

A G H

**AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE**

Wyszukiwanie Geometryczne

Implementacja struktur QuadTree i KD-drzew
Dokumentacja i Sprawozdanie

Algorytmy geometryczne
Informatyka WI AGH, II rok

Maciej Kmąk, Michał Szymocha
Styczeń 2025

Spis treści

I Dokumentacja	6
1 Wprowadzenie	6
2 Zależności	6
3 Klasy	6
3.1 Rectangle	6
3.1.1 Atrybuty	6
3.1.2 Metody	6
3.2 Node	7
3.2.1 Atrybuty	7
3.2.2 Metody	8
3.3 KdTree	8
3.3.1 Atrybuty	8
3.3.2 Metody	9
3.4 QuadTreeNode	10
3.4.1 Atrybuty	10
3.4.2 Metody	10
3.5 QuadTree	11
3.5.1 Atrybuty	11
3.5.2 Metody	11
4 Wizualizacje	13
4.1 KdTreeVis	14
4.1.1 Atrybuty	14
4.1.2 Metody	14
4.2 QuadTreeVis	15
4.2.1 Atrybuty	15
4.2.2 Metody	15
5 Przykład Użycia	18
5.1 Rectangle	18
5.1.1 Przykład Użycia Klasy Rectangle	18
5.1.2 Wyjaśnienie Przykładu Klasy Rectangle	18
5.2 KdTree	19
5.2.1 Przykład Użycia Klasy KdTree	19
5.2.2 Wyjaśnienie Przykładu Klasy KdTree	19
5.3 KdTreeVis	19
5.3.1 Przykład Użycia Klasy KdTreeVis	19
5.3.2 Wyjaśnienie Przykładu Klasy KdTreeVis	20
5.4 QuadTree	20
5.4.1 Przykład Użycia Klasy QuadTree	20
5.4.2 Wyjaśnienie Przykładu Klasy QuadTree	21
5.5 QuadTreeVis	22
5.5.1 Przykład Użycia Klasy QuadTreeVis	22
5.5.2 Wyjaśnienie Przykładu Klasy QuadTreeVis	22

6 Obsługa Błędów	23
7 Szczegółы Zależności	24
7.1 Klasa Rectangle	24
7.2 Klasa Node	25
7.3 Klasa KdTree	25
7.4 Klasa QuadTreeNode	25
7.5 Klasa QuadTree	26
8 Funkcje Testowe	26
8.1 random_uniform_test	26
8.1.1 Parametry	26
8.1.2 Zwraca	26
8.2 random_normal_test	27
8.2.1 Parametry	27
8.2.2 Zwraca	27
8.3 random_integer_test	27
8.3.1 Parametry	27
8.3.2 Zwraca	27
8.4 grid_test	27
8.4.1 Parametry	28
8.4.2 Zwraca	28
8.5 circle_test	28
8.5.1 Parametry	28
8.5.2 Zwraca	28
8.6 line_test	28
8.6.1 Parametry	28
8.6.2 Zwraca	29
8.7 cross_test	29
8.7.1 Parametry	29
8.7.2 Zwraca	29
8.8 rectangle_sides_test	29
8.8.1 Parametry	29
8.8.2 Zwraca	30
8.9 rectangle_with_diagonals_test	30
8.9.1 Parametry	30
8.9.2 Zwraca	30
8.10 test_two_clusters	30
8.10.1 Parametry	30
8.10.2 Zwraca	31
8.11 test_with_outliers	31
8.11.1 Parametry	31
8.11.2 Zwraca	31
II Sprawozdanie	33
1 Wstęp Teoretyczny	33
1.1 Cel Projektu	33
1.2 QuadTree	33

1.3	KdTree	35
2	Opis Implementacji	36
2.1	QuadTree	36
2.2	KdTree	36
3	Porównanie wyników dla testów wydajnościowych	36
3.1	Jak czytać i interpretować wykresy?	37
3.2	Rozkład jednostajny	38
3.3	Rozkład normalny	40
3.4	Współrzędne całkowite	42
3.5	Siatka	44
3.6	Okrąg	46
3.7	Prosta	48
3.8	Krzyż	50
3.9	Boki prostokąta	52
3.10	Prostokąt i przekątne	54
3.11	Dwie chmury	56
3.12	Punkty odstające	58
4	Podsumowanie wyników testów wydajnościowych	60
4.1	Zbiorcza analiza czasu budowy	60
4.2	Zbiorcza analiza czasu zapytań	60
4.3	Podsumowanie:	60
5	Testy QuadTree dla różnych wartości <code>max_capacity</code>	61
5.1	Czas budowy QuadTree	61
5.2	Czas zapytań w QuadTree	61
5.3	Podsumowanie	61
6	Wnioski i Podsumowanie	62
6.1	Wnioski	62
6.2	Podsumowanie	62

Dokumentacja

Struktury QuadTree i Kd-drzewa

Część I

Dokumentacja

1 Wprowadzenie

Niniejsza dokumentacja opisuje klasy `Rectangle`, `KdTree`, `QuadTree` oraz ich odpowiednie węzły `Node` i `QuadTreeNode` w języku Python. Te klasy implementują struktury danych służące do efektywnego przechowywania i wyszukiwania punktów w dwuwymiarowej przestrzeni. `KdTree` jest strukturą drzewiastą, natomiast `QuadTree` to drzewo czwórkowe, oba wykorzystywane w różnych zastosowaniach takich jak przeszukiwanie przestrzenne, grafika komputerowa czy analiza danych przestrzennych. Opisane zostały również klasy `QuadTreeVis` oraz `KdTreeVis`, które zawierają elementy wizualizacji, wykorzystujące narzędzie opracowane przez koło naukowe Bit.

2 Zależności

Implementacja korzysta z następujących bibliotek i modułów Pythona:

- `numpy`: Do operacji numerycznych i obsługi tablic.
- `matplotlib`: Do graficznej wizualizacji procesu budowy i działania struktur danych.
- `typing`: Do wskazywania typów, w tym `Optional`, `Generator`, `Tuple`, `List` oraz `Dict`.
- `rectangle.Rectangle`: Klasa `Rectangle` używana do definiowania granic węzłów `KdTree` i `QuadTree`.

3 Klasy

3.1 Rectangle

Opis: Klasa `Rectangle` reprezentuje prostokąt zdefiniowany przez minimalne i maksymalne współrzędne na osiach X i Y. Dostarcza metody do sprawdzania, czy dany punkt znajduje się wewnątrz prostokąta oraz czy dwa prostokąty się przecinają. Klasa została rozszerzona o obsługę wyjątków w celu zapewnienia poprawności definicji prostokąta.

3.1.1 Atrybuty

`extreme: List[List[float]]`

Lista zawierająca minimalne i maksymalne wartości współrzędnych na osiach X i Y. Struktura:

```
extreme = [[xmin, xmax], [ymin, ymax]]
```

3.1.2 Metody

__init__

```
def __init__(self, xmin: float, ymin: float, xmax: float, ymax: float):
```

Konstruktor klasy `Rectangle`. Inicjalizuje atrybut `extreme` na podstawie podanych współrzędnych granicznych.

Parametry:

- **xmin: float**
Minimalna wartość na osi X.
- **ymin: float**
Minimalna wartość na osi Y.
- **xmax: float**
Maksymalna wartość na osi X.
- **ymax: float**
Maksymalna wartość na osi Y.

Wyjątki:

- **ValueError:** Jeśli $x_{\text{min}} > x_{\text{max}}$ lub $y_{\text{min}} > y_{\text{max}}$, co oznacza nieprawidłową definicję prostokąta.

contains

```
def contains(self, point: Tuple[float, float]) -> bool:
```

Sprawdza, czy dany punkt znajduje się wewnątrz prostokąta (włącznie z krawędziami).

Parametry:

- **point: Tuple[float, float]**
Krotka reprezentująca punkt w przestrzeni 2D jako punkt postaci (x, y) .

Zwraca: bool

True jeśli punkt znajduje się wewnątrz prostokąta, w przeciwnym razie False.

intersects

```
def intersects(self, other: "Rectangle") -> bool:
```

Sprawdza, czy bieżący prostokąt przecina się z innym prostokątem other.

Parametry:

- **other: Rectangle**
Inny prostokąt (obiekt klasy Rectangle) do sprawdzenia przecięcia.

Zwraca: bool

True jeśli prostokąty się przecinają, w przeciwnym razie False.

3.2 Node

Opis: Klasa Node reprezentuje węzeł w strukturze KdTree. Każdy węzeł zawiera punkt w przestrzeni 2D oraz odniesienia do lewego i prawego dziecka. Dodatkowo, każdy węzeł może przechowywać liczbę duplikatów danego punktu.

3.2.1 Atrybuty

point: Tuple[float, float]

Współrzędne punktu przechowywanego w węźle.

left: Optional[Node]

Odniesienie do lewego dziecka węzła. Może być None, jeśli węzeł nie ma lewego potomka.

right: Optional[Node]

Odniesienie do prawego dziecka węzła. Może być None, jeśli węzeł nie ma prawego potomka.

count: int

Liczba duplikatów danego punktu. Domyslnie ustawiona na 1.

3.2.2 Metody

__init__

```
def __init__(
    self,
    point: Tuple[float, float],
    left: Optional["Node"] = None,
    right: Optional["Node"] = None,
    count: int = 1,
):
```

Konstruktor klasy `Node`. Inicjalizuje atrybuty węzła na podstawie podanych parametrów.

Parametry:

- `point: Tuple[float, float]`
Współrzędne punktu przechowywanego w węźle.
- `left: Optional[Node]`
Odniesienie do lewego dziecka węzła.
- `right: Optional[Node]`
Odniesienie do prawego dziecka węzła.
- `count: int`
Liczba zduplikowanych punktów w tym węźle. Domyślnie ustawiona na 1.

__repr__

```
def __repr__(self):
```

Zwraca reprezentację tekstową węzła. Umożliwia czytelne wyświetlanie węzłów i ułatwia debugowanie.

Zwraca: str

Reprezentacja węzła w formie `Node(point=(x, y), count=n)` lub "None" jeśli punkt jest `None`.

3.3 KdTree

Opis: Klasa `KdTree` implementuje drzewo k-wymiarowe (K-d), które jest strukturą danych służącą do przechowywania punktów w dwuwymiarowej przestrzeni. Umożliwia efektywne wyszukiwanie zakresowe w zadanym prostokącie.

3.3.1 Atrybuty

k: int

Liczba wymiarów przestrzeni. W tym przypadku ustawiona na 2.

points: np.ndarray

Oryginalna tablica punktów dostarczona do budowy drzewa.

counts: defaultdict

Słownik zliczający wystąpienia każdego unikalnego punktu. Kluczem jest krotka (x, y) , a wartością liczba wystąpień.

root: Optional[Node]

Korzeń drzewa `KdTree`. Jest to instancja klasy `Node` lub `None`, jeśli drzewo jest puste.

3.3.2 Metody

__init__

```
def __init__(self, points: np.ndarray):
```

Inicjalizuje KdTree z podanym zestawem punktów.

Parametry:

- `points: np.ndarray`
Tablica NumPy o kształcie (`n_points, 2`) reprezentująca punkty 2D.

Wyjątki:

- `TypeError`: Jeśli `points` nie jest tablicą NumPy ndarray.
- `ValueError`: Jeśli tablica `points` nie ma kształtu (`n_points, 2`).

Opis: Podczas inicjalizacji, klasa sprawdza poprawność wejściowych danych, zlicza duplikaty punktów, definiuje korzeń drzeca KdTree i buduje drzewo rekurencyjnie za pomocą metody `build`.

_count_duplicates

```
def _count_duplicates(self) -> defaultdict:
```

Prywatna metoda zliczająca duplikaty punktów w dostarczonym zestawie danych.

Zwraca: `defaultdict` mapujący każdy unikalny punkt na jego liczbę wystąpień.

build

```
def build(self, points: np.ndarray, depth: int = 0) -> Optional[Node]:
```

Rekurencyjnie buduje KdTree z dostarczonych punktów.

Parametry:

- `points: np.ndarray`
Tablica NumPy zawierająca punkty do dodania do drzewa.
- `depth: int`
Aktualny poziom drzewa, używany do wyboru osi podziału.

Zwraca: `Optional[Node]` reprezentujący korzeń poddrzewa.

search_rectangle

```
def search_rectangle(self, rectangle: Rectangle)
-> Generator[Node, None, None]:
```

Metoda inicjująca wyszukiwanie węzłów znajdujących się w zadany prostokącie.

Parametry:

- `rectangle: Rectangle`
Obiekt klasy `Rectangle` definiujący obszar wyszukiwania.

Zwraca: Generator zwracający instancje obiektów `Node` znajdujących się wewnątrz zadanej prostokąta.

__search_rectangle

```
def __search_rectangle(
    self,
    rectangle: Rectangle,
    node: Optional[Node],
    depth: int
) -> Generator[Node, None, None]:
```

Rekurencyjnie przeszukuje drzewo w poszukiwaniu węzłów w zadanym prostokącie.

Parametry:

- **rectangle: Rectangle**
Prostokąt definiujący obszar wyszukiwania.
- **node: Optional[Node]**
Aktualny węzeł drzewa.
- **depth: int**
Aktualna głębokość w drzewie, określająca oś podziału.

Zwraca: Generator zwracający instancje klasy `Node` znajdujących się wewnątrz zadanego prostokąta.

3.4 QuadTreeNode

Opis: Klasa `QuadTreeNode` reprezentuje węzeł w drzewie czwórkowym (QuadTree). Każdy węzeł przechowuje informacje o granicach prostokąta, liczbę punktów w nim zawartych oraz odniesienia do swoich potomków. Drzewo czwórkowe jest strukturą danych używaną do efektywnego przechowywania i wyszukiwania punktów w przestrzeni dwuwymiarowej.

3.4.1 Atrybuty

boundary: Rectangle

Prostokąt definiujący granice obszaru, który obejmuje dany węzeł drzewa czwórkowego

point_counts: Optional[Dict[Tuple[float, float], int]]

Słownik mapujący punkty do ich liczby wystąpień. Obecny tylko w liściach drzewa.

children: List[QuadTreeNode]

Lista potomków węzła. Każdy węzeł może mieć maksymalnie cztery dzieci, odpowiadające czterem podobszarom prostokąta (`boundary`).

count: int

Całkowita liczba punktów znajdujących się w tym węźle. Jeśli węzeł jest liściem, jest to suma wartości w słowniku `point_counts`. W przeciwnym razie jest to suma `count` wszystkich potomków.

3.4.2 Metody

__init__

```
def __init__(
    self,
    boundary: Rectangle,
    point_counts: Optional[Dict[Tuple[float, float], int]] = None,
    children: Optional[List["QuadTreeNode"]] = None
):
```

Inicjalizuje nową instancję klasy `QuadTreeNode`.

Parametry:

- `boundary: Rectangle`
Obiekt klasy `Rectangle` definiujący granice prostokąta węzła.
- `point_counts: Optional[Dict[Tuple[float, float], int]]`
Opcjonalny słownik punktów i ich liczności. Jeśli jest podany, węzeł jest liściem.
- `children: Optional[List[QuadTreeNode]]`
Opcjonalna lista potomków. Jeśli nie jest podany, inicjalizowana jest pusta lista.

Opis: Podczas inicjalizacji, jeśli `point_counts` jest dostarczony, `count` jest ustawiany jako suma wszystkich wartości w `point_counts`. W przeciwnym razie `count` jest sumą `count` wszystkich dzieci.

`is_leaf`

```
def is_leaf(self) -> bool:
```

Sprawdza, czy węzeł jest liściem (czy przechowuje punkty bez potomków).

Zwraca: `bool`

`True` jeśli węzeł jest liściem, w przeciwnym razie `False`.

3.5 QuadTree

Opis: Klasa `QuadTree` implementuje strukturę drzewa czwórkowego do przechowywania punktów w dwuwymiarowej przestrzeni. Umożliwia ona efektywne wyszukiwanie punktów w określonych obszarach, takich jak prostokąty.

3.5.1 Atrybuty

`points: np.ndarray`

Tablica NumPy zawierająca punkty do przechowywania w drzewie. Powinna mieć kształt (`n_points`, 2).

`max_capacity: int`

Maksymalna liczba unikalnych punktów, które może przechowywać węzeł będący liściem. Gdy liczba punktów przekracza tę wartość, węzeł jest dzielony na cztery potomki. Domyślna wartość to 4.

`boundary: Rectangle`

Obiekt klasy `Rectangle` definiujący granice całego drzewa czwórkowego `QuadTree`, obejmujące wszystkie przechowywane punkty.

`root: Optional[QuadTreeNode]`

Korzeń drzewa `QuadTree`. Jest to instancja klasy `QuadTreeNode`

3.5.2 Metody

`__init__`

```
def __init__(self, points: np.ndarray, max_capacity: int = 4):
```

Inicjalizuje drzewo `QuadTree` na podstawie podanych punktów i maksymalnej pojemności węzłów.

Parametry:

- **points: np.ndarray**
Tablica NumPy o kształcie (`n_points, 2`) reprezentująca punkty w przestrzeni dwuwymiarowej.
- **max_capacity: int**
Maksymalna liczba unikalnych punktów w liściu. Domyslnie ustawiona na 4.

Wyjątki:

- **TypeError:** Jeśli `points` nie jest tablicą NumPy ndarray.
- **ValueError:** Jeśli tablica `points` nie ma kształtu (`n_points, 2`).

Opis: Podczas inicjalizacji klasa sprawdza poprawność wejściowych danych, definiuje granice drzewa na podstawie minimalnych i maksymalnych wartości punktów, a następnie buduje drzewo rekurencyjnie za pomocą prywatnej metody `_build`.

_build

```
def _build(self, indices: np.ndarray) -> Optional[QuadTreeNode]:
```

Prywatna metoda rekurencyjna budująca drzewo `QuadTree` na podstawie indeksów punktów.

Parametry:

- **indices: np.ndarray**
Tablica NumPy zawierająca indeksy punktów w tablicy `points`, które mają być uwzględnione w aktualnym węźle.

Zwraca: `Optional[QuadTreeNode]`

Korzeń poddrzewa `QuadTreeNode` lub `None`, jeśli nie ma punktów do przetworzenia.

search_rectangle

```
def search_rectangle(self, rectangle: Rectangle)
-> Generator[Tuple[float, float], None, None]:
```

Metoda inicjująca wyszukiwanie punktów znajdujących się w zadanym prostokącie `Rectangle`.

Parametry:

- **rectangle: Rectangle**
Prostokąt (obiekt klasy `Rectangle`) definiujący obszar wyszukiwania.

Zwraca: Generator krotek (`x, y`) reprezentujących punkty znajdujące się w prostokącie.

_search_rectangle

```
def _search_rectangle(
    self,
    node: QuadTreeNode,
    rectangle: Rectangle,
    seen: set
) -> Generator[Tuple[float, float], None, None]:
```

Rekurencyjnie przeszukuje drzewo w poszukiwaniu punktów w zadanym prostokącie.

Parametry:

- **node: QuadTreeNode**
Aktualny węzeł drzewa `QuadTreeNode`.

- **rectangle: Rectangle**
Prostokąt definiujący obszar wyszukiwania.

- **seen: set**
Zbiór już przetworzonych punktów, aby uniknąć duplikatów.

Zwraca: Generator zwracający krotki (x, y) reprezentujące punkty znajdujące się w prostokącie.

search_rectangle_with_count

```
def search_rectangle_with_count(self, rectangle: Rectangle) -> Generator[Tuple[Tuple[float, float], int]]:
```

Wyszukuje wszystkie punkty znajdujące się w zadanym prostokącie wraz z ich liczbą wystąpień.

Parametry:

- **rectangle: Rectangle**
Prostokąt (obiekt klasy `Rectangle`) definiujący obszar wyszukiwania.

Zwraca: Generator zwracający krotki $((x, y), count)$ reprezentujące punkty znajdujące się w prostokącie oraz ich liczby wystąpień.

_search_rectangle_with_count

```
def _search_rectangle_with_count(
    self,
    node: QuadTreeNode,
    rectangle: Rectangle,
    seen: set
) -> Generator[Tuple[Tuple[float, float], int], None, None]:
```

Rekurencyjnie przeszukuje drzewo w poszukiwaniu punktów w zadanym prostokącie wraz z ich liczbą wystąpień.

Parametry:

- **node: QuadTreeNode**
Aktualny węzeł drzewa `QuadTreeNode`.
- **rectangle: Rectangle**
Prostokąt definiujący obszar wyszukiwania.
- **seen: set**
Zbiór już przetworzonych punktów, aby uniknąć duplikatów.

Zwraca: Generator zwracający krotki $((x, y), count)$ reprezentujące punkty znajdujące się w prostokącie oraz ich liczby wystąpień.

4 Wizualizacje

Wizualizacje zostały stworzone za pomocą narzędzia opracowanego przez koło naukowe Bit, które wykorzystuje bibliotekę `matplotlib`. W niniejszej sekcji przedstawione zostaną wyłącznie dodatkowe implementacje oraz rozszerzenia, które zostały wprowadzone w porównaniu do standardowych struktur (`KdTree` oraz `QuadTree` omówionych kolejno w sekcjach 3.3 oraz 3.5).

