszczyzny. Wykorzystano reprezentacje podstawowych obiektów geometrycznych (punkt, odcinek, prostokąt), a także zaimplementowano struktury danych niezbędne do optymalnego działania algorytmów.

Przygotowano trzy algorytmy do szukania przecięć w zbiorze odcinków: algorytm Bentleya-Ottmanna (korzysta ze zmodyfikowanego drzewa AVL) i algorytm Shamosa-Hoeya do odcinków w pozycji ogólnej oraz dodatkowy algorytm do odcinków pionowych i poziomych.

Dodatkowo zaimplementowano trzy algorytmy do szukania pary najbliższych punktów na płaszczyźnie. Pierwszy algorytm wykorzystuje technikę zamiatania, dwa następne technikę dziel i zwyciężaj.

Przedstawiono również implementację struktury danych drzewa czwórkowego, które może przechowywać punkty lub prostokątne obszary płaszczyzny. Drzewo czwórkowe pozwala na wykonanie szybkich operacji wyszukiwania danych na płaszczyźnie albo zwartego przechowywania informacji o prostokątnych fragmentach pewnego obszaru płaszczyzny.

Słowa kluczowe: geometria obliczeniowa, zamiatanie płaszczyzny, problem przecięcia odcinków, problem najbliższej pary, drzewo czwórkowe, drzewo AVL

English title: Sweep line technique

Abstract

Python implementation of selected computational geometry algorithms using a sweep line technique is presented. Standard geometrical objects are used (point, segment, rectangle) together with special data structures needed in optimal algorithms.

Three algorithms for finding line segment intersections are prepared: the Bentley-Ottmann algorithm (using a modified AVL tree) and the Shamos-Hoey algorithm (general position is assumed), the special algorithm for horizontal and vertical line segments.

Additionally, three algorithms for finding the closest pair of points in the plane are implemented. The first algorithm is using plane sweep, next algorithms are using the divide and conquer technique.

A tree data structure called a quadtree is also presented. It is used to partition a rectangular region into four subregions (with additional data) or to maintain point sets with efficient lookup.

Keywords: computational geometry, sweep line technique, line segment intersection problem, closest pair problem, quadtree, AVL tree

Spis treści

Spis rysunków

Listings

1. Wstęp

Tematem niniejszej pracy jest *technika zamiatania płaszczyzny* (ang. *sweep line technique*) [1], która jest jedną z kluczowych technik w geometrii obliczeniowej (ang. *computational geometry*) [2]. Inne nazwy spotykane w literaturze polskiej to wymiatanie, omiatanie, miotełkowanie.

Działanie algorytmów korzystających z techniki zamiatania płaszczyzny można wyobrazić sobie jako przesuwanie prostej pionowej z lewa na prawo lub prostej poziomej z dołu do góry. Prosta (miotła) zatrzymuje się w pewnych punktach, aby przetworzyć zdarzenie powiązane z danym punktem. Jest istotne, że operacje geometryczne są ograniczone do obiektów przecinających miotłę lub leżących w bezpośredniej bliskości miotły. Pełne rozwiązanie danego problemu otrzymuje się, gdy miotła minie wszystkie rozważane obiekty. Typowe obiekty omiatane przez miotłę to punkty lub odcinki. Wybrane algorytmy wykorzystujące technikę zamiatania płaszczyzny:

— Algorytm Bentleya-Ottmanna (przecinanie się odcinków) [3].
— Algorytm Shamosa-Hoeya (przecinanie się odcinków) [4].
— Algorytm Fortune'a (konstrukcja diagramu Voronoi) [5].

Wynalezienie techniki zamiatania płaszczyzny przyniosło przełom w geometrii obliczeniowej. Algorytm Shamosa-Hoeya wykrywania przecięć odcinków obniżył granicę złożoności z $O(n^2)$ do $O(n \log n)$. Pewne algorytmy używające techniki zamiatania są czułe na wyjście (ang. *output-sensitive algorithms*), czyli ich złożoność obliczeniowa zależy od wielkości rozwiązania, a nie tylko od wielkości danych wejściowych. Przykładem jest liczba k punktów przecięcia odcinków w algorytmie Bentleya-Ottmanna. Algorytm będzie wydajny dla k mniejszego lub porównywalnego z liczbą odcinków.

Algorytmy z geometrii obliczeniowej pojawiają się w ogólnych podręcznikach do algorytmów i struktur danych [6], [7], są też książki poświęcone tylko algorytmom geometrycznym [8], [9]. W opisach na ogół pomija się szczegóły użytych struktur danych, bez których trudno osiągnąć teoretyczną złożoność obliczeniową. Naszym celem jest podanie pełnego i przetestowanego kodu, który w pełni wyjaśni nieraz subtelne szczegóły. Do implementacji algorytmów użyto języka Python [10], ponieważ łączy on czytelność kodu, bogatą bibliotekę standardową i przyjemność programowania. Implementację podstawowych obiektów geometrycznych zaczerpnięto z pracy Permusa [11].

Organizacja niniejszej pracy jest następująca. Rozdział 1 podaje cele pracy. Rozdział 2 wprowadza w geometrię obliczeniową. Rozdział 3 prezentuje interfejs obiektów geometrycznych i przykładowe obliczenia. Rozdział 4 opisuje implementacje algorytmów geometrycznych. Rozdział 5 jest podsumowaniem pracy. Dodatek A przedstawia wyniki testów wydajnościowych.

2. Geometria obliczeniowa

Rozdział ten poświęcony jest wyjaśnieniu podstawowych zagadnień z zakresu algorytmów geometrii obliczeniowej na płaszczyźnie. Omówimy problemy związane z przecinaniem się odcinków i bliskością punktów.

2.1. Przecinanie się odcinków na płaszczyźnie

Problem znajdowania punktów przecięcia dla zbioru odcinków ma wiele zastosowań [16]:

— Obliczanie przecięcia, sumy lub różnicy dwóch wielokątów prostych lub dwóch grafów płaskich. Należy wyznaczyć wszystkie punkty przecięcia.
— Sprawdzanie, czy dwa wielokąty lub grafy płaskie przecinają się. Wystarczy znaleźć jeden punkt przecięcia.
— Sprawdzanie, czy wielokąt jest prosty. Tutaj trzeba sprawdzić przecinanie się każdej pary niesąsiednich krawędzi. Wystarczy znaleźć jeden punkt przecięcia.
— Rozkładanie wielokąta na części proste. Należy wyznaczyć wszystkie punkty przecięcia.

Zauważmy, że czasem szukamy przecięć dla każdej pary odcinków pochodzącej z jednego zbioru, a czasem wymagamy, aby odcinki w parze pochodziły z dwóch różnych zbiorów (ang. *red-blue intersection problem*).

Dla n odcinków może być w najgorszym razie $n(n-1)/2 = O(n^2)$ punktów przecięcia. Wtedy prosty algorytm siłowy może zbadać każdą parę odcinków i wyznaczyć punkt przecięcia w czasie $O(n^2)$. Jeżeli punktów przecięcia jest rzędu $O(n)$, to chcielibyśmy mieć szybszy algorytm. Rozwiązaniem może być algorytm czuły na wyjście, czyli algorytm o złożoności zależnej od rozmiaru wejścia i rozmiaru wyjścia.

2.2. Bliskość dla punktów na płaszczyźnie

Wybrane problemy związane z bliskością punktów na płaszczyźnie [9].

— Dla n danych punktów na płaszczyźnie, znaleźć takie dwa punkty, których wzajemna odległość jest najmniejsza. Może istnieć kilka takich par punktów, ale wystarczy znaleźć jedną.
— Dla n danych punktów na płaszczyźnie, znaleźć dla każdego z nich najbliższego sąsiada. W dwóch wymiarach punkt może mieć najwyżej sześciu najbliższych sąsiadów.

— Dla n danych punktów na płaszczyźnie, stworzyć drzewo o najmniejszej całkowitej długości, którego wierzchołki są danymi punktami. Jest to *minimalne drzewo euklidesowe EMST* (ang. *Euclidean minimum spanning tree*).
— Dla n danych punktów na płaszczyźnie, połączyć je nieprzecinającymi się odcinkami tak, aby każdy obszar wewnątrz otoczki wypukłej był trójkątem. Jest to problem *triangulacji* zbioru punktów.
— Dla n danych punktów na płaszczyźnie, znaleźć najbliższego sąsiada nowego punktu q.

3. Implementacja

W tym rozdziale zostaną przedstawione podstawowe obiekty wykorzystane w pracy, a także założenia naszej implementacji.

3.1. Założenia implementacyjne

W geometrii obliczeniowej niezwykle ważną kwestią jest poprawna reprezentacja liczb w komputerze. Przy korzystaniu z liczb **float** nieunikniona jest skończona dokładność działań, co należy uwzględnić przy implementacji działań matematycznych. W naszej pracy obliczenia będą oparte na liczbach wymiernych z klasy Fraction, dzięki czemu przy danych wejściowych całkowitych lub wymiernych wyniki będą dokładne (całkowite lub wymierne). W przypadku danych wejściowych typu **float** wyniki automatycznie będą też typu **float**.

3.2. Podstawowe obiekty geometryczne

Aby dany problem mógł zostać poprawnie rozwiązany, potrzebujemy dobrze zdefiniowanych reprezentacji obiektów geometrycznych w pamięci komputera. Nasza implementacja zakłada, że wszystkie obiekty są hashowalne i niezmienne, dzięki czemu mogą być m.in. umieszczane w zbiorach. Poniżej przedstawione są najczęściej wykorzystywane obiekty w geometrii obliczeniowej.

3.2.1. Punkt

Obiekt punktu wewnętrznie reprezentowany jest przez parę współrzędnych (x, y) w układzie kartezjańskim.

Listing 3.1. Użycie klasy Point z modułu points.

```
>>> from points import Point
# Inicjalizacja punktu.
>>> p = Point(x, y)
# Wyswietlanie punktu.
>>> print p
# Porownywanie punktow.
>>> p > q, p < q
# Dzialania na punktach jako wektorach.
>>> p + q, p - q, +p, -p
# Iloczyn skalarny.
>>> p * q
# Iloczyn wektorowy 2D.
>>> p.cross(q)
```

```
# Kopiowanie punktu.
>>> q = p.copy()
# Dlugosc wektora.
>>> p.length()
# Hashowanie punktu.
>>> hash(p)
```

3.2.2. Odcinek

Obiekt odcinka wewnętrznie reprezentowany jest przez parę punktów (pt1, pt2) oznaczających odpowiednio jego początek i koniec. Jest to więc odcinek skierowany.

Listing 3.2. Użycie klasy Segment z modułu segments.

```
>>> from segments import Segment
# Inicjalizacja odcinka.
>>> s = Segment(x1, y1, x2, y2)
>>> t = Segment(p, q)
# Wyswietlanie odcinka.
>>> print s
# Porownywanie odcinkow.
>>> s == t,  s != t
# Kopiowanie odcinka.
>>> t = s.copy()
# Wyznaczanie srodka odcinka.
>>> p = s.center()
# Dlugosc odcinka.
>>> s.length()
# Sprawdzanie czy punkt nalezy do odcinka.
>>> p in s, q not in s
# Przecinanie sie odcinkow (bool).
>>> s.intersect(t)
# Wyznaczenie punktu przeciecia sie dwoch odcinkow.
>>> p = s.intersection_point(t)
# Przesuniecie odcinka o wektor.
>>> t = s.move(x, y)
>>> t = s.move(p)
# Hashowanie odcinka.
>>> hash(s)
```

3.2.3. Prostokąt

Obiekt prostokąta wewnętrznie reprezentowany jest przez parę punktów (pt1, pt2) oznaczających jego lewy dolny i prawy górny róg. Boki prostokąta są ułożone równolegle do osi układu współrzędnych.

Listing 3.3. Użycie klasy Rectangle z modułu rectangles.

```
>>> from rectangles import Rectangle
# Inicjalizacja prostokata.
>>> s = Rectangle(x1, y1, x2, y2)
>>> t = Rectangle(p, q)
# Wyswietlanie prostokata.
>>> print s
```

```
# Porownywanie prostokatow.
>>> s == t, s != t
# Kopiowanie prostokata.
>>> t = s.copy()
# Wyznaczanie srodka prostokata.
>>> p = s.center()
# Obliczanie pola powierzchni prostokata.
>>> s.area()
# Podzial prostokata na cztery mniejsze jednakowe.
>>> tl, tr, bl, br = s.make4()
# Sprawdzanie czy punkt nalezy do prostokata.
>>> p in s, q not in s
# Prostokat pokrywajacy dwa prostokaty.
>>> r = s.cover(t)
# Wyznaczanie czesci wspolnej dwoch prostokatow.
>>> r = s.intersection(t)
# Test czy prostokat jest kwadratem (bool).
>>> s.is_square()
# Przesuniecie prostokata o wektor.
>>> t = s.move(x, y)
>>> t = s.move(p)
# Hashowanie prostokata.
>>> hash(s)
```

3.3. Struktury danych

Aby osiągnąć teoretyczną złożoność obliczeniową rozważanych algorytmów potrzebujemy dobrej podstawy w postaci odpowiednich struktur danych. Niektóre algorytmy zmuszają nas do modyfikacji tradycyjnych wersji struktur danych opisanych w literaturze.

3.3.1. Zdarzenie

Struktura wykorzystana jest do przechowywania informacji o zdarzeniu (ang. *event*), w którym prosta zamiatająca się zatrzymuje. Będzie to klasa Event, która definiuje atrybuty klasy opisujące typy zdarzeń. Jest reprezentowana za pomocą punktu, typu zdarzenia i odcinka (w pewnych przypadkach dwóch odcinków). Jedyną operacją wykonywaną na tym obiekcie jest operator porównywania, w naszej implementacji wyższy priorytet ma współrzędna x. Na podstawie typu zdarzenia w rozważanych algorytmach wybierana jest obsługa tego zdarzenia. Obiekt ten jest hashowalny.

Użycie klasy: Algorytmy Bentleya-Ottmanna (4.1), Shamosa-Hoeya (4.2), przecinanie się odcinków pionowych i poziomych (4.3).

Listing 3.4. Klasa Event z modułu events.

```
#!/usr/bin/python

class Event:
    """The class defining an event."""
    LEFT = 0
    CROSSING = 1
```

```
    RIGHT = 2
    HORIZONTAL = 3
    VERTICAL = 4

    def __init__(self, pt, event_type, *sequence):
        self.pt = pt
        self.type = event_type
        if len(sequence) == 1:
            self.segment = sequence[0]
        elif len(sequence) == 2:
            self.segment_above = sequence[0]
            self.segment_below = sequence[1]

    def __str__(self):
        return "Event({}, {})".format(self.pt, self.type)

    def __cmp__(self, other):
        return cmp(self.pt, other.pt)

    def __hash__(self):
        """Hashable events."""
        return hash((self.pt, self.type))    # hash based on tuple
```

3.3.2. Kolejka priorytetowa

Kolejka priorytetowa (ang. *priority queue*) jest kolejką, w której elementy są zwracane w kolejności od najmniejszego priorytetu [12]. Nasza implementacja klasy PriorityQueue wykorzystuje moduł kopca heapq z biblioteki standardowej Pythona [13]. Kolejka została rozbudowana dodatkowym zbiorem elementów item_set, który pozwala na wykonanie operacji dodawania, usuwania i wyszukiwania w czasie $O(1)$. Skutkiem tej modyfikacji jest wykluczenie powtórzeń (co wynika z założeń naszej pracy - pozycja ogólna) bez negatywnej zmiany złożoności czasowej operacji push() i pop(). Obie te metody działają w czasie $O(\log n)$.

Użycie klasy: Algorytmy Bentleya-Ottmanna (4.1), Shamosa-Hoeya (4.2).

Listing 3.5. Klasa PriorityQueue z modułu priority_queue.

```
#!/usr/bin/python

import heapq

class PriorityQueue:
    """A priority queue without repetitions, O(log n) time operations."""

    def __init__(self):
        self.heap = []
        self.item_set = set()    # quick search

    def __str__(self):
        return str(self.heap)

    def push(self, item):
        if item not in self.item_set:
```

```
            heapq.heappush(self.heap, item)
            self.item_set.add(item)

    def pop(self):
        if not self.is_empty():
            item = heapq.heappop(self.heap)
            self.item_set.remove(item)
            return item
        else:
            raise ValueError("Queue is empty")

    def is_empty(self):
        return len(self.heap) == 0

    def __len__(self):
        return len(self.heap)
```

3.3.3. Drzewo AVL - klucze statyczne

Jest to klasyczna struktura zrównoważonego drzewa poszukiwań binarnych (ang. *binary search tree*), umieszczona w klasie AVLTree. Wysokość lewego i prawego poddrzewa każdego węzła różni się co najwyżej o jeden. Wyklucza to przypadki zdegenerowane, gdy jedna strona drzewa jest nieproporcjonalnie większa od drugiej. Elementy w lewym poddrzewie są mniejsza od wierzchołka, w prawym - większe. Po każdej operacji dodawania i usuwania elementu wykonywane są odpowiednie rotacje, aby zachować zrównoważenie drzewa. Własności te gwarantują, że pesymistyczny czas wyszukiwania elementu w drzewie o n węzłach wynosi $O(\log n)$ [14].

