AGH AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Algorytmy geometryczne - projekt

Wyszukiwanie geometryczne przeszukiwanie obszarów ortogonalnych QuadTree oraz KD-drzewa

Dokumentacja

05.01.2025

Aleksander Jóźwik

Szymon Hołysz

Spis treści

1. Wprowadzenie

Niniejsza dokumentacja opisuje implementacje dwóch struktur danych: *KD-drzewa* i *QuadTree* w języku Python. Struktury te są fundamentalne w realizacji wydajnych zapytań przestrzennych, znajdujących zastosowanie w systemach geolokalizacyjnych, bazach danych przestrzennych i aplikacjach GIS.

KD-drzewa umożliwiają efektywne wykonywanie zapytań zakresowych w przestrzeni wielowymiarowej, co jest kluczowe np. w systemach nawigacyjnych przy wyszukiwaniu punktów zainteresowania w określonym obszarze czy w bazach danych geoprzestrzennych przy analizie rozkładu obiektów. *QuadTree* z kolei specjalizuje się w operacjach na danych dwuwymiarowych, co znajduje zastosowanie w mapach cyfrowych przy dynamicznym ładowaniu szczegółów terenu czy w systemach monitoringu przy śledzeniu obiektów w określonych sektorach przestrzeni.

2. Część techniczna

2.1. Wymagania techniczne

Kod programu był uruchamiany w interpreterze języka *Python* w wersji 3.13.1. Wymagane zależności zostały zapisane w pliku *requirements.txt*. Są to między innymi:

```
numpy >= 1.25.2
pandas >= 2.0.3
matplotlib >= 3.7.2
notebook >= 6.5.4
```

W celu instalacji należy wywołać następujące polecenie:

```
pip install -r requirements.txt
```

W projekcie umieszczono oraz wykorzystano narzędzie wizualizacji stworzone przez koło naukowe BIT. Kod źródłowy wraz z dokumentacją dostępny jest pod poniższym adresem:
https://github.com/aghbit/Algorytmy-Geometryczne.

Do poprawnego funkcjonowania nie są wymagane dodatkowe zależności.

2.2. Schemat pakietów

```
KDTree_QuadTree_project/
│
├── kdtree/
│   ├── kdtree.py
│   ├── kdtree_test.py
│   └── kdtree_visualizer.py
│
├── quadtree/
│   └── quad.py
│
├── visualizer/
│
├── automatic_tests.py
├── generators.py
├── gui_creator.py
├── main.ipynb
└── requirements.txt
```

2.3. Pakiet kdtree

2.3.1. Moduł kdtree

2.3.1.1. Klasa *Node*

Reprezentuje węzeł w strukturze KD-drzewa. Zawiera odniesienia do lewego i prawego dziecka.

2.3.1.1.1. Atrybuty

- self.line - wartość jednej ze współrzędnych, przez którą poprowadzono linię podziału,
- self.left - odniesienie do lewego dziecka węzła,
- self.right - odniesienie do prawego dziecka węzła,
- self.point - współrzędne punktu zawartego w węźle będącym liściem (krotka).

2.3.1.1.2. Metody

```
def __init__(
    self: Self@Node,
    line: Any | None = None,
    left: Any | None = None,
    right: Any | None = None,
    point: Any | None = None
) -> None
```

Konstruktor węzła w KD-drzewie.

Parametry:

- line - wartość jednej ze współrzędnych, przez którą poprowadzono linię podziału,
- left - odniesienie do lewego dziecka węzła,
- right - odniesienie do prawego dziecka węzła,
- point - współrzędne punktu zawartego w węźle będącym liściem (krotka).

```
def report_subtree(self: Self@Node) -> (Any | list)
```

Zwraca wszystkie punkty poddrzewa zakorzenionego w danym węźle.

Zwraca: listę punktów reprezentowanych jako krotki.

2.3.1.2. Klasa *KDtree*

Klasa KDtree jest implementacją drzewa k-wymiarowego, wykonaną na podstawie [1] oraz [2].
Pozwala ona na wydajne zapytania o punkty z obszaru k-wymiarowej przestrzeni. Złożoność pamięciowa $O(n)$, gdzie n to liczba punktów.

2.3.1.2.1. Atrybuty

- self.k - liczba wymiarów,
- self.eps - tolerancja dla zera,
- self.root - węzeł będący korzeniem KD-drzewa.

2.3.1.2.2. Metody

```
def __build_kdtree(
    self: Self@KDtree,
    P: Any,
    depth: Any
) -> (Node | None)
```

Buduje rekurencyjnie KD-drzewo.

Parametry:

- P - k-wymiarowa lista list punktów posortowanych ze względu na k-tą współrzędną,
- depth - aktualna głębokość w drzewie.

Zwraca: obiekt klasy *Node* będący korzeniem drzewa (poddrzewa).

```
def __init__(
    self: Self@KDtree,
    P: Any,
    k: int = 2,
    eps: float = 0
) -> None
```

Konstruktor inicjalizujący instancję KD-drzewa. Złożoność: $O(k \cdot n \log n)$, gdzie k jest liczbą wymiarów, a n to liczba punktów. W większości przypadków $k \ll n$, a więc za złożoność budowania drzewa można przyjąć $O(n \log n)$.

Parametry:

- P - lista punktów,
- k - liczba wymiarów,
- eps - tolerancja dla zera.

Wyjątki:

- ValueError - jeżeli przekazana lista punktów P jest pusta,
- TypeError - jeżeli punkty nie spełniają zadeklarowanego wymiaru k.

```
def __contains(
    self: Self@KDtree,
    lower_bound: Any,
    upper_bound: Any,
    lower_left: Any,
    upper_right: Any
) -> bool
```

Sprawdza czy dany region całkowicie zawiera się w regionie, z którego wyszukujemy punkty.

Parametry:
- lower_bound - punkt w postaci krotki lub listy, który reprezentuje dolny zakres aktualnie rozważanego zakresu,
- upper_bound - punkt w postaci krotki lub listy, który reprezentuje górny zakres aktualnie rozważanego zakresu,
- lower_left - punkt w postaci krotki lub listy, który reprezentuje dolny zakres regionu zapytania przestrzennego,
- upper_right - punkt w postaci krotki lub listy, który reprezentuje górny zakres regionu zapytania przestrzennego.

Zwraca: bool: True jeżeli region się całkowicie zawiera, False w przeciwnym przypadku.

```
def __intersects(
    self: Self@KDtree,
    lower_bound: Any,
    upper_bound: Any,
    lower_left: Any,
    upper_right: Any
) -> bool
```

Sprawdza czy dany region przecina się z regionem, o który stworzono zapytanie.

Parametry:
- lower_bound - punkt w postaci krotki lub listy, który reprezentuje dolny zakres aktualnie rozważanego zakresu,
- upper_bound - punkt w postaci krotki lub listy, który reprezentuje górny zakres aktualnie rozważanego zakresu,
- lower_left - punkt w postaci krotki lub listy, który reprezentuje dolny zakres regionu zapytania przestrzennego,
- upper_right - punkt w postaci krotki lub listy, który reprezentuje górny zakres regionu zapytania przestrzennego.

Zwraca: bool: True jeżeli regiony się przecinają, False w przeciwnym przypadku.

```
def __search_kdtree(
    self: Self@KDtree,
    v: Node,
    lower_bound: list,
    upper_bound: list,
    lower_left: Any,
    upper_right: Any,
    depth: Any
) -> (list[Any | None] | Any | list)
```

Rekurencyjnie przeszukuje KD-drzewo w celu znalezienia punktów z zadanego regionu.

Parametry:
- v - obecny węzeł w KD-drzewie,
- lower_bound - punkt w postaci listy, który reprezentuje dolny zakres aktualnie rozważanego zakresu,
- upper_bound - punkt w postaci listy, który reprezentuje górny zakres aktualnie rozważanego zakresu,
- lower_left - punkt w postaci krotki lub listy, który reprezentuje dolny zakres regionu zapytania przestrzennego,

- upper_right - punkt w postaci krotki lub listy, który reprezentuje górny zakres regionu zapytania przestrzennego,
- depth - aktualna głębokość w KD-drzewie.

Zwraca: listę punktów z danego obszaru.

```
def query(
    self: Self@KDtree,
    lower_left: Any,
    upper_right: Any
) -> (list[Any | None] | Any | list)
```

Tworzy zapytanie do KD-drzewa, aby znaleźć wszystkie punkty z zadanego obszaru.
Złożoność (przy zbalansowanym drzewie) dla $k = 2$ wynosi: $O(\sqrt{n} + d)$, gdzie d jest liczbą znalezionych punktów. Dla dowolnej liczby wymiarów jest to: $O\left(n^{1-\frac{1}{k}} + d\right)$.

Parametry:
- lower_left - punkt w postaci krotki lub listy, który reprezentuje dolny zakres regionu zapytania przestrzennego,
- upper_right - punkt w postaci krotki lub listy, który reprezentuje górny zakres regionu zapytania przestrzennego.

Zwraca: listę wszystkich punktów, które znajdują się w zadanym obszarze.

Wyjątki:
- TypeError - jeżeli wymiary wprowadzonych punktów nie zgadzają się z zadeklarowanym wymiarem KD-drzewa.

2.3.2. Moduł kdtree_visualizer

Moduł wykorzystuje narzędzie wizualizacji stworzone przez koło naukowe BIT oraz bibliotekę *matplotlib*.

2.3.2.1. Klasa *KDtreeVisualizer*

Klasa *KDtreeVisualizer* stanowi modyfikację klasy *KDtree* z pakietu kdtree, która ma dodaną możliwość graficznej wizualizacji krok po kroku budowania drzewa oraz wykonywania zapytań o punkty z zadanego obszaru. Poniżej zostaną opisane różnice pomiędzy tymi modułami.

2.3.2.1.1. Atrybuty

Dodano następujące atrybuty:

- self.points - lista punktów, które znajdują się w KD-drzewie (potrzebna do poprawnej wizualizacji dla wielokrotnych zapytań),
- self.lines - lista odcinków (w postaci krotek dwóch punktów: początku i końca) stanowiących linie podziału (wykorzystywana przy wizualizacji dla zapytań),
- self.vis_build - obiekt klasy Visualizer, który umożliwia wizualizację budowania drzewa,
- self.vis_query - obiekt klasy Visualizer, który umożliwia wizualizację zapytania,
- self.start_lower_bound - punkt, który reprezentuje dolny zakres obszaru, w którym znajdują się wszystkie punkty (wykorzystywany przy wizualizacji),
- self.start_upper_bound - punkt, który reprezentuje górny zakres obszaru, w którym znajdują się wszystkie punkty (wykorzystywany przy wizualizacji).

2.3.2.1.2. Metody

```
def __init__(
    self: Self@KDtreeVisualizer,
    P: Any,
    eps: int = 0
) -> None
```

Konstruktor KD-drzewa pozbawiony możliwości określenia liczby wymiarów - wizualizacja możliwa tylko dla dwóch wymiarów.