4.1 KdTreeVis

Opis: Klasa `KdTreeVis` rozszerza funkcjonalność klasy `KdTree` poprzez dodanie elementów wizualizacji procesu budowy drzewa k-wymiarowego. Umożliwia graficzne przedstawienie punktów w formacie `.JPG` (jako efekt końcowy podziału) oraz `.GIF` (jako wizualizację krokową tworzenia struktury), median oraz linii podziału, co ułatwia zrozumienie struktury drzewa oraz procesu jego tworzenia. Klasa wykorzystuje narzędzie `Visualizer` dostarczone przez koło naukowe `Bit`.

Klasa `KdTreeVis` dziedziczy wszystkie atrybuty i metody z klasy `KdTree`, co oznacza, że posiada wszystkie podstawowe funkcjonalności związane z przechowywaniem punktów, budowaniem drzewa K-d oraz wyszukiwaniem punktów w zadanym prostokącie.

4.1.1 Atrybuty

Oprócz domyślnych atrybutów odziedziczonych z klasy `KdTree` czyli: `points`, `k`, `counts`, `boundary`, `root`, klasa `KdTreeVis` wprowadza dodatkowe atrybuty niezbędne do wizualizacji:

`vis: Visualizer`

Obiekt klasy `Visualizer` odpowiedzialny za tworzenie i zarządzanie wizualizacjami punktów oraz struktur drzewa K-d. Umożliwia zapisywanie wizualizacji w formie krokowej - `.GIF` oraz w formie końcowej - `.JPG`.

4.1.2 Metody

Klasa `KdTreeVis` dziedziczy wszystkie metody z klasy `KdTree` i wykorzystuje je do przechowywania punktów, budowania drzewa K-d oraz wyszukiwania punktów w zadanym prostokącie. Dodatkowo, podczas wykonywania tych operacji, obiekty klasy `KdTreeVis` realizują dodatkowe operacje przy użyciu atrybutu `vis`, będącego instancją klasy `Visualizer`. Dzięki temu, każda operacja na drzewie K-d jest jednocześnie przedstawiana graficznie, co umożliwia śledzenie procesu budowy drzewa oraz wyników wyszukiwania.

```
build:     def build(self, points: np.ndarray, depth: int = 0)
           -> Optional[Node]:
```

Metoda `build` została rozszerzona o funkcje wizualizacyjne, które dodają graficzne elementy do wizualizacji podczas budowania drzewa K-d. Początkowo początkowy obszar zostaje otoczony prostokątem w kolorze ciemnoszarym (#2a2a2a). Po wybraniu punktu mediany, punkt ten jest zaznaczany na wykresie w kolorze **czerwonym (red)**, a linie podziału przestrzeni są rysowane w kolorze **czerwonym (red)** a następnie **szarym (#d3d3d3)**. Dodatkowo, punkty potomne są wizualizowane w kolorze **różowym (pink)**, a następnie przywracane do koloru **czarnego (black)** po zakończeniu podziału. Dzięki temu, użytkownik może śledzić każdy krok procesu budowy drzewa K-d.

Na rysunku 1. została przedstawiona przykładowa wizualizacja zbudowanego drzewa `KdTree`:

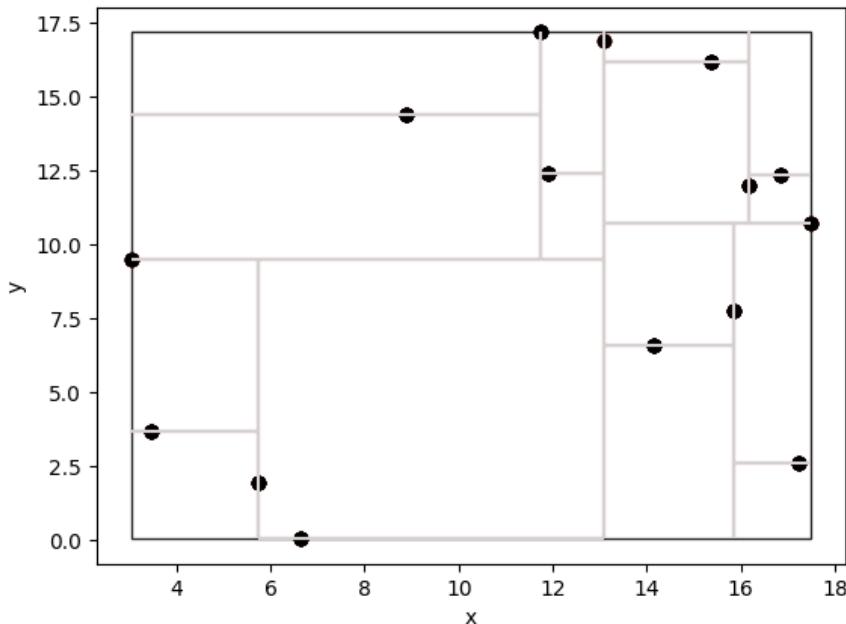
Dodatkowo Dodatkowo, klasa `KdTreeVis` zawiera prywatną metodę pomocniczą, która przekształca obiekt `Rectangle` na listę współrzędnych wierzchołków. Lista ta jest następnie przekazywana do metody `add_polygon` klasy `Visualizer`. Dzięki temu wizualizacje są bardziej i czytelne, co ułatwia analizę struktury drzewa oraz rozmieszczenia punktów w przestrzeni.

`_rectangle_as_polygon`

```
def _rectangle_as_polygon(self, rectangle: Rectangle) -> List[List[float]]:
```

Konwertuje obiekt `Rectangle` na listę wierzchołków potrzebnych do narysowania prostokąta jako poligona w wizualizacji.

Parametry:



Rysunek 1: Przykładowa wizualizacja zbudowanego drzewa K-d

- `rectangle`: Rectangle
Prostokąt do konwersji.

Zwraca: `List[List[float]]` listę wierzchołków prostokąta.

Opis: Metoda ta przekształca granice prostokąta na format odpowiedni do funkcji `add_polygon` używanej przez `Visualizer`.

4.2 QuadTreeVis

Opis: Klasa `QuadTreeVis` rozszerza funkcjonalność klasy `QuadTree` poprzez dodanie elementów wizualizacji procesu budowy drzewa czwórkowego. Umożliwia graficzne przedstawienie punktów w formacie `.JPG` (jako efekt końcowy podziału) oraz `.GIF` (jako wizualizację krokową tworzenia struktury), granic węzłów oraz linii podziału, co ułatwia zrozumienie struktury drzewa oraz procesu jego tworzenia. Klasa wykorzystuje narzędzie `Visualizer` dostarczone przez koło naukowe Bit.

Klasa `QuadTreeVis` dziedziczy wszystkie atrybuty i metody z klasy `QuadTree`, co oznacza, że posiada wszystkie podstawowe funkcjonalności związane z przechowywaniem punktów, budowaniem drzewa czwórkowego oraz wyszukiwaniem punktów w zadanym prostokącie.

4.2.1 Atrybuty

Oprócz domyślnych atrybutów odziedziczonych z klasy `QuadTree`, czyli: `points`, `max_capacity`, `boundary`, oraz `root`, klasa `QuadTreeVis` wprowadza dodatkowe atrybuty niezbędne do wizualizacji:

vis: `Visualizer`

Obiekt klasy `Visualizer` odpowiedzialny za tworzenie i zarządzanie wizualizacjami punktów oraz struktur drzewa czwórkowego. Umożliwia zapisywanie wizualizacji w formie krokowej (`.GIF`) oraz w formie końcowej (`.JPG`).

4.2.2 Metody

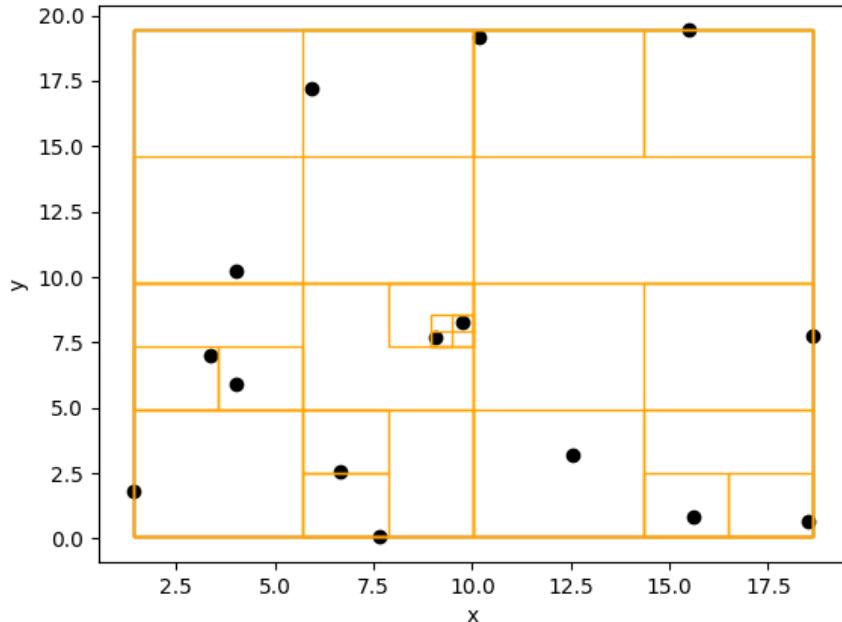
Oprócz domyślnych metod z klasy `QuadTree`, klasa `QuadTreeVis` wykorzystuje te same metody do przechowywania punktów, budowania drzewa czwórkowego oraz wyszukiwania punktów w zadanym

prostokącie. W trakcie wykonywania tych operacji obiekty klasy `QuadTreeVis` realizują dodatkowe operacje wizualizacyjne przy użyciu atrybutu `vis`, będącego instancją klasy `Visualizer`. Dzięki temu każda operacja na drzewie czwórkowym może być jednocześnie przedstawiana graficznie, co umożliwia śledzenie procesu budowy drzewa oraz wyników wyszukiwania.

```
_build:      def _build(self, indices: np.ndarray, boundary: Rectangle)
             -> Optional[QuadTreeNode]:
```

Metoda `_build` została rozszerzona o funkcje wizualizacyjne, które dodają graficzne elementy do wizualizacji podczas budowania drzewa czwórkowego. Początkowo obszar budowy drzewa zostaje otoczony prostokątem w kolorze `#2a2a2a` (ciemnoszary). Następnie, dla każdego węzła, granice są wizualizowane w kolorze **pomarańczowym (orange)**. W przypadku tworzenia węzła liścia, wszystkie punkty są przechowywane i wizualizowane jako punkty w kolorze **czarnym (black)**.

Na rysunku 2. została przedstawiona przykładowa wizualizacja zbudowanego drzewa czwórkowego `QuadTree`:



Rysunek 2: Przykładowa wizualizacja zbudowanego drzewa `QuadTree`.
Parametr `max_capacity` został ustawiony na 1

```
search_rectangle:      def search_rectangle(self, rectangle: Rectangle)
                      -> List[Tuple[float, float]]:
```

Metoda `search_rectangle` została rozszerzona o funkcje wizualizacyjne, które umożliwiają przedstawienie wyników wyszukiwania na wykresie. Podczas przeszukiwania drzewa czwórkowego, znalezione punkty znajdujące się w zadanym prostokącie, pokolorowanym na kolor **niebieski (blue)** są zaznaczane na wykresie w kolorze **zielonym (green)**, a inne punkty pozostają w kolorze **czarnym (black)**.

```
_search_rectangle:      def _search_rectangle(
                           self,
                           node: QuadTreeNode,
```

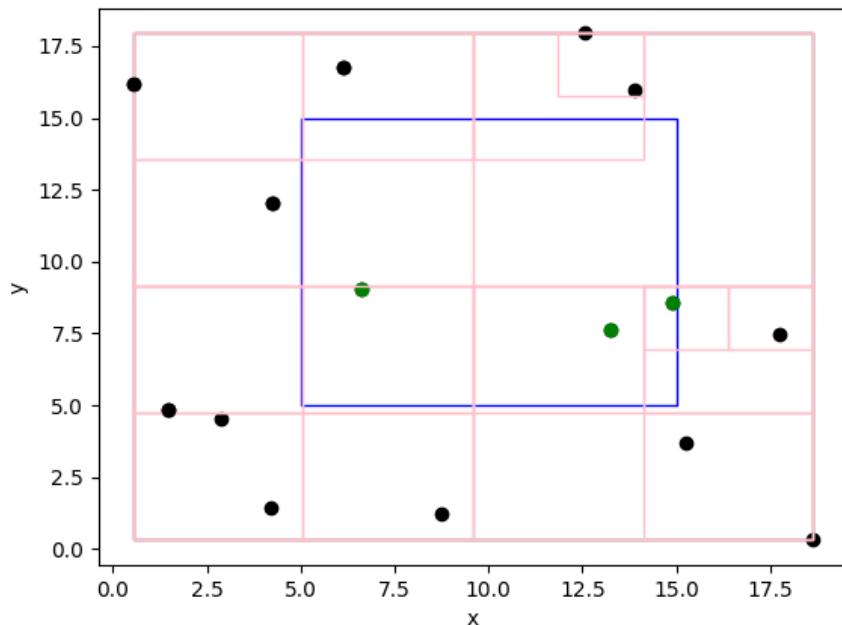
```

    rectangle: Rectangle,
    seen: set
) -> List[Tuple[float, float]]:

```

Prywatna metoda `_search_rectangle` przeszukuje drzewo czwórkowe w poszukiwaniu punktów znajdujących się w zadanym prostokącie. Metoda ta wizualizuje granice każdego aktualnie przeszukiwanego węzła w kolorze **różowym pink** oraz zaznacza znalezione punkty w kolorze **zielonym green**.

Na rysunku 3. została przedstawiona przykładowa wizualizacja przeszukiwania punktów drzewa czwórkowego `QuadTree` dla zadanego prostokąta `Rectangle`:



Rysunek 3: Przykładowa wizualizacja przeszukiwania punktów drzewa czwórkowego `QuadTree`

Dodatkowo

Zaimplementowano także prywatną metodę pomocniczą, która przekształca obiekt `Rectangle` na listę współrzędnych wierzchołków. Lista ta jest wykorzystywana przez metodę `add_polygon` klasy `Visualizer`.

`_rectangle_as_polygon`

```
def _rectangle_as_polygon(self, rectangle: Rectangle) -> List[List[float]]:
```

Konwertuje obiekt `Rectangle` na listę wierzchołków potrzebnych do narysowania prostokąta jako poligona w wizualizacji.

Parametry:

- `rectangle: Rectangle`
Prostokąt do konwersji.

Zwraca: `List[List[float]]` listę wierzchołków prostokąta.

Opis: Metoda ta przekształca granice prostokąta na format odpowiedni do funkcji `add_polygon` używanej przez `Visualizer`.

5 Przykład Użycia

Poniżej znajduje się przykładowa dokumentacja, jak wykorzystać klasy `Rectangle`, `KdTree`, `KdTreeVis`, `QuadTree`, `QuadTreeVis`, `Node` oraz `QuadTreeNode` do tworzenia struktur danych, wyszukiwania punktów w zadanym prostokącie oraz obsługi wyjątków.

5.1 Rectangle

5.1.1 Przykład Użycia Klasy Rectangle

```

1 from rectangle import Rectangle
2
3 def main():
4     try:
5         # Tworzenie dwóch prostokątów
6         rect1 = Rectangle(0, 0, 10, 10)
7         rect2 = Rectangle(5, 5, 15, 15)
8
9         # Próba utworzenia nieprawidłowego prostokąta
10        rect_invalid = Rectangle(10, 10, 5, 5)
11    except ValueError as e:
12        print(f"ŁaBd podczas tworzenia prostokąta: {e}")
13
14    # Punkt do sprawdzenia
15    point = (7, 7)
16
17    # Sprawdzanie, czy punkt znajduje się w rect1
18    if rect1.contains(point):
19        print(f"Punkt {point} znajduje się w rect1.")
20    else:
21        print(f"Punkt {point} nie znajduje się w rect1.")
22
23    # Sprawdzanie, czy rect1 przecina się z rect2
24    if rect1.intersects(rect2):
25        print("rect1 przecina się z rect2.")
26    else:
27        print("rect1 nie przecina się z rect2.")
28
29 if __name__ == "__main__":
30     main()

```

Listing 1: Przykład użycia klasy Rectangle

5.1.2 Wyjaśnienie Przykładu Klasy Rectangle

- `rect1 = Rectangle(0, 0, 10, 10)`: Tworzy prostokąt od (0,0) do (10,10).
- `rect2 = Rectangle(5, 5, 15, 15)`: Tworzy prostokąt od (5,5) do (15,15).
- `rect_invalid = Rectangle(10, 10, 5, 5)`: Próbuje utworzyć nieprawidłowy prostokąt, co powoduje zgłoszenie `ValueError`.
- `point = (7, 7)`: Definiuje punkt do sprawdzenia.
- `rect1.contains(point)`: Sprawdza, czy punkt (7,7) znajduje się w `rect1`.
- `rect1.intersects(rect2)`: Sprawdza, czy `rect1` przecina się z `rect2`.
- `print(...)`: Wypisuje wyniki sprawdzeń oraz ewentualne błędy.

5.2 KdTree

5.2.1 Przykład Użycia Klasy KdTree

```

1 import numpy as np
2 from kdtreeVis import KdTreeVis
3 from rectangle import Rectangle
4
5 def main():
6     try:
7         # Definicja punktów
8         points = np.array([
9             [2, 3], [5, 7], [9, 6], [4, 7],
10            [5, 7], [7, 2], [6, 6], [15, 15],
11            [5, 15], [16, 15], [5, 5]
12        ])
13
14         # Tworzenie KdTree z qWizualizacj
15         kdtree = KdTreeVis(points)
16
17         # Definicja qprostokta wyszukiwania
18         rectangle = Rectangle(5, 5, 15, 15)
19
20         # Wyszukiwanie qlwów w qprostokcie
21         result_nodes = kdtree.search_rectangle(rectangle)
22         print("qlwzy qznajdujce qsi w qprostokcie:")
23         for node in result_nodes:
24             print(node)
25
26     except (TypeError, ValueError) as e:
27         print(f"qaqBd: {e}")
28
29 if __name__ == "__main__":
30     main()

```

Listing 2: Przykład użycia klasy KdTree

5.2.2 Wyjaśnienie Przykładu Klasy KdTree

- `points`: Inicjalizuje tablicę NumPy z punktami 2D.
- `KdTree(points)`: Tworzy KdTree z podanych punktów.
- `Rectangle(5, 5, 15, 15)`: Definiuje prostokątny obszar od (5,5) do (15,15).
- `search_rectangle(rectangle)`: Wyszukuje wszystkie węzły znajdujące się w zadanym prostokącie.
- `print(node)`: Wypisuje znalezione węzły w prostokącie.
- `try-except`: Obsługuje potencjalne wyjątki `TypeError` i `ValueError`.