Nasza implementacja oparta jest o kod opublikowany na Rosetta Code [24] a także o bibliotekę PyBST [23].

Użycie klasy: Algorytm wyznaczania pary najbliższych punktów - technika zamiatania (4.5), przecinanie się odcinków pionowych i poziomych (4.3).

Listing 3.6. Moduł avltree - klucze statyczne.

```
#!/usr/bin/python

# https://rosettacode.org/wiki/AVL_tree#Python
# https://github.com/TylerSandman/py-bst

class Node:
    """The class defining a node."""

    def __init__(self, value):
        self.value = value
        self.parent = None
        self.left = None
        self.right = None

    def __repr__(self):
        return str(self.value)

    def insert(self, node):
```

```python
        if node is None:
            return
        if node.value < self.value:
            if self.left is None:
                node.parent = self
                self.left = node
            else:
                self.left.insert(node)
        else:
            if self.right is None:
                node.parent = self
                self.right = node
            else:
                self.right.insert(node)

    def find(self, value):
        if value == self.value:
            return self
        elif value < self.value:
            if self.left is None:
                return None
            else:
                return self.left.find(value)
        else:
            if self.right is None:
                return None
            else:
                return self.right.find(value)

    def find_min(self):
        current = self
        while current.left is not None:
            current = current.left
        return current

    def find_max(self):
        current = self
        while current.right is not None:
            current = current.right
        return current

    def successor(self):
        if self.right is not None:
            return self.right.find_min()
        current = self
        while current.parent is not None and current is current.parent.right:
            current = current.parent
        return current.parent

    def predecessor(self):
        if self.left is not None:
            return self.left.find_max()
        current = self
        while current.parent is not None and current is current.parent.left:
            current = current.parent
        return current.parent
```

```python
def height(node):
    if node is None:
        return -1
    else:
        return node.height


def update_height(node):
    node.height = max(height(node.left), height(node.right)) + 1


class AVLTree:
    """The class defining an AVL tree."""

    def __init__(self):
        self.root = None

    def insert(self, value):
        node = Node(value)
        if self.root is None:
            self.root = node
        else:
            self.root.insert(node)
        self.rebalance(node)

    def find(self, k):
        return self.root and self.root.find(k)

    def delete(self, value):
        node = self.find(value)
        if node:
            if not (node.left or node.right):
                self._delete_leaf(node)
            elif not (node.left and node.right):
                self._delete_leaf_parent(node)
            else:
                self._delete_node(node)

    def _delete_leaf(self, node):
        parent_node = node.parent

        if parent_node:
            if parent_node.left == node:
                parent_node.left = None
            else:
                parent_node.right = None

            del node
            self.rebalance(parent_node)
        else:
            self.root = None

    def _delete_leaf_parent(self, node):
        parent_node = node.parent
        if node.value == self.root.value:
            if node.right:
```

```
                self.root = node.right
                node.right = None
                self.root.parent = None
            else:
                self.root = node.left
                node.left = None
                self.root.parent = None

        else:
            if parent_node.right == node:
                if node.right:
                    parent_node.right = node.right
                    parent_node.right.parent = parent_node
                    node.right = None
                else:
                    parent_node.right = node.left
                    parent_node.right.parent = parent_node
                    node.left = None
            else:

                if node.right:
                    parent_node.left = node.right
                    parent_node.left.parent = parent_node
                    node.right = None
                else:
                    parent_node.left = node.left
                    parent_node.left.parent = parent_node
                    node.left = None

        del node
        self.rebalance(parent_node)

    def _switch_nodes(self, node1, node2):
        switch1 = node1
        switch2 = node2
        temp_value = switch1.value

        if switch1.value == self.root.value:
            self.root.value = node2.value
            switch2.value = temp_value
        elif switch2.value == self.root.value:
            switch1.value = self.root.value
            self.root.value = temp_value
        else:
            switch1.value = node2.value
            switch2.value = temp_value

    def _delete_node(self, node):
        if height(node.left) > height(node.right):
            to_switch = node.predecessor()
            self._switch_nodes(node, to_switch)

            if not (to_switch.right or to_switch.left):
                to_delete = node.predecessor()
                self._delete_leaf(to_delete)
            else:
                to_delete = node.predecessor()
```

```python
            self._delete_leaf_parent(to_delete)
    else:
        to_switch = node.successor()
        self._switch_nodes(node, to_switch)

        if not (to_switch.right or to_switch.left):
            to_delete = node.successor()
            self._delete_leaf(to_delete)
        else:
            to_delete = node.successor()
            self._delete_leaf_parent(to_delete)

def successor(self, value):
    node = self.find(value)
    return node and node.successor()

def predecessor(self, value):
    node = self.find(value)
    return node and node.predecessor()

def rebalance(self, node):
    while node is not None:
        update_height(node)
        if height(node.left) >= 2 + height(node.right):
            if height(node.left.left) >= height(node.left.right):
                self.right_rotate(node)
            else:
                self.left_rotate(node.left)
                self.right_rotate(node)
        elif height(node.right) >= 2 + height(node.left):
            if height(node.right.right) >= height(node.right.left):
                self.left_rotate(node)
            else:
                self.right_rotate(node.right)
                self.left_rotate(node)
        node = node.parent

def left_rotate(self, x):
    y = x.right
    y.parent = x.parent
    if y.parent is None:
        self.root = y
    else:
        if y.parent.left is x:
            y.parent.left = y
        elif y.parent.right is x:
            y.parent.right = y
    x.right = y.left
    if x.right is not None:
        x.right.parent = x
    y.left = x
    x.parent = y
    update_height(x)
    update_height(y)

def right_rotate(self, x):
    y = x.left
```

```
        y.parent = x.parent
        if y.parent is None:
            self.root = y
        else:
            if y.parent.left is x:
                y.parent.left = y
            elif y.parent.right is x:
                y.parent.right = y
        x.left = y.right
        if x.left is not None:
            x.left.parent = x
        y.right = x
        x.parent = y
        update_height(x)
        update_height(y)
```

3.3.4. Drzewo AVL - klucze dynamiczne

Jest to zmodyfikowana wersja klasycznego drzewa AVL niezbędna do poprawnego działania algorytmów szukania przecięć odcinków. Struktura ta pozwala na umieszczanie węzłów z kluczami, których wartości zmieniają się podczas działania algorytmu (przesuwaniu się miotły). Jest to wymagana modyfikacja ze względu na charakterystykę odcinków, które mogą zmieniać swoje położenie względem innych przy zmianie punktu odniesienia - w naszym przypadku jest to współrzędna x wyznaczająca położenie miotły.

Główną różnicą w stosunku do drzewa z kluczami statycznymi są dodatkowe atrybuty klasy AVLTree:
— Położenie miotły w postaci współrzędnej x. Jest ona kluczowa w trakcie dodawania nowego węzła, aby poprawnie określić położenie odcinka względem innych, obecnych już w drzewie. Na jej podstawie wyliczana jest wartość y odcinka w danym momencie.
— Słownik zawierający wszystkie węzły, potrzebny do szybkiego wyszukiwania węzła. Bez tej modyfikacji nie mamy pewności poprawnego wyszukiwania odcinka, ponieważ w przypadku przecięć dwa odcinki mają taki sam klucz (wartość y), mimo że są unikalne.

Użycie klasy: Algorytmy Bentleya-Ottmanna (4.1), Shamosa-Hoeya (4.2).

Poniżej przedstawione są główne zmiany implementacji w stosunku do klasycznego drzewa AVL (3.7).

Listing 3.7. Moduł avltree2 - klucze dynamiczne.

```
class Node:
    """The class defining a node."""

    def calculate(self, x):
        return self.value.calculate_y(x)

    def insert(self, node, x):
        if node is None:
            return
```

```
        if node.calculate(x) < self.calculate(x):
            if self.left is None:
                node.parent = self
                self.left = node
            else:
                self.left.insert(node, x)
        else:
            if self.right is None:
                node.parent = self
                self.right = node
            else:
                self.right.insert(node, x)


class AVLTree:
    """The class defining an AVL tree."""

    def __init__(self):
        self.root = None
        self.D = dict()
        self.current_x = None

    def insert(self, value):
        node = Node(value)
        if self.root is None:
            self.root = node
        else:
            self.root.insert(node, self.current_x)
        self.D[value] = node
        self.rebalance(node)

    def delete(self, value):
        node = self.D.get(value)
        if node:
            if not (node.left or node.right):
                self._delete_leaf(node)
                del self.D[value]
            elif not (node.left and node.right):
                self._delete_leaf_parent(node)
                del self.D[value]
            else:
                self._delete_node(node)

    def _switch_nodes(self, node1, node2):
        self.D[node1.value] = node2
        self.D[node2.value] = node1

        switch1 = node1
        switch2 = node2
        temp_value = switch1.value

        if switch1.value == self.root.value:
            self.root.value = node2.value
            switch2.value = temp_value
        elif switch2.value == self.root.value:
            switch1.value = self.root.value
            self.root.value = temp_value
```

```
        else:
            switch1.value = node2.value
            switch2.value = temp_value

    def find(self, value):
        return self.D.get(value)

    def successor(self, value):
        node = self.D.get(value)
        return node and node.successor()

    def predecessor(self, value):
        node = self.D.get(value)
        return node and node.predecessor()
```

3.4. Przykładowe użycie algorytmów

Poniżej przedstawiono przykładowe użycie w praktyce algorytmów zaimplementowanych w tej pracy.

Przykład 1: Przecinanie się odcinków na płaszczyźnie.

```
>>> from segments import Segment

# Utworzenie zbioru odcinkow w pozycji ogolnej.
>>> segments = set()
>>> segments.add(Segment(9, 11, 0, 2))
>>> segments.add(Segment(4, 0, 11, 7))
>>> segments.add(Segment(10, 2, 1, 11))
>>> segments.add(Segment(2, 6, 7, 1))

# Wyznaczenie wszystkich punktow przeciecia.
>>> from bentleyottmann import BentleyOttmann
>>> i_points = BentleyOttmann(segments).run()
>>> print(i_points)
[ ... ]

# Sprawdzenie czy zbior odcinkow ma jakiekolwiek przeciecie.
>>> from shamoshoey import ShamosHoey
>>> print(ShamosHoey(segments).run())
True
>>> segments2 = set()
>>> segments2.add(Segment(1, 2, 3, 0))
>>> segments2.add(Segment(2, 3, 4, 4))
>>> print(ShamosHoey(segments2).run())
False

# Utworzenie zbioru odcinkow pionowych i poziomych.
>>> segments3 = set()
>>> segments3.add(Segment(1, 3, 1, 7))
>>> segments3.add(Segment(3, 1, 3, 5))
>>> segments3.add(Segment(5, 0, 5, 8))
>>> segments3.add(Segment(2, 2, 6, 2))
>>> segments3.add(Segment(0, 6, 6, 6))
```

```
# Wyznaczanie wszystkich przeciec odcinkow pionowych i poziomych.
>>> from horizontalvertical import HorizontalVertical
>>> i_points2 = HorizontalVertical(segments3).run()
>>> print(i_points2)
[ ... ]
```

Przykład 2: Bliskość punktów na płaszczyźnie.

```
>>> from points import Point
>>> from random import random

# Wygenerowanie zbioru 10 punktow na plaszczyznie.
>>> plist = set([Point(random(), random()) for _ in range(10)])

# Wyznaczenie pary najblizszych punktow.
>>> from closestpair2 import ClosestPair
>>> print(ClosestPair(plist).run())
(..., ...)

# Interfejs wszystkich zaimplementowanych algorytmow
# do wyznaczania pary najblizszych punktow jest identyczny
# ClosestPair(points).run()
```

Przykład 3: Drzewo czwórkowe.

```
>>> from rectangles import Rectangle
>>> from points import Point
>>> from quadtree import QuadTree

# Utworzenie drzewa czworkowego na kwadracie 1x1
# z pojemnoscia rowna 4.
>>> qt = QuadTree(Rectangle(0, 0, 1, 1), 4)

# Wprowadzanie punktow do drzewa.
>>> qt.insert(Point(0.1, 0.1))
True
>>> qt.insert(Point(0.2, 0.2))
True
>>> qt.insert(Point(0.2, 0.1))
True
>>> qt.insert(Point(0.1, 0.2))
True

# Liczba punktow na najwyzszym wezle drzewa.
>>> print(len(qt.point_list))
4

# Dodanie piatego punktu wyzwala podzial drzewa.
>>> qt.insert(Point(0.9, 0.9))
True
>>> print(len(qt.point_list))
0
>>> print(len(qt.top_left.point_list))
0
>>> print(len(qt.top_right.point_list))
1
```

```
>>> print(len(qt.bottom_left.point_list))
4
>>> print(len(qt.bottom_right.point_list))
0

# Proba dodania punktu lezacego poza plaszczyzna.
>>> qt.insert(Point(2.0, 1.5))
False

# Szukanie punktow nalezacych do danej plaszczyzny.
>>> qt.insert(Point(0.4, 0.6))
True
>>> qt.insert(Point(0.6, 0.6))
True
>>> p_in_range = qt.query(Rectangle(0.2, 0.1, 0.7, 0.7))
>>> print(p_in_range)
[Point(0.4, 0.6),
Point(0.6, 0.6),
Point(0.2, 0.1),
Point(0.2, 0.2)]

# Szukanie najblizszego sasiada danego punktu.
>>> print(qt.nearest(Point(0.7, 0.5)))
Point(0.6, 0.6)
```

4. Algorytmy

W tym rozdziale zaprezentujemy implementację niektórych algorytmów wykorzystywanych w geometrii obliczeniowej. Skupimy uwagę na specjalnej grupie takich algorytmów, które wykorzystują technikę zamiatania.

4.1. Algorytm Bentleya-Ottmanna

Algorytm Bentleya-Ottmanna (1979) nazywany jest algorytmem prostej zamiatającej (ang. *sweep line algorithm*), ponieważ działanie algorytmu można opisać jako przesuwanie się prostej (miotły) po zbiorze odcinków i odpowiednie przetwarzanie informacji o odcinkach [16].

Założenia implementacji są następujące.

— Żadne dwa końce odcinków ani punkty przecięcia nie mogą mieć takiej samej współrzędnej x. Zatem zabronione są odcinki pionowe. Koniec jednego odcinka nie może leżeć powyżej końca innego odcinka.

— Żadne trzy odcinki nie przecinają się w tym samym punkcie.

Dane wejściowe: Zbiór n odcinków.

Dane wyjściowe: Zbiór wszystkich k punktów przecięcia odcinków.

Problem: Znajdowanie punktów przecięcia zbioru odcinków.

Opis algorytmu: Algorytm wykorzystuje dwie struktury danych: X-strukturę i Y-strukturę. X-struktura jest kolejką priorytetową zdarzeń (ang. *events*) opartą zwykle na kopcu (ang. *heap*). Zdarzenie może być początkiem odcinka, końcem odcinka lub punktem przecięcia dwóch odcinków. Podczas inicjalizacji X-strukturę wypełniamy wyłącznie końcami odcinków wejściowych. Zdarzenia w X-strukturze są uporządkowane niemalejąco względem pierwszej współrzędnej (x) punktów.

Y-struktura jest zbiorem odcinków, które w danym położeniu prostej zamiatającej przecinają tę prostą. Odcinki są uporządkowane niemalejąco względem drugiej współrzędnej (Y) punktów przecięcia odcinków z miotłą. Wykorzystamy tutaj zmodyfikowane drzewo AVL, czyli zrównoważone binarne drzewo przeszukiwań, pozwalające na wykonanie poniższych operacji w czasie $O(\log n)$, gdzie n oznacza tutaj liczbę odcinków w Y-strukturze.

— insert(s) - dodanie odcinka s do Y-struktury.

— delete(s) - usunięcie odcinka s z Y-struktury.

— successor(s) - wyszukanie następnika odcinka s w Y-strukturze.

— predecessor(s) - wyszukanie poprzednika odcinka s w Y-strukturze.

Początkowo Y-struktura jest pusta, ponieważ możemy sobie wyobrazić, że miotła startuje w punkcie $x = -\infty$. Algorytm rozpoczynamy od inicjalizacji dwóch powyższych struktur. Następnie sprawdzamy, czy kolejka zdarzeń jest pusta. Jeśli nie, to pobieramy kolejne zdarzenie i obsługujemy je w zależności od typu.