Wyjątki:
- TypeError - jeżeli punkty nie są z dwuwymiarowej przestrzeni.

```
def show_build_visualization(
    self: Self@KDtreeVisualizer,
    interval: int = 400
) -> Image
```

Wyświetla wizualizację budowania KD-drzewa.

Parametry:
- interval - interwał w milisekundach pomiędzy klatkami animacji (domyślnie 400 ms).

Zwraca: obraz w formacie *gif*.

```
def show_query_visualization(
    self: Self@KDtreeVisualizer,
    interval: int = 600
) -> Image
```

Wyświetla wizualizację procesu zapytania.

Parametry:
- interval - interwał w milisekundach pomiędzy klatkami animacji (domyślnie 600 ms).

Zwraca: obraz w formacie *gif*.

2.3.2.2. Funkcja *visualize_queried_points*

```
def visualize_queried_points(
    P: Any,
    lower_left: Any,
    upper_right: Any,
    result: Any
) -> None
```

Funkcja służy do wizualizacji punktów, obszaru zapytania oraz znalezionych punktów.

Parametry:
- P - lista punktów,
- lower_left - punkt reprezentujący lewy dolny róg prostokątnego obszaru zapytania,
- upper_right - punkt reprezentujący prawy górny róg prostokątnego obszaru zapytania,
- result - lista punktów zwrócona w zapytaniu.

Wyjątki:
- TypeError - jeżeli punkty nie są z dwuwymiarowej przestrzeni.

2.3.3. Moduł kdtree_test

Moduł ten zawiera 10 wielowymiarowych testów jednostkowych służących do przetestowania poprawności implementacji KD-drzewa.

2.3.3.1. Funkcja *runtests*

```
def runtests() -> None
```

Uruchamia testy. Wypisuje odpowiednio informacje na ekranie:
- w przypadku zaliczonego i-tego testu: „Test i: zaliczony!”,
- w przypadku niezaliczonego i-tego testu: „Test i: niezaliczony!!!”.

2.4. Pakiet quadtree

2.4.1. Klasa *Quarter*

Typ wyliczeniowy (enum) reprezentujący cztery typy wezłów w podziale na ćwiartki:
- NE (prawa górna)
- NW (lewa górna)
- SW (lewa dolna)
- SE (prawa dolna)

2.4.2. Klasa *Rectangle*

Reprezentuje przedział ortogonalny. Zawiera współrzędne (x. y) minimalnego i maksymalnego punktu przedziału.

2.4.2.1. Atrybuty

- self.min_x - współrzędna x minimalnego punktu
- self.min_y - współrzędna y minimalnego punktu
- self.max_x - współrzędna x maksymalnego punktu
- self.max_y - współrzędna y maksymalnego punktu

2.4.2.2. Metody

```
def med_y(self): return (self.max_y + self.min_y) / 2.0
def med_x(self): return (self.max_x + self.min_x) / 2.0
```

Zwracają średnią ze współrzędnych x i y punktu maksymalnego i minimalnego.

```
def rectangle_partition(self):
    med_y = self.med_y()
    med_x = self.med_x()
    s_ne = Rectangle(med_x, med_y, self.max_x, self.max_y)
    s_nw = Rectangle(self.min_x, med_y, med_x, self.max_y)
    s_sw = Rectangle(self.min_x, self.min_y, med_x, med_y)
    s_se = Rectangle(med_x, self.min_y, self.max_x, med_y)

    return s_ne, s_nw, s_sw, s_se
```

Zwraca cztery obiekty klasy `Rectangle()` odpowiadające podziałowi prostokąta na ćwiartki.

```
def intersects(self, other):
    return (self.min_x < other.max_x and self.max_x > other.min_x
            and self.min_y < other.max_y and self.max_y > other.min_y)
```

Przyjmuje wskazanie na inny obiekt `Rectangle()` i sprawdza, czy się przecinają. Zwraca wartość logiczną.

```
  def contains(self, point):
      x, y = point
          if self.min_x <= x <= self.max_x and self.min_y <= y <= self.max_y:
              return True
          return False
```

Przyjmuje krotkę dwóch współrzędnych punktu i sprawdza, czy należy do przedziału. Zwraca wartość logiczną.

```
def draw(self, visualizer, color):
    return visualizer.add_line_segment((
                      ((self.min_x, self.min_y), (self.max_x, self.min_y)),
                      ((self.min_x, self.min_y), (self.min_x, self.max_y)),
                      ((self.min_x, self.max_y), (self.max_x, self.max_y)),
                      ((self.max_x, self.min_y), (self.max_x, self.max_y))), color = color)
```

Przyjmuje wskazanie na obiekt `Visualizer()` i parametr `color`. Dodaje do wizualizatora prostokąt odpowiadający przedziałowi i zwraca wskazanie na ten prostokąt.

2.4.3. Klasa *Node*

Reprezentuje węzeł w strukturze drzewa ćwiartek. Zawiera odniesienia do rodzica i dzieci, drzewa, przedziału, któremu odpowiada i zbioru punktów, które przechowuje.

2.4.3.1. Atrybuty

- self.tree - wskazanie na obiekt klasy `Quad()`
- self.quarter - wartość `Quarter(Enum)`
- self.parent - wskazanie na rodzica, obiekt klasy `Node()`
- self.square - wskazanie na przedział, któremu odpowiada; obiekt klasy `Rectangle()`
- self.ne, self.nw, self.se, self.sw - wskazania na dzieci obiekty klasy `Node()`
- self.points = wskazanie na zbiór `set()` punktów, które przechowuje; jeżeli węzeł nie jest liściem, to `self.points = None`

2.4.3.2. Metody

```
def is_leaf(self): return self.points is not None and self != self.tree.root
```

Sprawdza, czy węzeł jest liściem. Zwraca wartość logiczną.

```
def construct_subtree(self, points, forced = False):
    if len(points) <= BUCKET_SIZE and not forced:
        self.points = points
        self.tree.leaves.append(self)
    else:
        x = self.square.med_x()
        y = self.square.med_y()
        p_ne, p_nw, p_sw, p_se = set_partition(points, x, y)
        s_ne, s_nw, s_sw, s_se = self.square.rectangle_partition(

        self.ne = Node(self.tree, Quarter.NE, s_ne, self)
        self.nw = Node(self.tree, Quarter.NW, s_nw, self)
        self.sw = Node(self.tree, Quarter.SW, s_sw, self)
        self.se = Node(self.tree, Quarter.SE, s_se, self)

        self.ne.construct_subtree(p_ne)
        self.nw.construct_subtree(p_nw)
        self.sw.construct_subtree(p_sw)
        self.se.construct_subtree(p_se)
```

Rekurencyjnie buduje drzewo ćwiartek. Przyjmuje zbiór punktów i w zależności od ich liczby tworzy liść (umieszcza punkty w self.points) albo dzieli punkty na ćwiartki i umieszcza je w poddrzewach.

```
def insert_subtree(self, point):
    if not self.square.contains(point): return False
    if self.is_leaf():
        if len(self.points) < BUCKET_SIZE:
            self.points.add(point)
            return True
        else:
            points_to_add = self.points.copy()
            points_to_add.add(point)
            self.construct_subtree(points_to_add)
            return True
    if self.ne.insert_subtree(point): return True
    if self.nw.insert_subtree(point): return True
    if self.se.insert_subtree(point): return True
    if self.sw.insert_subtree(point): return True
```

Rekurencyjnie wstawia punkt do drzewa. Przyjmuje krotkę współrzędnych punktów i w zależności od tego, czy przedział bieżącego węzła może zawierać dany punkt, albo wstawia go do jednego z poddrzew, albo zwraca `False`.

```
def query_range_subtree(self, range_rect):
    result = set()
    if self.square.intersects(range_rect):
        if self.points is not None:
            for point in self.points:
                if range_rect.contains(point):
                    result.add(point)
        if self.ne is not None:
            r_ne = self.ne.query_range_subtree(range_rect)
            if r_ne is not None: result.update(r_ne)
        if self.nw is not None:
            r_nw = self.nw.query_range_subtree(range_rect)
            if r_nw is not None: result.update(r_nw)
        if self.sw is not None:
            r_sw = self.sw.query_range_subtree(range_rect)
            if r_sw is not None: result.update(r_sw)
        if self.se is not None:
            r_se = self.se.query_range_subtree(range_rect)
            if r_se is not None: result.update(r_se)
    return result
```

Rekurencyjnie wyszukuje punktów należących do przedziału ortogonalnego. Przyjmuje obiekt klasy `Rectangle()`, sprawdza czy przedział węzła przecina zadany przedział i jeżeli bieżący węzeł jest liściem, sprawdza, czy punkty w nim przechowywane należą do zadanego przedziału. W przeciwnym wypadku przeszukuje dzieci bieżącego węzła.

```
def graphic_query_range_subtree(self, range_rect, visualizer, color):
    result = set()
    temp_square = self.square.draw(visualizer, color)
    stay = False
    if self.square.intersects(range_rect):
        if self.points is not None:
            stay = True
            for point in self.points:
                if range_rect.contains(point):
                    visualizer.add_point(point, color = color)
                    result.add(point)
        if self.ne is not None:
            r_ne = self.ne.graphic_query_range_subtree(range_rect, visualizer, color)
            if r_ne is not None: result.update(r_ne)
        if self.nw is not None:
            r_nw = self.nw.graphic_query_range_subtree(range_rect, visualizer, color)
            if r_nw is not None: result.update(r_nw)
        if self.sw is not None:
            r_sw = self.sw.graphic_query_range_subtree(range_rect, visualizer, color)
            if r_sw is not None: result.update(r_sw)
        if self.se is not None:
            r_se = self.se.graphic_query_range_subtree(range_rect, visualizer, color)
            if r_se is not None: result.update(r_se)
    if not stay:
       visualizer.remove_figure(temp_square)
    return result
```

Wersja graficzna powyższej metody, która dodatkowo przyjmuje wskazanie na wizualizator `Visualizer()` i dodaje do niego kolejne etapy wyszukiwania używając koloru `color`.

```
def draw(self, visualizer, color):
    self.square.draw(visualizer, color)
    if self.ne is not None: self.ne.draw(visualizer, color)
    if self.nw is not None: self.nw.draw(visualizer, color)
    if self.sw is not None: self.sw.draw(visualizer, color)
    if self.se is not None: self.se.draw(visualizer, color)
```

Przyjmuje wskazanie na obiekt `Visualizer()` i parametr `color`. Dodaje do wizualizatora prostokąt odpowiadający bieżącemu węzłowi i rekurencyjnie wywołuje metodę dla swoich dzieci.

2.4.4. Klasa *Quad*

Reprezentuje drzewo ćwiartek. Zawiera odniesienie do korzenia drzewa, zbiór punktów należących do drzewa i listę liści.