5.3 KdTreeVis

5.3.1 Przykład Użycia Klasy KdTreeVis

```

1 import numpy as np
2 from code.kdtreeVis import KdTreeVis
3 from code.rectangle import Rectangle
4

```

```

5 def main():
6     try:
7         # Definicja punktów
8         points = np.array([
9             [2, 3], [5, 7], [9, 6], [4, 7],
10            [5, 7], [7, 2], [6, 6], [15, 15],
11            [5, 15], [16, 15], [5, 5]
12        ])
13
14         # Tworzenie KdTree z wizualizacją
15         kdtree = KdTreeVis(points)
16
17         kdtree.vis.save("kdtree_build")
18         kdtree.vis.save_gif("kdtree_build.gif", interval=100)
19
20         # Definicja prostokąta wyszukiwania
21         rectangle = Rectangle(5, 5, 15, 15)
22
23         # Wyszukiwanie elementów w prostokącie
24         result_nodes = kdtree.search_rectangle(rectangle)
25         print("Elementy znajdujące się w prostokącie:")
26         for node in result_nodes:
27             print(node)
28
29     except (TypeError, ValueError) as e:
30         print(f"ŁąBd: {e}")
31
32 if __name__ == "__main__":
33     main()

```

Listing 3: Przykład użycia klasy KdTreeVis

5.3.2 Wyjaśnienie Przykładu Klasy KdTreeVis

- `points`: Inicjalizuje tablicę NumPy z punktami 2D, które będą przechowywane w drzewie K-d.
- `KdTreeVis(points)`: Tworzy instancję `KdTreeVis` z podanych punktów, co automatycznie buduje drzewo K-d i inicjalizuje wizualizację procesu budowy drzewa.
- `vis.save("kdtree_build")`: Zapisuje aktualny (końcowy) stan wizualizacji procesu budowy drzewa K-d jako obraz (np. `kdtree_build.png`).
- `vis.save_gif("kdtree_build.gif")`: Tworzy animację GIF całego procesu budowy drzewa K-d i zapisuje ją jako `kdtree_build.gif` z interwalem 100 ms między klatkami.
- `Rectangle(5, 5, 15, 15)`: Definiuje prostokątny obszar od (5,5) do (15,15), w którym będą wyszukiwane węzły drzewa K-d.
- `search_rectangle(rectangle)`: Wyszukuje wszystkie węzły drzewa K-d znajdujące się w zadanym prostokącie `rectangle`.
- `print(node)`: Wypisuje znalezione węzły w prostokącie, umożliwiając użytkownikowi przegląd wyników wyszukiwania.
- `try-except`: Blok `try-except` obsługuje potencjalne wyjątki `TypeError` i `ValueError`

5.4 QuadTree

5.4.1 Przykład Użycia Klasy QuadTree

```

1 import numpy as np
2 from quadtree import QuadTree
3 from rectangle import Rectangle
4
5 def main():
6     try:
7         # Definicja punktów
8         points = np.array([
9             [2, 3], [5, 7], [9, 6], [4, 7],
10            [5, 7], [7, 2], [6, 6], [15, 15],
11            [5, 15], [16, 15], [5, 5]
12        ])
13
14         # Tworzenie QuadTree z maksymalną pojemnością 1 punktu na liść
15         quadtree = QuadTree(points, max_capacity=1)
16
17         # Definicja prostokąta wyszukiwania
18         rectangle = Rectangle(5, 5, 15, 15)
19
20         # Wyszukiwanie punktów w prostokącie
21         result_points = quadtree.search_rectangle(rectangle)
22         print("Punkty znajdujące się w prostokącie:")
23         for (x, y) in result_points:
24             print(f"Point: ({x}, {y})")
25
26         # Wyszukiwanie punktów z liczbą wystąpień w prostokącie
27         result_all = quadtree.search_rectangle_with_count(rectangle)
28         print("\nPunkty i ich liczba wystąpień w prostokącie:")
29         for (x, y), count in result_all:
30             print(f"Point: ({x}, {y}), Count: {count}")
31
32         # Próba wyszukiwania w nieistniejącym prostokącie
33         empty_rectangle = Rectangle(100, 100, 200, 200)
34         empty_results = quadtree.search_rectangle(empty_rectangle)
35         print("\nPunkty w pustym prostokącie:")
36         for point in empty_results:
37             print(f"Point: {point}")
38
39     except (TypeError, ValueError) as e:
40         print(f"\nBłąd: {e}")
41
42 if __name__ == "__main__":
43     main()

```

Listing 4: Przykład użycia klasy QuadTree

5.4.2 Wyjaśnienie Przykładu Klasy QuadTree

- `points`: Inicjalizuje tablicę NumPy z punktami 2D.
- `QuadTree(points, max_capacity=1)`: Tworzy QuadTree z podanych punktów, ustawiając maksymalną pojemność liścia na 1 punkt.
- `Rectangle(5, 5, 15, 15)`: Definiuje prostokątny obszar od (5,5) do (15,15).
- `search_rectangle(rectangle)`: Wyszukuje wszystkie punkty znajdujące się w zadanym prostokącie.
- `search_rectangle_with_count(rectangle)`: Wyszukuje wszystkie punkty w prostokącie wraz z ich liczbą wystąpień.

- `Rectangle(100, 100, 200, 200)`: Definiuje pusty prostokąt, w którym nie ma żadnych punktów.
- `try-except`: Obsługuje potencjalne wyjątki `TypeError` i `ValueError`.
- `print(...)`: Wypisuje wyniki wyszukiwania punktów oraz ewentualne błędy.

5.5 QuadTreeVis

5.5.1 Przykład Użycia Klasy QuadTreeVis

```

1 import numpy as np
2 from code.QuadtreeVis import QuadTreeVis
3 from code.rectangle import Rectangle
4
5 def main():
6     try:
7         # Definicja punktów
8         points = np.array([
9             [2, 3], [5, 7], [9, 6], [4, 7],
10            [5, 7], [7, 2], [6, 6], [15, 15],
11            [5, 15], [16, 15], [5, 5]
12        ])
13
14         # Tworzenie QuadTreeVis z maksymalną pojemnością 1 punktu na liść
15         quadtree = QuadTreeVis(points, max_capacity=1)
16
17         # Definicja prostokąta wyszukiwania
18         rectangle = Rectangle(5, 5, 15, 15)
19
20         # Tworzenie wizualizacji procesu budowy drzewa QuadTree
21         quadtree.vis.save("quadtree_build")
22         quadtree.vis.save_gif("quadtree_build_gif", interval=100)
23
24         # Wyszukiwanie punktów w prostokącie
25         quadtree.search_rectangle(rectangle)
26
27         # Tworzenie wizualizacji procesu wyszukiwania w QuadTree
28         quadtree.vis.save("quadtree_search")
29         quadtree.vis.save_gif("quadtree_search_gif", interval=100)
30
31     except (TypeError, ValueError) as e:
32         print(f"ŁaBd: {e}")
33
34 if __name__ == "__main__":
35     main()

```

Listing 5: Przykład użycia klasy QuadTreeVis

5.5.2 Wyjaśnienie Przykładu Klasy QuadTreeVis

- `points`: Inicjalizuje tablicę NumPy z punktami 2D, które będą przechowywane w drzewie QuadTree.
- `QuadTreeVis(points, max_capacity=1)`: Tworzy instancję `QuadTreeVis` z podanych punktów, ustawiając maksymalną pojemność liścia na 1 punkt. To powoduje, że każdy liść w drzewie QuadTree będzie zawierał tylko jeden punkt.
- `Rectangle(5, 5, 15, 15)`: Definiuje prostokątny obszar od (5,5) do (15,15), w którym będą wyszukiwane punkty w drzewie QuadTree.

- `vis.save("quadtree_build")`: Zapisuje aktualny (końcowy) stan wizualizacji procesu budowy drzewa QuadTree jako obraz (np. `quadtree_build.png`).
- `vis.save_gif("quadtree_build_gif", interval=100)`: Tworzy animację GIF całego procesu budowy drzewa QuadTree i zapisuje ją jako `quadtree_build.gif` z interwałem 100 ms między klatkami.
- `search_rectangle(rectangle)`: Wyszukuje wszystkie punkty znajdujące się w zadanym prostokącie `rectangle` w drzewie QuadTree, jednocześnie aktualizując wizualizację procesu wyszukiwania.
- `vis.save("quadtree_search")`: Zapisuje aktualny (końcowy) stan wizualizacji procesu wyszukiwania jako obraz (np. `quadtree_search.png`).
- `vis.save_gif("quadtree_search_gif", interval=100)`: Tworzy animację GIF procesu wyszukiwania punktów w drzewie QuadTree i zapisuje ją jako `quadtree_search.gif` z interwałem 100 ms między klatkami.
- `try-except`: Blok `try-except` obsługuje potencjalne wyjątki `TypeError` i `ValueError`

6 Obsługa Błędów

Klasy `Rectangle`, `KdTree`, `KdTreeVis`, `QuadTree`, `QuadTreeVis` oraz ich węzły `Node` i `QuadTreeNode` zawierają mechanizmy obsługi błędów, aby zapewnić poprawność danych wejściowych oraz integralność struktury drzewa. Poniżej przedstawiono szczegółowe informacje na temat typów wyjątków, które mogą być zgłaszane przez poszczególne klasy oraz przykłady ich obsługi.

- `Rectangle`:
 - `ValueError`: Zgłasza się, jeśli `xmin > xmax` lub `ymin > ymax`, co oznacza nieprawidłową definicję prostokąta.
 - * **Przykład:** Próba utworzenia prostokąta z `xmin = 10` i `xmax = 5` spowoduje zgłoszenie `ValueError` z komunikatem "`Invalid rectangle definition`".
 - * **Dlaczego to ważne:** Zapobiega tworzeniu prostokątów, które nie mają logicznego sensu, co mogłoby prowadzić do błędów w dalszych operacjach na drzewie przestrzennym.
- `KdTree` i `KdTreeVis`:
 - `TypeError`: Zgłasza się, jeśli `points` nie jest tablicą NumPy ndarray.
 - * **Przykład:** Próba utworzenia `KdTreeVis` z listą zamiast tablicy NumPy spowoduje zgłoszenie `TypeError`.
 - * **Dlaczego to ważne:** Gwarantuje, że dane wejściowe są w odpowiednim formacie, co jest niezbędne do prawidłowego działania algorytmów drzewa K-d.
 - `ValueError`: Zgłasza się, jeśli tablica `points` nie ma kształtu (`n_points, 2`).
 - * **Przykład:** Tablica punktów o kształcie (`n_points, 3`) zamiast (`n_points, 2`) spowoduje zgłoszenie `ValueError`.
 - * **Dlaczego to ważne:** Upewnia się, że każdy punkt ma dokładnie dwie współrzędne (`x` i `y`), co jest wymogiem dla drzewa K-d skoncentrowanego na dwuwymiarowych przestrzeniach.
- `QuadTree` i `QuadTreeVis`:
 - `TypeError`: Zgłasza się, jeśli `points` nie jest tablicą NumPy ndarray.

- * **Przykład:** Próba utworzenia `QuadTreeVis` z listą zamiast tablicy NumPy spowoduje zgłoszenie `TypeError`.
- * **Dlaczego to ważne:** Zapewnia, że dane wejściowe są w odpowiednim formacie, co jest niezbędne do efektywnego zarządzania danymi przestrzennymi w drzewie QuadTree.
 - `ValueError`: Zgłasza, jeśli tablica `points` nie ma kształtu (`n_points, 2`).
 - * **Przykład:** Tablica punktów o kształcie (`n_points, 3`) zamiast (`n_points, 2`) spowoduje zgłoszenie `ValueError`.
 - * **Dlaczego to ważne:** Upewnia się, że każdy punkt ma dokładnie dwie współrzędne (`x` i `y`), co jest niezbędne dla prawidłowego działania drzewa QuadTree w dwuwymiarowych przestrzeniach.
- `QuadTreeNode`:
 - `ValueError`: Zgłasza przez klasę `Rectangle`, jeśli `xmin > xmax` lub `ymin > ymax`, co oznacza nieprawidłową definicję prostokąta.
 - * **Przykład:** Tworzenie węzła `QuadTreeNode` z nieprawidłowym prostokątem spowoduje zgłoszenie `ValueError`.
 - * **Dlaczego to ważne:** Zapewnia, że każdy węzeł QuadTree jest zdefiniowany poprawnie, co jest kluczowe dla prawidłowego podziału przestrzeni w drzewie.

Użytkownik powinien obsługiwać te wyjątki za pomocą bloków `try-except`, aby zapewnić, że tworzone obiekty są zawsze prawidłowe i unikać nieoczekiwanych błędów podczas działania programu.

7 Szczegółы Zależności

7.1 Klasa Rectangle

Implementacja klas `KdTree` oraz `QuadTree` opiera się na klasie `Rectangle`, zakładanej jako zdefiniowana w module `rectangle`. Klasa ta zapewnia metody do definiowania prostokątnego obszaru oraz sprawdzania, czy dany punkt znajduje się w nim lub czy dwa prostokąty się przecinają.

- `Rectangle(x_min, y_min, x_max, y_max)`: Inicjalizuje prostokąt z podanymi granicami. Podczas inicjalizacji sprawdza, czy $x_{\min} < x_{\max}$ oraz $y_{\min} < y_{\max}$. Jeśli warunek nie jest spełniony, zgłasza `ValueError`.
- `contains(point: Tuple[float, float]) -> bool`: Zwraca `True`, jeśli punkt znajduje się w prostokącie, w przeciwnym razie `False`.
- `intersects(other: Rectangle) -> bool`: Zwraca `True`, jeśli prostokąty się przecinają, w przeciwnym razie `False`.

Zależności:

- Klasy `KdTree` i `QuadTree` wykorzystują `Rectangle` do definiowania obszarów, w których przechowywane są punkty.
- Węzły drzewa czwórkowego (`QuadTreeNode`) używają obiektów `Rectangle` do opisania swoich granic.
- Wyszukiwanie punktów w zadanych obszarach w obu strukturach opiera się na metodach klasy `Rectangle`.

7.2 Klasa Node

Klasa `Node` jest wewnętrzną klasą używaną przez `KdTree` do reprezentowania węzłów drzewa. Każdy węzeł zawiera punkt w przestrzeni 2D oraz odniesienia do lewego i prawego dziecka, co umożliwia budowanie drzewa k-wymiarowego.

- `Node(point: Tuple[float, float], left: Optional[Node], right: Optional[Node], count: int = 1)`: Inicjalizuje węzeł drzewa, zawierający punkt, wskaźniki na lewego i prawego potomka oraz liczbę wystąpień tego punktu w danych.
- `__repr__() -> str`: Zwraca czytelną reprezentację węzła, zawierającą informacje o punkcie i liczbie wystąpień.

Zależności:

- `Node` jest używane wyłącznie przez `KdTree` do reprezentowania węzłów.
- Węzły drzewa są podstawą do przechowywania punktów i umożliwiają rekurencyjne dzielenie przestrzeni w `KdTree`.

7.3 Klasa KdTree

Klasa `KdTree` implementuje strukturę drzewa k-wymiarowego, która umożliwia efektywne przechowywanie punktów oraz szybkie wyszukiwanie w zadanym obszarze.

- `KdTree(points: np.ndarray)`: Inicjalizuje strukturę drzewa, przyjmując tablicę punktów (x, y). Sprawdza poprawność wejściowych danych, tj. czy są one tablicą NumPy o odpowiednich wymiarach.
- `build(points: np.ndarray, depth: int = 0) -> Optional[Node]`: Rekurencyjnie buduje drzewo k-wymiarowe, dzieląc dane na podstawie mediany wzduż cyklicznie zmienianych osi.
- `search_rectangle(rectangle: Rectangle) -> Generator[Node, None, None]`: Przeszukuje drzewo w poszukiwaniu punktów znajdujących się w zadanym prostokącie.

Zależności:

- Używa klasy `Node` do reprezentacji węzłów drzewa.
- Operacje wyszukiwania i budowy drzewa wykorzystują obiekty `Rectangle` do definiowania obszarów przestrzeni.

7.4 Klasa QuadTreeNode

Klasa `QuadTreeNode` reprezentuje węzeł w strukturze drzewa czwórkowego. Może być węzłem wewnętrznym lub liściem, przechowującym punkty.

- `QuadTreeNode(boundary: Rectangle, point_counts: Optional[Dict[Tuple[float, float], int]], children: Optional[List[QuadTreeNode]])`: Inicjalizuje węzeł, określając jego granice, przechowywane punkty (dla liści) lub listę dzieci (dla węzłów wewnętrznych).
- `is_leaf() -> bool`: Zwraca `True`, jeśli węzeł jest liściem, w przeciwnym razie `False`.

Zależności:

- Węzły korzystają z `Rectangle`, aby opisywać swoje granice w przestrzeni.
- Punkty w węzłach są dzielone i delegowane do dzieci, co pozwala na hierarchiczne przechowywanie danych w `QuadTree`.

7.5 Klasa QuadTree

Klasa `QuadTree` implementuje strukturę drzewa czwórkowego, która umożliwia przechowywanie punktów w 2D oraz efektywne wyszukiwanie ich w zadanym obszarze.

- `QuadTree(points: np.ndarray, max_capacity: int = 4)`: Inicjalizuje drzewo, przyjmując tablicę punktów (x, y) oraz maksymalną pojemność liści. Sprawdza poprawność danych wejściowych.
- `_build(indices: np.ndarray) -> Optional[QuadTreeNode]`: Rekurencyjnie buduje drzewo czwórkowe, dzieląc obszar na cztery podobszary.
- `search_rectangle(rectangle: Rectangle) -> Generator[Tuple[float, float], None, None]`: Przeszukuje drzewo w poszukiwaniu punktów znajdujących się w zadanym prostokącie.
- `search_rectangle_with_count(rectangle: Rectangle) -> Generator[Tuple[Tuple[float, float], int], None, None]`: Przeszukuje drzewo, zwracając punkty znajdujące się w zadanym prostokącie wraz z ich liczbą wystąpień.

Zależności:

- Używa klasy `QuadTreeNode` do reprezentacji węzłów drzewa.
- Klasa `Rectangle` jest wykorzystywana do określenia granic obszarów, które są dzielone w `QuadTree`.
- Wyszukiwanie punktów w prostokącie polega na iteracyjnym porównywaniu obszarów za pomocą metod klasy `Rectangle`.

8 Funkcje Testowe

Poniżej znajduje się dokumentacja funkcji testowych, które umożliwiają generowanie różnych zestawów danych punktów oraz prostokątów do testowania struktur `KdTree` i `QuadTree`.

8.1 random_uniform_test

Opis: Generuje losowe punkty rozłożone równomiernie w zadanym zakresie oraz prostokąt testowy.

8.1.1 Parametry

count: int

Liczba punktów do wygenerowania.

minval: float

Minimalna wartość dla każdego wymiaru (domyślnie -100).

maxval: float

Maksymalna wartość dla każdego wymiaru (domyślnie 100).

dimension: int

Liczba wymiarów punktów (domyślnie 2).

8.1.2 Zwraca

points: np.ndarray

Tablica punktów o kształcie (`count, dimension`).

rect: Rectangle

Prostokąt testowy definiujący obszar wyszukiwania.

8.2 random_normal_test

Opis: Generuje losowe punkty rozłożone normalnie (rozkład Gaussa) oraz prostokąt testowy.

8.2.1 Parametry

count: int

Liczba punktów do wygenerowania.

mean: float

Średnia rozkładu normalnego (domyślnie 0).

std: float

Odchylenie standardowe rozkładu normalnego (domyślnie 1).

dimension: int

Liczba wymiarów punktów (domyślnie 2).

8.2.2 Zwraca

points: np.ndarray

Tablica punktów o kształcie (`count, dimension`).

rect: Rectangle

Prostokąt testowy definiujący obszar wyszukiwania.

8.3 random_integer_test

Opis: Generuje losowe punkty o wartościach całkowitych w zadanym zakresie oraz prostokąt testowy.

8.3.1 Parametry

count: int

Liczba punktów do wygenerowania.

low: int

Dolna granica zakresu wartości punktów (domyślnie 0).

high: int

Górna granica zakresu wartości punktów (domyślnie 100).

dimension: int

Liczba wymiarów punktów (domyślnie 2).

8.3.2 Zwraca

points: np.ndarray

Tablica punktów o kształcie (`count, dimension`).

rect: Rectangle

Prostokąt testowy definiujący obszar wyszukiwania.

8.4 grid_test

Opis: Generuje punkty rozmieszczone w siatce oraz prostokąt testowy.

8.4.1 Parametry

count: int

Liczba punktów do wygenerowania (powinna być kwadratem liczby punktów na jednym wymiarze).

minval: float

Minimalna wartość dla każdego wymiaru (domyślnie -100).

maxval: float

Maksymalna wartość dla każdego wymiaru (domyślnie 100).

dimension: int

Liczba wymiarów punktów (domyślnie 2).

8.4.2 Zwraca

points: np.ndarray

Tablica punktów rozmieszczonych w siatce.

rect: Rectangle

Prostokąt testowy definiujący obszar wyszukiwania.

8.5 circle_test

Opis: Generuje punkty rozmieszczone w okręgu oraz prostokąt testowy.

8.5.1 Parametry

count: int

Liczba punktów do wygenerowania.

radius: float

Promień okręgu (domyślnie 1).

dimension: int

Liczba wymiarów punktów (domyślnie 2).

8.5.2 Zwraca

points: np.ndarray

Tablica punktów rozmieszczone w okręgu.

rect: Rectangle

Prostokąt testowy definiujący obszar wyszukiwania.

8.6 line_test

Opis: Generuje punkty rozmieszczone wzdłuż linii oraz prostokąt testowy.

8.6.1 Parametry

count: int

Liczba punktów do wygenerowania.

x1: float

Współrzędna X punktu początkowego linii (domyślnie -50).

y1: float

Współrzędna Y punktu początkowego linii (domyślnie -50).

x2: float

Współrzędna X punktu końcowego linii (domyślnie 50).

y2: float

Współrzędna Y punktu końcowego linii (domyślnie 50).

8.6.2 Zwraca

points: np.ndarray

Tablica punktów rozmieszczonych wzdłuż linii.

rect: Rectangle

Prostokąt testowy definiujący obszar wyszukiwania.

8.7 cross_test

Opis: Generuje punkty tworzące krzyż oraz prostokąt testowy.

8.7.1 Parametry

count: int

Łączna liczba punktów do wygenerowania (powinna być parzysta).

minval: float

Minimalna wartość dla każdego wymiaru (domyślnie -100).

maxval: float

Maksymalna wartość dla każdego wymiaru (domyślnie 100).

dimension: int

Liczba wymiarów punktów (domyślnie 2).

8.7.2 Zwraca

points: np.ndarray

Tablica punktów tworzących krzyż.

rect: Rectangle

Prostokąt testowy definiujący obszar wyszukiwania.