— Początek odcinka s.
  Dodajemy odcinek s do Y-struktury. Niech s_1 i s_2 będą kolejno poprzednikiem i następnikiem odcinka s w aktualnym położeniu miotły. Jeśli odcinek s_1 istnieje, sprawdzamy, czy przecina się z odcinkiem s na prawo od miotły. Wykryte przecięcie dodajemy do kolejki zdarzeń. Analogicznie sprawdzamy przecięcie odcinka s_2 z odcinkiem s.
— Koniec odcinka s.
  Niech s_1 i s_2 będą kolejno poprzednikiem i następnikiem odcinka s w aktualnym położeniu miotły. Usuwamy odcinek s z Y-struktury. Jeśli s_1 i s_2 istnieją, sprawdzamy ich potencjalne przecięcie na prawo od miotły i dodajemy do kolejki zdarzeń.
— Przecięcie dwóch odcinków s_1 i s_2.
  Dodajemy punkt zdarzenia do wynikowego zbioru punktów przecięć. Niech s_1 będzie odcinkiem znajdującym się nad odcinkiem s_2 w aktualnym położeniu miotły. Zamieniamy ich kolejność w drzewie tak, aby odcinek s_2 był bezpośrednio nad odcinkiem s_1. Niech odcinek s_2' będzie górnym sąsiadem (następnikiem) odcinka s_2, natomiast odcinek s_1' będzie dolnym sąsiadem (poprzednikiem) odcinka s_1. Sprawdzamy potencjalne przecięcia odcinka s_2 z odcinkiem s_2' i odcinka s_1 z odcinkiem s_1' na prawo od miotły i dodajemy je do kolejki zdarzeń.

Po wyczerpaniu się zdarzeń z X-struktury otrzymujemy wszystkie możliwe punkty przecięcia odcinków wejściowych.

Złożoność: Złożoność czasowa algorytmu wynosi $O((n+k)\log n)$, złożoność pamięciowa wynosi $O(n + k)$. Testy potwierdzające złożoność zamieszczone są w dodatku A.

Uwaga 1: Warto zwrócić uwagę, że ruch miotły w kierunku x powoduje przesuwanie się punktów przecięcia odcinków z miotłą. Y-struktura utrzymuje wzajemne uporządkowanie odcinków, a w punktach przecięcia odcinków zmienia ich kolejność.

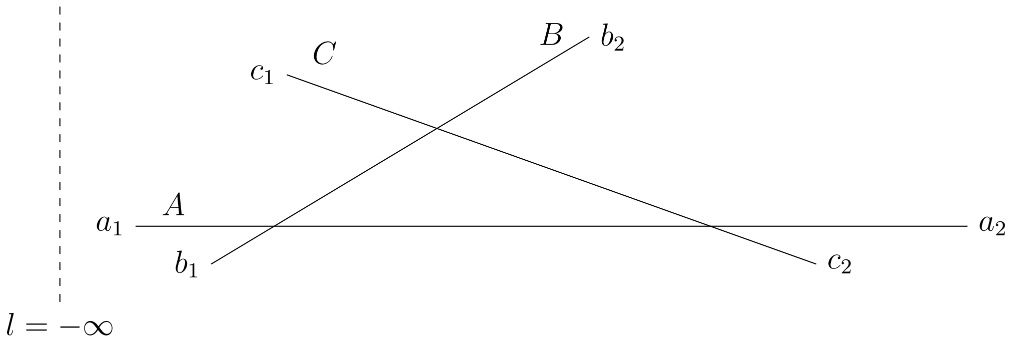
Uwaga 2: Istotną sprawą jest taka implementacja kolejki priorytetowej, która nie pozwala na powtórne wstawienie do kolejki tego samego zdarzenia (punktu przecięcia odcinków). Aby to osiągnąć, obok struktury kopca wykorzystaliśmy zbiór zdarzeń, który umożliwia szybkie wyszukiwanie zdarzenia w kolejce. Zdarzenia zostały zaimplementowane jako obiekty hashowalne.

Uwaga 3: Algorytm może być zmodyfikowany tak, aby obliczał wszystkie punkty przecięcia prostych, a nie odcinków. Wtedy już na początku ($x = -\infty$) należy wstawić do miotły wszystkie proste w kolejności malejącego nachylenia. W kolejce zdarzeń znajdą się tylko punkty przecięcia prostych.

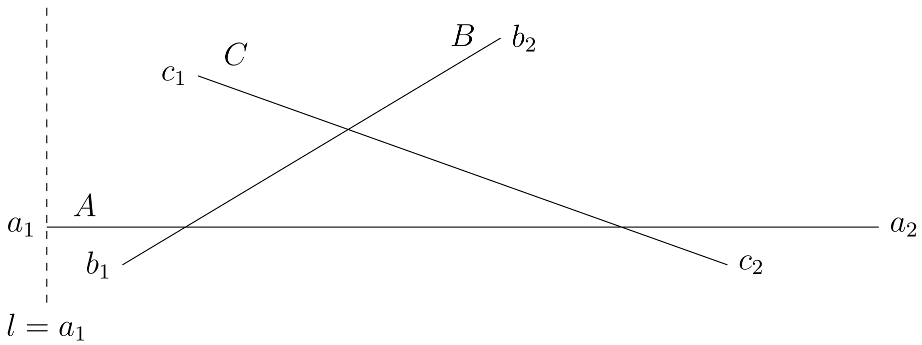
Kolejka zdarzeń musi być zainicjalizowana punktami przecięcia prostych sąsiadujących ze sobą dla $x = -\infty$ [17].

Etapy działania algorytmu: Poniżej przedstawiono kolejne etapy przesuwania się miotły i zmiany w obu strukturach.

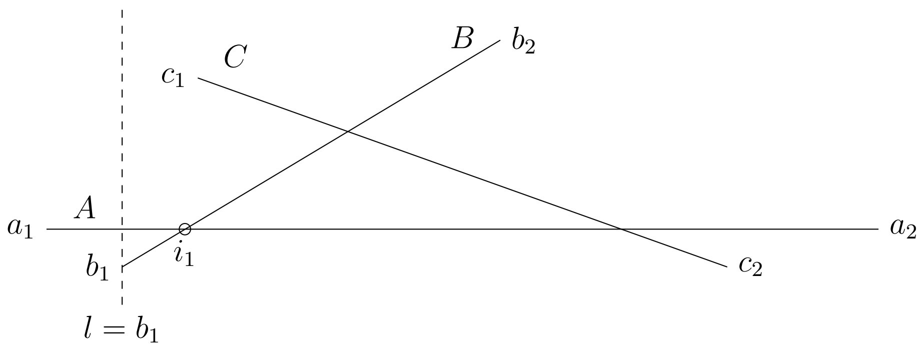
Rysunek 4.1. Przykład działania algorytmu Bentleya-Ottmanna.



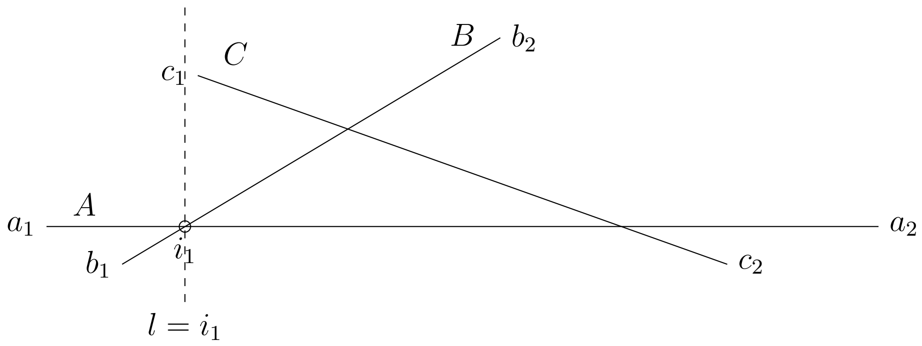
X-struktura: $a_1,\ b_1,\ c_1,\ b_2,\ c_2,\ a_2$
Y-struktura: $\emptyset$



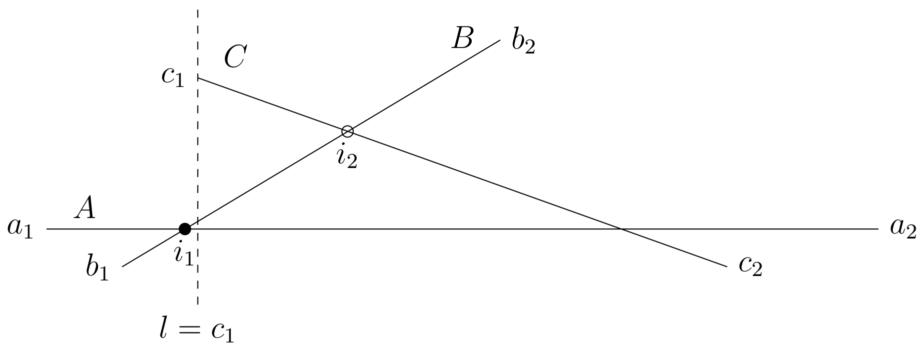
X-struktura: $b_1,\ c_1,\ b_2,\ c_2,\ a_2$
Y-struktura: A



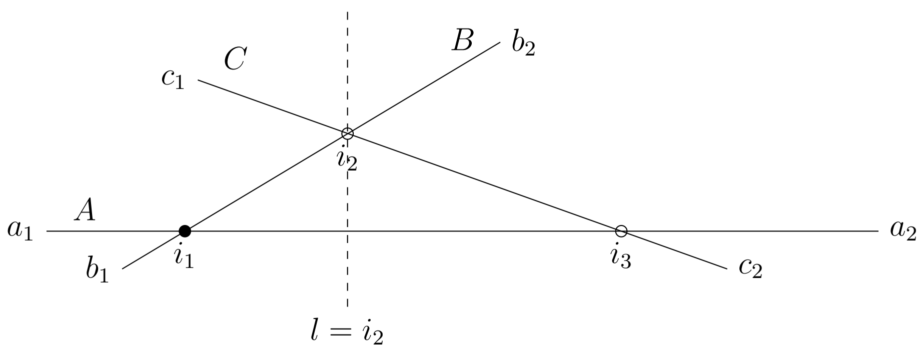
X-struktura: $i_1,\ c_1,\ b_2,\ c_2,\ a_2$
Y-struktura: $A,\ B$



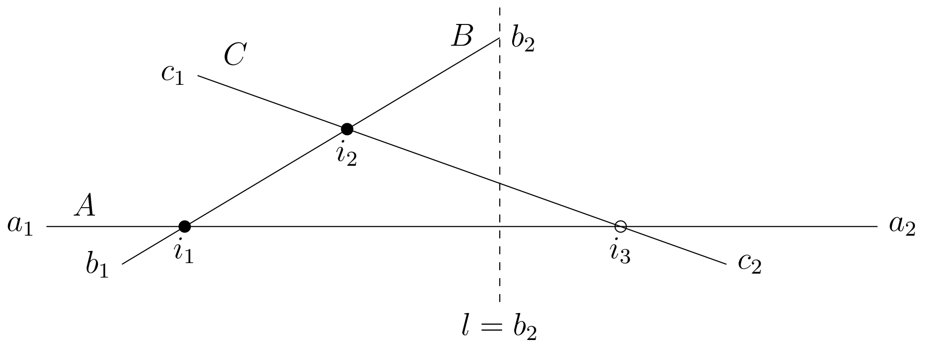
X-struktura: $c_1,\ b_2,\ c_2,\ a_2$
Y-struktura: $B,\ A$



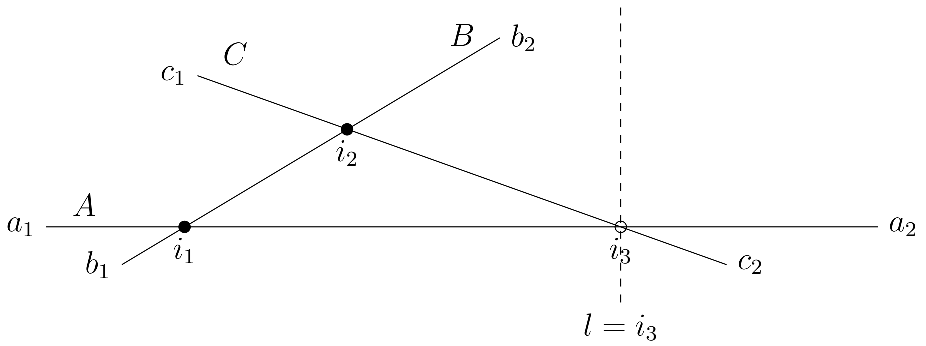
X-struktura: $i_2,\ b_2,\ c_2,\ a_2$
Y-struktura: $C,\ B,\ A$



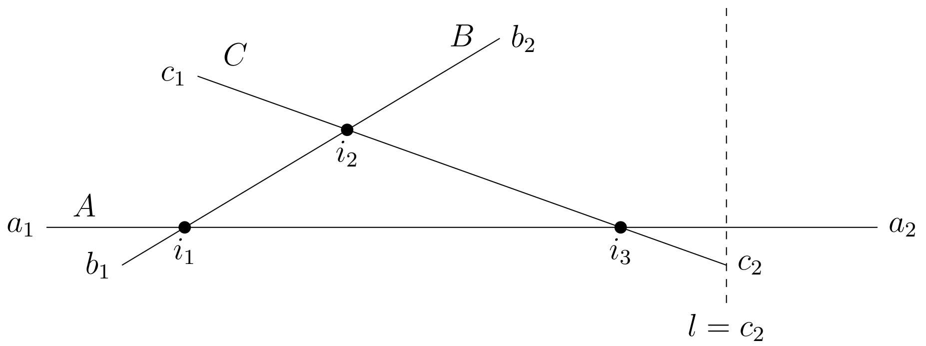
X-struktura: $b_2,\ i_3,\ c_2,\ a_2$
Y-struktura: $B,\ C,\ A$



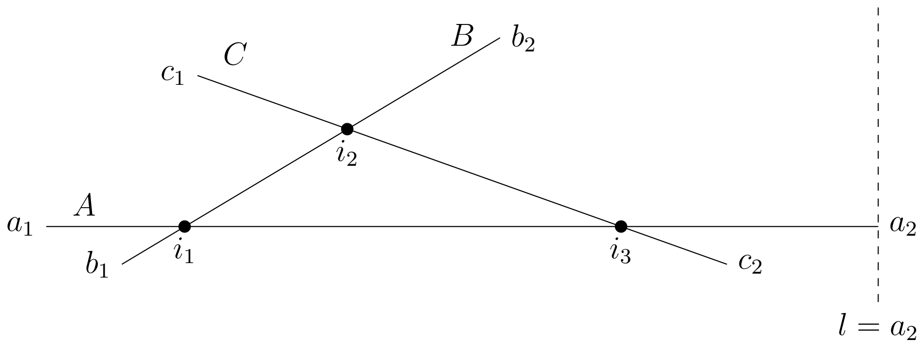
X-struktura: $i_3,\ c_2,\ a_2$
Y-struktura: $C,\ A$



X-struktura: $c_2,\ a_2$
Y-struktura: $A,\ C$



X-struktura: a_2
Y-struktura: A



X-struktura: $\emptyset$
Y-struktura: $\emptyset$

Listing 4.1. Moduł bentleyottmann.

```python
#!/usr/bin/python

from priority_queue import PriorityQueue
from avltree2 import AVLTree
from events import Event


class BentleyOttmann:
    """Bentley-Ottmann algorithm."""

    def __init__(self, segment_set):
        self.eq = PriorityQueue()  # event queue
        self.sl = AVLTree()  # sweep line

        for segment in segment_set:
            if segment.pt1.x > segment.pt2.x:
                segment = ~segment
            self.eq.push(Event(segment.pt1, Event.LEFT, segment))
            self.eq.push(Event(segment.pt2, Event.RIGHT, segment))

        self.il = []  # intersection list

    def run(self):
        while not self.eq.is_empty():
            event = self.eq.pop()
            if event.type == Event.LEFT:
                self._handle_left_endpoint(event)
            elif event.type == Event.RIGHT:
                self._handle_right_endpoint(event)
            elif event.type == Event.CROSSING:
                self._handle_crossing(event)
            else:
                raise ValueError("unknown event")

        del self.eq
        del self.sl
        return self.il

    def _handle_left_endpoint(self, event):
        segment_e = event.segment
        self.sl.current_x = event.pt.x
```

```
        self.sl.insert(segment_e)

        segment_above = self.sl.successor(segment_e)
        segment_below = self.sl.predecessor(segment_e)

        if segment_above is not None:
            segment_above = segment_above.value
            point = segment_e.intersection_point(segment_above)
            if point and point.x > event.pt.x:
                self.eq.push(Event(point,
                                   Event.CROSSING,
                                   segment_above,
                                   segment_e))

        if segment_below is not None:
            segment_below = segment_below.value
            point = segment_e.intersection_point(segment_below)
            if point and point.x > event.pt.x:
                self.eq.push(Event(point,
                                   Event.CROSSING,
                                   segment_e,
                                   segment_below))

    def _handle_right_endpoint(self, event):
        segment_e = event.segment

        self.sl.current_x = event.pt.x
        segment_above = self.sl.successor(segment_e)
        segment_below = self.sl.predecessor(segment_e)

        if segment_above is not None and segment_below is not None:
            segment_above = segment_above.value  # get segment from node
            segment_below = segment_below.value  # get segment from node
            point = segment_above.intersection_point(segment_below)
            if point and point.x > event.pt.x:
                self.eq.push(Event(point,
                                   Event.CROSSING,
                                   segment_above,
                                   segment_below))

        self.sl.delete(segment_e)

    def _handle_crossing(self, event):
        self.il.append(event.pt)

        self.sl.current_x = event.pt.x
        segment_e1 = event.segment_above
        segment_e2 = event.segment_below

        self.swap(segment_e1, segment_e2)

        segment_above_e2 = self.sl.successor(segment_e2)
        segment_below_e1 = self.sl.predecessor(segment_e1)

        if segment_above_e2 is not None:
            segment_above_e2 = segment_above_e2.value
            point = segment_above_e2.intersection_point(segment_e2)
```

```
            if point and point.x > event.pt.x:
                self.eq.push(Event(point,
                                   Event.CROSSING,
                                   segment_above_e2,
                                   segment_e2))

        if segment_below_e1 is not None:
            segment_below_e1 = segment_below_e1.value
            point = segment_below_e1.intersection_point(segment_e1)
            if point and point.x > event.pt.x:
                self.eq.push(Event(point,
                                   Event.CROSSING,
                                   segment_e1,
                                   segment_below_e1))

    def swap(self, segment1, segment2):
        self.sl.delete(segment1)
        self.sl.delete(segment2)

        self.sl.insert(segment1)
        self.sl.insert(segment2)
```

4.2. Algorytm Shamosa-Hoeya

Algorytm Shamosa-Hoeya (1976) jest okrojoną wersją algorytmu Bentleya-Ottmanna. Sprawdza on tylko, czy w pewnym zbiorze odcinków istnieje jakiekolwiek przecięcie dwóch odcinków.