2.4.4.1. Atrybuty

- self.points - zbiór punktów, na podstawie których skonstruowane zostało drzewo
- self.leaves - lista liści
- self.root wskazanie na obiekt `Node()` będący korzeniem

2.4.4.2. Metody

```
def __init__(self, points):
    self.points = points
    self.leaves = []
    self.root = Node(self, None, min_square(points))
    self.root.construct_subtree(points)
```

Konstruktor inicjalizujący instancję drzewa ćwiartek. Złożoność budowy drzewa zależy od głębokości drzewa d i liczby punktów n. Średni czas budowy drzewa wynosi $O(n \log n)$, natomiast w pesymistycznym przypadku (głębokość drzewa w przybliżeniu równa liczbie punktów) może wynieść $O(n^2)$.

```
def insert(self, point): return self.root.insert_subtree(point)
```

Przyjmuje krotkę współrzędnych punktu i wywołuje rekurencyjną metodę `insert_subtree()` dla korzenia drzewa.

```
def query_range(self, min_point, max_point):
    range_rect = Rectangle(min_point[0], min_point[1], max_point[0], max_point[1])
    return self.root.query_range_subtree(range_rect)
```

Przyjmuje minimalny i maksymalny punkt przedziału ortogonalnego i wywołuje rekurencyjną metodę przeszukiwania `query_range_subtree` dla korzenia drzewa. Pesymistyczna złożoność takiego wyszukiwania to $O(n)$.

```
def graphic_query_range(self, min_point, max_point, visualizer, color):
    range_rect = Rectangle(min_point[0], min_point[1], max_point[0], max_point[1])
    range_rect.draw(visualizer, 'brown')
    return self.root.graphic_query_range_subtree(range_rect, visualizer, color)
```

Wersja graficzna powyższej metody. Wywołuje graficzną wersję metody rekurencyjnej.

```
def draw(self, visualizer, color):
    self.root.draw(visualizer, color)
```

Przyjmuje wskazanie na wizualizator i wywołuje rekurencyjną metodę rysowania poddrzewa dla korzenia drzewa.

2.5. Moduł *generators*

Zawiera funkcje generujące zbiory punktów o różnych charakterystykach. Wykorzystano biblioteki *numpy* oraz *random*.

2.5.1. Funkcja *generate_uniform_points*

```
def generate_uniform_points(
    left: Any,
    right: Any,
    n: int = 10 ** 5
) -> list[tuple[Any, ...]]
```

Funkcja generuje równomiernie n punktów na kwadratowym obszarze od left do right (jednakowo na osi y) o współrzędnych rzeczywistych.

Parametry:

- left - lewy kraniec przedziału,
- right - prawy kraniec przedziału,
- n - liczba generowanych punktów.

Zwraca: tablicę punktów w postaci krotek współrzędnych np. $[(x_1, y_1), (x_2, y_2), ...(x_n, y_n)]$.

2.5.2. Funkcja *generate_normal_points*

```
def generate_normal_points(
    mean: Any,
    std: Any,
    n: int = 10 ** 5
) -> list[tuple[Any, ...]]
```

Funkcja generuje n punktów o rozkładzie normalnym na płaszczyźnie o współrzędnych rzeczywistych.

Parametry:

- mean - średnia wartość rozkładu,
- std - odchylenie standardowe rozkładu,
- n - liczba generowanych punktów.

Zwraca: tablicę punktów w postaci krotek współrzędnych np. $[(x_1, y_1), (x_2, y_2), ...(x_n, y_n)]$.

2.5.3. Funkcja *generate_collinear_points*

```
def generate_collinear_points(
    a: Any,
    b: Any,
    n: int = 100,
    x_range: int = 1000
) -> list[tuple]
```

Funkcja generuje równomiernie n współliniowych punktów leżących na prostej ab pomiędzy punktami a i b.

Parametry:
- a - krotka współrzędnych oznaczająca początek wektora tworzącego prostą,
- b - krotka współrzędnych oznaczająca koniec wektora tworzącego prostą,
- n - liczba generowanych punktów.

Zwraca: tablicę punktów w postaci krotek współrzędnych.

2.5.4. Funkcja *generate_rectangle_points*

```
def generate_rectangle_points(
    a: Any = (-10, -10),
    b: Any = (10, -10),
    c: Any = (10, 10),
    d: Any = (-10, 10),
    n: int = 100
) -> list[tuple]
```

Funkcja generuje n punktów na obwodzie prostokąta o wierzchołkach w punktach a, b, c i d.

Parametry:
- a - lewy-dolny wierzchołek prostokąta,
- b - prawy-dolny wierzchołek prostokąta,
- c - prawy-górny wierzchołek prostokąta,
- d - lewy-górny wierzchołek prostokąta,
- n - liczba generowanych punktów.

Zwraca: tablicę punktów w postaci krotek współrzędnych.

2.5.5. Funkcja *generate_square_points*

```
def generate_square_points(
    a: Any = (0, 0),
    b: Any = (10, 0),
    c: Any = (10, 10),
    d: Any = (0, 10),
    axis_n: int = 25,
    diag_n: int = 20
) -> list[tuple]
```

Funkcja generuje axis_n punktów na dwóch bokach kwadratu leżących na osiach x i y oraz diag_n punktów na przekątnych kwadratu, którego wyznaczają punkty a, b, c i d.

Parametry:
- a - lewy-dolny wierzchołek kwadratu,
- b - prawy-dolny wierzchołek kwadratu,
- c - prawy-górny wierzchołek kwadratu,
- d - lewy-górny wierzchołek kwadratu,
- axis_n - liczba generowanych punktów na każdym z dwóch boków kwadratu równoległych do osi x i y,
- diag_n - liczba generowanych punktów na każdej przekątnej kwadratu.

Zwraca: tablicę punktów w postaci krotek współrzędnych.

2.5.6. Funkcja *generate_grid_points*

```
def generate_grid_points(n: int = 100) -> list[tuple[int, int]]
```

Funkcja generuje punkty na siatce $n \times n$.

Parametry:
- n - liczba punktów wzdłuż jednej osi.

Zwraca: tablicę punktów w postaci krotek współrzędnych.

2.5.7. Funkcja *generate_clustered_points*

```
def generate_clustered_points(
    cluster_centers: Any,
    cluster_std: Any,
    points_per_cluster: Any
) -> list[tuple]
```

Funkcja generuje punkty w klastrach wokół podanych centrów.

Parametry:
- cluster_centers - lista krotek współrzędnych centrów klastrów,
- cluster_std - odchylenie standardowe dla każdego klastra,
- points_per_cluster - liczba punktów w każdym klastrze.

Zwraca: tablicę punktów w postaci krotek współrzędnych.

2.6. Moduł *automatic_tests*

Służy do automatycznego testowania dwóch struktur danych: KD-drzewa i QuadTree. Testy mają na celu sprawdzenie, czy obie struktury danych zwracają zgodne wyniki dla różnych zestawów punktów i zapytań zakresowych.

2.6.1. Funkcja *runtests_all*

```
def runtests_all() -> None
```

Funkcja generuje zbiory testowe oraz porównuje wyniki zapytań dla obu struktur danych.

Jeśli wyniki są różne, wypisuje komunikat o błędzie i kończy działanie.

Jeśli wszystkie testy przejdą pomyślnie, wypisuje komunikat o zaliczeniu testów.

2.7. Moduł *gui_creator*

Zapewnia GUI do zadawania punktów i definiowania prostokątnego obszaru na płaszczyźnie 2D. Wykorzystuje biblioteki *matplotlib* oraz *tkinter*.

2.7.1. Funkcja create_gui

```
def create_gui() -> (tuple[list, Any, Any] | None)
```

Tworzy GUI do zbierania punktów i definiowania prostokątnego obszaru na płaszczyźnie 2D.

Zwraca: Krotkę zawierającą:
- points (lista krotek) - lista współrzędnych (x, y) punktów,
- lower_left (krotka) - współrzędne (x, y) dolnego lewego rogu prostokąta,
- upper_right (krotka) - współrzędne (x, y) górnego prawego rogu prostokąta.

2.8. Plik *main.ipynb*

Plik Jupyter Notebook stworzony, aby zapewnić wygodny interfejs do prezentacji oraz wizualizacji funkcjonowania KD-tree i QuadTree. Jest to także narzędzie do przetestowania poprawności implementacji powyższych struktur oraz porównania ich efektywności dla poszczególnych zbiorów danych.

3. Część użytkownika

W tej części pokazane zostaną przykłady uruchamiania programu oraz korzystania z jego poszczególnych modułów.

3.1. Pakiet *kdtree*

3.1.1. Moduł *kdtree*

Stanowi implementację KD-drzewa.

3.1.1.1. Inicjalizacja struktury danych

```
from kdtree.kdtree import KDtree

points_set = [(0, 0), (20, 10), (20, 70), (60, 10), (60, 40), (70, 80), (75, 90), (80, 85),
(80, 80), (80, 83)]

kdtree = KDtree(points_set, k = 2, eps = 1e-12)
```

W powyższym przykładzie dokonano importu struktury *KDtree* z odpowiedniego pakietu.
Następnie zainicjalizowano drzewo z wykorzystaniem konstruktora. Zostały przekazane do niego:
- points_set - lista punktów,
- k - liczba wymiarów (domyślnie 2),
- eps - tolerancja dla zera (opcjonalne).

3.1.1.2. Zapytanie o punkty z zadanego obszaru

```
lower_left = (20, 10)
upper_right = (90, 80)

result = kdtree.query(lower_left, upper_right)

print(result)
```

Wyjście:

```
[(20, 10), (60, 10), (20, 70), (60, 40), (70, 80), (80, 80)]
```

Do metody *query* zostały przekazane:
- lower_left - punkt w postaci krotki, który reprezentuje dolny zakres regionu zapytania przestrzennego,
- upper_right - punkt w postaci krotki, który reprezentuje górny zakres regionu zapytania przestrzennego.

Program wypisał listę wszystkich punktów, które zostały znalezione w zdefiniowanym obszarze.

3.1.2. Moduł *kdtree_visualizer*

Stanowi implementację KD-drzewa, wzbogaconą o możliwość wizualizacji poszczególnych kroków

3.1.2.1. Inicjalizacja struktury danych

```
from kdtree.kdtree_visualizer import KDtreeVisualizer

points_set = [(0, 0), (20, 10), (20, 70), (60, 10), (60, 40), (70, 80), (75, 90), (80, 85),
(80, 80), (80, 83)]
kdtree = KDtreeVisualizer(points_set, eps = 1e-12)
```

W powyższym przykładzie dokonano importu struktury *KDtreeVisualizer* z odpowiedniego pakietu. Następnie zainicjalizowano drzewo z wykorzystaniem konstruktora. Zostały przekazane do niego:
- points_set - lista punktów,
- eps - tolerancja dla zera (opcjonalne).

Brak możliwości zdefiniowania liczby wymiarów (wizualizacja możliwa tylko dla dwuwymiarowej przestrzeni).

3.1.2.2. Wizualizacja procesu budowania KD-drzewa

```
kdtree.show_build_visualization(interval = 400)
```

Funkcja zwraca plik *gif* będący animacją procesu budowania KD-drzewa. Parametr *interval* odpowiada za czas interwału pomiędzy kolejnymi klatkami animacji (domyślnie 400 ms).

Oznaczenia kolorystyczne na wizualizacji:
- kolorem niebieskim oznaczono punkty z przestrzeni,
- kolorem szarym oznaczany jest aktualnie rozpatrywany obszar (do podziału),
- odcinkami pomarańczowymi oznaczono pionowe linie podziału,
- odcinkami zielonymi oznaczono poziome linie podziału.