8.8 rectangle_sides_test

Opis: Generuje punkty rozmieszczone na bokach prostokąta oraz prostokąt testowy.

8.8.1 Parametry

count: int

Łączna liczba punktów do wygenerowania (powinna być podzielna przez 4).

minval: float

Minimalna wartość dla każdego wymiaru (domyślnie -100).

maxval: float

Maksymalna wartość dla każdego wymiaru (domyślnie 100).

dimension: int

Liczba wymiarów punktów (domyślnie 2).

8.8.2 Zwraca

points: np.ndarray

Tablica punktów rozmieszczonych na bokach prostokąta.

rect: Rectangle

Prostokąt testowy definiujący obszar wyszukiwania.

8.9 rectangle_with_diagonals_test

Opis: Generuje punkty rozmieszczone zarówno na bokach, jak i na przekątnych prostokąta oraz prostokąt testowy.

8.9.1 Parametry

count: int

Łączna liczba punktów do wygenerowania.

a: float

Długość pierwszej krawędzi prostokąta (domyślnie 50).

b: float

Długość drugiej krawędzi prostokąta (domyślnie 50).

dimension: int

Liczba wymiarów punktów (domyślnie 2).

on_diagonal_ratio: float

Procent punktów rozmieszczenych na przekątnych (domyślnie 0.5).

8.9.2 Zwraca

points: np.ndarray

Tablica punktów rozmieszczonej na bokach i przekątnych prostokąta.

rect: Rectangle

Prostokąt testowy definiujący obszar wyszukiwania.

8.10 test_two_clusters

Opis: Generuje dwa odrębne skupiska punktów oraz prostokąt testowy.

8.10.1 Parametry

count: int

Łączna liczba punktów do wygenerowania.

minval: float

Minimalna wartość dla każdego wymiaru w pierwszym skupisku (domyślnie -100).

maxval: float

Maksymalna wartość dla każdego wymiaru w drugim skupisku (domyślnie 100).

dimension: int

Liczba wymiarów punktów (domyślnie 2).

8.10.2 Zwraca

points: np.ndarray

Tablica punktów tworzących dwa oddzielne skupiska.

rect: Rectangle

Prostokąt testowy definiujący obszar wyszukiwania.

8.11 test_with_outliers

Opis: Generuje punkty z dodatkowymi wartościami odstającymi (outliers) oraz prostokąt testowy.

8.11.1 Parametry

count: int

Liczba punktów do wygenerowania.

minval: float

Minimalna wartość dla każdego wymiaru (domyślnie -100).

maxval: float

Maksymalna wartość dla każdego wymiaru (domyślnie 100).

dimension: int

Liczba wymiarów punktów (domyślnie 2).

8.11.2 Zwraca

points: np.ndarray

Tablica punktów z dodatkowymi wartościami odstającymi.

rect: Rectangle

Prostokąt testowy definiujący obszar wyszukiwania.

Sprawozdanie

Struktury QuadTree i Kd-drzewa

Część II

Sprawozdanie

1 Wstęp Teoretyczny

1.1 Cel Projektu

Celem projektu było zaimplementowanie dwóch struktur danych QuadTree i KdTree, umożliwiających efektywne zarządzanie i przeszukiwanie zbioru punktów w dwuwymiarowej przestrzeni. Projekt koncentrował się na rozwiązyaniu problemu szybkiego wyszukiwania punktów znajdujących się w zadanym obszarze, przy jednoczesnym zapewnieniu przejrzystości i elastyczności w działaniu obu struktur.

Projekt obejmował następujące założenia:

- **Implementacja podstawowych operacji:**

- Budowa struktur z podanych zbiorów punktów, przy zachowaniu poprawności i wydajności w procesie dzielenia przestrzeni.
- Wyszukiwanie punktów w określonych prostokątnych obszarach za pomocą rekurencyjnych algorytmów dostosowanych do specyfiki każdej struktury.

- **Analiza i porównanie wydajności:**

- Testowanie obu struktur na danych o różnych rozkładach, takich jak rozkłady jednorodne, normalne, siatki regularne czy punkty skupione w klastrach.
- Porównanie czasów budowy struktur oraz czasu przeszukiwania w zależności od liczby punktów, kształtu obszaru oraz charakteru danych wejściowych.

- **Wizualizacja działania struktur:**

- Zastosowanie narzędzi wizualizacyjnych, które umożliwiają graficzne przedstawienie podziału przestrzeni przez QuadTree i KdTree.
- Ilustracja procesu budowy drzew i wyszukiwania punktów w zadanych obszarach, co pozwala na intuicyjne zrozumienie działania algorytmów.

- **Zastosowanie i testy:**

- Implementacja wszechstronnych testów jednostkowych oraz testów porównawczych w celu potwierdzenia poprawności i wydajności zaimplementowanych struktur.
- Przeanalizowanie zachowania obu struktur w skrajnych przypadkach, takich jak dane z dużą liczbą punktów odstających czy dane mocno wspólnotowe.

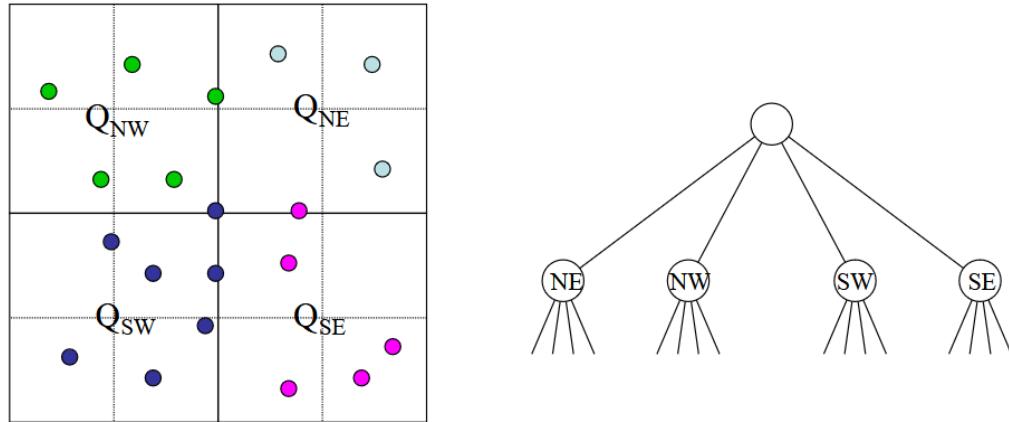
Efektem końcowym projektu była kompleksowa analiza działania QuadTree i KdTree, wraz z przejrzystą implementacją oraz praktycznym zestawem narzędzi do testowania i wizualizacji ich działania.

1.2 QuadTree

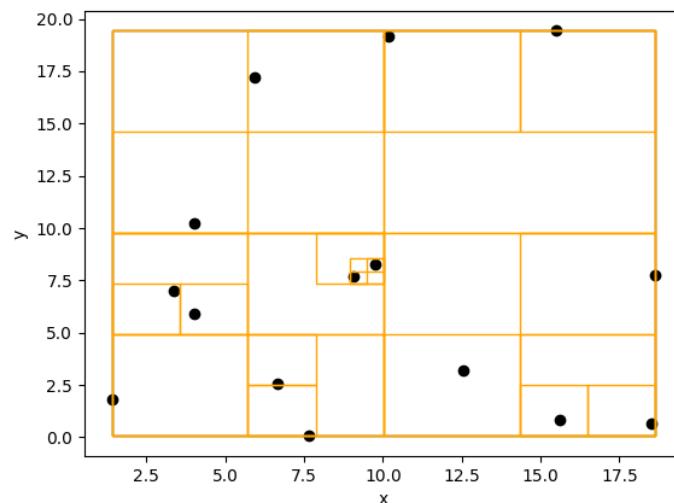
QuadTree to struktura danych umożliwiająca przechowywanie punktów w przestrzeni dwuwymiarowej. Jej głównym zadaniem jest rekurencyjny podział przestrzeni na mniejsze części poprzez dzielenie jej na cztery równe ćwiartki. Każda z tych ćwiartek może być dalej dzielona na kolejne cztery części, dopóki liczba punktów w danym obszarze nie przekroczy zdefiniowanej maksymalnej pojemności węzła. Węzły drzewa reprezentują prostokątne obszary przestrzeni, natomiast liście to najmniejsze prostokąty, które nie są już dzielone z powodu ograniczenia liczby punktów.

Teoretyczna złożoność operacji w QuadTree jest następująca:

- Budowa struktury: $O(n \log n)$, gdzie n to liczba punktów w zbiorze, przy założeniu równomiernego rozkładu punktów.
- Wyszukiwanie punktów w zadanym obszarze: $O(d \cdot l)$, gdzie d to głębokość drzewa, a l to liczba liści. W przypadku dobrze zrównoważonego drzewa głębokość d może być logarytmiczna względem liczby punktów.



Rysunek 1: Struktura QuadTree.



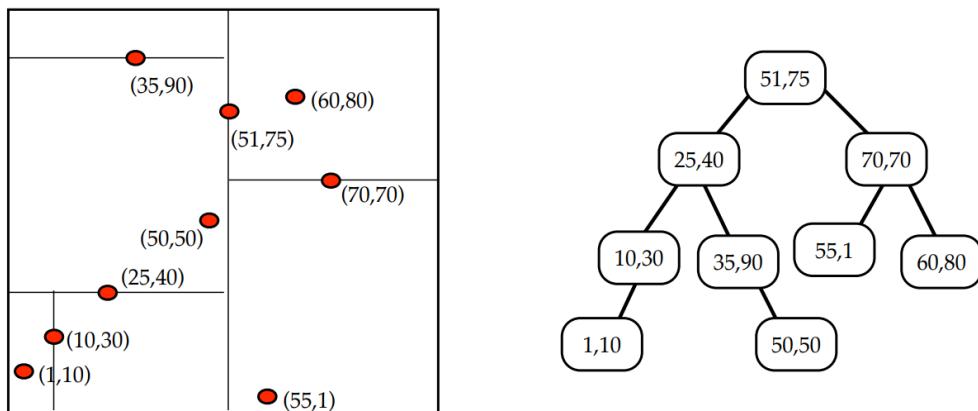
Rysunek 2: Przykładowa wizualizacja zbudowanego drzewa QuadTree

1.3 KdTree

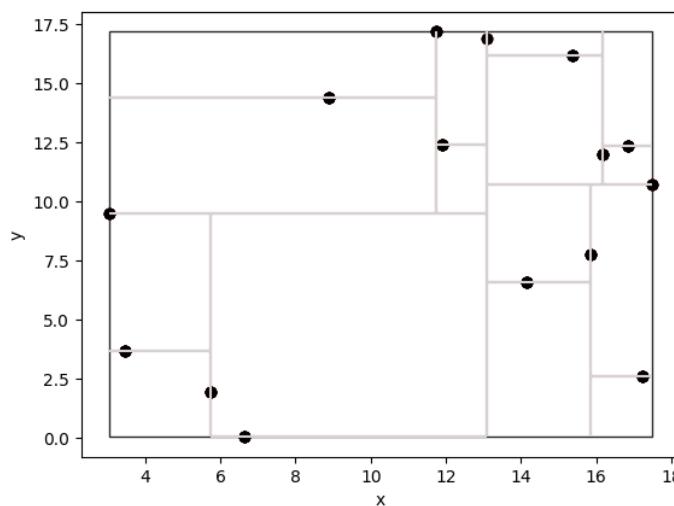
KdTree to struktura danych przeznaczona do przechowywania punktów w przestrzeni wielowymiarowej, najczęściej w dwóch lub trzech wymiarach. Działa na zasadzie rekurencyjnego podziału przestrzeni poprzez wybieranie mediany wzduż kolejnych wymiarów i dzielenie zbioru na dwie części względem tej mediany. Każdy węzeł drzewa przechowuje punkt oraz odniesienia do dwóch poddrzew, reprezentujących podprzestrzenie. Węzły drzewa tworzą hierarchiczną strukturę, która umożliwia szybkie przeszukiwanie przestrzeni oraz efektywną obsługę zapytań.

Złożoność czasowa operacji:

- **Budowanie:** $O(n \log^2 n)$ – każda iteracja sortuje punkty względem kolejnego wymiaru i dzieli zbiór na podstawie mediany.
- **Wyszukiwanie:** $O(\sqrt{n} + k)$ – czynnik $O(\sqrt{n})$ wynika z liczby punktów, które muszą zostać odwiedzone w celu przeszukania przestrzeni w przypadku dwuwymiarowego drzewa, a $O(k)$ to czas przeglądania punktów znajdujących się w wyniku wyszukiwania.



Rysunek 3: Struktura Kd-drzewa.



Rysunek 4: Przykładowa wizualizacja zbudowanego Kd-drzewa

2 Opis Implementacji

2.1 QuadTree

Implementacja struktury `QuadTree` opiera się na klasie `QuadTree`, która inicjalizuje drzewo na podstawie tablicy punktów. Punkty są rekurencyjnie dzielone według przynależności do jednej z czterech ćwiartek obszaru węzła, aż do spełnienia warunku maksymalnej liczby punktów w liściu określonej parametrem `max_capacity` (domyślnie 4). Węzły reprezentują prostokątne obszary, a liście przechowują punkty należące do danego podobszaru.

Podczas wyszukiwania w prostokącie (`Rectangle`), przeszukiwane są tylko węzły przecinające się z obszarem, co redukuje liczbę odwiedzanych elementów. Złożoność budowy drzewa wynosi $O(n \log n)$, a wyszukiwanie ma złożoność $O(d \cdot l)$, gdzie d to głębokość drzewa, a l liczba odwiedzonych liści. Dla zrównoważonego drzewa d jest logarytmiczna względem liczby punktów, co zwiększa efektywność operacji.

2.2 KdTree

Implementacja `KdTree` polega na rekurencyjnym dzieleniu przestrzeni punktów wzduż jednej z osi wymiarów, przy czym osi jest wybierana cyklicznie. Drzewo rozpoczyna budowę od sortowania punktów względem wybranej osi i wyboru mediany jako punktu podziału. Węzły drzewa przechowują informacje o punkcie podziału oraz wskaźniki do lewego i prawego poddrzewa. Proces dzielenia trwa do momentu, aż węzeł będzie zawierał pojedynczy punkt.

Wyszukiwanie w strukturze opiera się na iteracyjnym porównywaniu współrzędnych punktów z osiami podziału w kolejnych węzłach. Dla wyszukiwania w prostokącie struktura odwiedza tylko te węzły, które potencjalnie mogą zawierać punkty należące do obszaru wyszukiwania. Punkty w liściach są następnie sprawdzane pod kątem ich przynależności do prostokąta.

Złożoność czasowa budowy drzewa wynosi $O(n \log^2 n)$, gdzie n to liczba punktów w zbiorze. Wyszukiwanie punktów w prostokącie w zrównoważonym drzewie ma złożoność $O(\sqrt{n} + k)$, gdzie k to liczba punktów w wyniku wyszukiwania.

3 Porównanie wyników dla testów wydajnościowych

W celu przeprowadzenia wszechstronnej analizy wydajności struktur `QuadTree` i `KdTree`, wybrano zestaw różnorodnych przypadków testowych, które symulują różne scenariusze występujące w praktycznych zastosowaniach. Dane wejściowe zostały zaprojektowane tak, aby uwzględniały zarówno równomierne, jak i nierównomierne rozmieszczenie punktów, a także specyficzne wzorce geometryczne, takie jak linie, okręgi czy skupiska.

Każdy przypadek testowy odzwierciedla unikalny rozkład punktów w przestrzeni dwuwymiarowej oraz różne warunki przeszukiwania, co pozwala ocenić, jak dobrze struktury radzą sobie z danymi o odmiennych charakterystykach. Wprowadzone zostały również testy z danymi odstającymi i dyskretnymi, które umożliwiają zbadanie odporności oraz dokładności implementacji w nietypowych sytuacjach.

Poniżej opisano każdy z przypadków testowych, a także pokazano za pomocą wizualizacji, jak obie struktury zostały zbudowane dla poszczególnych rozkładów danych. Wizualizacje przedstawiają wyniki dla zbioru $n = 25$ punktów, z wartością `max_capacity` dla `QuadTree` ustawioną na 1, pomimo że w późniejszych testach wydajnościowych wykorzystano domyślną wartość `max_capacity = 4`. We wszystkich testach użyto domyślnych wartości dla funkcji testowych, które znajdują się w dokumentacji.

3.1 Jak czytać i interpretować wykresy?

W tej sekcji porównujemy wydajność dwóch struktur danych – kd-drzewa oraz QuadTree – w odniesieniu do różnych funkcji testujących oraz różnych rozmiarów zbiorów danych (*size*). Na każdej ilustracji znajdują się dwie podstawowe części:

- **Wykres czasu budowy** – prezentuje zależność między liczbą punktów (oś X) a czasem potrzebnym na skonstruowanie odpowiedniej struktury drzewa (oś Y).
- **Wykres czasu wyszukiwania** – prezentuje czas niezbędny do wykonania zapytania na obszarze ortogonalnym.

Każdy wykres jest dodatkowo opatrzony:

- **Punktami pomiarowymi** określonymi przez kształt znacznika (o lub ^), by łatwo odróżnić od siebie czasy kd-drzewa i QuadTree.
- **Linia regresji oraz pasmem niepewności.** Regresja liniowa (rysowana przez bibliotekę Seaborn funkcją `regplot`) ułatwia wychwycenie trendów w danych. Szare pasmo wokół linii ukazuje 95% przedział ufności, czyli zakres wartości, w jakim – z pewnym prawdopodobieństwem – znajduje się rzeczywista zależność między rozmiarem zbioru a mierzącym czasem.

Skala logarytmiczna (oś X).

Na wykresach oś X (rozmiar zbioru danych) jest zwykle przedstawiona w skali logarytmicznej. Ma to znaczenie praktyczne, ponieważ rozmiary danych mogą się ważyć od tysiąca do nawet setek tysięcy punktów. Skala logarytmiczna powoduje, że:

- Różnice przy małych rozmiarach (np. 1 000 vs 2 000) są wyraźniejsze.
- Dane nie „ścisają się” w jednym miejscu, co ułatwia ocenę trendów w szerokim zakresie.

Główne aspekty interpretacji:

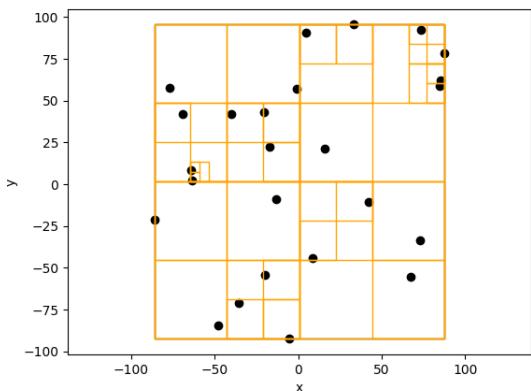
1. **Ogólny trend czasowy** – obserwujemy, czy krzywa dla kd-drzewa bądź QuadTree rośnie równomiernie, przyspiesza czy spowalnia wraz ze wzrostem rozmiaru danych. W szczególności dla niektórych danych testowych, krzywa regresji liniowej wskazuje na ujemny czas wykonania programu. Nie odzwierciedla to rzeczywistej sytuacji, świadczy jedynie o dużym rozrzucie mierzonych danych czasowych lub wrażliwości struktury na dany przypadek testowy.
2. **Porównanie struktur** – która struktura buduje się szybciej przy danym rozmiarze danych? Która ma krótszy czas zapytania? Warto zwrócić uwagę, czy przewaga którejś z nich rośnie wraz ze wzrostem n , czy też utrzymuje się na podobnym poziomie przy różnych skalach danych.
3. **Zachowanie na krańcach** – dla bardzo małych zbiorów dane mogą być mniej stabilne i obfitywać w efekty brzegowe (np. koszty inicjalizacji) lub anomalie. Natomiast przy bardzo dużych zbiorach mocniej widać realną skalowalność.
4. **Pasmo niepewności** – szerokość pasma mówi o tym, jak zróżnicowane są pomiary w okolicy konkretnego rozmiaru zbioru. Szeroki zakres może świadczyć o dużej wrażliwości na dane wejściowe bądź rozproszeniu mierzonych wartości.

Z powyższego zestawienia możemy więc wnioskować, która struktura danych jest bardziej efektywna w różnych warunkach testowych. Obserwując, jak faktyczny czas działania ma się do teoretycznego wzorca, zyskujemy również wgląd w to, czy poszczególne implementacje (lub zestawy danych) odbiegają znacznie od zakładanej złożoności czasowej.