Założenia implementacji są następujące.
- Żadne dwa końce odcinków ani punkty przecięcia nie mogą mieć takiej samej współrzędnej x. Zatem zabronione są odcinki pionowe. Koniec jednego odcinka nie może leżeć powyżej końca innego odcinka.
- Żadne trzy odcinki nie przecinają się w tym samym punkcie.

Dane wejściowe: Zbiór n odcinków.

Dane wyjściowe: Wartość logiczna True, jeżeli wykryto przecięcie, albo False przy braku przecięć.

Problem: Znajdowanie przecięcia w zbiorze odcinków.

Opis algorytmu: Algorytm Shamosa-Hoeya również wykorzystuje X-strukturę i Y-strukturę. Inicjalizacja struktur przebiega identycznie jak w algorytmie Bentleya-Ottmanna. Przetwarzanie zdarzeń z X-struktury różni się tym, że wykrycie pierwszego przecięcia odcinków kończy działanie algorytmu. W związku z tym w X-strukturze nie pojawi się zdarzenie odpowiadające przecięciu odcinków.

Algorytm kończy się zwróceniem wartości True przy pierwszym napotkanym przecięciu lub False jeśli przecięć nie było.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n \log n)$, złożoność pamięciowa wynosi $O(n)$. Zostało to potwierdzone w dodatku A. X-struktura po zainicjalizowaniu zawiera tylko $2n$ końców odcinków, a potem się zmniejsza. Ponadto nie przechowuje się listy punktów przecięcia.

Uwagi: Y-struktura musi zapewnić prawidłowe wstawianie odcinków w różnych punktach osi x. X-struktura nie pozwala na przechowywanie dwóch jednakowych zdarzeń w danym momencie, co jest rozszerzeniem typowej kolejki priorytetowej.

Listing 4.2. Moduł shamoshoey.

```
#!/usr/bin/python

from avltree2 import AVLTree
from priority_queue import PriorityQueue
from events import Event


class ShamosHoey:
    """Shamos-Hoey algorithm."""

    def __init__(self, segment_set):
        self.eq = PriorityQueue()    # event queue
        self.sl = AVLTree()          # sweep line

        for segment in segment_set:
            if segment.pt1.x > segment.pt2.x:
                segment = ~segment
            self.eq.push(Event(segment.pt1, Event.LEFT, segment))
            self.eq.push(Event(segment.pt2, Event.RIGHT, segment))

    def run(self):
        while not self.eq.is_empty():
            event = self.eq.pop()
            if event.type == Event.LEFT:
                if self._handle_left_endpoint(event):
                    return True
            elif event.type == Event.RIGHT:
                if self._handle_right_endpoint(event):
                    return True
            else:
                raise ValueError("unknown event")

        del self.eq
        del self.sl
        return False

    def _handle_left_endpoint(self, event):
        segment_e = event.segment

        self.sl.current_x = event.pt.x
        self.sl.insert(segment_e)

        segment_above = self.sl.successor(segment_e)
        segment_below = self.sl.predecessor(segment_e)

        if segment_above is not None:
            segment_above = segment_above.value      # get segment from node
```

```
            if segment_e.intersect(segment_above):
                return True

        if segment_below is not None:
            segment_below = segment_below.value      # get segment from node
            if segment_e.intersect(segment_below):
                return True

        return False

    def _handle_right_endpoint(self, event):
        segment_e = event.segment

        segment_above = self.sl.successor(segment_e)
        segment_below = self.sl.predecessor(segment_e)

        if segment_above is not None and segment_below is not None:
            segment_above = segment_above.value     # get segment from node
            segment_below = segment_below.value     # get segment from node
            if segment_above.intersect(segment_below):
                return True

        self.sl.delete(segment_e)
        return False
```

4.3. Przecinanie się odcinków pionowych i poziomych

Dla zbioru odcinków poziomych i pionowych nie można wykorzystać algorytmu Bentleya-Ottmanna, ponieważ nie spełnia on wymagań pozycji ogólnej. W takim przypadku potrzebny jest kolejny algorytm, który również polega na przesuwaniu miotły po płaszczyźnie [22].

Dane wejściowe: Zbiór n poziomych lub pionowych odcinków.

Dane wyjściowe: Zbiór k punktów przecięcia odcinków.

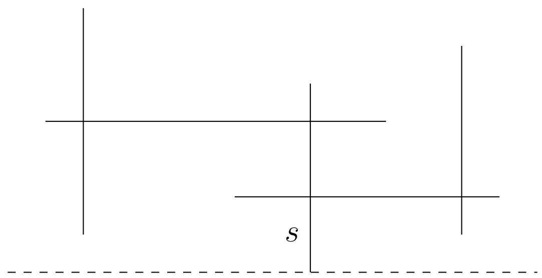
Problem: Znajdowanie punktów przecięcia zbioru poziomych lub pionowych odcinków.

Opis algorytmu: Algorytm wykorzystuje typowe dwie struktury danych: X-strukturę i Y-strukturę. Tym razem miotłę przesuwamy od dołu do góry, zatem X-struktura jest zbiorem odcinków pionowych, które w danym położeniu miotły przecinają ją. Odcinki są uporządkowane względem pierwszej współrzędnej x. Wykorzystamy tutaj zrównoważone binarne drzewo przeszukiwań pozwalające na znalezienie k odcinków przecinających w czasie $O(k \log n)$.

Y-struktura jest kolejką zdarzeń, przez które przechodzi prosta zamiatająca. Zdarzeniem może być albo odcinek poziomy, albo początek lub koniec pionowego odcinka. Tym razem struktura ta jest tylko inicjalizowana i sortowana na początku, bez ingerencji w trakcie działania algorytmu. Z tego powodu wykorzystano tutaj standardową listę, na której operacja sortowania działa w czasie $O(n \log n)$.

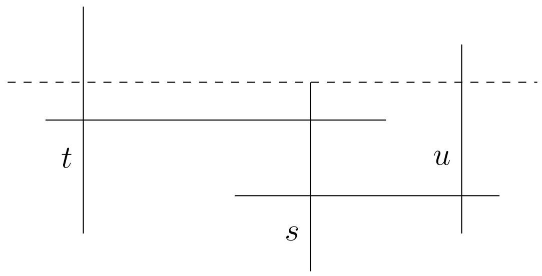
Algorytm pobiera kolejno zdarzenia z kolejki zdarzeń i obsługuje je w zależności od typu:

— Początek odcinka pionowego s - dodajemy odcinek s do X-struktury.



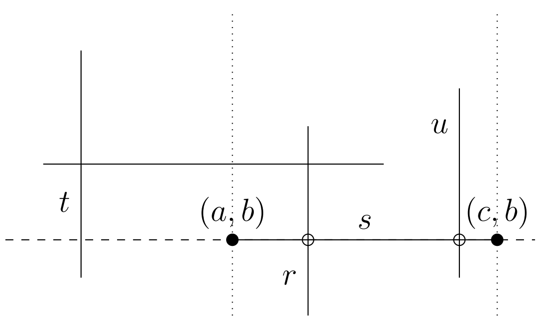
X-struktura: s

— Koniec odcinka pionowego s - usuwamy odcinek s z X-struktury.



X-struktura: $t,\ u$

— Odcinek poziomy s.
Niech odcinek s będzie odcinkiem o początku w punkcie (a, b) i końcu w punkcie (c, b), gdzie $c > a$. W takim wypadku należy znaleźć wszystkie odcinki pionowe, których współrzędna x mieści się w przedziale $[a, c]$. Aby to osiągnąć, dodajemy odcinek poziomy do naszej X-struktury, a następnie szukamy wszystkie odcinki, które spełniają powyższą zależność. Dla każdego znalezionego odcinka wyznaczamy punkt przecięcia z odcinkiem s i dodajemy do listy wynikowej przecięć. Na końcu usuwamy odcinek s z X-struktury.



X-struktura: $t,\ r,\ u$

Złożoność: Złożoność czasowa algorytmu wynosi $O((n + k) \log n)$, złożoność pamięciowa wynosi $O(n)$. Testy w dodatku A potwierdziły złożoność z literatury.

Uwagi: Według opisu Michiela Smida [22] do szukania k odcinków pionowych przecinających jeden odcinek poziomy najlepsze rezultaty osiągniemy wykorzystując "orthogonal range query". Jest to dodatkowa struktura danych pozwalająca na szybkie szukanie punktów na płaszczyźnie w pewnym przedziale (przecięcia pomiędzy końcami poziomego odcinka).

Listing 4.3. Moduł horizontalvertical.

```
#!/usr/bin/python

from events import Event
from avltree import AVLTree


class HorizontalVertical:
    """Intersections of horizontal and vertical line segments."""

    def __init__(self, segment_set):
        """Initialize structures."""
        self.eq = []   # event queue
        self.sl = AVLTree()   # sweep line

        for segment in segment_set:
            if segment.pt1.x > segment.pt2.x:
                segment = ~segment
            if segment.pt1.x == segment.pt2.x:   # vertical segment
                self.eq.append(Event(segment.pt1,
                                     Event.LEFT, segment))
                self.eq.append(Event(segment.pt2,
                                     Event.RIGHT, segment))
            elif segment.pt1.y == segment.pt2.y:   # horizontal segment
                self.eq.append(Event(segment.pt1,
                                     Event.HORIZONTAL, segment))
            else:
                raise ValueError("horizontal or "
                                 "vertical segments are allowed")

        self.eq.sort(key=lambda event: event.pt.y)
        self.il = []   # intersection list

    def run(self):
        """Processing events."""
        for event in self.eq:
            if event.type == Event.LEFT:
                self._handle_bottom_endpoint(event)
            elif event.type == Event.RIGHT:
                self._handle_top_endpoint(event)
            elif event.type == Event.HORIZONTAL:
                self._handle_crossing(event)
            else:
                raise ValueError("unknown event")
        del self.eq
```

```python
        del self.sl
        return self.il

    def _handle_bottom_endpoint(self, event):
        self.sl.insert(event.segment)

    def _handle_top_endpoint(self, event):
        self.sl.delete(event.segment)

    def _handle_crossing(self, event):
        segment_e = event.segment
        x_min = segment_e.pt1.x
        x_max = segment_e.pt2.x
        self.sl.insert(segment_e)

        node = self.sl.successor(segment_e)
        while node and node.value.pt1.x < x_max:
            point = segment_e.intersection_point(node.value)
            self.il.append(point)
            node = self.sl.successor(node.value)
        self.sl.delete(segment_e)

# EOF
```

4.4. Para najbliższych punktów - algorytm siłowy

Algorytm szukania pary najbliżej położonych punktów metodą siłową daje nam pewność poprawnego rozwiązania przy każdym zbiorze wejściowym.

Dane wejściowe: Zbiór n punktów, $n > 1$.

Dane wyjściowe: Para najbliżej położonych punktów.

Problem: Znajdowanie pary najbliższych punktów.

Opis algorytmu: Klasyczne użycie pętli zagnieżdżonych. Każdy obieg wylicza długość wektora między dwoma punktami ze zbioru i porównuje z aktualnie najmniejszą odległością.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n^2)$, złożoność pamięciowa wynosi $O(1)$.

Uwagi: Odległość pomiędzy parą punktów w przestrzeni dwuwymiarowej rozumiemy jako odległość euklidesową

$$d(p,q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}. \tag{4.1}$$

Zauważmy, że w implementacji nie korzystamy z kosztownego obliczania pierwiastka, tylko porównujemy kwadraty odległości między punktami.

Listing 4.4. Funkcja find_two_closest_points().

```
def find_two_closest_points(point_list):
    """Find two closest points in O(n^2) time (brute force)."""
    dist = float("inf")
    pair = None
    n = len(point_list)
    for i in xrange(n):
        for j in xrange(i+1, n):
            vec = point_list[j] - point_list[i]
            new_dist = vec * vec
            if new_dist < dist:
                dist = new_dist
                pair = point_list[i], point_list[j]
    return pair
```

4.5. Para najbliższych punktów - technika zamiatania

Tym razem do rozwiązania tego problemu wykorzystamy technikę zamiatania, która znacząco redukuje złożoność czasową szukania pary najbliższych punktów. Algorytm polega na przesuwaniu miotły po płaszczyźnie i porównywaniu odległości tylko niektórych punktów. Tym razem miotłę będziemy przesuwać od dołu do góry, aby pokazać elastyczność techniki zamiatania. W naszej implementajcji wygodniej jest przesuwać miotłę z dołu do góry, ponieważ wtedy punkty należące do miotły są porównywane najpierw względem współrzędnej x, co jest naturalnym sposobem porównywania punktów.

Dane wejściowe: Zbiór n punktów, $n > 1$.

Dane wyjściowe: Para najbliżej położonych punktów.

Problem: Znajdowanie pary najbliższych punktów.

Opis algorytmu: Algorytm tradycyjnie wykorzystuje dwie struktury danych: X-strukturę i Y-strukturę. Y-struktura jest tablicą przechowującą punkty posortowane względem współrzędnej y. X-struktura jest zbalansowanym drzewem poszukiwań binarnych zawierającym punkty aktywne. Zauważmy, że wybór X-struktury i Y-struktury zależy od wyboru kierunku przesuwania się miotły (z lewej na prawo, czy z dołu do góry).

Niech p będzie punktem ze zbioru początkowego S, a d aktualną najmniejszą odległością pomiędzy punktami znajdującymi się poniżej p. W momencie przecięcia prostej zamiatającej punkt p, punktami aktywnymi są punkty znajdujące się w pasie poziomym o wysokości równej d, na którego górnej krawędzi leży punkt p. Następnym krokiem jest znalezienie sześciu sąsiadów punktu p w zbiorze punktów aktywnych, trzech z lewej strony i trzech z prawej. Na końcu obliczamy odległości od p do każdego z tych sześciu sąsiadów w celu znalezienia nowej odległości d i przesuwamy miotłę do góry.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n \log n)$, złożoność pamięciowa wynosi $O(n)$. Wykazano to podczas testów, które można znaleźć w dodatku A.

Uwaga 1: Po każdym obiegu pętli mamy obliczoną najmniejszą odległość pomiędzy wszystkimi punktami poniżej miotły.

Uwaga 2: Punkty aktywne należą do poziomego pasa o grubości d, który przesuwa się z dołu do góry razem z miotłą, przy czym grubość tego pasa maleje wraz ze zmniejszaniem się wartości d.

Uwaga 3: W każdym obiegu pętli sprawdzamy co najwyżej 3 punkty aktywne na lewo od p i co najwyżej 3 punkty aktywne na prawo od p. Wynika to z następującego stwierdzenia [18]:

Stwierdzenie: Jeżeli S jest zbiorem n punktów, a d odległością pomiędzy najmniej odległą parą punktów z S, to każdy kwadrat o boku d zawiera co najwyżej cztery punkty z S.