3.1.2.3. Zapytanie o punkty z zadanego obszaru

```
lower_left = (20, 10)
upper_right = (90, 80)

result = kdtree.query(lower_left, upper_right)

print(result)
```

Wyjście:

```
[(20, 10), (60, 10), (20, 70), (60, 40), (70, 80), (80, 80)]
```

Do metody *query* zostały przekazane:
- lower_left - punkt w postaci krotki, który reprezentuje dolny zakres regionu zapytania przestrzennego,
- upper_right - punkt w postaci krotki, który reprezentuje górny zakres regionu zapytania przestrzennego.

3.1.2.4. Wizualizacja procesu zapytania

```
kdtree.show_query_visualization(interval = 600)
```

Funkcja zwraca plik *gif* będący animacją procesu zapytania do KD-drzewa. Parametr *interval* odpowiada za czas interwału pomiędzy kolejnymi klatkami animacji (domyślnie 600 ms).

Oznaczenia kolorystyczne:
- kolorem niebieskim oznaczono punkty,
- kolorem ciemnym niebieskim (półprzezroczystym) zaznaczono obszar, z którego punkty chcemy znaleźć,
- na różowo zaznaczono odcinki, które dzielą płaszczyznę
- na szaro zaznaczany jest obszar, który zostanie zawężony w celu dalszego wyszukiwania,
- kolorem zielonym oznaczane są punkty wyszukane w trakcie zapytania oraz obszary, w których te punkty się znajdowały (w przypadku gdy obszar w całości znajduje się w obszarze, z którego chcemy znaleźć punkty, to wszystkie punkty z niego są kolorowane na zielono),
- kolorem pomarańczowym oznaczane są odcinki, które pokazują w jaki sposób w trakcie wyszukiwania rozpatrywane są i dzielone poszczególne obszary (czy algorytm rozpatruje stronę „lewą” czy „prawą”).

3.2. Pakiet *quadtree*

Pakiet quadtree zawiera implementację drzewa ćwiartek.

3.2.1. Inicjalizacja struktury danych

```
from quadtree.quad import Quad
points_set = [(0, 0), (20, 10), (20, 70), (60, 10), (60, 40), (70, 80), (75, 90), (80, 85),
(80, 80), (80, 83)]
quad_tree = Quad(points_set)
```

Powyższy kod importuje pakiet `Quad` i inicjalizuje strukturę danych przekazując do niej listę punktów `points_set`.

3.2.2. Wizualizacja budowy struktury danych

```
from visualizer.main import Visualizer
vis = Visualizer()
quad_tree.draw(vis, 'blue')
vis.show_gif()
```

Powyższy kod uruchamia animację budowy drzewa ćwiartek. Importowany jest pakiet `visualizer`, a następnie generowana jest animacja.W animacji użyto następujących oznaczeń:

- kolorem zielonym oznaczono zadane punkty,
- kolorem niebieskim oznaczono prostokąty reprezentujące węzły drzewa.

3.2.3. Zapytanie o punkty z obszaru ortogonalnego

```
lower_left = (20, 10)
upper_right = (90, 80)
print(quad_tree.query_range(lower_left, upper_right))
```

Wyjście:

```
{(60, 40), (80, 80), (60, 10), (20, 70), (70, 80), (20, 10)}
```

Do metody *query_range* zostały przekazane dwa punkty w postaci krotek współrzędznych; lower_left - punkt minimalny i upper_right - punkt maksymalny.

Program wypisał zbiór wszystkich znalezionych punktów.

3.2.4. Wizualizacja zapytania

```
vis.clear()
quad_tree.graphic_query_range(lower_left, upper_right)
vis.show_gif)
```

Powyższy kod uruchamia animację wyszukiwania obszaru ortogonalnego w drzewie. W animacji użyto następujących oznaczeń:

- kolorem zielonym oznaczone są punkty przechowywane w drzewie,
- kolorem czerwonym oznaczony jest zadany prostokąt,
- kolorem niebieskim oznaczone są prostokąty, które przecinają zadany prostokąt oraz wykryte punkty, które należą do zadanego prostokąta.

3.3. Plik *main.ipynb*

Plik Jupyter Notebook został stworzony, aby zapewnić wygodny interfejs do prezentacji oraz wizualizacji funkcjonowania KD-drzewa i QuadTree. Jest to także narzędzie do przetestowania poprawności implementacji powyższych struktur oraz porównania ich efektywności dla poszczególnych zbiorów danych.

Wykonywane są w nim kolejno:

1. Importowanie bibliotek i modułów.
   - Importowanie niezbędnych bibliotek oraz modułów do obsługi KD-tree i QuadTree, testów, wizualizacji oraz generowania danych.

2. Testy poprawności implementacji:
   1. Uruchomienie testów jednostkowych dla KD-tree.
   2. Uruchomienie testów integralnościowych dla KD-tree i QuadTree.
3. Graficzne zadawanie punktów oraz obszaru wyszukiwania.
   - Interaktywne narzędzie do zadawania punktów na płaszczyźnie oraz obszaru wyszukiwania.
4. Wizualizacja punktów i obszaru wyszukiwania.
   - Wizualizacja zadanych punktów oraz obszaru wyszukiwania.
5. Wizualizacja budowania KD-tree i QuadTree.
6. Zapytania o punkty w zadanym obszarze.
   - Wykonanie zapytania o punkty w zadanym obszarze i porównanie wyników.
7. Wizualizacja wyszukiwania punktów:
   - Wizualizacja procesu wyszukiwania punktów w KD-tree.
   - Wizualizacja procesu wyszukiwania punktów w QuadTree.
8. Ostateczny wynik zapytania:
   - Wizualizacja ostatecznego wyniku zapytania.
9. Generowanie zbiorów punktów o różnych charakterystykach i ich wizualizacja.
10. Testy wydajnościowe.
    - Porównanie wydajności między implementacjami KD-tree oraz QuadTree.

4. Sprawozdanie

4.1. Dane techniczne

- System operacyjny: Windows 10 22H2 (x86-64)
- Procesor: AMD Ryzen 5 1600AF (3.20 - 3.60 GHz)
- Pamięć RAM: 16GB (3400 MHz CL14)
- Środowisko: Jupyter Notebook
- Język: Python 3.13.1

Użyta precyzja przechowywania zmiennych i obliczeń w testach to *float64*.

4.2. Opis problemu

Efektywne przeszukiwanie przestrzenne stanowi istotne zagadnienie w wielu dziedzinach informatyki, od systemów GIS po grafikę komputerową. W niniejszym sprawozdaniu analizujemy wydajność dwóch struktur danych służących do partycjonowania przestrzeni: KD-drzewa oraz drzewa czwórkowego (QuadTree). Ograniczamy się do przestrzeni dwuwymiarowej.

Struktury te umożliwiają szybkie wyszukiwanie punktów w zadanym obszarze poprzez rekurencyjny podział przestrzeni. KD-drzewo dzieli przestrzeń naprzemiennie wzdłuż kolejnych wymiarów, podczas gdy QuadTree dzieli obszar na cztery równe części. W przypadku zrównoważonego KD-drzewa, złożoność czasowa budowania wynosi $O(n \log n)$, a wyszukiwanie punktów w zadanym obszarze zajmuje $O(\sqrt{n} + d)$, gdzie d to liczba znalezionych punktów. Dla drzewa czwórkowego, złożoność budowania również wynosi $O(n \log n)$, jednak w najgorszym przypadku może zdegenerować się do $O(n^2)$. Wyszukiwanie w QuadTree ma złożoność $O(\log n + d)$ dla równomiernie rozłożonych danych.

Badanie porównawcze koncentruje się na analizie czasu konstrukcji struktury oraz wydajności zapytań przestrzennych dla różnych rozkładów danych wejściowych.

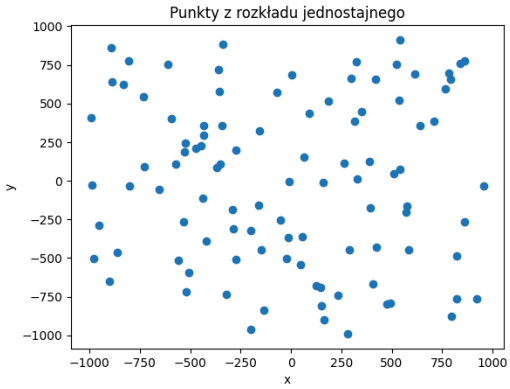
4.3. Realizacja

W celu porównania obu struktur danych wygenerowano punkty z dwuwymiarowej przestrzeni o różnych charakterystykach. Dla każdego zbioru zostanie zaprezentowana wizualizacja (o niewielkiej liczności zbioru punktów), graficzna reprezentacja wyniku zapytania oraz porównanie czasowe. Animacje pokazujące krok po kroku funkcjonowanie algorytmów zostały zamieszczone w pliku *Jupyter*.

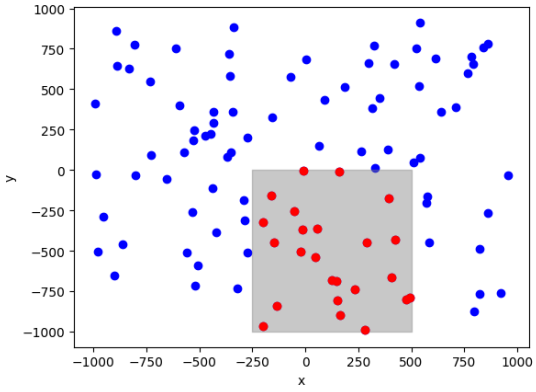
4.4. Wyniki

Dla każdego ze zbiorów testowych, wyniki zapytania zwrócone przez obie struktury danych były tożsame, co stanowi dowód poprawności implementacji.