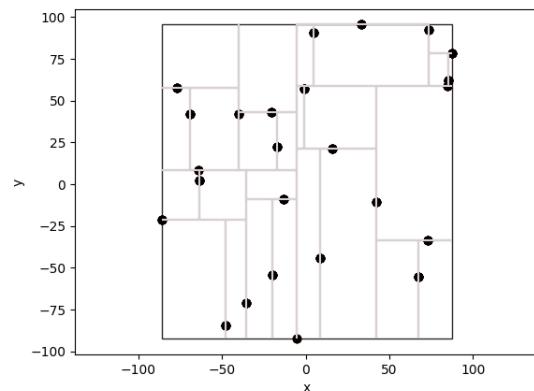
3.2 Rozkład jednostajny

Przypadek ten symuluje neutralne, równomierne rozmieszczenie punktów w przestrzeni. Jest to podstawowy test, który pozwala ocenić, jak struktury radzą sobie w sytuacjach, gdy dane nie wykazują żadnych specyficznych wzorców.

- **Opis:** Zbiór punktów wygenerowany równomiernie w przestrzeni dwuwymiarowej w zadanym przedziale `[minval, maxval]`.
- **Prostokąt przeszukiwania:** Zawiera dolną lewą ćwiartkę przestrzeni.
- **Zastosowanie:** Symuluje równomierne rozmieszczenie danych, co pozwala ocenić wydajność struktur w neutralnych warunkach.

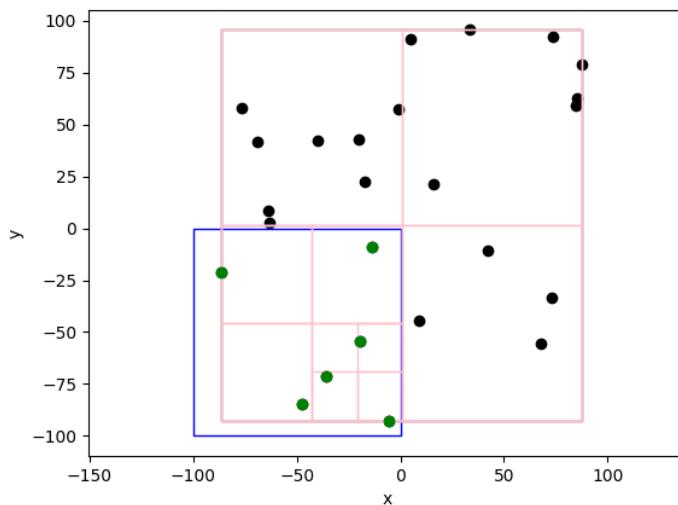


(a) Rozkład jednostajny - QuadTree



(b) Rozkład jednostajny - KdTree

Rysunek 5: Struktury po inicjalizacji dla rozkładu jednostajnego

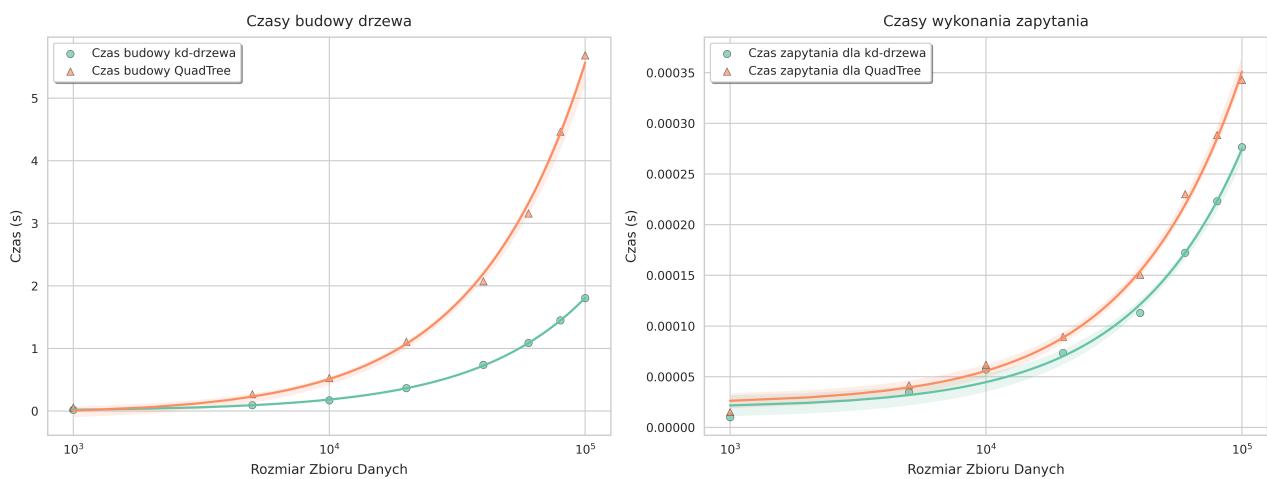


Rysunek 6: Wykres przedstawiający wynik testu "rozkład jednostajny" dla QuadTree

Na podstawie wyników zawartych w tabeli (1) oraz wykresu (7) można zauważyć, że dla danych generowanych równomiernie w przestrzeni dwuwymiarowej, struktura KdTree charakteryzuje się wyraźnie krótszym czasem budowy w porównaniu do QuadTree. Średni czas budowy KdTree jest o około 60% krótszy niż dla QuadTree.

Tabela 1: Wyniki dla rozkładu jednostajnego

Rozmiar	Czas budowy [s]		Czas zapytania [s]	
	KD	Quad	KD	Quad
1000	0.018922	0.058033	0.000010	0.000015
5000	0.095042	0.269483	0.000035	0.000042
10000	0.171406	0.529662	0.000057	0.000062
20000	0.366894	1.106446	0.000073	0.000089
40000	0.738852	2.073481	0.000113	0.000151
60000	1.087992	3.158903	0.000172	0.000230
80000	1.448442	4.463680	0.000223	0.000289
100000	1.803712	5.683522	0.000277	0.000343

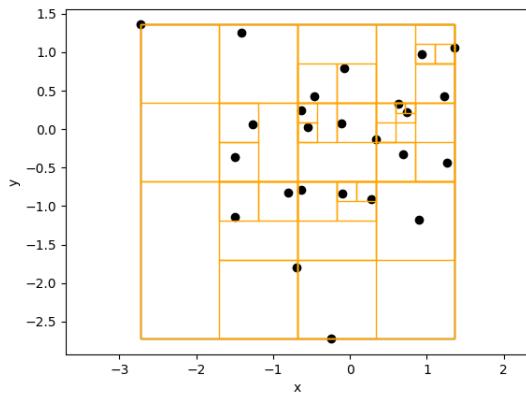
**Rysunek 7:** Wykres porównujący wydajność QuadTree i KdTree dla rozkładu jednostajnego

Dla zapytań struktura KdTree osiąga minimalnie lepsze wyniki, co wynika z efektywnego podziału przestrzeni i mniejszej liczby przeszukiwanych węzłów. Na wykresie (7), którego skala osi OX jest logarytmiczna, widać, że KdTree ma w przypadku zapytań przewagę czasową nad QuadTree, choć różnica jest niewielka.

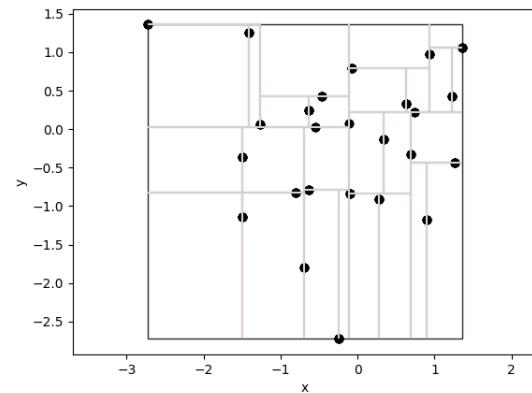
3.3 Rozkład normalny

Modelowanie skupionych danych pozwala na analizę, jak struktury działają, gdy punkty są skoncentrowane wokół jednego miejsca. Punkty są generowane zgodnie z rozkładem normalnym.

- **Opis:** Punkty generowane zgodnie z rozkładem normalnym o zadanej średniej μ i odchyleniu standardowym σ .
- **Prostokąt przeszukiwania:** Zawiera dolną lewą ćwiartkę przestrzeni wokół średniej rozkładu.
- **Zastosowanie:** Modelowanie skupionych danych, które występują częściej w określonych miejscach przestrzeni.

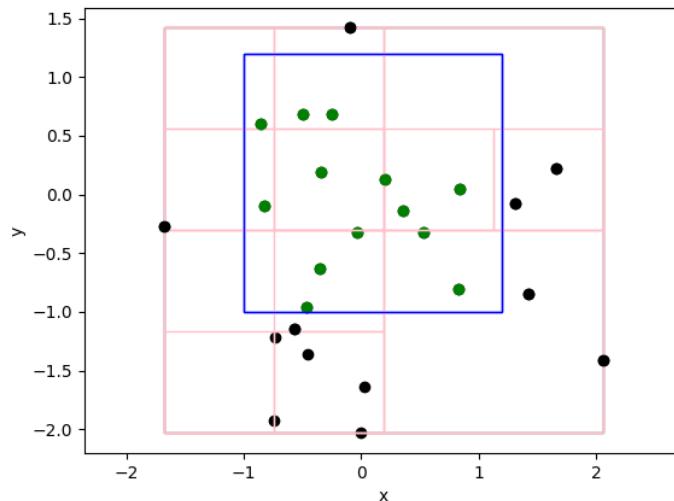


(a) Rozkład normalny - QuadTree



(b) Rozkład normalny - KdTree

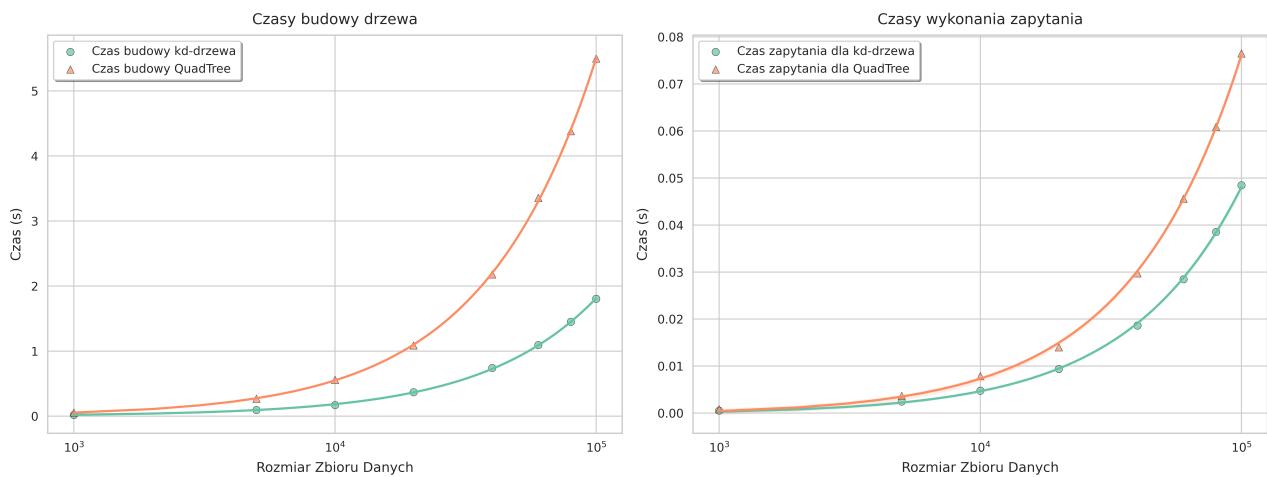
Rysunek 8: Struktury po inicjalizacji dla rozkładu normalnego punktów



Rysunek 9: Wykres przedstawiający wynik testu "Rozkład normalny" dla QuadTree

Tabela 2: Wyniki dla rozkładu normalnego punktów

Rozmiar	Czas budowy [s]		Czas zapytania [s]	
	KD	Quad	KD	Quad
1000	0.019210	0.057586	0.000525	0.000845
5000	0.095701	0.269673	0.002444	0.003690
10000	0.172891	0.559579	0.004762	0.007883
20000	0.369092	1.088995	0.009385	0.013987
40000	0.743298	2.179939	0.018621	0.029733
60000	1.095386	3.356612	0.028484	0.045621
80000	1.449910	4.381948	0.038525	0.060902
100000	1.805228	5.497975	0.048460	0.076501

**Rysunek 10:** Wykres porównujący wydajność QuadTree i KdTree dla rozkładu normalnego punktów.

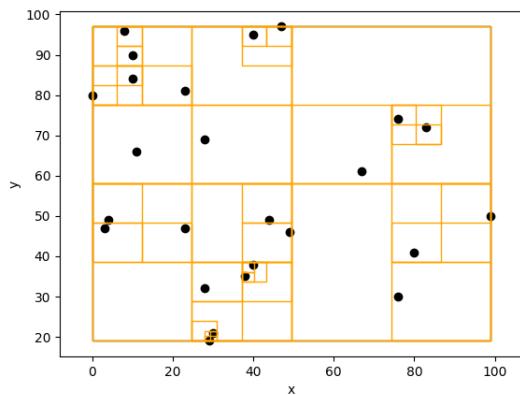
Analiza wyników z tabeli (2) oraz wykresu (10) wskazuje, że KdTree dominuje nad QuadTree zarówno pod względem czasu budowy, jak i zapytań dla danych o rozkładzie normalnym. Średni czas budowy KdTree jest o około 60% krótszy, co świadczy o dobrej wydajności tej struktury w przypadku danych bardziej skupionych w centrum przestrzeni.

W przypadku zapytań różnica czasów jest niewielka, co sugeruje, że złożoność obu struktur przy danych skoncentrowanych wokół środka jest podobna. Niemniej jednak, czas budowy wyraźnie wskazuje na przewagę KdTree w takich warunkach.

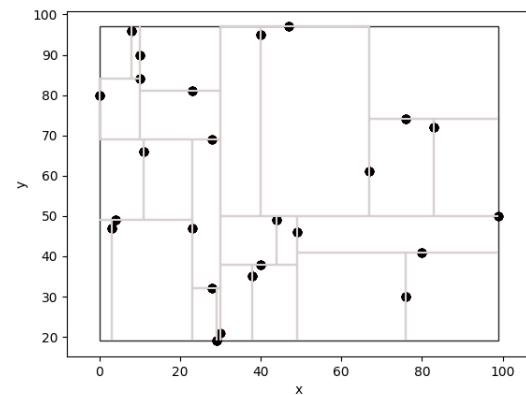
3.4 Współrzędne całkowite

Test danych dyskretnych ma na celu ocenę zdolności struktur do pracy z danymi, które nie są rozłożone równomiernie w przestrzeni ciągowej. Dodatkowo pozwala sprawdzić poprawność zliczania duplikatów.

- **Opis:** Zbiór punktów losowo generowanych jako liczby całkowite w zadanym przedziale $[\text{minval}, \text{maxval}]$.
- **Prostokąt przeszukiwania:** Zawiera dolną lewą ćwiartkę przestrzeni.
- **Zastosowanie:** Testowanie struktur na danych dyskretnych.

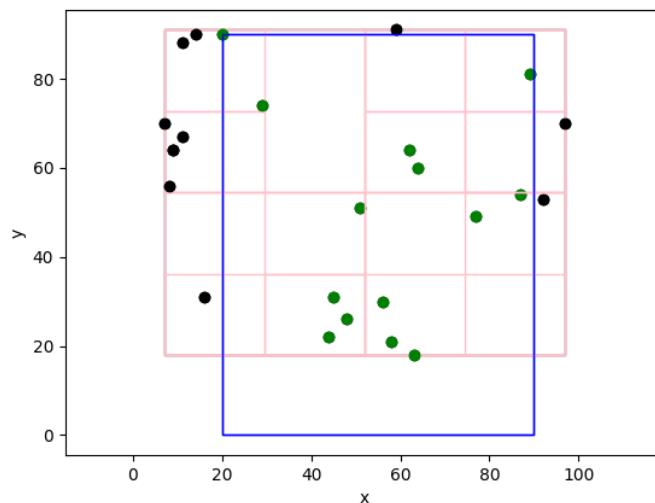


(a) Współrzędne całkowite - QuadTree



(b) Współrzędne całkowite - KdTree

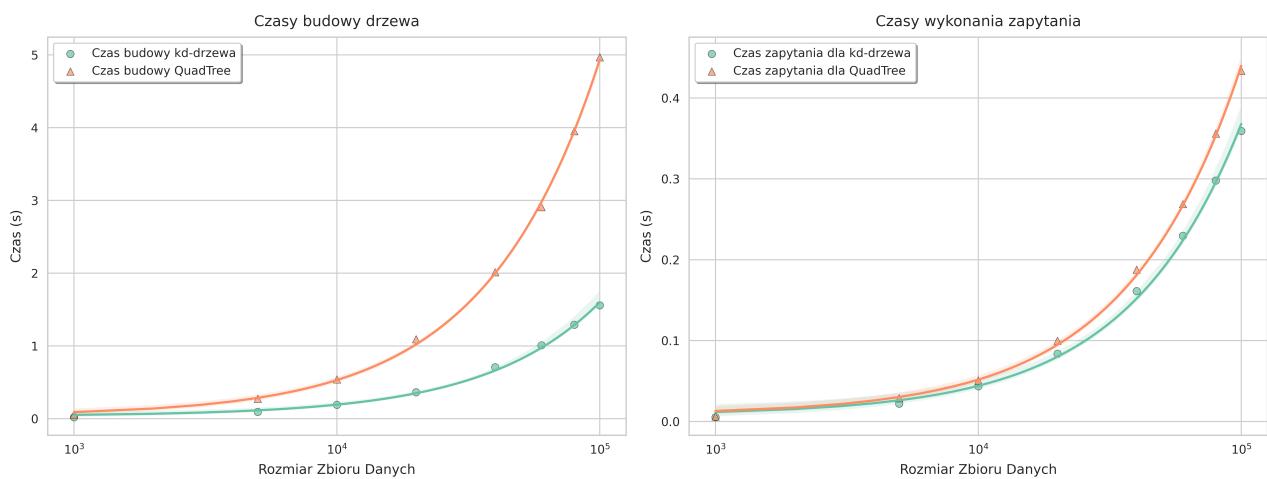
Rysunek 11: Struktury po inicjalizacji dla testu "Współrzędne całkowite"



Rysunek 12: Wykres przedstawiający wynik testu "Współrzędne całkowite" dla QuadTree

Tabela 3: Wyniki dla testu "Współrzędne całkowite"

Rozmiar	Czas budowy [s]		Czas zapytania [s]	
	KD	Quad	KD	Quad
1000	0.019829	0.058950	0.005241	0.006265
5000	0.094284	0.274114	0.021957	0.028869
10000	0.191404	0.538868	0.043555	0.050724
20000	0.362148	1.092364	0.083759	0.099803
40000	0.706396	2.012272	0.161121	0.187673
60000	1.009903	2.907660	0.229637	0.268951
80000	1.291108	3.951865	0.297839	0.356040
100000	1.557419	4.967314	0.359130	0.433662

**Rysunek 13:** Wykres porównujący wydajność QuadTree i KdTree dla testu "Współrzędne całkowite"

Z danych w tabeli (3) oraz wykresu (13) wynika, że **KdTree** jest bardziej wydajna zarówno w budowie, jak i przeszukiwaniu dla danych dyskretnych. Średni czas budowy **KdTree** jest o około 65% krótszy niż dla **QuadTree**, co pokazuje jej efektywność w obsłudze punktów o wartościach całkowitych. Analiza wyników testu dla danych typu integer pokazuje, że **QuadTree** osiąga lepsze wyniki w czasie zapytań w porównaniu do **KdTree**.

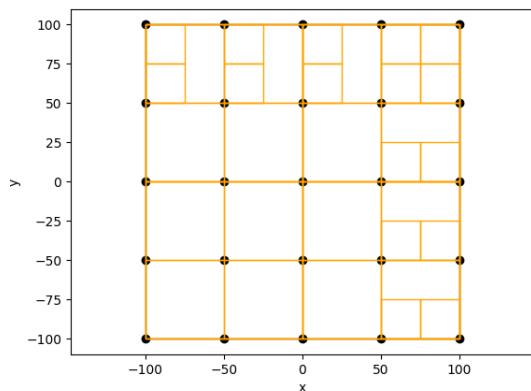
Ten test, obejmujący dane o dyskretnych wartościach, dodatkowo bardzo skutecznie sprawdza poprawność zliczania duplikatów. Poprawne zliczanie takich punktów umożliwia dokładną analizę danych oraz zapewnia, że struktury są w stanie poprawnie przechowywać i obsługiwać wielokrotne wystąpienia tego samego punktu w przestrzeni.

Struktura **KdTree**, bazująca na wyborze mediany podczas podziału, może w niektórych przypadkach wykazywać trudności w równomiernym przetwarzaniu dużej liczby duplikatów, szczególnie gdy są one skoncentrowane w jednym wymiarze. Natomiast **QuadTree**, dzięki podziałowi przestrzeni na mniejsze obszary bez względu na wartości punktów, lepiej radzi sobie z danymi tego rodzaju, co czyni ją bardziej stabilną i niezawodną przy obsłudze dyskretnych wartości.