Listing 4.5. Klasa ClosestPair z modułu closestpair2.

```
#!/usr/bin/python

from avltree import AVLTree

class ClosestPair:
    """Solving the closest pair problem."""

    def __init__(self, point_list):
        """Initialize structures."""
        if len(point_list) < 2:
            raise ValueError("minimum 2 points")

        self.points = point_list
        self.points.sort(key=lambda point: point.y)
        self.active_points = AVLTree()
        self.closest_pair = self.points[0], self.points[1]
        self.min_distance = (self.points[0] - self.points[1]).length()

    def run(self):
        """Finding the closest pair."""
        if len(self.points) == 2:
            return self.closest_pair
        top = 2
        new_point = self.points[top]
        bottom = 0
        while self.points[bottom].x <= new_point.x - self.min_distance:
            bottom += 1
        for i in range(bottom, top):
            self.active_points.insert(self.points[i])

        while top < len(self.points):
            self.active_points.insert(new_point)
            neighbors = []
            # three points on right side
            point = new_point
            for _ in range(3):
                node = self.active_points.successor(point)
                if node is not None:
```

```
                point = node.value
                neighbors.append(point)
            else:
                break
        # three points on left side
        point = new_point
        for _ in range(3):
            node = self.active_points.predecessor(point)
            if node is not None:
                point = node.value
                neighbors.append(node.value)
            else:
                break

        for point in neighbors:
            new_distance = (new_point - point).length()
            if new_distance < self.min_distance:
                self.min_distance = new_distance
                self.closest_pair = point, new_point

        if top < len(self.points) - 1:
            new_point = self.points[top + 1]
            while self.points[bottom].x <= new_point.x - self.min_distance:
                self.active_points.delete(self.points[bottom])
                bottom += 1

        top += 1
    return self.closest_pair
```

4.6. Para najbliższych punktów - technika dziel i zwyciężaj

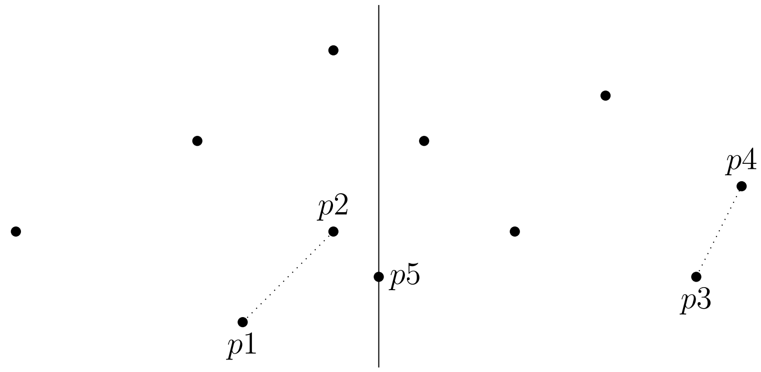
Kolejna metoda rozwiązywania problemu szukania pary najbliższych punktów wykorzystuje technikę dziel i zwyciężaj. Algorytm polega na rekurencyjnym podziale zbioru posortowanych punktów na dwie równe części i szukaniu w nich pary najbliższych punktów. Po każdym podziale należy zbadać niektóre punkty z obu części, które mogą potencjalnie tworzyć bliższą parę od znalezionej wcześniej [19].

Dane wejściowe: Zbiór n punktów, $n > 1$.

Dane wyjściowe: Para najbliżej położonych punktów.

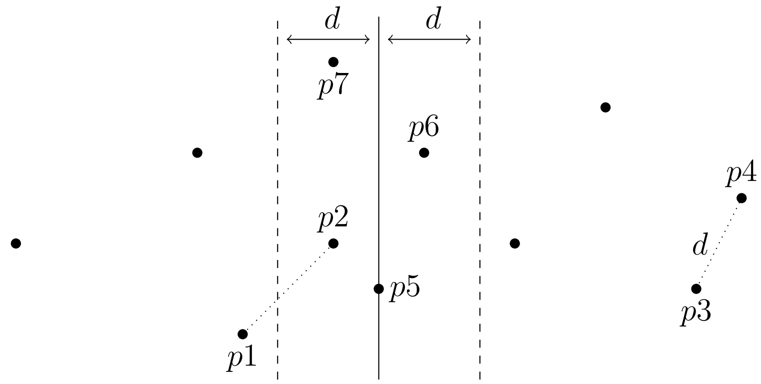
Problem: Znajdowanie pary najbliższych punktów.

Opis algorytmu: Algorytm rozpoczynamy od posortowania punktów wejściowych względem współrzędnej x. Następnym krokiem jest podział punktów na dwa równe zbiory, w których następuje rekurencyjne szukanie pary najbliższych punktów. Oznaczmy te pary jako pair_left i pair_right, a punkt środkowy mid_point.



pair_left = <p1, p2>
pair_right = <p3, p4>
mid_point = p5

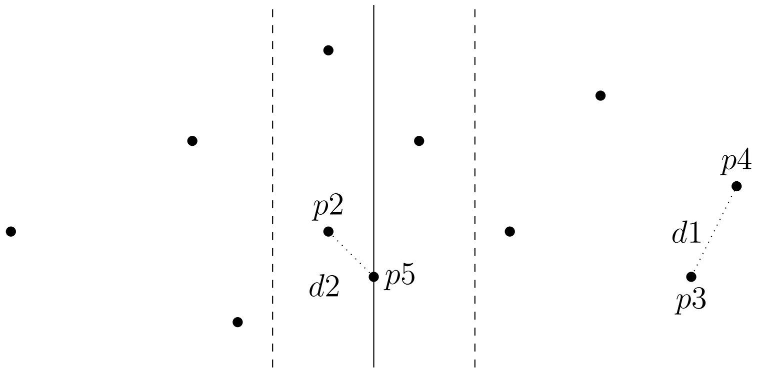
Kolejnym kluczowym krokiem jest wybranie punktów, które leżą w niewielkiej odległości od punktu mid_point. Są to punkty leżące po obu stronach, a które potencjalnie mogą tworzyć parę bliższą od par pair_left i pair_right. Aby wyznaczyć takie punkty, należy nałożyć ograniczenie na oś x wokół punktu mid_point dzielącego zbiór na dwa podzbiory. Takim ograniczeniem jest mniejsza odległość między punktami z par pair_left i pair_right, oznaczmy ją jako d. Dodajemy więc tylko te punkty, których współrzędna x jest w odległości co najwyżej d od współrzędnej x punktu mid_point. Sortujemy je względem współrzędnej y.



pair_left = <p1, p2>
pair_right = <p3, p4>
mid_point = p5
strip = {p5, p2, p6, p7}

Następnie musimy znaleźć parę najbliższych punktów, które należą do wyznaczonego pasa (ang. *strip*). Standardowo, taka operacja wydaje się być czasu $O(n^2)$, ale w tym przypadku złożoność redukuje się do $O(n)$, co można geometrycznie udowodnić. Każdy kolejny punkt z tego zbioru wystarczy porównać maksymalnie z siedmioma następnymi punktami.

Na końcu zwracamy najbliższą parę, porównując wyszukaną wcześniej w obu podziałach i znalezioną przed momentem na pasie.



$d2 < d1$

A więc najbliższą parą jest para <p2, p5>

Złożoność: Złożoność czasową algorytmu $T(n)$ można zapisać rekurencyjnie jako $T(n) = 2T(n/2) + O(n) + O(n \log n) + O(n)$, gdzie tworzenie pasa zajmuje czas $O(n)$, sortowanie punktów w pasie $O(n \log n)$, znalezienie najbliższych punktów w pasie $O(n)$. Złożoność czasowa algorytmu wynosi łącznie $O(n(\log n)^2)$. Złożoność pamięciowa wynosi $O(n)$, ponieważ wykorzystujemy dodatkową pamięć na punkty należące do pasa. Testy potwierdzające złożoność dostępne są w dodatku A.

Uwaga 1: Zbiór dzielimy do momentu, gdy opłacalne będzie użycie metody siłowej, w naszym przypadku jest to zbiór liczący trzy punkty.

Listing 4.6. Klasa ClosestPair z modułu closestpair3.

```
#!/usr/bin/python

from tools1 import find_two_closest_points

class ClosestPair:
    """Solving the closest pair problem using divide and conquer."""

    def __init__(self, point_list):
        if len(point_list) < 2:
            raise ValueError("minimum 2 points")
        self.points = point_list
        self.points.sort()
        self.closest_pair = None
        self.min_distance = None

    def run(self):
        self.closest_pair = self._closest(0, len(self.points)-1)
        self.min_distance = self.pair_length(self.closest_pair)
        return self.closest_pair

    def _closest(self, left, right):
        if (right - left) <= 2:
            return find_two_closest_points(self.points[left:right+1])

        middle = (left + right) // 2
        closest_left = self._closest(left, middle)
        closest_right = self._closest(middle + 1, right)
```

```
        left_d = self.pair_length(closest_left)
        right_d = self.pair_length(closest_right)

        if left_d < right_d:
            closest_pair = closest_left
            current_d = left_d
        else:
            closest_pair = closest_right
            current_d = right_d
        strip = []
        for i in xrange(left, right+1):    # O(n) time
            if abs(self.points[middle].x - self.points[i].x) < current_d:
                strip.append(self.points[i])

        if len(strip) < 2:
            closest_strip = closest_pair
        else:
            closest_strip = self._closest_on_strip(strip, current_d)

        if current_d < self.pair_length(closest_strip):
            return closest_pair
        else:
            return closest_strip

    def _closest_on_strip(self, strip, distance):    # O(n log n) time
        min_distance = distance
        strip.sort(key=lambda point: point.y)
        closest_pair = strip[0], strip[1]
        for i in range(len(strip)):
            for j in range(i + 1, len(strip)):
                if strip[j].y - strip[i].y > min_distance:
                    break
                new_distance = (strip[i] - strip[j]).length()
                if new_distance < min_distance:
                    min_distance = new_distance
                    closest_pair = strip[i], strip[j]
        return closest_pair

    def pair_length(self, pair):
        return (pair[0] - pair[1]).length()
```

Uwaga 2: W literaturze można znaleźć informację, że da się wyznaczyć parę najbliższych punktów w pasie w czasie $O(n)$ [6]. Wtedy łączny czas pracy algorytmu wyniesie $O(n \log n)$. Istotne jest jednorazowe posortowanie wszystkich punktów względem współrzędnej y. Następnie podczas kolejnych podziałów zbioru punktów względem x należy zachowywać uporządkowanie względem y. Ten krok jest jakby odwrotnością procesu scalania, który występuje w sortowaniu przez scalanie. Wymaganą wydajność uzyskujemy w naszej implementacji przez przechowywanie punktów w trzech kontenerach: w zbiorze P (szybkie wyszukiwanie), na liście X posortowanej względem współrzędnej x, oraz na liście Y posortowanej względem współrzędnej y. Aby to potwierdzić, wykonano testy dostępne w dodatku A. W kolejnych wywołaniach rekurencyjnych tworzone są nowe mniejsze zbiory i listy, ale zajęta pamięć jest rzędu $O(n)$.

Listing 4.7. Klasa ClosestPair z modułu closestpair4.

```
#!/usr/bin/python

from tools1 import find_two_closest_points

class ClosestPair:
    """Solving the closest pair problem using divide and conquer."""

    def __init__(self, point_list):
        if len(point_list) < 2:
            raise ValueError("minimum 2 points")
        self.P = set(point_list)
        self.closest_pair = None
        self.min_distance = None

    def run(self):
        X = list(self.P)
        X.sort(key=lambda point: point.x)
        Y = list(self.P)
        Y.sort(key=lambda point: point.y)
        self.closest_pair = self._closest(self.P, X, Y)
        self.min_distance = self.pair_length(self.closest_pair)
        return self.closest_pair

    def _closest(self, P, X, Y):
        if len(P) <= 3:
            return find_two_closest_points(X)

        middle = len(P) // 2
        middle_point = X[middle]
        X_left = X[:middle]
        X_right = X[middle:]
        P_left = set(X_left)
        P_right = set(X_right)

        Y_left = []
        Y_right = []
        for point in Y:    # O(n) time
            if point in P_left:
                Y_left.append(point)
            else:
                Y_right.append(point)
        closest_left = self._closest(P_left, X_left, Y_left)
        closest_right = self._closest(P_right, X_right, Y_right)
        left_d = self.pair_length(closest_left)
        right_d = self.pair_length(closest_right)

        if left_d < right_d:
            closest_pair = closest_left
            current_d = left_d
        else:
            closest_pair = closest_right
            current_d = right_d

        strip = []
        for point in Y:    # O(n) time
            if abs(middle_point.x - point.x) < current_d:
```

```
                strip.append(point)

        if len(strip) < 2:
            closest_strip = closest_pair
        else:
            closest_strip = self._closest_on_strip(strip, current_d)

        if current_d < self.pair_length(closest_strip):
            return closest_pair
        else:
            return closest_strip

    def _closest_on_strip(self, strip, distance):    # O(n) time
        min_distance = distance
        closest_pair = strip[0], strip[1]
        for i in range(len(strip)):
            for j in range(i + 1, len(strip)):
                if strip[j].y - strip[i].y > min_distance:
                    break
                new_distance = (strip[i] - strip[j]).length()
                if new_distance < min_distance:
                    min_distance = new_distance
                    closest_pair = strip[i], strip[j]
        return closest_pair

    def pair_length(self, pair):
        return (pair[0] - pair[1]).length()
```

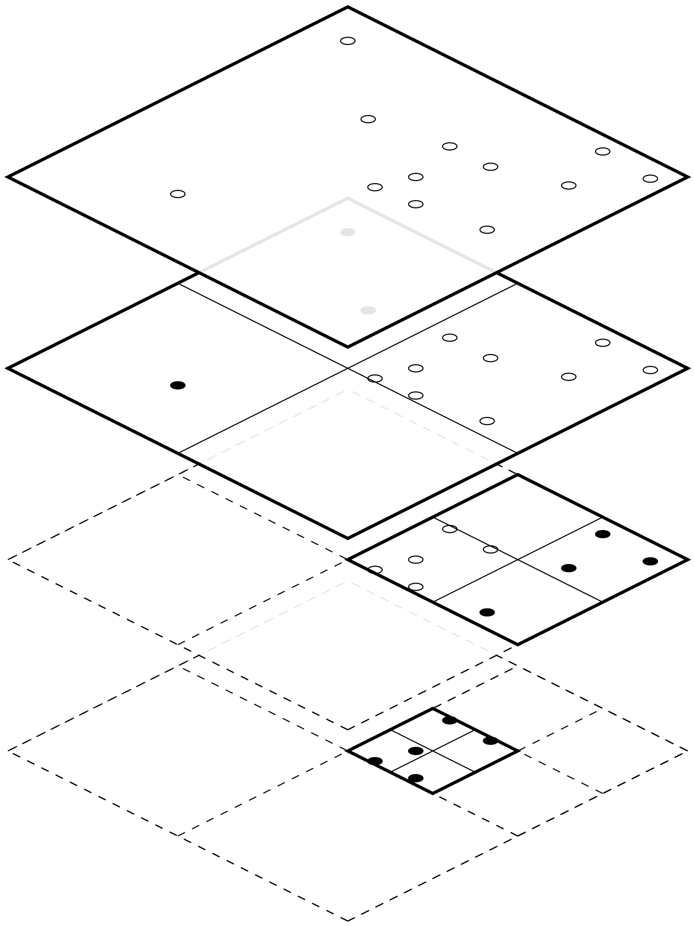
4.7. Drzewo czwórkowe

Definicja: *Drzewo czwórkowe* (ang. *quadtree*) jest to drzewowa struktura danych, w której każdy węzeł wewnętrzny ma dokładnie czworo dzieci. Struktura ta jest używana zwykle do podziału prostokątnego obszaru płaszczyzny na cztery równe ćwiartki, a każda ćwiartka z kolei może być dzielona na cztery kolejne ćwiartki itd. [20].

Drzewo czwórkowe jest stosowane w procesie wykrywania kolizji obiektów w dwóch wymiarach, w kompresji bitmap czarno-białych, do przechowywania zbioru punktów płaszczyzny (szybkie wyszukiwanie), a także do wielu innych zadań.

Drzewa czwórkowe można podzielić ze względu na typ danych, jakie przechowuje. Wyróżnia się:
— Region quadtree
— Point quadtree
— Point-region quadtree
— Edge quadtree
— Polygonal map quadtree
— Compressed quadtree
W naszej pracy skupimy się na trzecim typie, który łączy cechy dwóch pierwszych.

Struktura ta przechowuje punkty leżące na wyznaczonym prostokątnym obszarze powierzchni dwuwymiarowej. Każdy węzeł takiego drzewa może przechowywać maksymalnie k punktów. Próba dodania kolejnego punktu skutkuje podziałem obszaru na cztery równe części i przenosi wszystkie punkty z węzła, na którym następuje podział do jego potomków. Po tej operacji nowy punkt może zostać dodany do jednego z nowo powstałych węzłów. Dzięki temu nasze drzewo przechowuje wartości tylko w swoich liściach.