4.4.1. Zbiór punktów z rozkładu jednostajnego



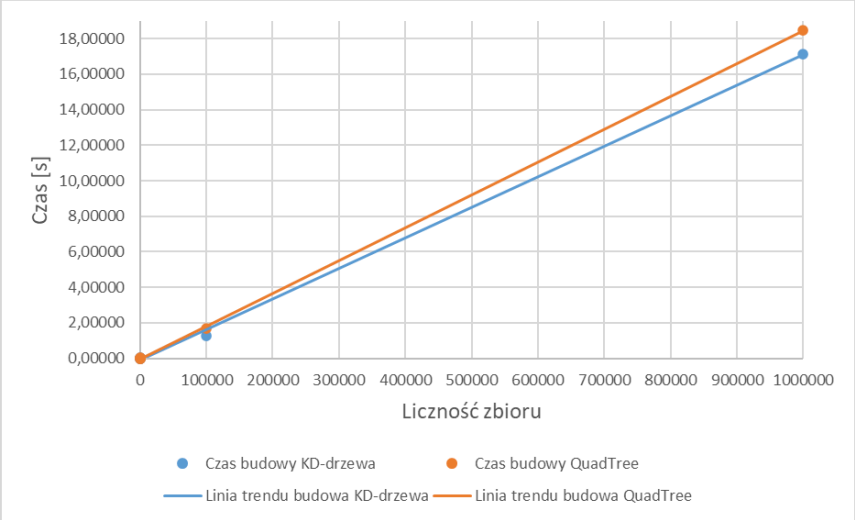
Rysunek 1: Przykładowy zbiór 100 punktów z rozkładu jednostajnego



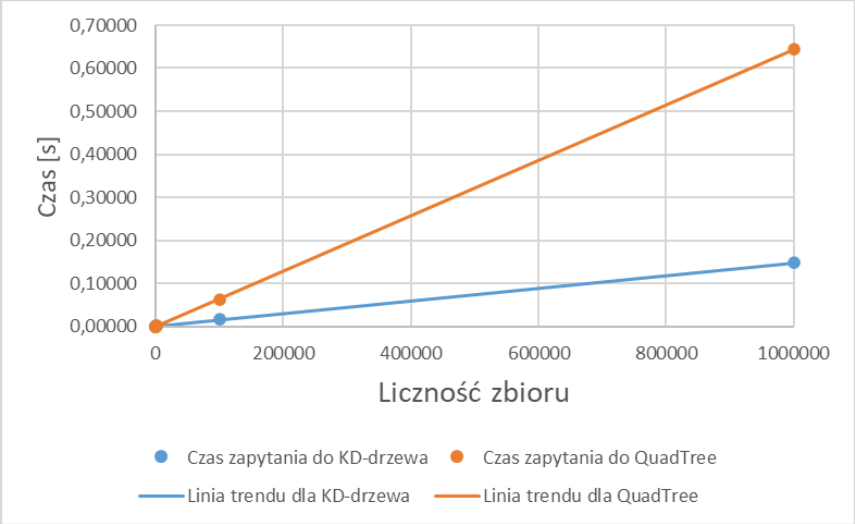
Rysunek 2: Wynik przykładowego zapytania dla zbioru jednostajnego

Liczba punktów		**Czas** [s]			
Zbioru	**Znalezionych**	**Budowy KD-drzewa**	**Budowy QuadTree**	**Zapytania do KD-drzewa**	**Zapytania do QuadTree**
50	14	0.00052	0.00031	0.00022	0.00008
100	26	0.00057	0.00060	0.00034	0.00012
500	101	0.00290	0.00328	0.00072	0.00044
1000	155	0.00642	0.00676	0.00074	0.00055
100000	18912	1.25	1.68	0.02	0.06
1000000	188010	17.13	18.48	0.15	0.64

Tabela 1: Porównanie czasowe dla różnych liczności punktów z rozkładu jednostajnego



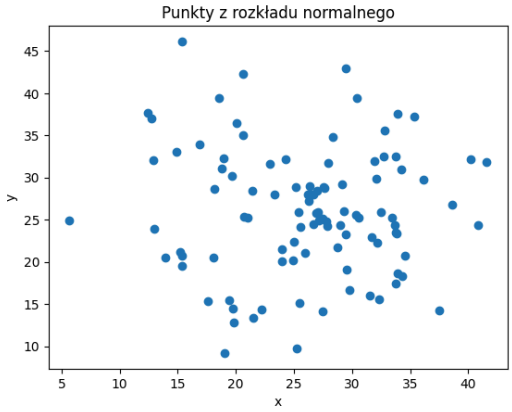
Rysunek 3: Wykres porównawczy czasu budowania struktur od liczby punktów dla zbioru o rozkładzie jednostajnym



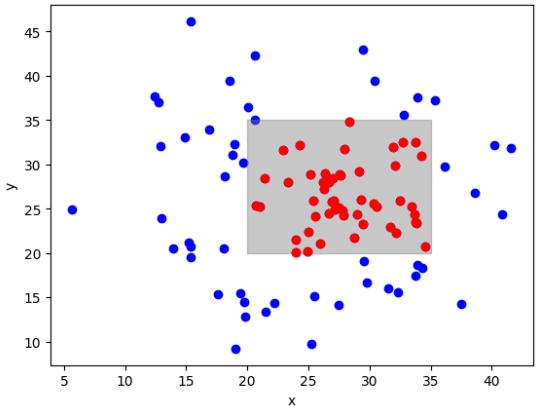
Rysunek 4: Wykres porównawczy czasu zapytania do struktur od liczby punktów dla zbioru o rozkładzie jednostajnym

Każdy z wygenerowanych zbiorów testowych wygląda podobnie do tego przedstawionego na Rysunku 1, różnicą jest liczność. Testy przeprowadzono dla obszaru zaznaczonego kolorem szarym na Rysunku 2. Jak można zauważyć w Tabeli 1, czas budowy KD-drzewa jest w większości przypadków mniejszy, niż dla QuadTree (co potwierdza Rysunek 3). Dla niewielkiej liczności zbiorów, zapytania były wykonywane szybciej przez QuadTree, KD-drzewo okazało się szybsze dla większej liczności. Zależność czasu wykonania zapytania została przedstawiona na Rysunku 4.

4.4.2. Zbiór punktów z rozkładu normalnego



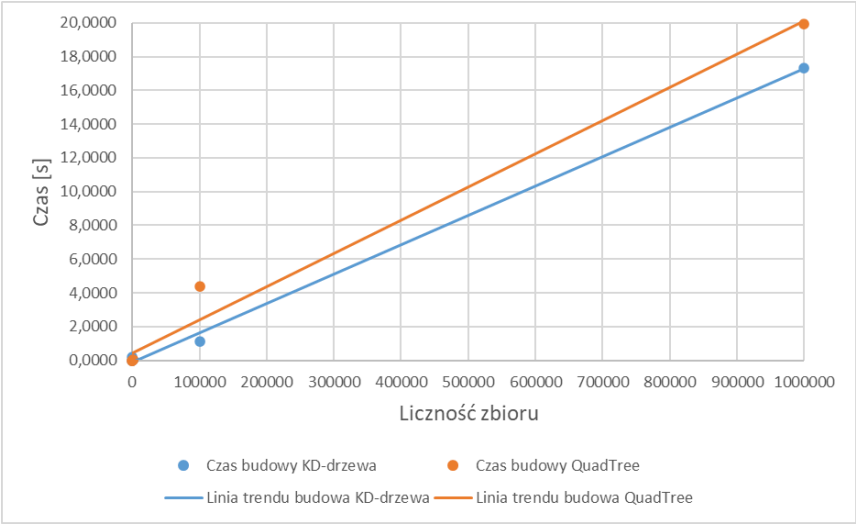
Rysunek 5: Przykładowy zbiór 100 punktów z rozkładu normalnego



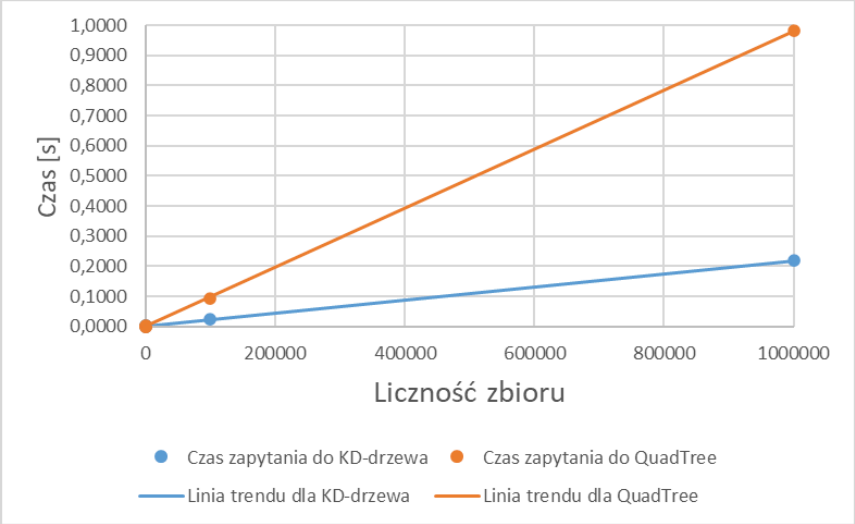
Rysunek 6: Wynik przykładowego zapytania dla zbioru normalnego

Liczba punktów		**Czas [s]**			
Zbioru	**Znalezionych**	**Budowy KD-drzewa**	**Budowy QuadTree**	**Zapytania do KD-drzewa**	**Zapytania do QuadTree**
50	9	0.1955	0.0003	0.0021	0.0040
100	31	0.0005	0.0006	0.0003	0.0001
500	130	0.0035	0.0032	0.0007	0.0005
1000	279	0.0064	0.0086	0.0012	0.0010
100000	28386	1.09	4.39	0.03	0.09
1000000	284255	17.33	19.93	0.22	0.98

Tabela 2: Porównanie czasowe dla różnych liczności punktów z rozkładu normalnego



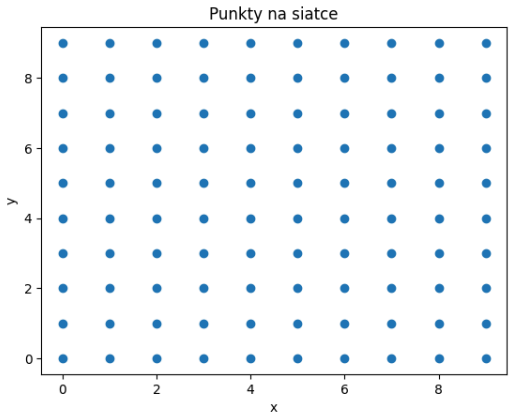
Rysunek 7: Wykres porównawczy czasu budowania struktur od liczby punktów dla zbioru o rozkładzie normalnym



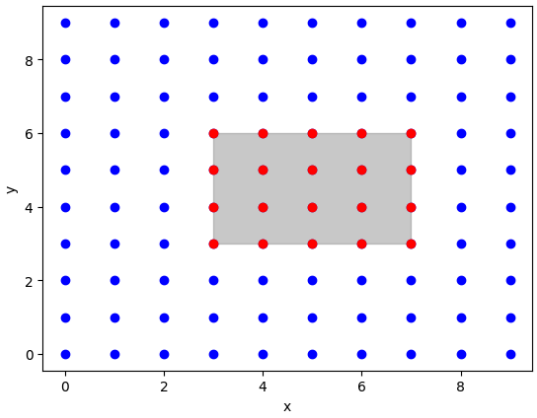
Rysunek 8: Wykres porównawczy czasu zapytania do struktur od liczby punktów dla zbioru o rozkładzie normalnym

Dla zbioru widocznego na Rysunku 5, wyniki dla zapytania przedstawionego na Rysunku 6 (dla różnych liczności) zostały umieszczone w Tabeli 2. Dla zbioru o liczbie punktów wynoszącej 50, czas budowy QuadTree jest znacząco mniejszy od czasu dla KD-drzewa. Druga struktura natomiast realizuje zapytanie niemal 2 razy szybciej. Dla liczby punktów od 100 do 1000 czasy budowy są podobne, by dla zbiorów o większej liczności KD-drzewo okazało się w szybsze (zależność widoczna na Rysunku 7). Dla przypadków od 100 do 1000 punktów, QuadTree wykonuje zapytania szybciej. Dla zbioru 50 punktów oraz powyżej 100000, KD-drzewo jest kolejno dwukrotnie i trzykrotnie szybsze (co można zauważyć na Rysunku 8).