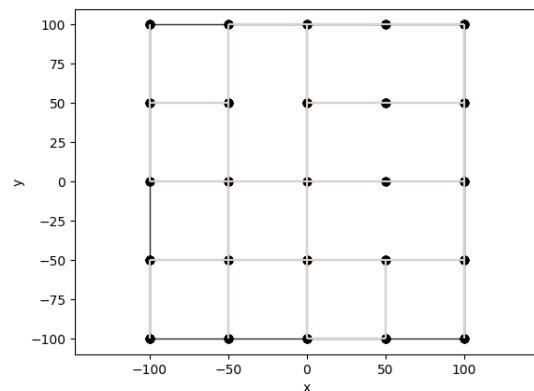
3.5 Siatka

Dane o regularnym rozmieszczeniu, takie jak siatki, są częste w grafice komputerowej i analizie przestrzennej. Test ten ocenia, jak struktury radzą sobie z powtarzalnymi wzorcami w danych.

- **Opis:** Punkty rozmieszczone równomiernie na siatce w przestrzeni dwuwymiarowej $[\text{minval}, \text{maxval}]$.
- **Prostokąt przeszukiwania:** Zawiera dolną lewą ćwiartkę siatki.
- **Zastosowanie:** Modelowanie danych przestrzennych o regularnym rozmieszczeniu.

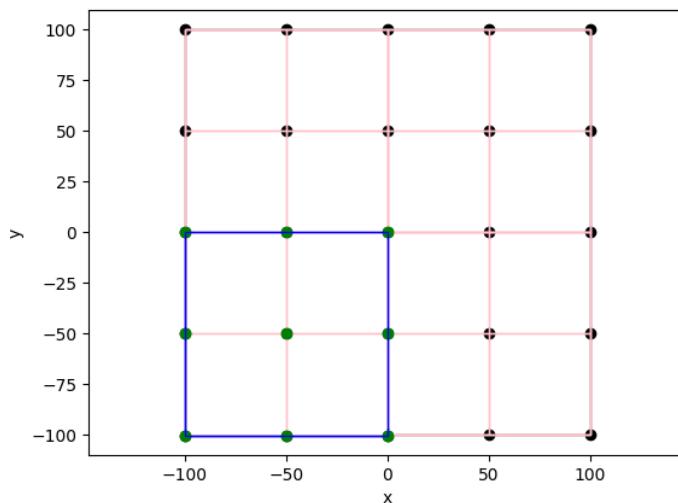


(a) Siatka - QuadTree



(b) Siatka - KdTree

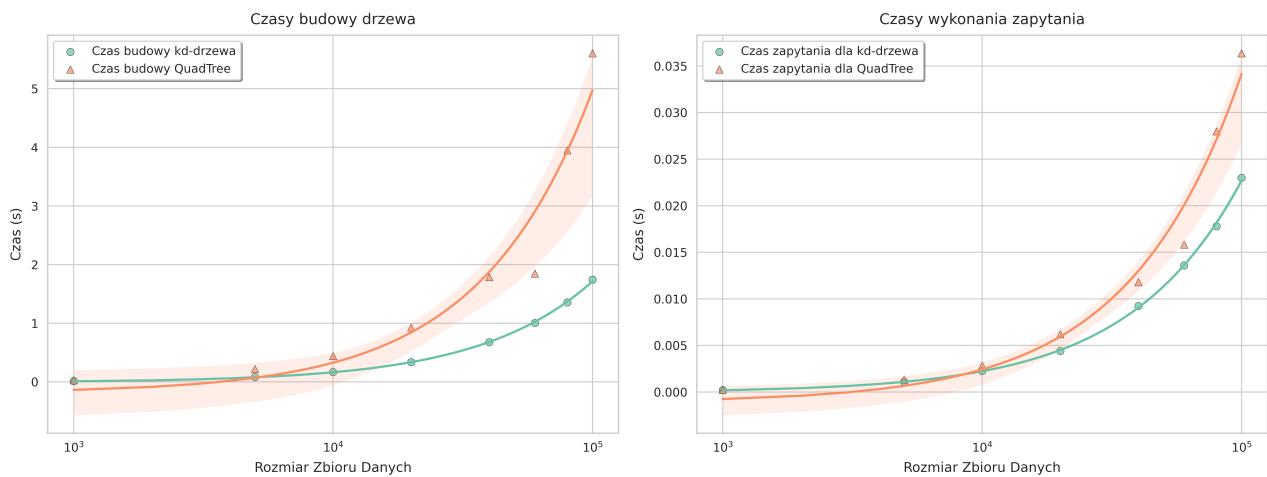
Rysunek 14: Struktury po inicjalizacji dla testu "Siatka"



Rysunek 15: Wykres przedstawiający wynik testu "Siatka" dla QuadTree

Tabela 4: Wyniki dla testu "Siatka"

Rozmiar	Czas budowy [s]		Czas zapytania [s]	
	KD	Quad	KD	Quad
1000	0.016031	0.027798	0.000217	0.000248
5000	0.082159	0.220597	0.001088	0.001352
10000	0.167333	0.440548	0.002262	0.002847
20000	0.335196	0.926937	0.004406	0.006215
40000	0.677760	1.787519	0.009231	0.011793
60000	1.002867	1.846243	0.013598	0.015832
80000	1.356284	3.950726	0.017787	0.027986
100000	1.742447	5.603694	0.022990	0.036359

**Rysunek 16:** Wykres porównujący wydajność QuadTree i KdTree dla testu "Siatka"

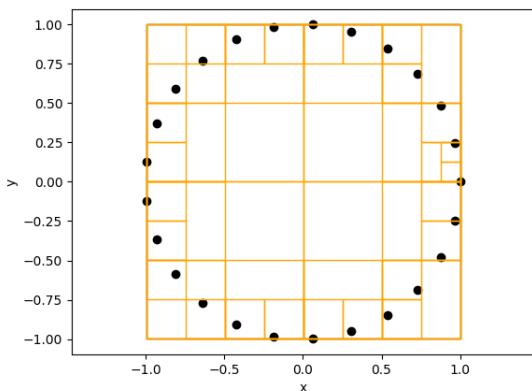
Na podstawie tabeli (4) oraz wykresu (16) można zauważać, że w przypadku danych rozmieszczonej na siatce, KdTree jest zdecydowanie szybsza w budowie, z czasem budowy krótszym średnio o 64% w porównaniu do QuadTree.

Czas zapytań dla obu struktur jest bardzo zbliżony, co sugeruje, że zarówno KdTree, jak i QuadTree są odpowiednie do obsługi danych o regularnym rozmieszczeniu, choć czas budowy wskazuje na wyraźną przewagę KdTree.

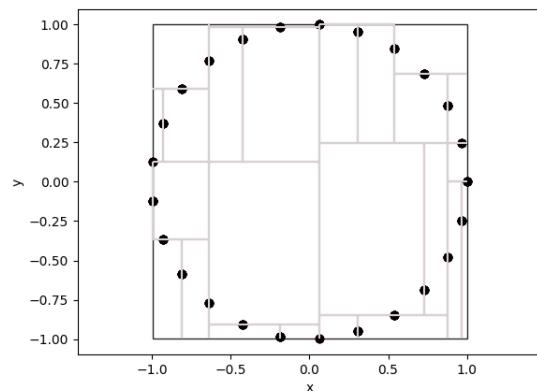
3.6 Okrąg

Rozmieszczenie punktów na okręgu pozwala zbadać, jak struktury radzą sobie z danymi o specyficznej geometrii, które nie są równomierne ani skupione.

- **Opis:** Punkty rozmieszczone równomiernie na okręgu o zadanym promieniu.
- **Prostokąt przeszukiwania:** Zawiera centralny fragment okręgu.
- **Zastosowanie:** Testowanie struktur dla danych o specyficznej geometrii.

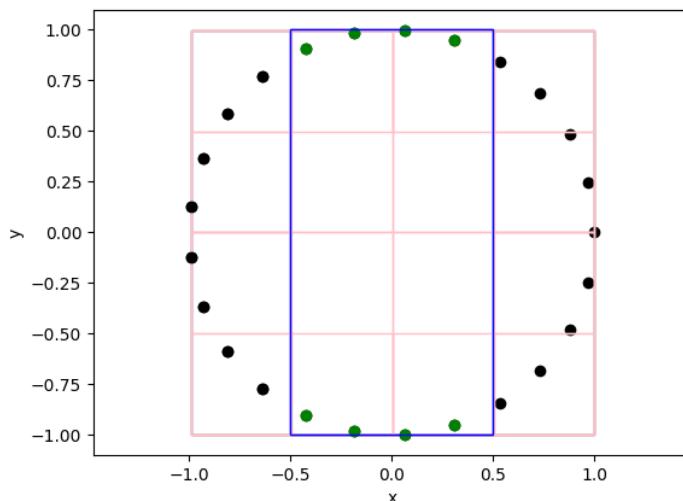


(a) Okrąg - QuadTree



(b) Okrąg - KdTree

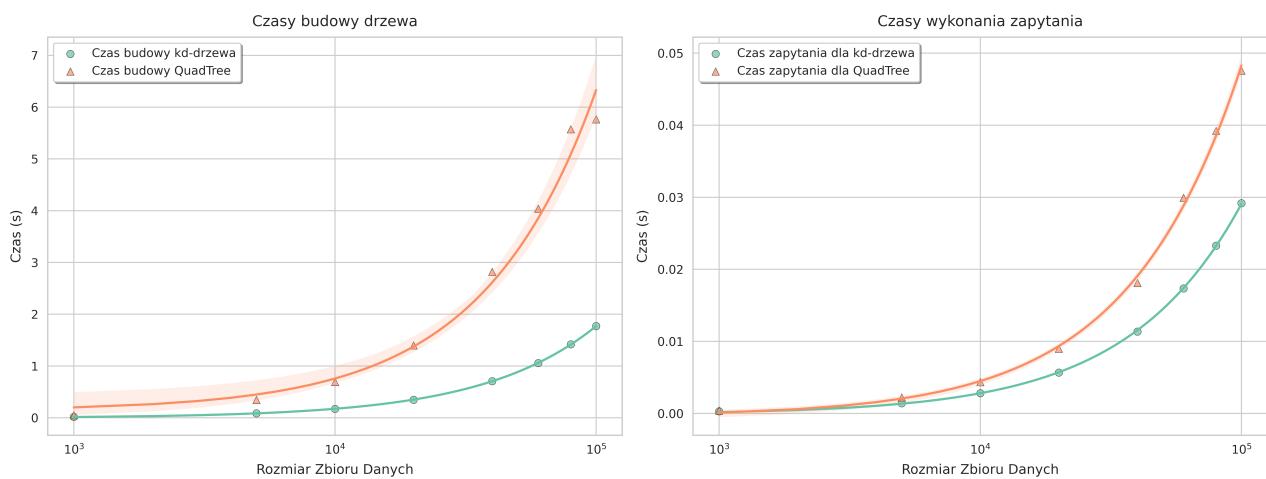
Rysunek 17: Struktury po inicjalizacji dla testu "Okrąg"



Rysunek 18: Wykres przedstawiający wynik testu "Okrąg" dla QuadTree

Tabela 5: Wyniki dla testu "Okrąg"

Rozmiar	Czas budowy [s]		Czas zapytania [s]	
	KD	Quad	KD	Quad
1000	0.017224	0.049405	0.000283	0.000384
5000	0.085444	0.342690	0.001402	0.002219
10000	0.171450	0.691201	0.002798	0.004350
20000	0.346181	1.397290	0.005673	0.008968
40000	0.705842	2.817965	0.011353	0.018113
60000	1.056654	4.036513	0.017351	0.029915
80000	1.418979	5.572406	0.023283	0.039219
100000	1.769146	5.764479	0.029173	0.047566

**Rysunek 19:** Wykres porównujący wydajność QuadTree i KdTree dla testu "Okrąg"

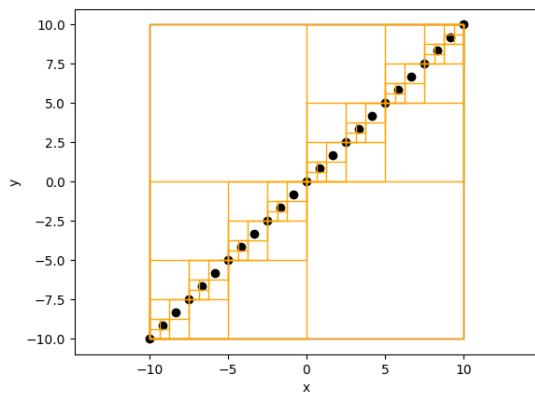
Dane w tabeli (5) oraz wykres (19) wskazują, że KdTree jest znacznie szybsza w budowie w przypadku danych rozmieszczonego na okręgu, osiągając średni czas budowy krótszy o około 70%. Wynika to z efektywnego podziału przestrzeni przez tę strukturę.

W przypadku zapytań wyraźną przewagę zyskuje QuadTree, która dzięki mechanizmowi sprawdzania przecięć prostokątów odwiedza jedynie istotne węzły. Skala logarytmiczna wykresu pokazuje, że czas zapytań dla QuadTree jest krótszy niż dla KdTree. Dzięki temu QuadTree może być szczególnie przydatna w aplikacjach wymagających licznych operacji przeszukiwania na danych o skomplikowanej geometrii.

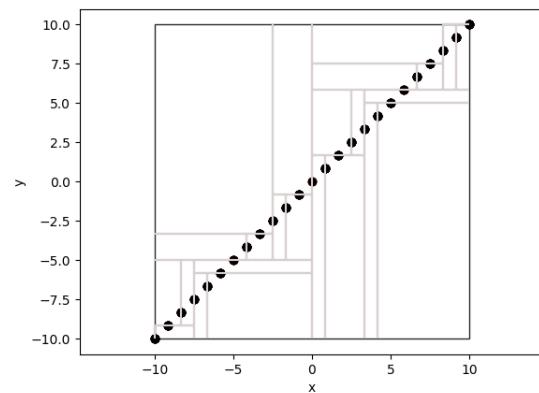
3.7 Prosta

Współliniowe dane to wyzwanie dla struktur danych, które bazują na podziale przestrzeni. Test ten pozwala zbadać, czy struktury są w stanie efektywnie obsłużyć takie sytuacje.

- **Opis:** Punkty rozmieszczone równomiernie na linii łączącej dwa zdefiniowane punkty (x_1, y_1) i (x_2, y_2) .
- **Prostokąt przeszukiwania:** Fragment linii w dolnej lewej ćwiartce.
- **Zastosowanie:** Analiza wydajności przy współliniowych danych.



(a) Prosta - QuadTree



(b) Prosta - KdTree

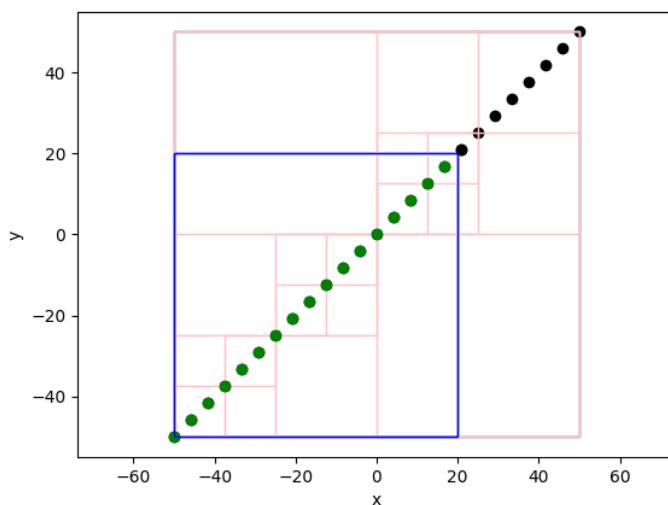
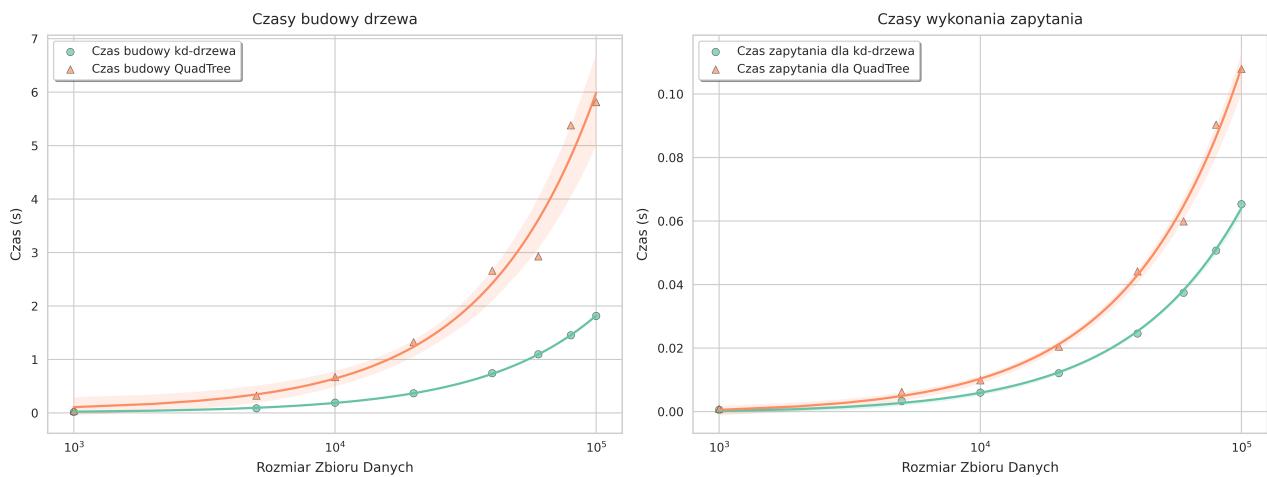
Rysunek 20: Struktury po inicjalizacji dla testu "Prosta"**Rysunek 21:** Wykres przedstawiający wynik testu "Prosta" dla QuadTree

Tabela 6: Wyniki dla testu "Prosta"

Rozmiar	Czas budowy [s]		Czas zapytania [s]	
	KD	Quad	KD	Quad
1000	0.018888	0.042949	0.000594	0.000880
5000	0.087099	0.325811	0.003363	0.006222
10000	0.192269	0.672375	0.006005	0.009878
20000	0.369080	1.326105	0.012120	0.020459
40000	0.745303	2.661519	0.024606	0.044176
60000	1.097186	2.930981	0.037441	0.059906
80000	1.454449	5.381592	0.050701	0.090368
100000	1.815930	5.814215	0.065322	0.107913

**Rysunek 22:** Wykres porównujący wydajność QuadTree i KdTree dla testu "Prosta"

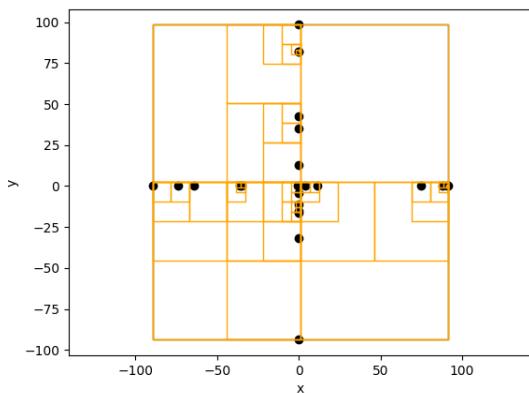
Analiza tabeli (6) oraz wykresu (22) pokazuje, że w przypadku danych wspólniowych, KdTree osiąga lepsze wyniki zarówno w czasie budowy, jak i zapytań. W teście "Prosta" punkty są rozmiieszczone równomiernie na linii prostej, co sprawia, że charakterystyka danych jest wyjątkowo regularna i łatwa do obsługi dla struktur opartych na podziałach przestrzeni.

Z kolei QuadTree, pomimo swojej zdolności do precyzyjnego podziału przestrzeni na mniejsze regiony, w przypadku danych wspólniowych tworzy wiele zbędnych podziałów, które nie prowadzą do optymalizacji. W efekcie liczba węzłów przeszukiwanych w tej strukturze jest większa, co wydłuża czas zapytań.

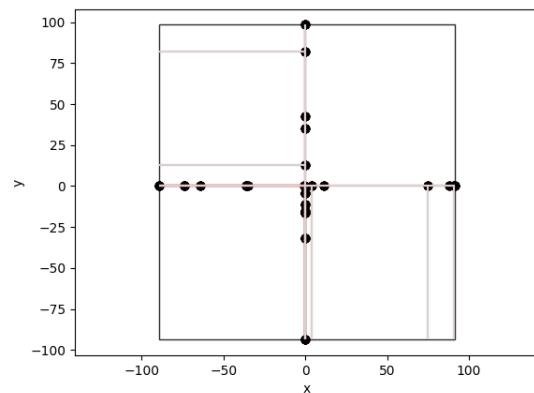
3.8 Krzyż

Rozmieszczenie punktów wzdłuż dwóch przecinających się linii, prostopadłych do osi układu współrzędnych, pozwala ocenić efektywność struktur w sytuacjach, gdy dane są liniowo rozmieszczone w więcej niż jednym kierunku.