Rysunek 4.2. Struktura drzewa czwórkowego dla $k = 4$.

Szukanie punktów należących do danej powierzchni: Mając poprawnie zaimplementowane drzewo czwórkowe, możemy w łatwy sposób dostać listę punktów, które zawierają się w pewnym ograniczonym obszarze. Metoda ta polega na rekurencyjnym przeszukiwaniu tylko tych węzłów, których powierzchnia przecina się z naszym ograniczonym obszarem. Odrzucamy zatem wszystkie niższe poziomy węzła, który nie posiada takiego przecięcia.

Szukanie najbliższego sąsiada danego punktu: Tym razem chcemy znaleźć najbliższego sąsiada pewnego punktu wejściowego. Również tutaj należy rekurencyjnie przeszukiwać tylko odpowiednie węzły. Kluczowym krokiem tej metody jest poprawne ustalenie kolejności przeszukiwania potomków. Aby to osiągnąć, należy wyznaczyć środek powierzchni aktualnego węzła i na jego podstawie zadecydować o porządku przeszukiwania [21].

Uwaga 1: Punkt może zostać dodany do drzewa czwórkowego tylko wtedy, gdy jego współrzędne zawierają się w całkowitej powierzchni drzewa, która jest podana podczas inicjalizacji struktury. W przeciwnym wypadku taki punkt zostaje odrzucony.

Uwaga 2: Maksymalna ilość punktów w każdym węźle (k) ustalana jest podczas inicjalizacji struktury. Jej wartość jest stała w całym drzewie. Domyślnie są to cztery punkty ($k = 4$).

Złożoność: Złożoność czasowa wstawiania nowego elementu do drzewa wynosi $O(\log n)$, złożoność pamięciowa wynosi $O(n)$. Wykazano to w dodatku A.

Listing 4.8. Klasa QuadTree z modułu quadtree.

```python
#!/usr/bin/python

class QuadTree:
    """The class defining a quadtree."""

    def __init__(self, rect, capacity=4):
        self.rect = rect
        self.capacity = capacity
        self.point_list = []
        self.top_left = None
        self.top_right = None
        self.bottom_left = None
        self.bottom_right = None

    def __str__(self):
        return "QuadTree({}, {})".format(self.rect, self.capacity)

    def is_divided(self):
        """Test if the quadtree is divided."""
        return self.top_left is not None

    def height(self):
        """Return the height of the quadtree."""
        if self.is_divided():
            tl = self.top_left.height()
            tr = self.top_right.height()
            bl = self.bottom_left.height()
            br = self.bottom_right.height()
            return 1 + max(tl, tr, bl, br)
        else:
            return 1

    def insert(self, point):
        """Insert a point into the quadtree."""
        if point not in self.rect:
            return False
        if len(self.point_list) < self.capacity and not self.is_divided():
            self.point_list.append(point)
            return True
        if not self.is_divided():
            self.subdivide()
```

```
            # move values to leaf nodes
            while self.point_list:
                current_point = self.point_list.pop()
                self.insert(current_point)
        if self.top_left.insert(point):
            return True
        if self.top_right.insert(point):
            return True
        if self.bottom_left.insert(point):
            return True
        if self.bottom_right.insert(point):
            return True

    def subdivide(self):
        """Subdividing the currect rect."""
        tl, tr, bl, br = self.rect.make4()
        self.top_left = QuadTree(tl, self.capacity)
        self.top_right = QuadTree(tr, self.capacity)
        self.bottom_left = QuadTree(bl, self.capacity)
        self.bottom_right = QuadTree(br, self.capacity)

    def query(self, query_rect):
        """Find all points that appear within a range."""
        points_in_rect = []
        try:
            self.rect.intersection(query_rect)
        except ValueError:
            return []
        for pt in self.point_list:
            if pt in query_rect:
                points_in_rect.append(pt)
        if not self.is_divided():
            return points_in_rect
        children = (self.top_left, self.top_right,
                    self.bottom_left, self.bottom_right)
        for child in children:
            points_in_rect.extend(child.query(query_rect))
        return points_in_rect

    def nearest(self, point, best=None):
        """Find a nearest point."""
        if best is None:
            if len(self.point_list) > 0:
                best = self.point_list[0]
                distance = (point - best).length()
            else:
                distance = float('Inf')
        else:
            distance = (point - best).length()

        if (point.x < self.rect.pt1.x - distance or
            point.x > self.rect.pt2.x + distance or
            point.y < self.rect.pt1.y - distance or
            point.y > self.rect.pt2.y + distance):
                return best

        for pt in self.point_list:
```

```python
        new_distance = (point-pt).length()
        if new_distance < distance:
            best = pt
            distance = new_distance
    if not self.is_divided():
        return best
    # find best children order
    c = self.rect.center()
    if point.x > c.x:   # right, left
        if point.y > c.y:   # top, bottom
            children = (self.top_right, self.top_left,
                        self.bottom_right, self.bottom_left)
        else:   # bottom, top
            children = (self.bottom_right, self.bottom_left,
                        self.top_right, self.top_left)
    else:   # left, right
        if point.y > c.y:   # top, bottom
            children = (self.top_left, self.top_right,
                        self.bottom_left, self.bottom_right)
        else:   # bottom, top
            children = (self.bottom_left, self.bottom_right,
                        self.top_left, self.top_right)

    for child in children:
        best = child.nearest(point, best)
    return best
```

5. Podsumowanie

Jednym z podstawowych zagadnień w geometrii obliczeniowej jest problem przecinania się odcinków z danego zbioru. Prosty algorytm siłowy działający w czasie $O(n^2)$ był już prezentowany, dlatego rozważyliśmy zaawansowane algorytmy wykorzystujące technikę zamiatania płaszczyzny.

Algorytm Bentleya-Ottmana w czasie $O((n+k)\log n)$ znajduje wszystkie k punktów przecięcia w zbiorze n odcinków. Algorytm Shamosa-Hoeya w czasie $O(n \log n)$ sprawdza, czy w zbiorze odcinków jakaś para odcinków przecina się. Oba algorytmy korzystają ze struktury danych drzewa AVL z kluczami dynamicznymi, która jest nietrywialnym rozszerzeniem zwykłego drzewa AVL. W obu algorytmach przyjęto założenie, że odcinki są w pozycji ogólnej, czyli końce odcinków i ich punkty przecięcia nie mają tych samych współrzędnych. W związku z tym rozważyliśmy dodatkowy problem przecinania się odcinków pionowych i poziomych. Zaimplementowane rozwiązanie działa w czasie $O((n+k)\log n)$.

W ramach pracy rozważono problem znajdowania pary najbliższych punktów. Przedstawiono kilka metod rozwiązywania tego problemu: metodę siłową z czasem $O(n^2)$, algorytm z techniką zamiatania i czasem $O(n \log n)$, dwa algorytmy typu dziel i zwyciężaj z czasami $O(n(\log n)^2)$ i $O(n \log n)$.

Z problemem znajdowania pary najbliższych punktów wiąże się problem znajdowania punktu najbliższego do punktu danego z zewnątrz. W rozwiązaniu tego drugiego problemu i wielu innych pomaga struktura danych o nazwie drzewo czwórkowe (*quadtree*). W pracy przedstawiono implementację i sprawdzono wydajność tej struktury danych.

Każdy algorytm został sprawdzony pod względem poprawności (moduł unittest), a także wydajności obliczeniowej (moduł timeit).

A. Testy algorytmów

Dodatek ten zawiera testy wydajnościowe wszystkich algorytmów, aby potwierdzić ich złożoność znaną z literatury. Wyniki działania niektórych z nich przedstawiono również w formie graficznej.

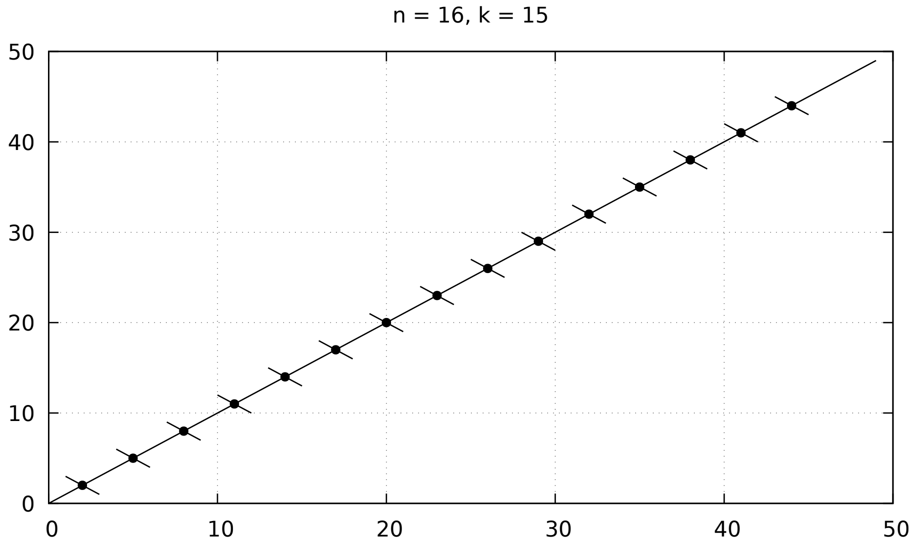
A.1. Testy przecinania się odcinków

Aby potwierdzić złożoność zaimplementowanych algorytmów do wykrywania przecięć odcinków na płaszczyźnie, zastosowano następujące zbiory:

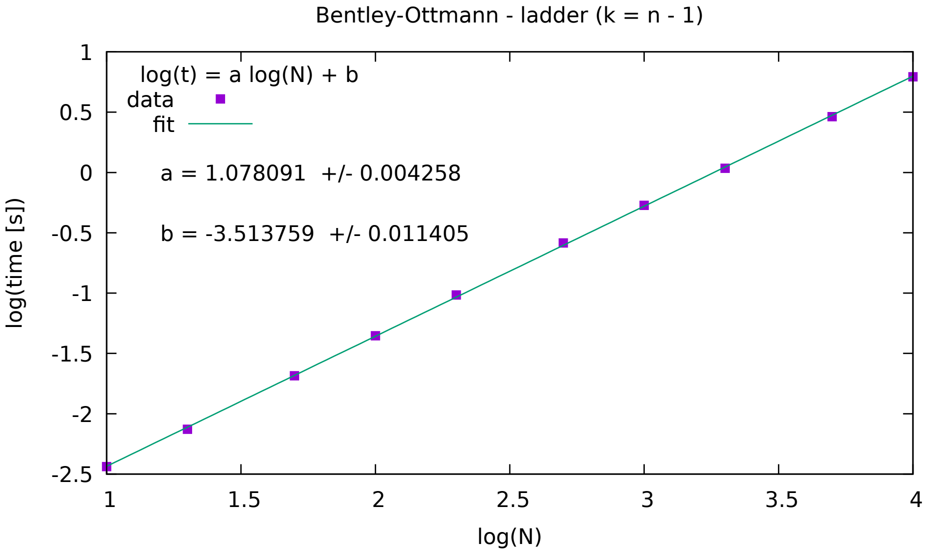
— odcinki jednakowej długości rozmieszczone na dużej powierzchni ($k < n$),
— odcinki jednakowej długości rozmieszczone na małej powierzchni ($k \sim n$),
— odcinki ułożone w formie drabiny ($k = n - 1$).

Złożoność naszych algorytmów z założenia jest czuła na liczbę przecięć, dlatego nie stosowano całkowicie losowych odcinków. Takie zbiory dają nam bardzo dużą ilość przecięć, przez co algorytm naiwny $O(n^2)$ już przy stosunkowo małej liczbie odcinków wejściowych jest wydajniejszy.

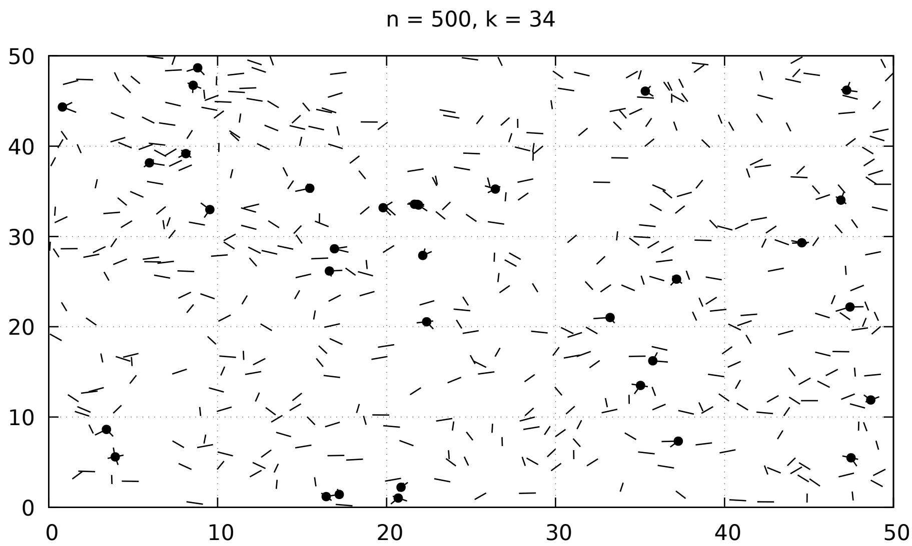
A.1.1. Algorytm Bentleya-Ottmanna



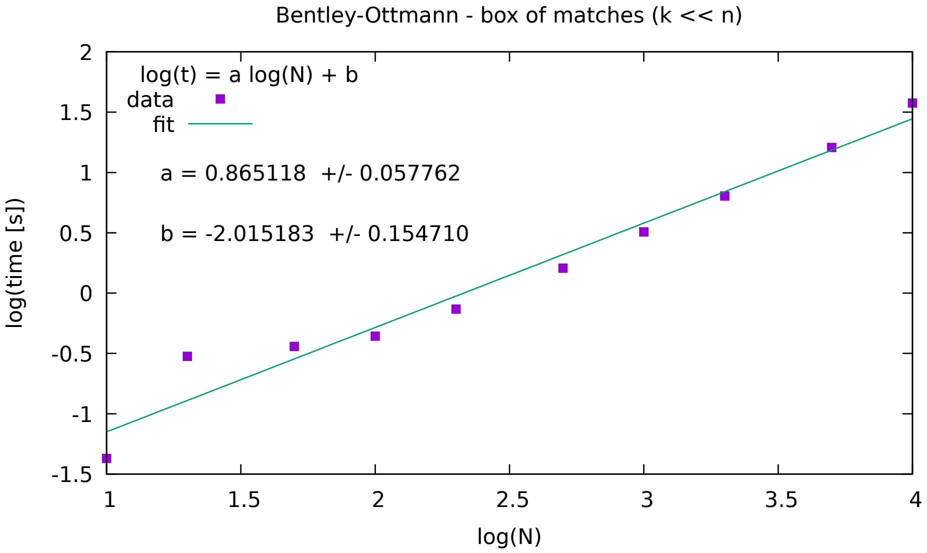
Rysunek A.1. Wynik działania algorytmu Bentleya-Ottmanna dla odcinków ułożonych w formie drabiny.



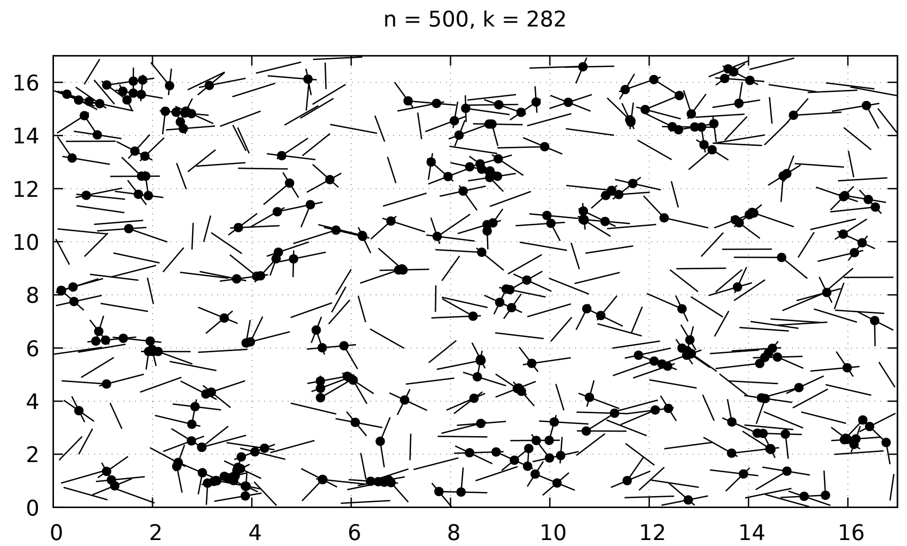
Rysunek A.2. Wykres wydajności algorytmu Bentleya-Ottmanna dla odcinków ułożonych w formie drabiny. Współczynik a lekko przekraczający 1 potwierdza złożoność $O(n \log n)$.