4.4.3. Zbiór punktów na siatce



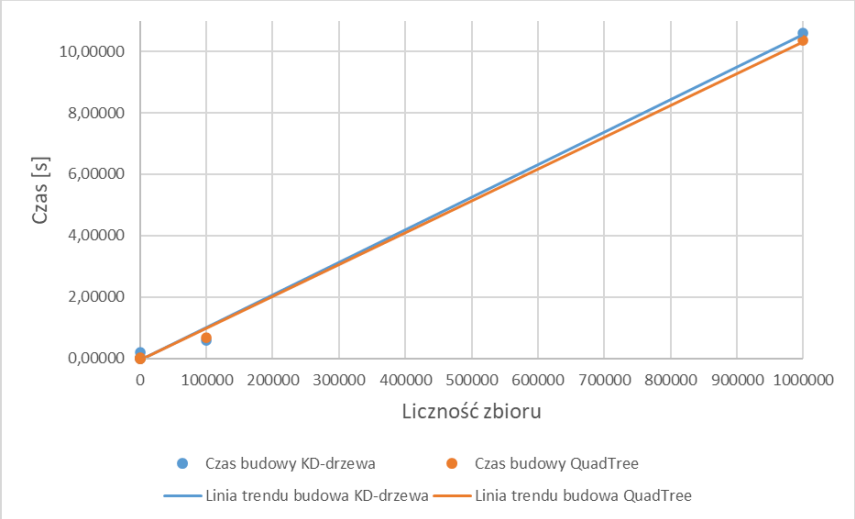
Rysunek 9: Przykładowy zbiór 100 punktów na siatce



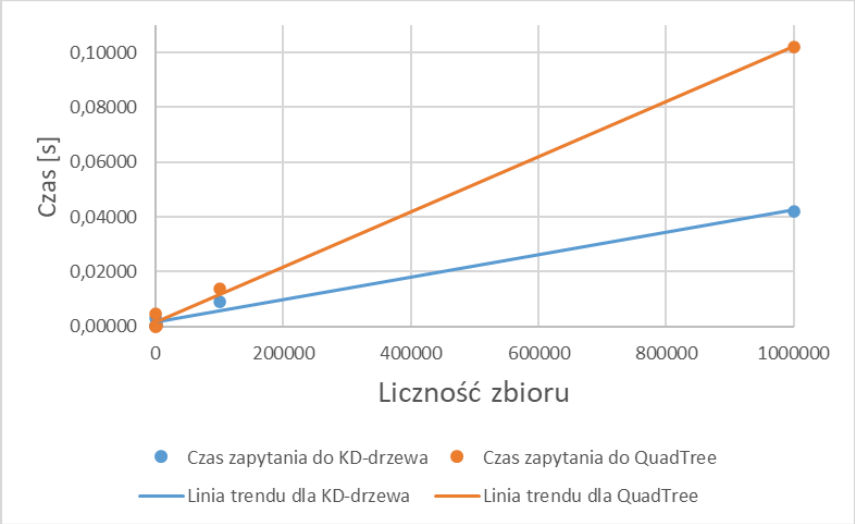
Rysunek 10: Wynik przykładowego zapytania dla zbioru punktów na siatce

Liczba punktów		Czas [s]			
Zbioru	**Znalezionych**	**Budowy KD-drzewa**	**Budowy QuadTree**	**Zapytania do KD-drzewa**	**Zapytania do QuadTree**
50	4	0.19232	0.00020	0.00323	0.00484
100	9	0.00040	0.00039	0.00014	0.00003
500	36	0.00202	0.00482	0.00038	0.00008
1000	64	0.00411	0.00349	0.00048	0.00011
100000	6400	0.578	0.665	0.009	0.014
1000000	63001	10.602	10.356	0.042	0.102

Tabela 3: Porównanie czasowe dla różnych liczności punktów z siatki



Rysunek 11: Wykres porównawczy czasu budowania struktur od liczby punktów dla zbioru punktów z siatki

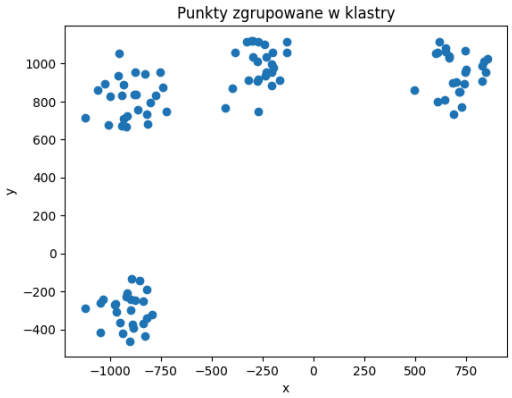


Rysunek 12: Wykres porównawczy czasu zapytania do struktur od liczby punktów dla zbioru punktów z siatki

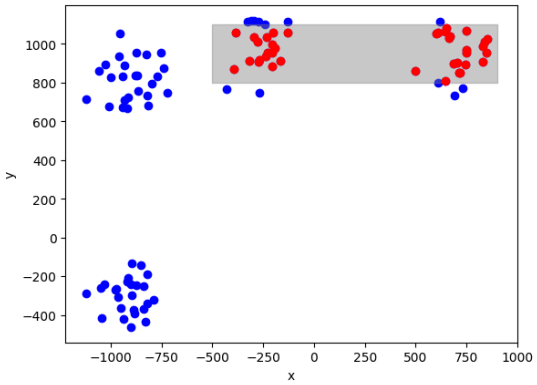
Na Rysunku 9 zaprezentowano przykładowy zbiór 100 punktów równomiernie rozłożonych na siatce, a Rysunek 10 pokazuje wynik przykładowego zapytania. Szczegółowe porównanie czasów wykonania operacji dla różnych rozmiarów zbiorów przedstawiono w Tabeli 3. Dane te zobrazowano również na

dwóch wykresach porównawczych (Rysunek 11 i 12). Na podstawie tych wyników można zaobserwować, że proces budowy obu struktur charakteryzuje się bardzo podobną wydajnością - czasy budowy KD--drzewa i QuadTree są niemal identyczne. Dla największego testowanego zbioru, zawierającego milion punktów, czas budowy wynosi około 10.6 sekundy dla KD-drzewa i 10.4 sekundy dla QuadTree. Znaczące różnice ujawniają się jednak w czasie wykonywania zapytań przestrzennych. QuadTree wykazuje gorszą wydajność przy rosnącej liczbie punktów - dla zbioru miliona punktów czas zapytania wynosi 0.102 sekundy, podczas gdy KD-drzewo wykonuje tę samą operację w 0.042 sekundy. Warto zauważyć, że dla mniejszych zbiorów (do 1000 punktów) różnice w wydajności są minimalne - obie struktury osiągają czasy rzędu tysięcznych części sekundy. Jednak wraz ze wzrostem rozmiaru zbioru różnica w wydajności staje się coraz bardziej wyraźna, na korzyść KD-drzewa.

4.4.4. Zbiór punktów zgrupowanych w klastry



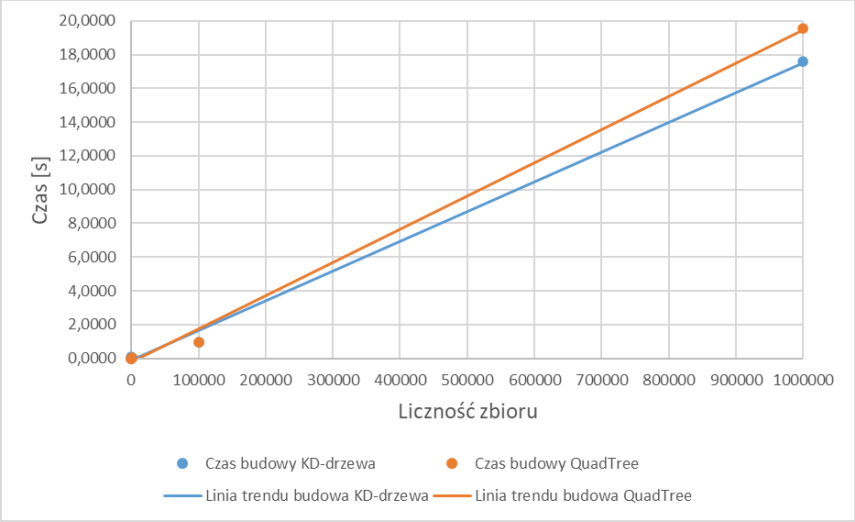
Rysunek 13: Przykładowy zbiór 100 punktów zgrupowanych w klastry



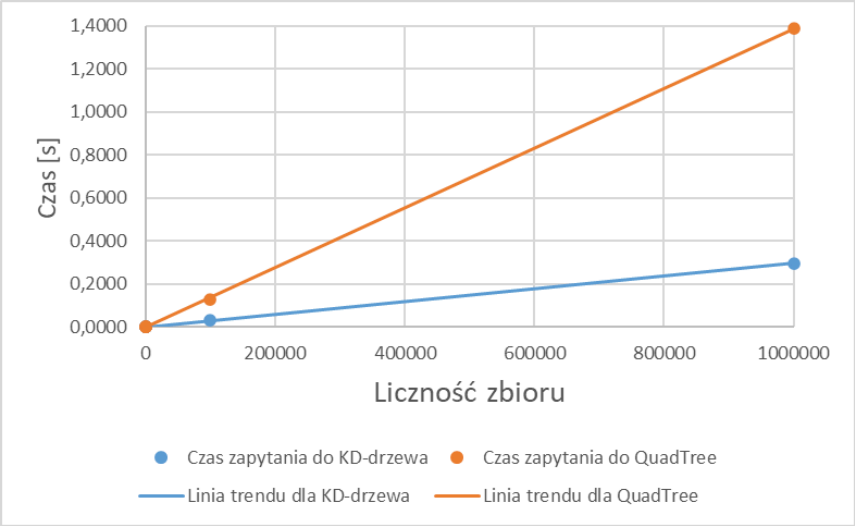
Rysunek 14: Wynik przykładowego zapytania dla zbioru punktów zgrupowanych w klastry

Liczba punktów		**Czas [s]**			
Zbioru	**Znalezionych**	**Budowy KD-drzewa**	**Budowy QuadTree**	**Zapytania do KD-drzewa**	**Zapytania do QuadTree**
50	17	0.1026	0.0004	0.0006	0.0008
100	41	0.0007	0.0007	0.0003	0.0002
500	196	0.0035	0.0039	0.0009	0.0008
1000	410	0.0069	0.0098	0.0014	0.0015
100000	40623	0.97	0.95	0.03	0.13
1000000	405720	17.57	19.55	0.30	1.39

Tabela 4: Porównanie czasowe dla różnych liczności punktów zgrupowanych w klastry