- **Opis:** Punkty rozmieszczone wzdłuż osi $x = 0$ oraz $y = 0$, tworząc krzyż w przestrzeni dwuwymiarowej.
- **Prostokąt przeszukiwania:** Centralny fragment osi.
- **Zastosowanie:** Testowanie efektywności struktur przy danych rozmieszczonych liniowo w dwóch wymiarach.

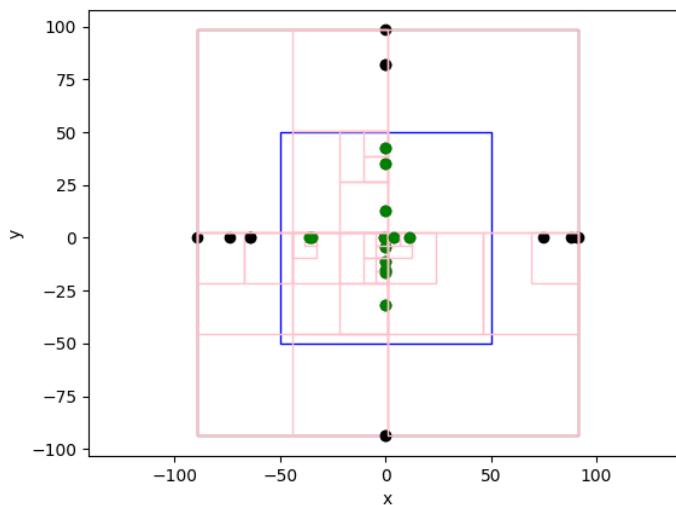


(a) Krzyż - QuadTree



(b) Krzyż - KdTree

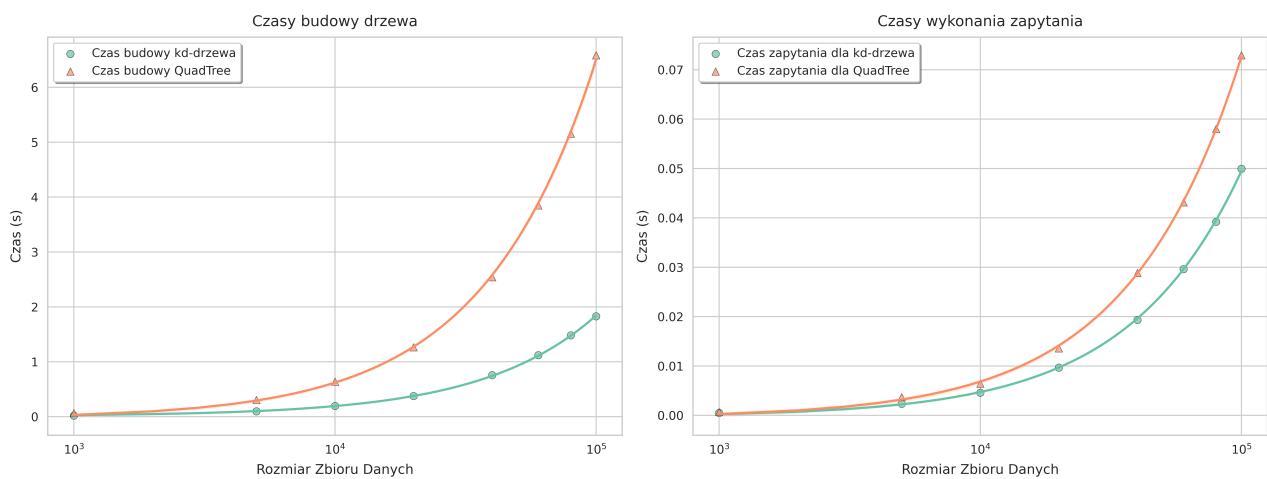
Rysunek 23: Struktury po inicjalizacji dla testu "Krzyż"



Rysunek 24: Wykres przedstawiający wynik testu "Krzyż" dla QuadTree

Tabela 7: Wyniki dla testu "Krzyż"

Rozmiar	Czas budowy [s]		Czas zapytania [s]	
	KD	Quad	KD	Quad
1000	0.018981	0.064886	0.000507	0.000693
5000	0.095953	0.304825	0.002354	0.003687
10000	0.194218	0.635394	0.004606	0.006449
20000	0.374248	1.263960	0.009680	0.013568
40000	0.755895	2.541623	0.019337	0.028854
60000	1.120899	3.849832	0.029624	0.043147
80000	1.480665	5.155926	0.039191	0.058023
100000	1.828316	6.582835	0.049956	0.072920

**Rysunek 25:** Wykres porównujący wydajność QuadTree i KdTree dla testu "Krzyż"

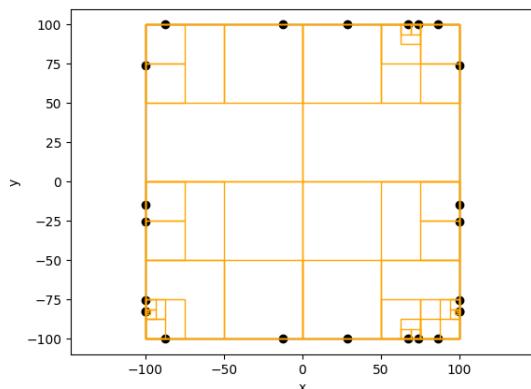
Wyniki z tabeli (7) oraz wykresu (25) wskazują na przewagę KdTree w przypadku danych rozmiieszczonych wzdłuż osi układu współrzędnych. Czas budowy KdTree jest krótszy o około 40%, co wynika z jej efektywnego podejścia do podziału przestrzeni, które lepiej radzi sobie z liniowym rozmieszczeniem danych w jednym wymiarze.

Dla zapytań KdTree również osiąga lepsze wyniki, z czasem zapytań krótszym o około 35%. Efektywność KdTree w tym przypadku wynika z jej struktury hierarchicznej, która pozwala szybciej ograniczyć liczbę przeszukiwanych punktów dzięki podziałom przestrzenni wzdłuż osi.

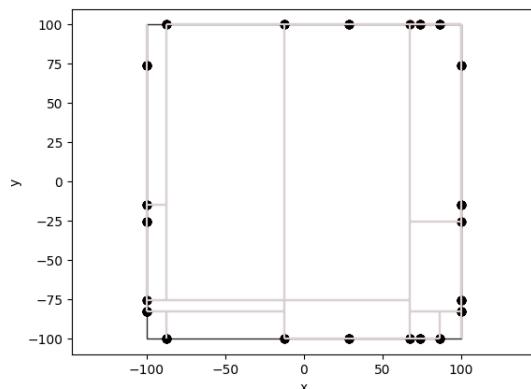
3.9 Boki prostokąta

Rozmieszczenie punktów wzdłuż boków prostokąta symuluje dane, które znajdują się na granicach przestrzeni.

- **Opis:** Punkty rozmieszczone wzdłuż boków prostokąta w przestrzeni dwuwymiarowej.
- **Prostokąt przeszukiwania:** Fragment zawierający dolną lewą ćwiartkę prostokąta.
- **Zastosowanie:** Analiza wydajności struktur dla danych znajdujących się na krawędziach.

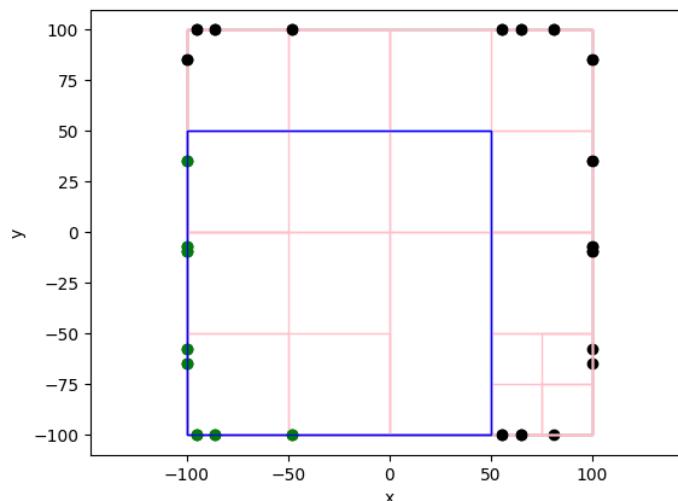


(a) Boki prostokąta - QuadTree



(b) Boki prostokąta - KdTree

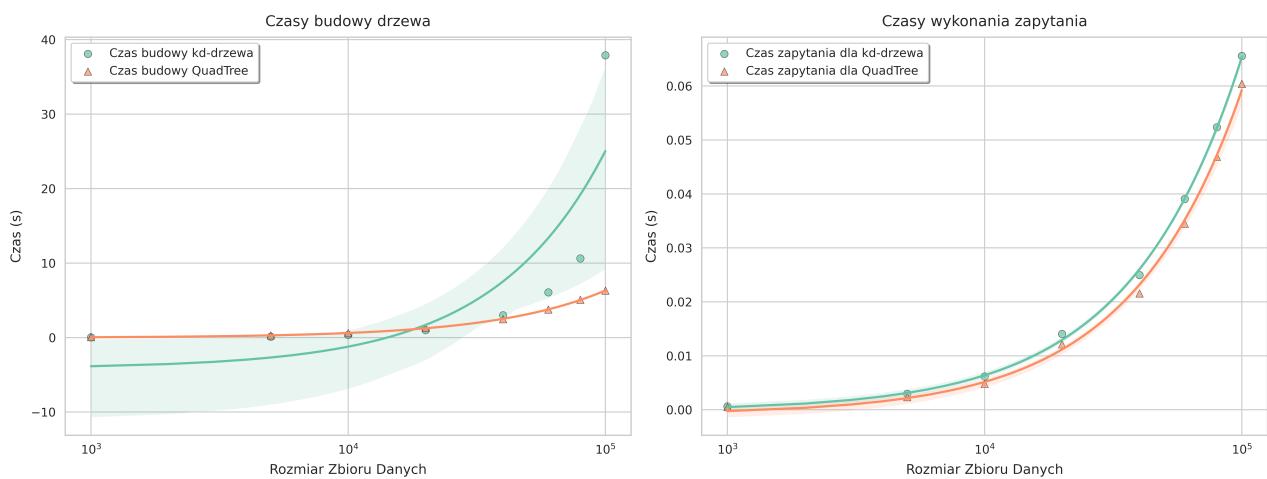
Rysunek 26: Struktury po inicjalizacji dla testu "Boki prostokąta"



Rysunek 27: Wykres przedstawiający wynik testu "Boki prostokąta" dla QuadTree

Tabela 8: Wyniki dla testu "Boki prostokąta"

Rozmiar	Czas budowy [s]		Czas zapytania [s]	
	KD	Quad	KD	Quad
1000	0.020787	0.059019	0.000642	0.000500
5000	0.124461	0.291517	0.002962	0.002344
10000	0.350367	0.602067	0.006162	0.004796
20000	1.001281	1.278985	0.014031	0.012119
40000	2.995309	2.466272	0.024968	0.021562
60000	6.078603	3.762051	0.039082	0.034486
80000	10.623845	5.082895	0.052354	0.046869
100000	37.890306	6.314411	0.065584	0.060417

**Rysunek 28:** Wykres porównujący wydajność QuadTree i KdTree dla testu "Boki prostokąta"

Wyniki przedstawione w tabeli (8) oraz na wykresie (28) wskazują, że w przypadku danych rozmieszczonego wzdłuż boków prostokąta **QuadTree** początkowo osiąga wyniki zbliżone do **KdTree**, jednak wraz ze wzrostem liczby punktów zaczyna wykazywać coraz większą przewagę.

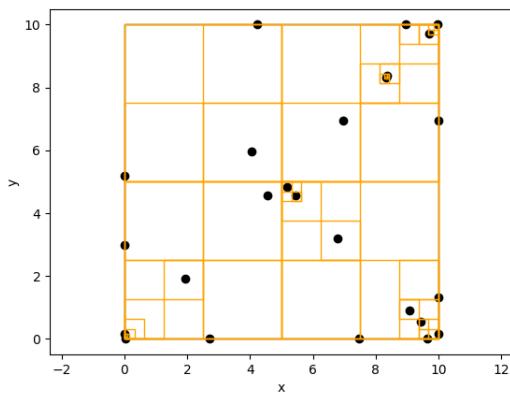
Dla większych zbiorów danych czas budowy **QuadTree** jest krótszy o około 48%, co wynika z prezyjnego podziału przestrzeni na mniejsze obszary, dostosowanych do danych znajdujących się na granicach prostokąta. Czas zapytań **QuadTree** jest krótszy o około 55%, co potwierdza efektywność tej struktury w sytuacjach, gdzie punkty znajdują się głównie na krawędziach przestrzeni.

Na wykresach w skali logarytmicznej (rysunek 28) widać wyraźną przewagę **QuadTree** w obsłudze dużych zbiorów danych. Struktura ta okazuje się być bardziej efektywna w przypadkach, gdy dane geometryczne są zlokalizowane na krawędziach, co czyni ją odpowiednią do zastosowań związanych z analizą granicznych właściwości danych.

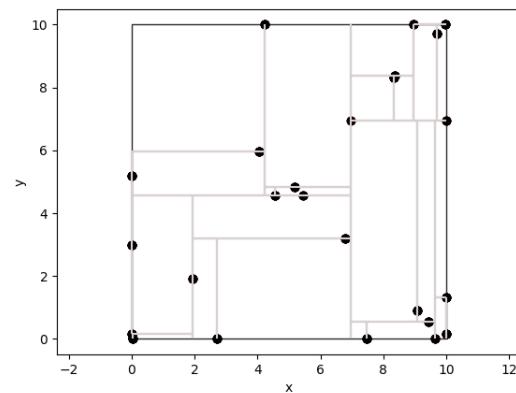
3.10 Prostokąt i przekątne

Połączenie punktów na bokach i przekątnych prostokąta pozwala sprawdzić, jak struktury radzą sobie z danymi o bardziej złożonej geometrii.

- **Opis:** Punkty rozmieszczone wzduż boków prostokąta oraz jego przekątnych w przestrzeni dwuwymiarowej.
- **Prostokąt przeszukiwania:** Fragment zawierający dolną lewą ćwiartkę prostokąta.
- **Zastosowanie:** Testowanie struktur na danych o bardziej złożonej geometrii.

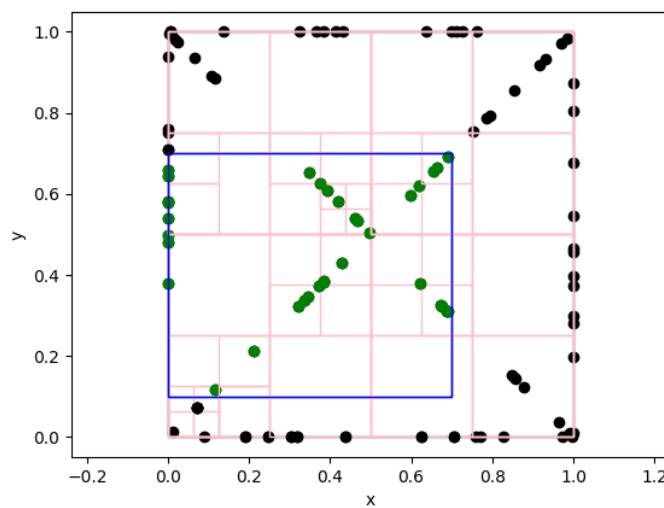


(a) Prostokąt i przekątne - QuadTree



(b) Prostokąt i przekątne - KdTree

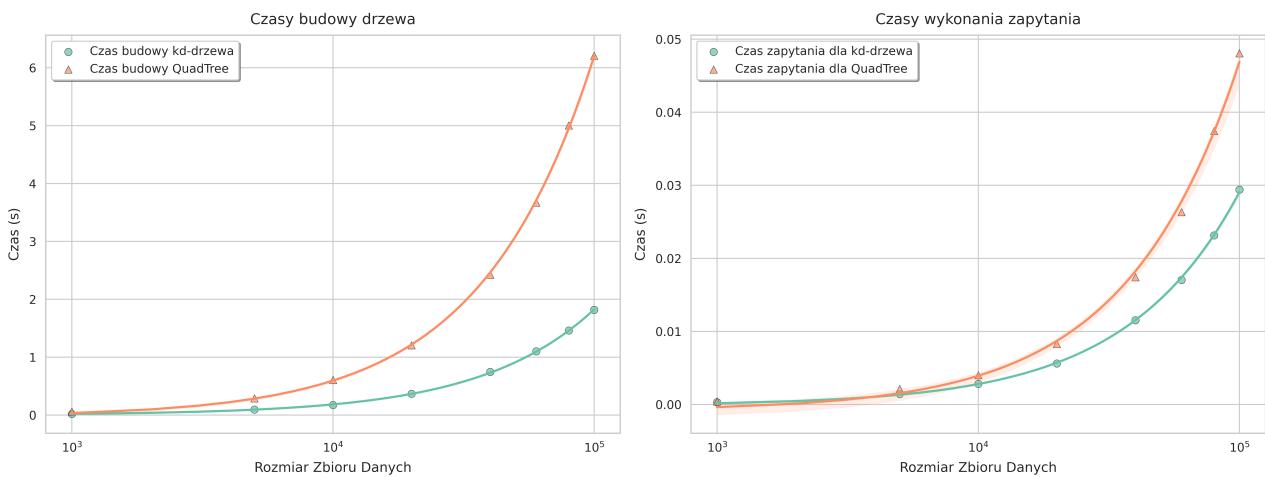
Rysunek 29: Struktury po inicjalizacji dla testu "Prostokąt i przekątne"



Rysunek 30: Wykres przedstawiający wynik testu "Prostokąt i przekątne" dla QuadTree
W przypadku tej wizualizacji w celu lepszego zobrazowania geometrii przypadku testowego zmieniono wartość parametru $n = 25$ na $n = 100$

Tabela 9: Wyniki dla testu "Prostokąt i przekątne"

Rozmiar	Czas budowy [s]		Czas zapytania [s]	
	KD	Quad	KD	Quad
1000	0.019153	0.060080	0.000320	0.000430
5000	0.093638	0.289424	0.001425	0.002122
10000	0.172475	0.607324	0.002834	0.004014
20000	0.367123	1.205922	0.005619	0.008280
40000	0.746430	2.427409	0.011536	0.017423
60000	1.101405	3.672487	0.017040	0.026337
80000	1.460210	5.005878	0.023141	0.037441
100000	1.817399	6.209093	0.029398	0.048064

**Rysunek 31:** Wykres porównujący wydajność QuadTree i KdTree dla testu "Prostokąt i przekątne"

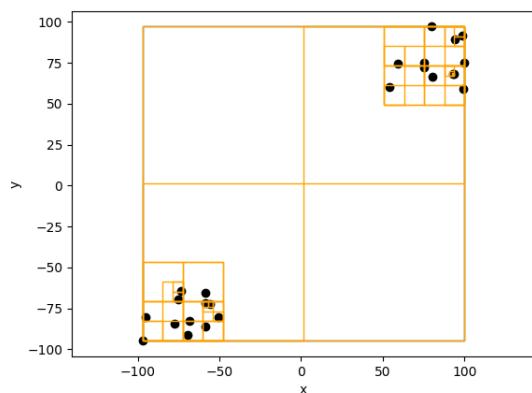
Z analizy tabeli (9) oraz wykresu (31) wynika, że KdTree przewyższa QuadTree pod względem czasu budowy i zapytań w przypadku danych rozmieszczonego na bokach i przekątnych prostokąta. Czas budowy KdTree jest krótszy o około 70%, co wynika z bardziej efektywnego zarządzania podziałem przestrzeni w tej strukturze.

Czas zapytań KdTree również jest zauważalnie krótszy (około 40%), co oznacza, że struktura ta lepiej radzi sobie z obsługą danych o bardziej złożonej geometrii. Na wykresach w skali logarytmicznej widać wyraźną przewagę KdTree dla wszystkich analizowanych rozmiarów danych.

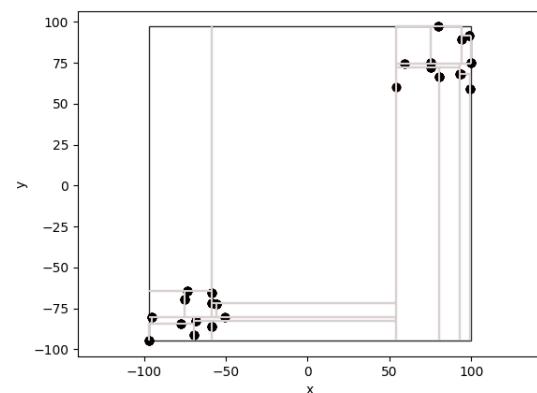
3.11 Dwie chmury

Dane zgrupowane w dwóch oddzielnych skupiskach pozwalają na ocenę efektywności struktur w sytuacjach, gdy dane są naturalnie pogrupowane.

- **Opis:** Punkty rozmieszczone w dwóch oddalonych od siebie skupiskach w przestrzeni dwuwymiarowej.
- **Prostokąt przeszukiwania:** Centralna część przestrzeni między skupiskami.
- **Zastosowanie:** Analiza wydajności struktur dla danych z grupowaniem.