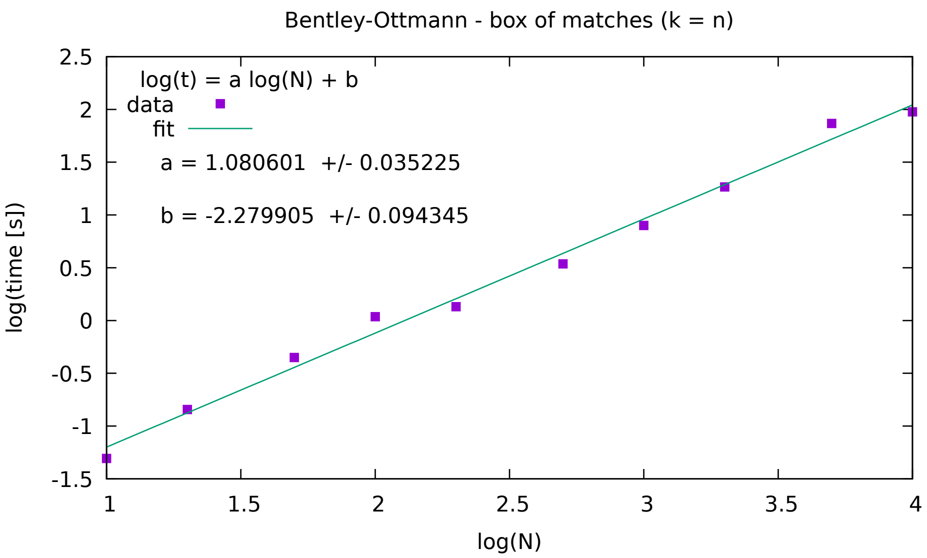
Rysunek A.3. Wynik działania algorytmu Bentleya-Ottmanna dla odcinków jednakowej długości (zapałki) na dużej powierzchni.



Rysunek A.4. Wykres wydajności algorytmu Bentleya-Ottmanna dla odcinków jednakowej długości (zapałki) na dużej powierzchni. Współczynik a bliski 1 potwierdza złożoność $O(n \log n)$.

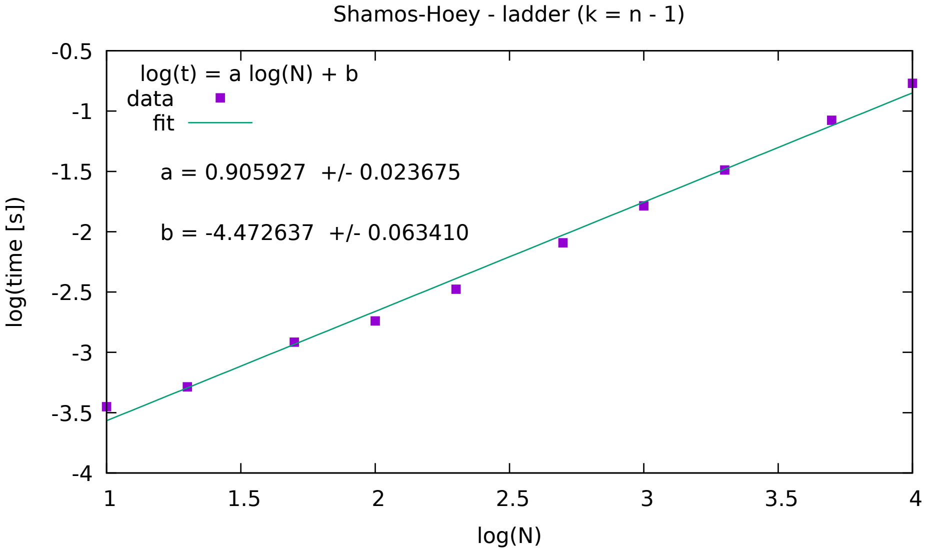


Rysunek A.5. Wynik działania algorytmu Bentleya-Ottmanna dla odcinków jednakowej długości (zapałki) na małej powierzchni.

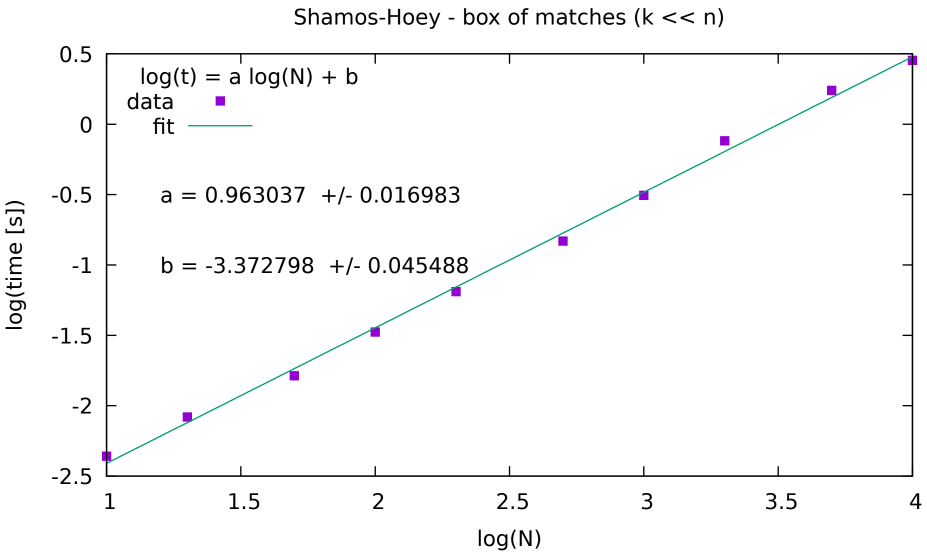


Rysunek A.6. Wykres wydajności algorytmu Bentleya-Ottmanna dla odcinków jednakowej długości (zapałki) na małej powierzchni. Współczynik a lekko przekraczający 1 potwierdza złożoność $O(n \log n)$.

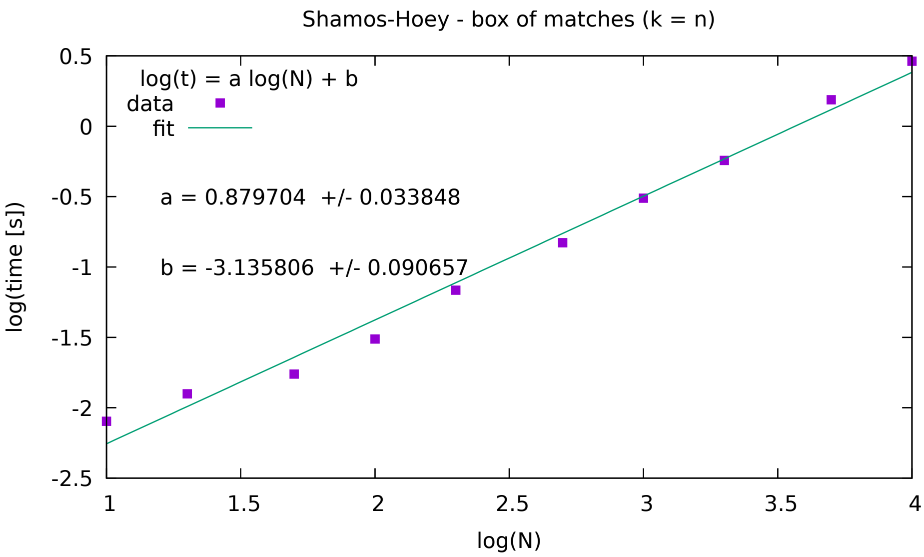
A.1.2. Algorytm Shamosa-Hoeya



Rysunek A.7. Wykres wydajności algorytmu Shamosa-Hoeya dla odcinków ułożonych w formie drabiny. Współczynik a bliski 1 potwierdza złożoność $O(n \log n)$.

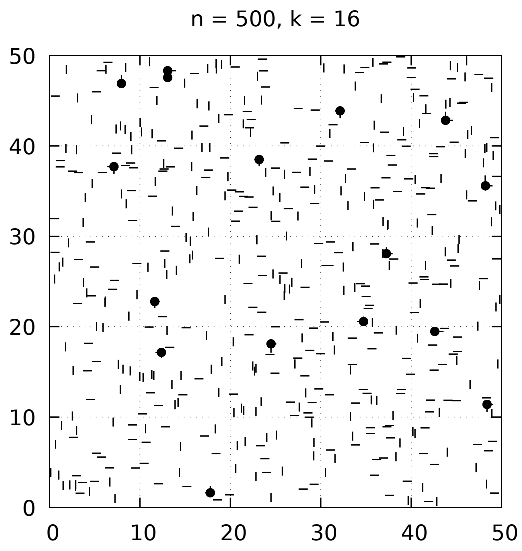


Rysunek A.8. Wykres wydajności algorytmu Shamosa-Hoeya dla odcinków jednakowej długości (zapałki) na dużej powierzchni. Współczynik a bliski 1 potwierdza złożoność $O(n \log n)$.

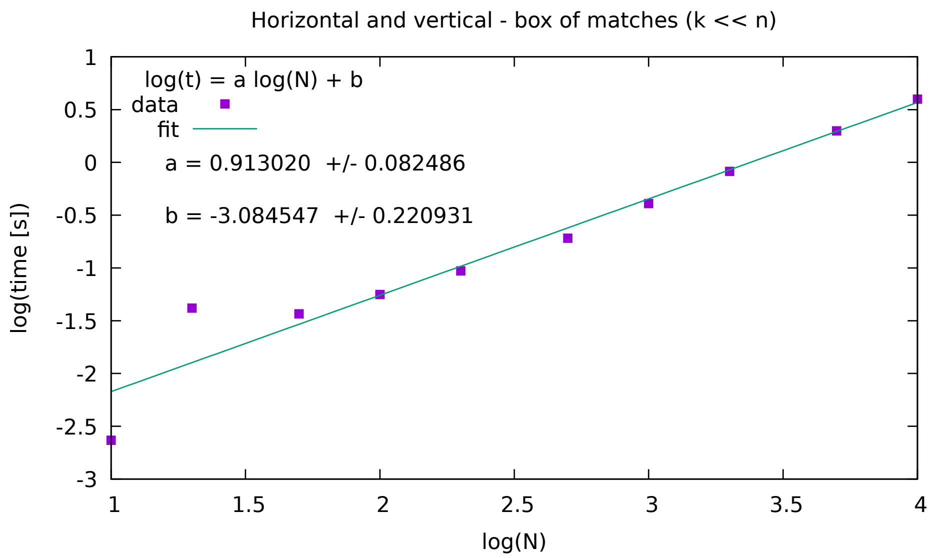


Rysunek A.9. Wykres wydajności algorytmu Shamosa-Hoeya dla odcinków jednakowej długości (zapałki) na małej powierzchni. Współczynik a bliski 1 potwierdza złożoność $O(n \log n)$.

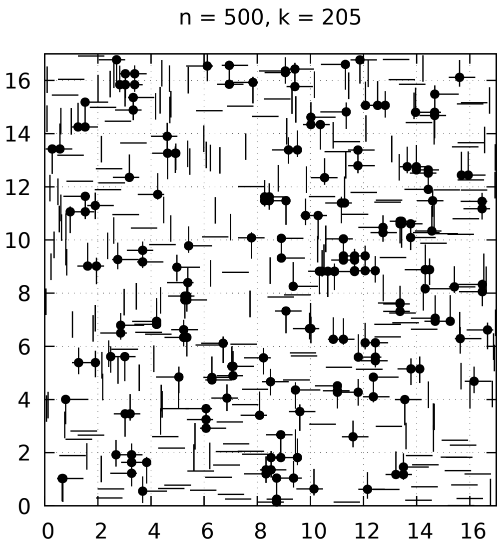
A.1.3. Algorytm dla odcinków pionowych i poziomych



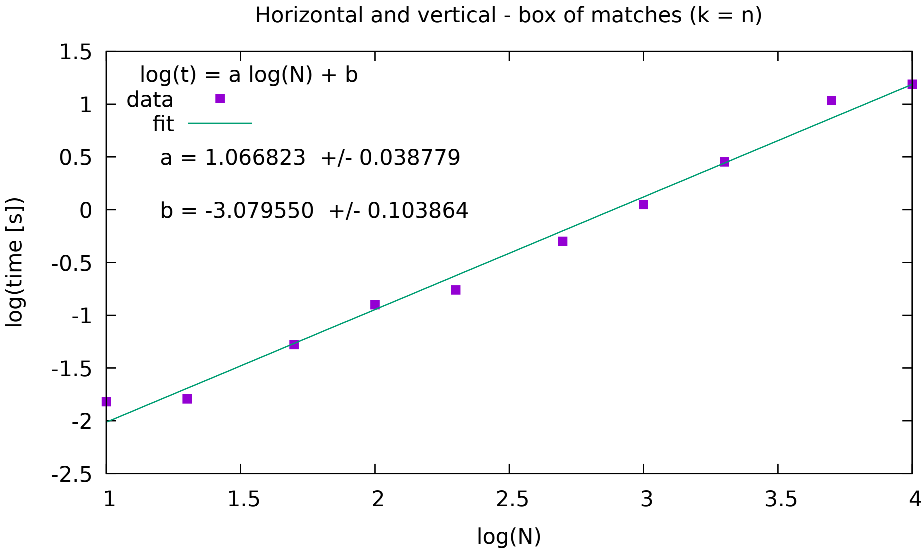
Rysunek A.10. Wynik działania algorytmu wykrywania przecięć dla pionowych i poziomych odcinków jednakowej długości (zapałki) na dużej powierzchni.



Rysunek A.11. Wykres wydajności wykrywania przecięć dla pionowych i poziomych dla odcinków jednakowej długości (zapałki) na dużej powierzchni. Współczynnik a bliski 1 potwierdza złożoność $O(n \log n)$.



Rysunek A.12. Wynik działania algorytmu wykrywania przecięć dla pionowych i poziomych odcinków jednakowej długości (zapałki) na małej powierzchni.

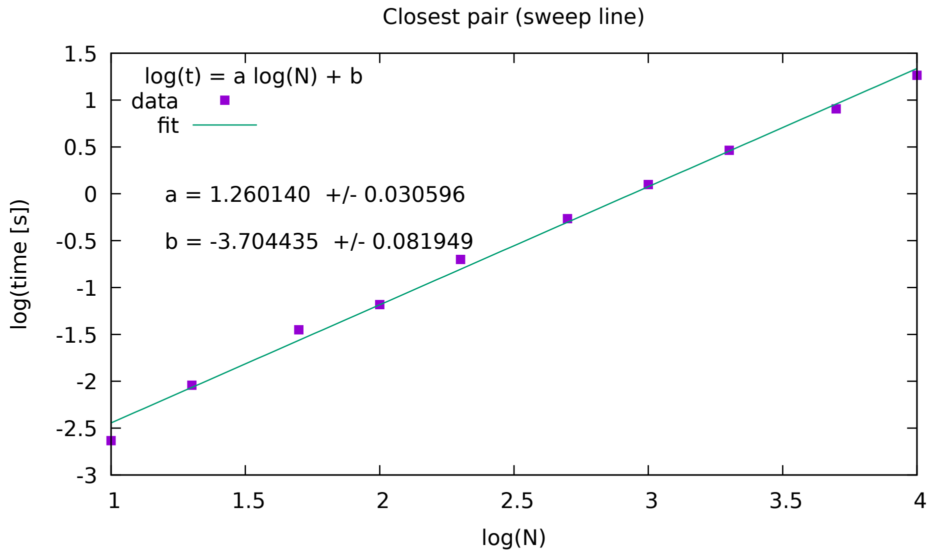


Rysunek A.13. Wykres wydajności algorytmu wykrywania przecięć dla pionowych i poziomych odcinków jednakowej długości (zapałki) na małej powierzchni. Współczynnik a lekko przekraczający 1 potwierdza złożoność $O(n \log n)$.

A.2. Testy wyszukiwania najbliższej pary punktów

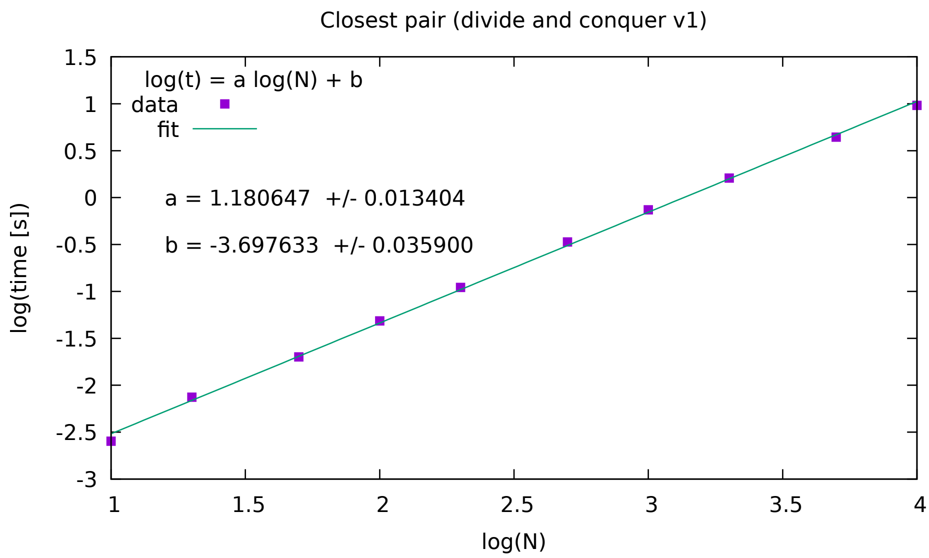
Do testowania algorytmów związanych z punktami wykorzystano zbiory losowych punktów na płaszczyźnie.