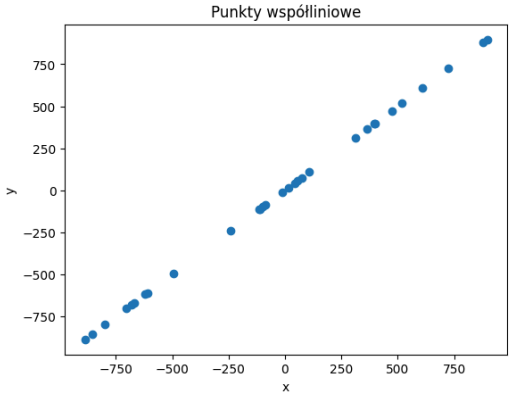
Rysunek 15: Wykres porównawczy czasu budowania struktur od liczby punktów dla zbioru punktów zgrupowanych w klastry



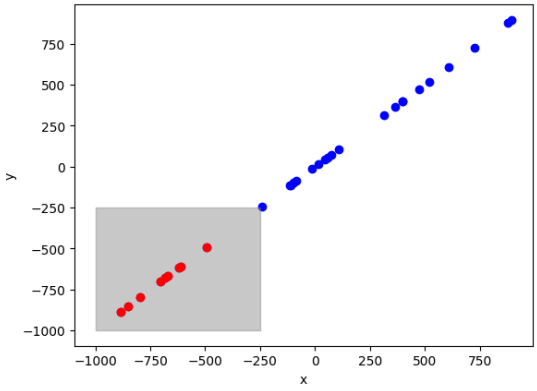
Rysunek 16: Wykres porównawczy czasu zapytania do struktur od liczby punktów dla zbioru punktów zgrupowanych w klastry

Na Rysunku 13 zaprezentowano przykładowy zbiór 100 punktów pogrupowanych w 4 klastry. Rysunek 14 pokazuje wynik przykładowego zapytania. Szczegółowe porównanie czasów wykonania operacji dla różnych rozmiarów zbiorów przedstawiono w Tabeli 4. Dane te zobrazowano również na dwóch wykresach porównawczych (Rysunek 15 i 16). W tym przypadku czas budowy struktury dla obu algorytmów jest bardzo podobny. Znacząco różnią się czasy przeszukiwania drzewa, gdzie KD-drzewo jest znacznie wydajniejsze. Wynika to z rozłożenia punktów, które sprawia, że drzewo ćwiartek ma dużą głębokość i jest silnie niezrównoważone. Gdyby klastry były umieszczone w oddzielnych ćwiartkach, wtedy algorytm dla drzewa ćwiartek osiągnąłby lepsze wyniki.

4.4.5. Zbiór punktów wygenerowanych na prostej



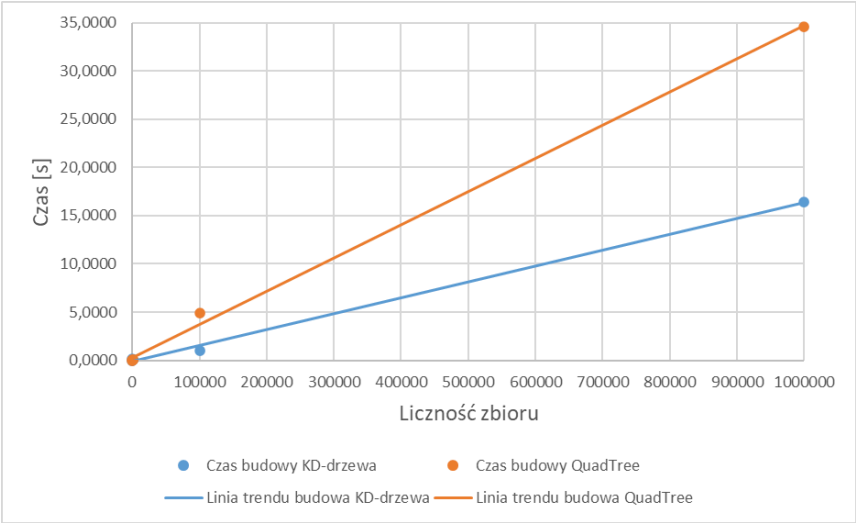
Rysunek 17: Przykładowy zbiór 100 punktów współliniowych



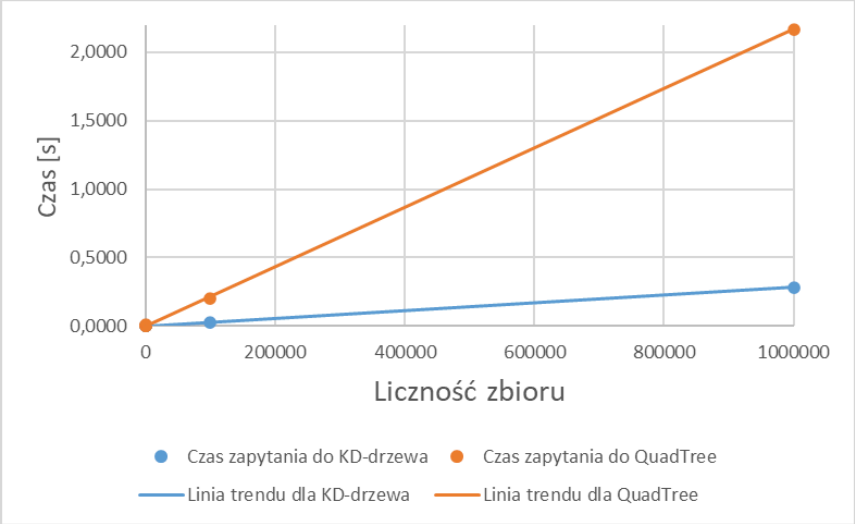
Rysunek 18: Wynik przykładowego zapytania dla zbioru punktów współliniowych

Liczba punktów		**Czas [s]**			
Zbioru	**Znalezionych**	**Budowy KD-drzewa**	**Budowy QuadTree**	**Zapytania do KD-drzewa**	**Zapytania do QuadTree**
50	20	0.1924	0.0005	0.0040	0.0082
100	42	0.0005	0.0031	0.0003	0.0002
500	187	0.0029	0.0059	0.0007	0.0010
1000	342	0.0063	0.0140	0.0010	0.0018
100000	37482	1.05	4.86	0.03	0.21
1000000	374479	16.44	34.64	0.28	2.17

Tabela 5: Porównanie czasowe dla różnych liczności punktów współliniowych



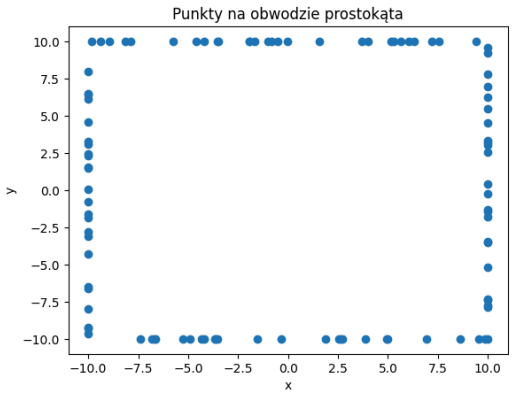
Rysunek 19: Wykres porównawczy czasu budowania struktur od liczby punktów dla zbioru punktów współliniowych



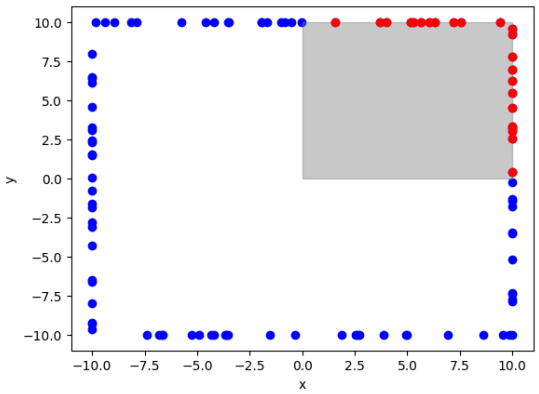
Rysunek 20: Wykres porównawczy czasu zapytania do struktur od liczby punktów dla zbioru punktów współliniowych

Na Rysunku 17 zaprezentowano przykładowy zbiór 100 współliniowych punktów. Rysunek 18 pokazuje wynik przykładowego zapytania. Szczegółowe porównanie czasów wykonania operacji dla różnych rozmiarów zbiorów przedstawiono w Tabeli 5. Dane te zobrazowano również na dwóch wykresach porównawczych (Rysunek 19 i 20). Dla punktów współliniowych budowa i wyszukiwanie punktów KD-drzewa są znacznie krótsze, ponieważ KD-drzewo jest w tym przypadku znacznie płytsze i bardziej zrównoważone.

4.4.6. Zbiór punktów na obwodzie prostokąta



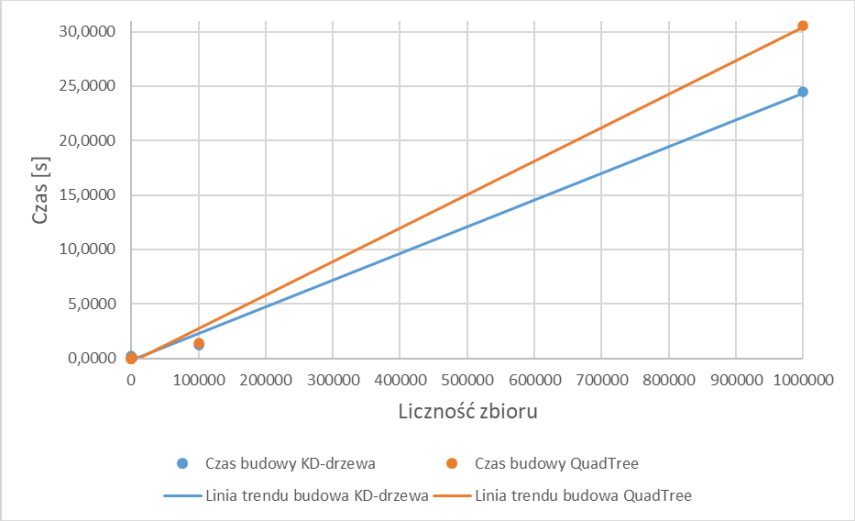
Rysunek 21: Przykładowy zbiór 100 punktów na obwodzie prostokąta



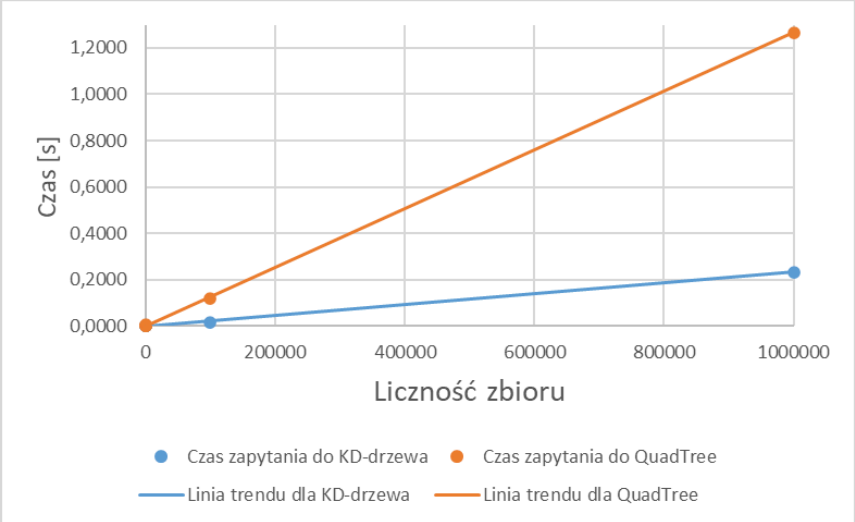
Rysunek 22: Wynik przykładowego zapytania dla zbioru punktów na obwodzie prostokąta