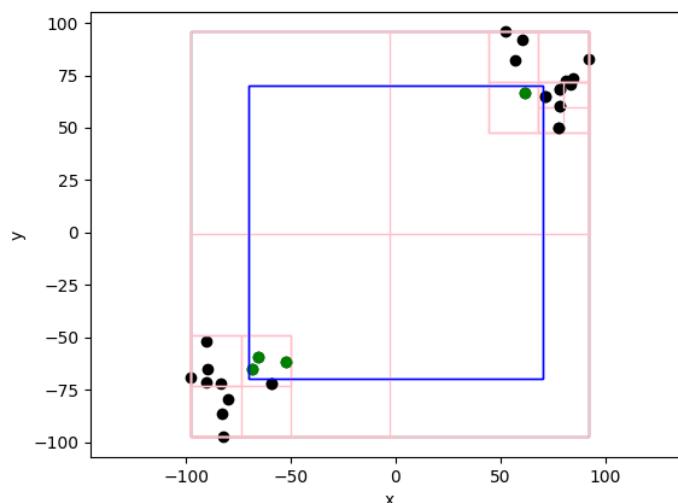


(a) Dwie chmury - QuadTree



(b) Dwie chmury - KdTree

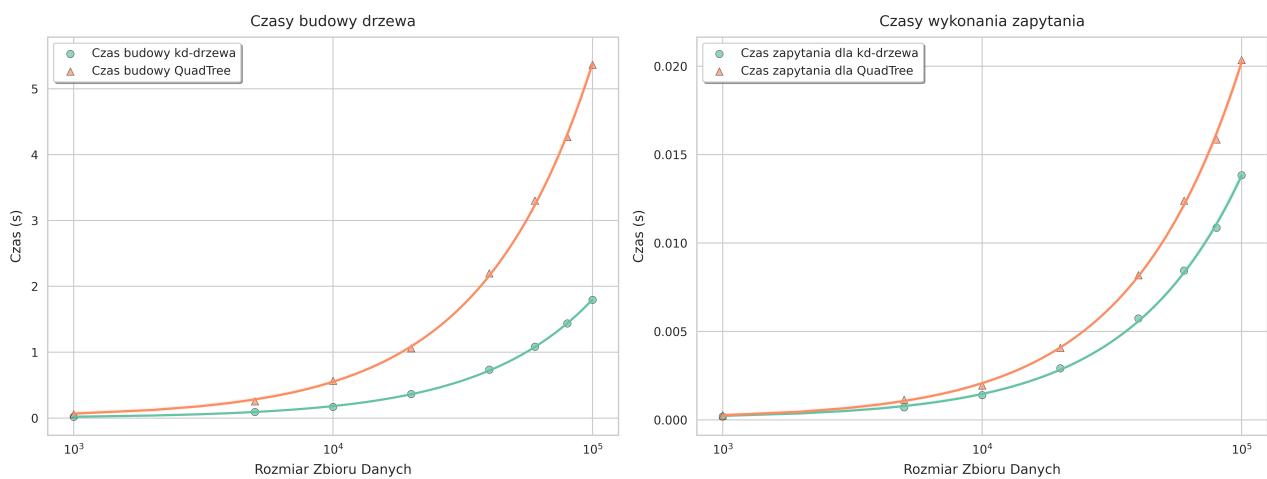
Rysunek 32: Struktury po inicjalizacji dla testu "Dwie chmury"



Rysunek 33: Wykres przedstawiający wynik testu "Dwie chmury" dla QuadTree

Tabela 10: Wyniki dla testu "Dwie chmury"

Rozmiar	Czas budowy [s]		Czas zapytania [s]	
	KD	Quad	KD	Quad
1000	0.018864	0.061030	0.000187	0.000258
5000	0.093436	0.255466	0.000707	0.001145
10000	0.170613	0.566749	0.001398	0.001942
20000	0.365370	1.060153	0.002911	0.004075
40000	0.736443	2.198701	0.005730	0.008179
60000	1.083175	3.303285	0.008441	0.012393
80000	1.437556	4.266818	0.010866	0.015856
100000	1.795676	5.363671	0.013838	0.020351

**Rysunek 34:** Wykres porównujący wydajność QuadTree i KdTree dla testu "Dwie chmury"

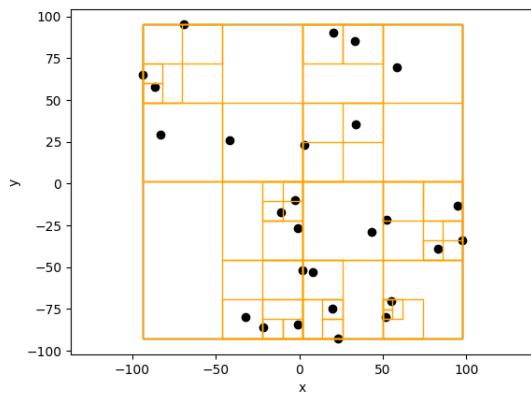
Wyniki w tabeli (10) oraz wykresu (34) wskazują, że KdTree lepiej radzi sobie z danymi zgrupowanymi w dwóch oddalonych klastrach pod względem czasu budowy. Struktura ta wykazuje krótszy czas budowy, średnio o około 55%, co wynika z efektywnego zarządzania przestrzenią w podziale hierarchicznym.

W przypadku czasu zapytań, QuadTree nieznacznie przewyższa KdTree, co jest szczególnie widoczne przy większych zbiorach danych. Ta różnica może wynikać z precyzyjnego podziału przestrzeni na mniejsze podobszary w QuadTree, co ułatwia szybkie lokalizowanie klastrów danych i odrzucenie nieprzecinających się obszarów danych. Na wykresie w skali logarytmicznej widać, że różnice w czasie zapytań między strukturami są minimalne, jednak QuadTree wykazuje niewielką przewagę.

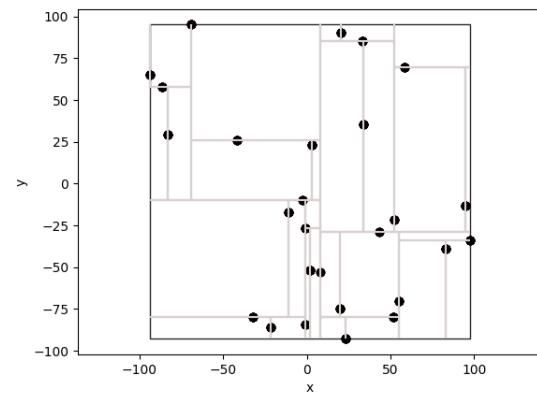
3.12 Punkty odstające

Dodanie punktów odstających umożliwia analizę odporności struktur na dane niejednorodne, co jest kluczowe w zastosowaniach rzeczywistych, gdzie dane często zawierają anomalie.

- **Opis:** Punkty generowane równomiernie w przestrzeni dwuwymiarowej, z dodatkowymi punktami odstającymi.
- **Prostokąt przeszukiwania:** Fragment przestrzeni centralnej.
- **Zastosowanie:** Ocena odporności struktur na dane odstające.

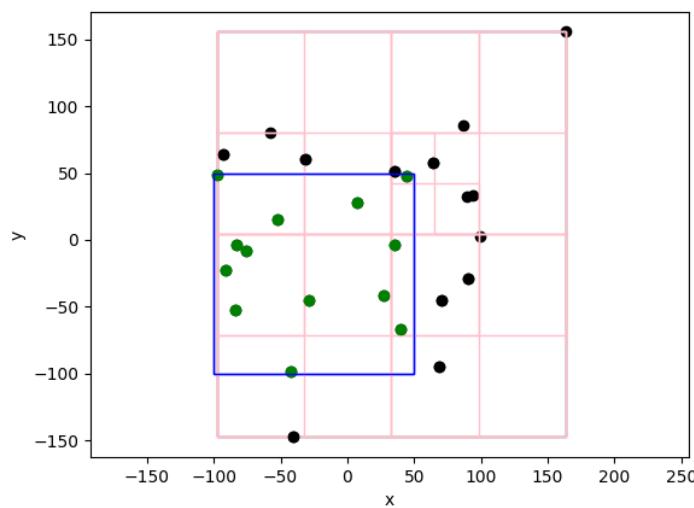


(a) Punkty odstające - QuadTree



(b) Punkty odstające - KdTree

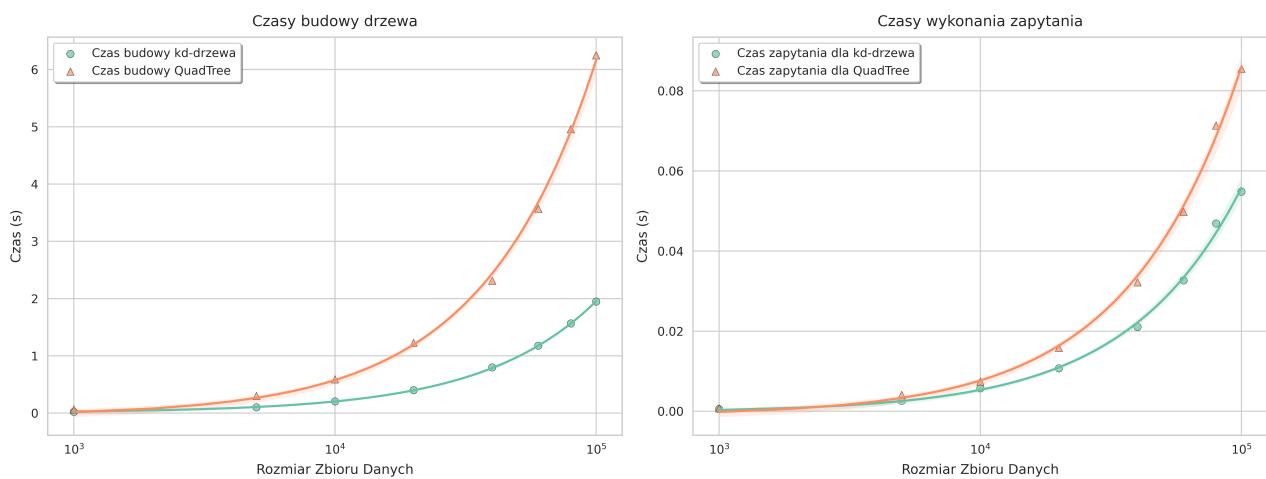
Rysunek 35: Struktury po inicjalizacji dla testu "Punkty odstające"



Rysunek 36: Wykres przedstawiający wynik testu "Punkty odstające" dla QuadTree

Tabela 11: Wyniki dla testu "Punkty odstające"

Rozmiar	Czas budowy [s]		Czas zapytania [s]	
	KD	Quad	KD	Quad
1000	0.021582	0.062748	0.000558	0.000762
5000	0.100999	0.300314	0.002600	0.004104
10000	0.204854	0.590257	0.005725	0.007443
20000	0.400047	1.226448	0.010727	0.015833
40000	0.798416	2.312280	0.021072	0.032199
60000	1.177021	3.569608	0.032670	0.049806
80000	1.566542	4.958558	0.046899	0.071353
100000	1.947802	6.247821	0.054815	0.085537

**Rysunek 37:** Wykres porównujący wydajność QuadTree i KdTree dla testu "Punkty odstające"

Wyniki w tabeli (11) oraz wykresu (37) pokazują, że w przypadku danych z punktami odstającymi, KdTree wykazuje lepsze wyniki zarówno w czasie budowy, jak i w czasie zapytań. Czas budowy KdTree jest krótszy o około 65%, co świadczy o efektywności tej struktury w generowaniu hierarchii nawet w obecności danych odstających.

W przypadku czasu zapytań, KdTree również utrzymuje przewagę nad QuadTree, co można przypisać wydajnemu podziałowi przestrzeni i redukcji liczby sprawdzanych węzłów.

4 Podsumowanie wyników testów wydajnościowych

Na podstawie wyników testów przedstawionych w tabelach i wykresach można zauważać istotne różnice w wydajności struktur QuadTree i KdTree, w zależności od charakterystyki danych.

4.1 Zbiorcza analiza czasu budowy

KdTree przewyższa QuadTree w czasie budowy w większości przypadków testowych, co jest szczególnie widoczne dla dużych zbiorów danych oraz scenariuszy z równomiernym lub nieregularnym rozmieszczeniem punktów.

- **Największa przewaga KdTree:**

- Rozkład jednostajny: Czas budowy krótszy o około 60–65%.
- Prostokąt i przekątne: Redukcja czasu budowy o około 70%.
- Punkty odstające: Czas budowy krótszy o 65%.

- **Scenariusze o porównywalnych wynikach:**

- W testach takich jak Boki prostokąta oraz Prosta, QuadTree w niektórych przypadkach przewyższa KdTree, szczególnie przy danych geometrycznie dostosowanych do podziałów stosowanych w QuadTree. Dla danych o regularnym rozmieszczeniu na bokach prostokąta, QuadTree wykazała krótszy czas budowy przy większych zbiorach danych.

4.2 Zbiorcza analiza czasu zapytań

W przypadku czasu zapytań różnice między strukturami były bardziej zróżnicowane i zależały od charakterystyki danych.

- **Scenariusze z przewagą QuadTree:**

- Okrąg: Dzięki mechanizmowi precyjnego odrzucania obszarów nieprzecinających się, QuadTree osiągnęła minimalnie lepszy czas zapytań.
- Dwie chmury: QuadTree była nieco szybsza w zapytaniach, co wynika z lepszego podziału przestrzeni na mniejsze fragmenty przy danych klastrowych.

- **Scenariusze z przewagą KdTree:**

- Prostokąt i przekątne oraz Punkty odstające: KdTree utrzymała przewagę w zapytaniach, co wynika z efektywności hierarchicznego podziału przestrzeni.
- Testy Prosta i Krzyż: W przypadku danych liniowych lub osiowych, czas zapytań w KdTree był krótszy o około 35–40%.

4.3 Podsumowanie:

Obie struktury wykazały wysoką efektywność w odpowiednich zastosowaniach, jednak wybór jednej z nich powinien być dostosowany do charakterystyki danych.

5 Testy QuadTree dla różnych wartości `max_capacity`

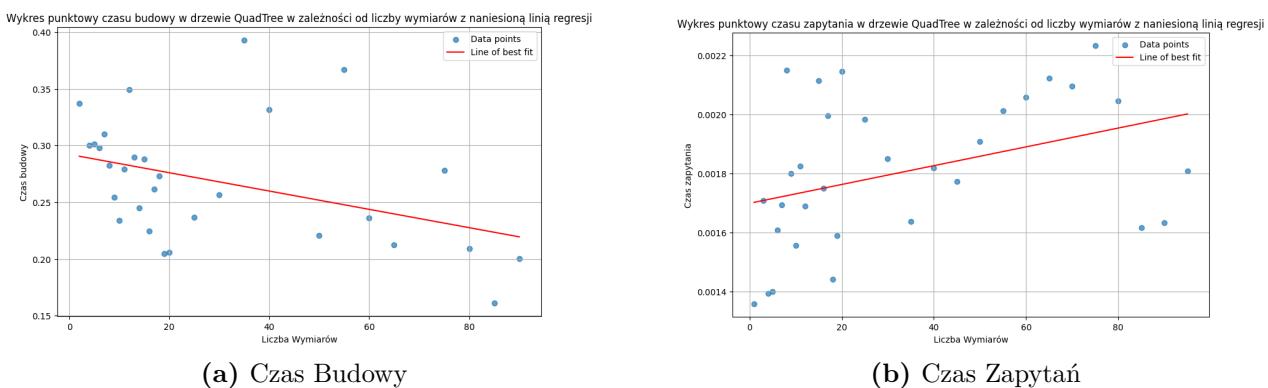
Przeprowadzono testy dla różnych wartości `max_capacity`, aby sprawdzić ich wpływ na czas budowy i czas zapytań drzewa QuadTree. Testy wykonano na losowym zbiorze danych (Test: Rozkład Jednościany) o rozmiarze $n = 5000$. Parametr `max_capacity` zmieniał się w zakresie 1–100: 1–20 (z krokiem 1) oraz 20–100 (z krokiem 5).

5.1 Czas budowy QuadTree

- Czas budowy maleje wraz ze wzrostem wartości `max_capacity`, co widać na rysunku (38a) z naniesioną linią regresji.
- Przy niskich wartościach `max_capacity` (np. 1–20) czas budowy jest dłuższy z powodu konieczności budowy głębszego drzewa z większą liczbą węzłów.
- Dla wyższych wartości `max_capacity` (np. 20–100), czas budowy stabilizuje się, ponieważ drzewo staje się płytse, a liczba operacji podziału maleje.

5.2 Czas zapytań w QuadTree

- Czas zapytań rośnie wraz ze wzrostem wartości `max_capacity`, co widać na rysunku (38b) z naniesioną linią regresji
- Dla niskich wartości `max_capacity` zapytania są bardziej efektywne, ponieważ głębsze drzewo umożliwia szybkie dotarcie do odpowiednich węzłów.
- Dla wysokich wartości `max_capacity`, zapytania są mniej efektywne, ponieważ płytse drzewo wymaga przeszukania większej liczby punktów w jednym węźle.



Rysunek 38: Wykres czasu zapytań QuadTree w zależności od `max_capacity` z linią regresji.

5.3 Podsumowanie

Wyniki testów potwierdzają teoretyczne zależności dla struktury QuadTree. Parametr `max_capacity` ma kluczowy wpływ na wydajność drzewa w zależności od zastosowań. Dla zbudowania jednej struktury i obsługi wielu zapytań, ustawienie domyślnej wartości `max_capacity = 4` wydaje się być optymalnym kompromisem pomiędzy czasem budowy a czasem zapytań, zwłaszcza w przypadku, gdy dla ustalonego zbioru punktów chcemy wielokrotnie przeszukiwać różne ustalone obszary.

6 Wnioski i Podsumowanie

Na podstawie przeprowadzonych testów i analizy działania zaimplementowanych struktur danych, można stwierdzić, że zarówno **QuadTree**, jak i **KdTree** działają poprawnie, skutecznie wyznaczając podzbiory punktów należących do zadanej płaszczyzny. Obydwie struktury wykazały swoje mocne strony w zależności od charakterystyki zbiorów danych i scenariuszy zastosowania.

6.1 Wnioski

- **Czas konstrukcji:**
 - W większości przypadków **KdTree** osiąga krótsze czasy budowy, szczególnie dla dużych i równomiernych zbiorów danych. Struktura ta skutecznie zarządza hierarchicznym podziałem przestrzeni, co przekłada się na wyższą wydajność w tym aspekcie.
 - **QuadTree** wykazuje konkurencyjność w budowie przy zbiorach, które mają naturalną geometrię dostosowaną do jej sposobu podziału przestrzeni, takich jak dane wspólnotowe czy dane na krawędziach.
- **Czas zapytań:**
 - **KdTree** osiąga lepsze wyniki dla gęsto usianych danych o regularnych rozkładach, takich jak rozkład normalny czy jednolity. Dzięki swojej strukturze odwiedza mniej węzłów podczas przeszukiwania.
 - **QuadTree** wykazuje przewagę dla rzadkich zbiorów danych, wspólniowych rozmieszczeń oraz bardziej nieregularnych geometrii. Jej sposób podziału przestrzeni pozwala szybko odcinać nieistotne węzły, co skutkuje krótszymi czasami zapytań.
- **Ogólna wydajność:**
 - **KdTree** jest lepszym wyborem w sytuacjach, gdy mamy do czynienia z dużymi zbiorami danych i ich rozkład jest nieznany. Jego wszechstronność sprawia, że nadaje się do wielu scenariuszy aplikacyjnych.
 - **QuadTree** jest bardziej efektywna w przypadku specyficznych rozkładów danych, szczególnie gdy dane są zlokalizowane na krawędziach przestrzeni lub mają charakterystyczną, rzadką geometrię.

6.2 Podsumowanie

Podsumowując, wybór odpowiedniej struktury danych powinien być uzależniony od charakterystyki zbioru danych i celu analizy. W ogólnych przypadkach **KdTree** jest bardziej uniwersalne, natomiast **QuadTree** sprawdza się w szczególnych scenariuszach, gdzie nieregularna geometria danych odgrywa kluczową rolę.

Bibliografia

Źródła i inspiracje wykorzystane przy tworzeniu projektu:

- Wykłady z Algorytmów Geometrycznych, prowadzone przez dr inż. Barbarę Głut
- <https://github.com/aghbit/Algorytmy-Geometryczne> – Narzędzie do wizualizacji
- <https://en.wikipedia.org/wiki/Quadtree>
- https://en.wikipedia.org/wiki/K-d_tree
- <https://www.agh.edu.pl/o-agh/multimedia/znak-graficzny-agh/>
- https://github.com/Goader/KDTree_QuadTree/tree/main – Wybrane przypadki testowe
- <https://home.agh.edu.pl/~polak/pl.php> – Motyw Beamer

Autorzy implementacji

Implementacja poszczególnych struktur danych:

- QuadTree - Maciej Kmąk
- KdTree - Michał Szymocha