A.2.1. Para najbliższych punktów - technika zamiatania

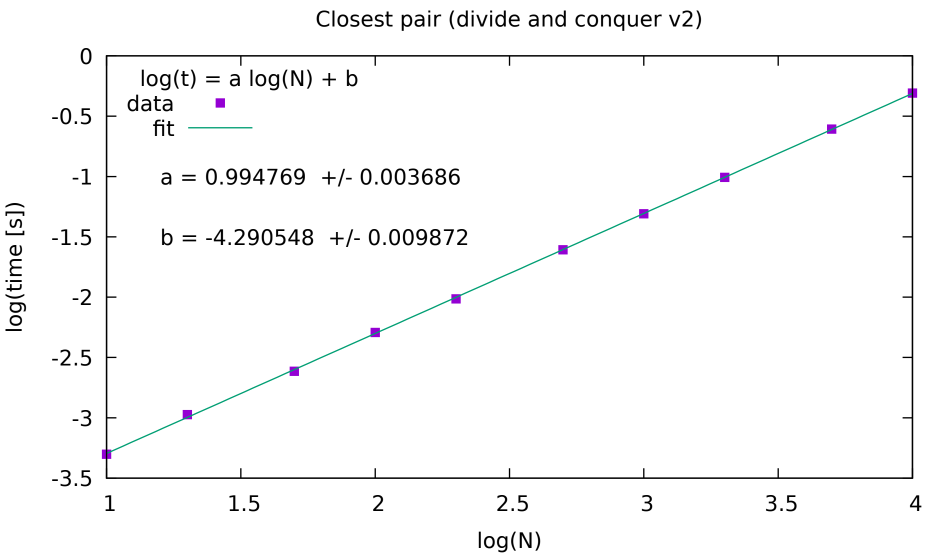


Rysunek A.14. Wykres wydajności algorytmu wyszukiwania pary najbliższych punktów techniką zamiatania. Współczynik a lekko przekraczający 1 potwierdza złożoność $O(n \log n)$.

A.2.2. Para najbliższych punktów - technika dziel i zwyciężaj



Rysunek A.15. Wykres wydajności algorytmu wyszukiwania pary najbliższych punktów techniką dziel i zwyciężaj. Współczynik a lekko przekraczający 1 potwierdza złożoność $O(n(\log n)^2)$.

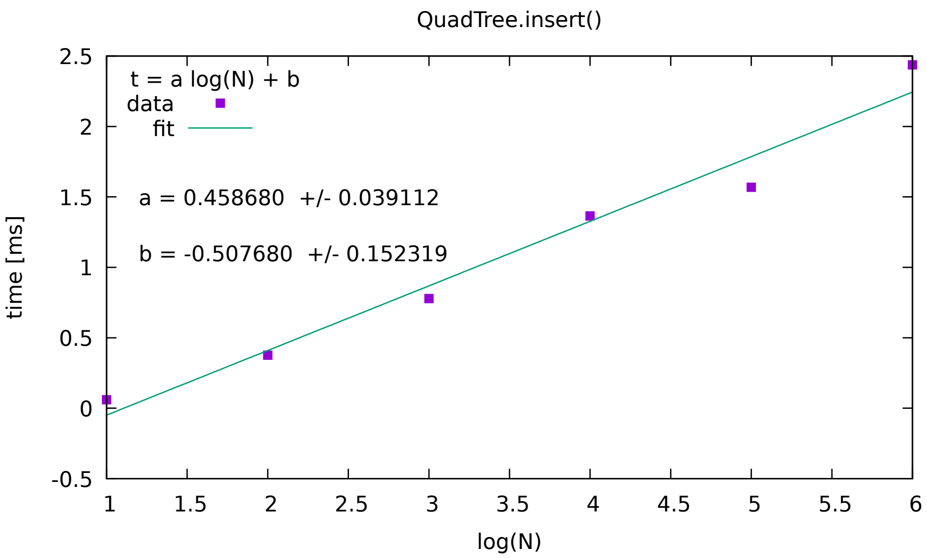


Rysunek A.16. Wykres wydajności algorytmu wyszukiwania pary najbliższych punktów techniką dziel i zwyciężaj z dodatkowym podziałem zbioru punktów. Współczynik a bliski 1 potwierdza złożoność $O(n \log n)$.

A.3. Testy drzewa czwórkowego

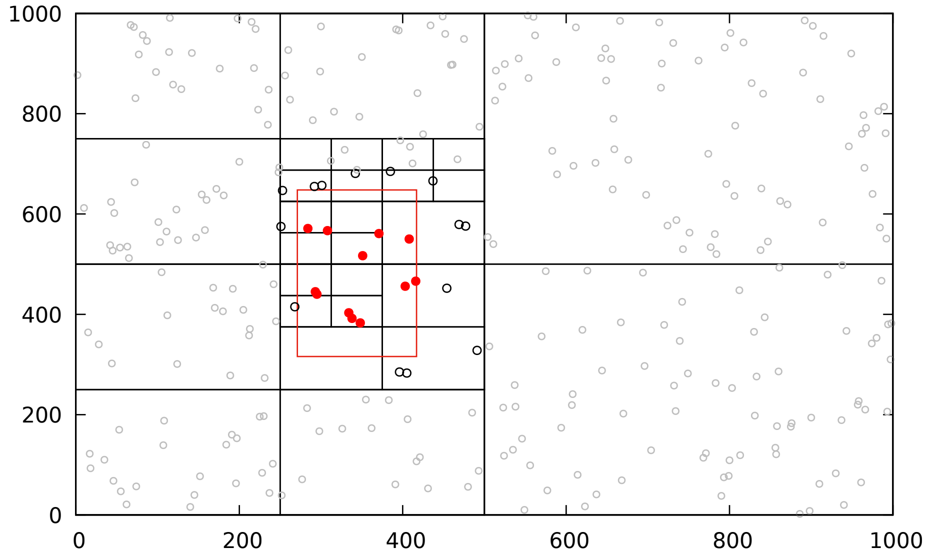
Do testowania wydajności drzewa czwórkowego wykorzystano losowo rozmieszczone punkty na powierzchni kwadratu.

A.3.1. Wstawianie elementu do drzewa czwórkowego

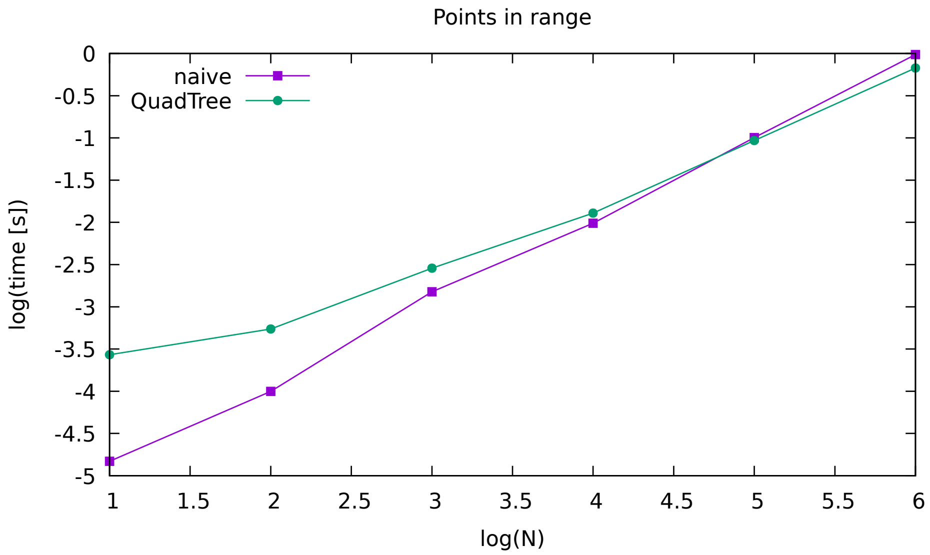


Rysunek A.17. Wykres wydajności wstawiania elementu do struktury drzewa czwórkowego. Bardzo niski współczynik a potwierdza złożoność $O(\log n)$.

A.3.2. Algorytm wyszukiwania punktów należących do pewnej płaszczyzny

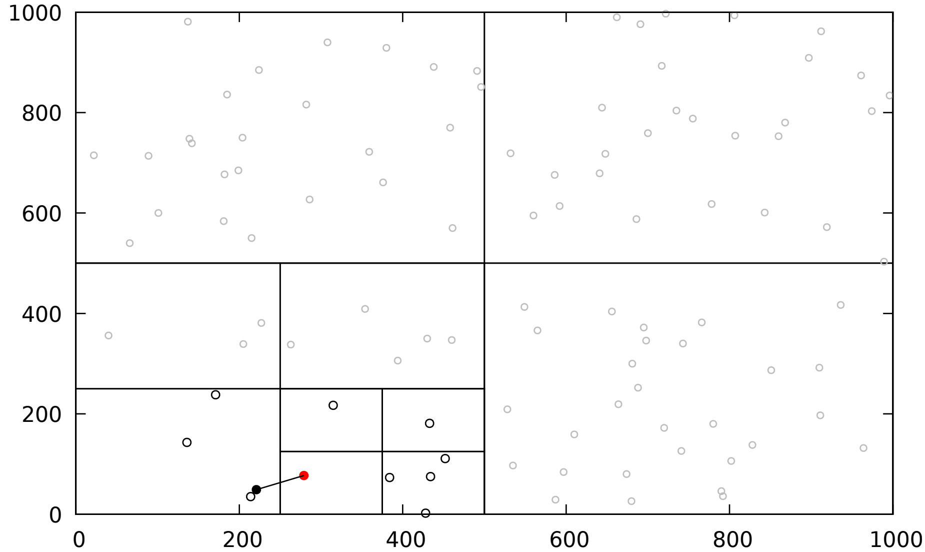


Rysunek A.18. Wynik działania algorytmu wyszukiwania punktów na danej powierzchni. Punkty czarne to punkty odwiedzone przez algorytm, czerwone to punkty należące do pewnej płaszczyzny (czerwony prostokąt).

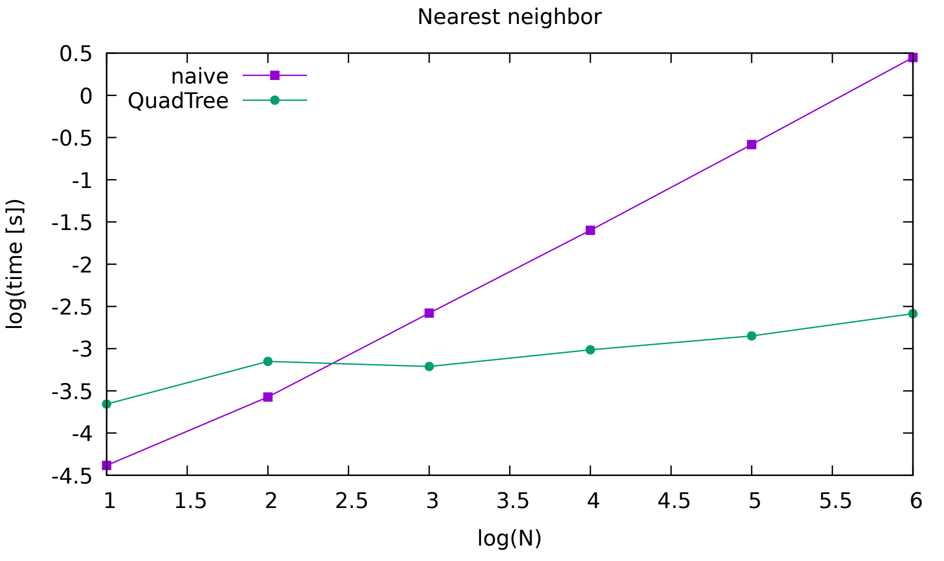


Rysunek A.19. Wykres porównujący wydajność algorytmu wyszukiwania punktów na danej powierzchni wykorzystując drzewo czwórkowe z algorytmem naiwnym. Dla n większych od ~ 50000 drzewo czwórkowe jest wydajeniejsze od metody siłowej.

A.3.3. Algorytm wyszukiwania najbliższego sąsiada pewnego punktu



Rysunek A.20. Wynik działania algorytmu wyszukiwania najbliższego sąsiada pewnego punktu. Punkty czarne bez wypełnienia to punkty odwiedzone przez algorytm, czerwony to punkt wejściowy.



Rysunek A.21. Wykres porównujący wydajność algorytmu wyszukiwania najbliższego sąsiada pewnego punktu wykorzystując drzewo czwórkowe z algorytmem naiwnym. Dla n większych od ~ 500 drzewo czwórkowe jest wydajeniejsze od metody siłowej.

Bibliografia

[1] Wikipedia, Sweep line algorithm, 2019,
https://en.wikipedia.org/wiki/Sweep_line_algorithm.

[2] Wikipedia, Computational geometry, 2019,
https://en.wikipedia.org/wiki/Computational_geometry.

[3] Wikipedia, Bentley-Ottmann algorithm, 2019,
https://en.wikipedia.org/wiki/Bentley%E2%80%93Ottmann_algorithm.

[4] M. I. Shamos, D. Hoey, *Geometric intersection problems*, 17th Annual Symposium on Foundations of Computer Science, pp. 208-215 (1976).

[5] Wikipedia, Fortune's algorithm, 2019,
https://en.wikipedia.org/wiki/Fortune's_algorithm.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Wprowadzenie do algorytmów*, Wydawnictwo Naukowe PWN, Warszawa 2012.

[7] Lech Banachowski, Krzysztof Diks, Wojciech Rytter, *Algorytmy i struktury danych*, Wydawnictwo naukowe PWN, Warszawa 2018.

[8] M. Berg, M. Kreveld, M. Overmars, O. Schwarzkopf, *Geometria obliczeniowa. Algorytmy i zastosowania*, WNT, Warszawa 2007.

[9] Franco P. Preparata, Michael Ian Shamos, *Geometria obliczeniowa. Wprowadzenie*, Helion, Gliwice 2003.

[10] Python Programming Language - Official Website,
https://www.python.org/.

[11] Marcin Permus, *Algorytmy geometryczne w języku Python*, Praca licencjacka, Uniwersytet Jagielloński, Kraków 2018.

[12] Wikibooks, Kolejka priorytetowa, 2019,
https://pl.wikibooks.org/wiki/Struktury_danych/Kolejki/Kolejka_priorytetowa.

[13] Python Docs, Heap queue algorithm, 2019,
https://docs.python.org/2/library/heapq.html.

[14] Wikipedia, Drzewo AVL, 2019,
https://pl.wikipedia.org/wiki/Drzewo_AVL.

[15] Michiel Smid, *The closest pair problem: A plane sweep algorithm*, 2019,
https://people.scs.carleton.ca/~michiel/lecturenotes/ALGGEOM/sweepclosestpair.pdf.

[16] Dan Sunday, Geometry Algorithms, 2019,
http://geomalgorithms.com/.

[17] Diane Souvaine, *Line Segment Intersection Using a Sweep Line Algorithm*, Tufts University, 2005,
http://www.cs.tufts.edu/comp/163/notes05/seg_intersection_handout.pdf.

[18] Lech Banachowski, Krzysztof Diks, Wojciech Rytter, *Algorytmy i struktury danych*, Wydawnictwo Naukowe PWN, Warszawa 2018.

[19] GeeksforGeeks, Closest Pair of Points using Divide and Conquer algorithm, 2019,
https://www.geeksforgeeks.org/.

[20] Wikipedia, Quadtree, 2019,
https://en.wikipedia.org/wiki/Quadtree.

[21] Patrick Surry, D3JS quadtree nearest neighbor algorithm, 2018,
http://bl.ocks.org/patricksurry/6478178.

[22] Michiel Smid, *Computing intersections in a set of horizontal and vertical line segments*, 2019,
https://people.scs.carleton.ca/~michiel/lecturenotes/ALGGEOM/horverintersect.pdf.

[23] Tyler Sandman, *PyBST*, GitHub repository, 2019,
https://github.com/TylerSandman/py-bst.

[24] Rosetta Code, *AVL Tree*, 2019,
https://rosettacode.org/wiki/AVL_tree#Python.