Liczba punktów		Czas [s]			
Zbioru	**Znalezionych**	**Budowy KD-drzewa**	**Budowy QuadTree**	**Zapytania do KD-drzewa**	**Zapytania do QuadTree**
50	9	0.1908	0.0004	0.0039	0.0084
100	25	0.0006	0.0011	0.0002	0.0001
500	120	0.0040	0.0067	0.0006	0.0006
1000	252	0.0082	0.0100	0.0004	0.0012
100000	25059	1.26	1.38	0.02	0.12
1000000	250451	24.47	30.53	0.23	1.27

Tabela 6: Porównanie czasowe dla różnych liczności punktów na obwodzie prostokąta



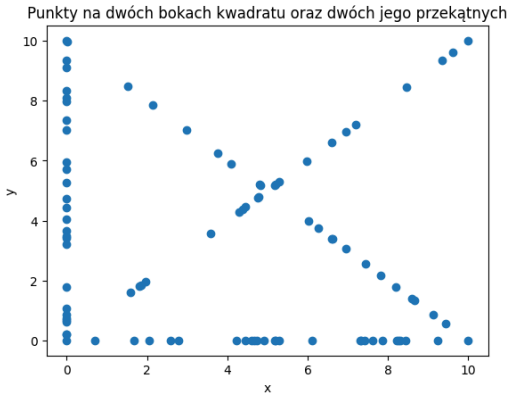
Rysunek 23: Wykres porównawczy czasu budowania struktur od liczby punktów na obwodzie prostokąta



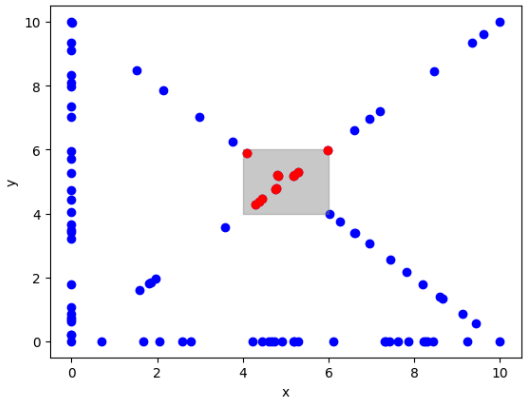
Rysunek 24: Wykres porównawczy czasu zapytania do struktur od liczby punktów na obwodzie prostokąta

Na Rysunku 21 zaprezentowano przykładowy zbiór 100 punktów na obwodzie. Rysunek 22 pokazuje wynik przykładowego zapytania. Szczegółowe porównanie czasów wykonania operacji dla różnych rozmiarów zbiorów przedstawiono w Tabeli 6. Dane te zobrazowano również na dwóch wykresach porównawczych (Rysunek 23 i 24). Czas budowania struktur jest podobny, natomiast operacja przeszukiwania jest znacznie wolniejsza dla QuadTree (nawet sześciokrotnie).

4.4.7. Zbiór punktów na dwóch bokach kwadratu oraz dwóch jego przekątnych



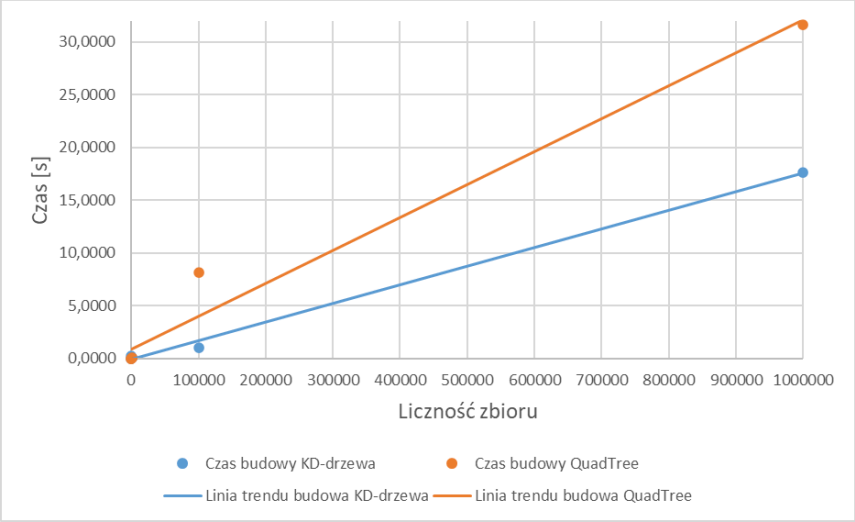
Rysunek 25: Przykładowy zbiór 100 punktów na dwóch bokach kwadratu oraz dwóch jego przekątnych



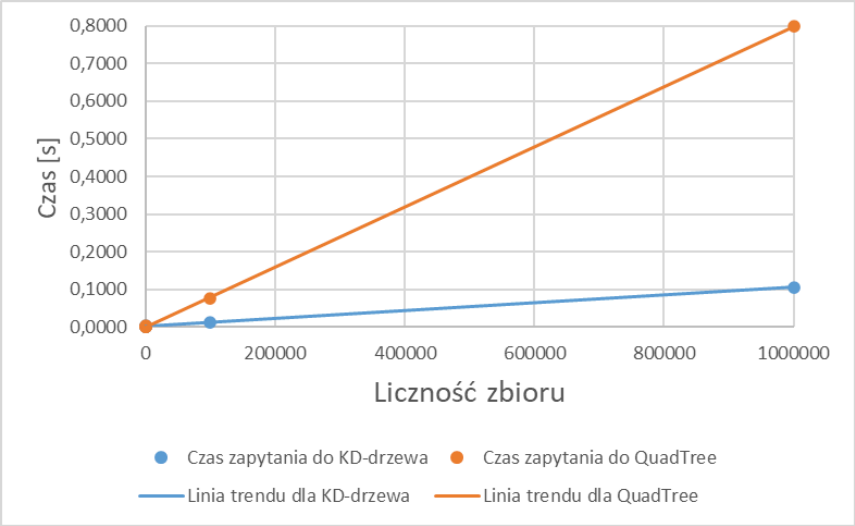
Rysunek 26: Wynik przykładowego zapytania dla zbioru punktów na dwóch bokach kwadratu oraz dwóch jego przekątnych

Liczba punktów		**Czas [s]**			
Zbioru	**Znalezionych**	**Budowy KD-drzewa**	**Budowy QuadTree**	**Zapytania do KD-drzewa**	**Zapytania do QuadTree**
50	4	0.2551	0.0004	0.0027	0.0045
100	18	0.0006	0.0010	0.0001	0.0001
500	76	0.0031	0.0049	0.0003	0.0004
1000	145	0.0068	0.0116	0.0005	0.0007
100000	15989	1.04	8.13	0.01	0.08
1000000	160160	17.63	31.67	0.11	0.80

Tabela 7: Porównanie czasowe dla różnych liczności punktów na dwóch bokach kwadratu oraz dwóch jego przekątnych



Rysunek 27: Wykres porównawczy czasu budowania struktur od liczby punktów na dwóch bokach kwadratu oraz dwóch jego przekątnych



Rysunek 28: Wykres porównawczy czasu zapytania do struktur od liczby punktów na dwóch bokach kwadratu oraz dwóch jego przekątnych

Na Rysunku 25 zaprezentowano przykładowy zbiór 100 punktów na dwóch bokach kwadratu i dwóch przekątnych. Rysunek 26 pokazuje wynik przykładowego zapytania. Szczegółowe porównanie czasów wykonania operacji dla różnych rozmiarów zbiorów przedstawiono w Tabeli 7. Dane te zobrazowano również na dwóch wykresach porównawczych (Rysunek 27 i 28). Stosunek czasu budowy struktur jest podobny do powyższych problemów, ale z racji na mniejszą liczbę punktów w zbiorze wynikowym czas przeszukiwania jest odpowiednio niższy. Dowodzi to, że oba algorytmy mają złożoność zależną od liczby punktów wyniku.

4.5. Wnioski

1. Zarówno KD-drzewo, jak i QuadTree poprawnie realizują wyszukiwanie punktów w zadanym obszarze, niezależnie od charakterystyki danych wejściowych.
2. KD-drzewo wykazuje przewagę w czasie budowy i przeszukiwania dla dużych zbiorów punktów, szczególnie gdy dane są równomiernie rozłożone lub współliniowe. Jest to efektem bardziej zrównoważonej struktury, co minimalizuje głębokość drzewa.
3. QuadTree jest bardziej wydajne w przypadku małych zbiorów punktów oraz w sytuacjach, gdy punkty są równomiernie rozłożone w przestrzeni. Wydajność tej struktury może być lepsza także dla mniej zagęszczonych zbiorów, gdzie rekurencyjny podział przestrzeni w QuadTree skutecznie ogranicza liczbę operacji w porównaniu z bardziej złożonymi układami.
4. Rozkład danych ma kluczowy wpływ na wydajność obu struktur. KD-drzewo radzi sobie lepiej w przypadku danych z rozkładów jednostajnego, normalnego i współliniowego, podczas gdy QuadTree wykazuje większy potencjał dla danych o mniejszej gęstości lub bardziej jednorodnym rozmieszczeniu w przestrzeni.
5. KD-drzewo jest bardziej uniwersalne i sprawdza się w aplikacjach wymagających obsługi dużych zbiorów punktów lub specyficznych rozkładów, podczas gdy QuadTree może być preferowane w przypadku mniejszych zbiorów, danych mniej zagęszczonych lub gdy prostota implementacji ma znaczenie.
6. Chociaż teoretyczna złożoność obu struktur jest podobna dla budowy i przeszukiwania, w praktyce KD-drzewo okazuje się bardziej stabilne wydajnościowo przy wzroście liczby punktów, podczas gdy QuadTree wykazuje większe zróżnicowanie w czasie operacji zależnie od rozmieszczenia danych i ich zagęszczenia.

Podsumowując, oba podejścia mają swoje mocne i słabe strony, a wybór odpowiedniej struktury danych powinien być dostosowany do charakterystyki danych i specyfiki zastosowania. KD-drzewo oferuje większą stabilność i przewagę w aplikacjach wymagających przetwarzania dużych i gęsto rozmieszczonych zbiorów punktów, natomiast QuadTree może być bardziej efektywne w przypadku mniejszych zbiorów, mniej zagęszczonych danych oraz tam, gdzie kluczowa jest prostota implementacji.

Bibliografia

[1] dr inż. Barbara Głut, „Algorytmy geometryczne — wykład”.

[2] *Computational Geometry*, 3. wyd. Springer Berlin, Heidelberg.