AGH Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Algorytmy Geometryczne - Projekt

Przeszukiwanie obszarów ortogonalnych – Quadtree oraz Kd-drzewa

Algorytmy geometryczne

5 stycznia 2025

Dariusz Marecik

Piotr Sękulski

Spis treści

1. Wstęp ... 3
1.1. Cel projektu ... 3
1.2. Dane techniczne ... 3
1.2.1. Dane techniczne środowiska programistycznego ... 3
1.2.2. Dane techniczne sprzętowe ... 3
2. Dokumentacja ... 3
2.1. Point ... 3
2.2. Rectangle ... 5
2.3. Kd_tree ... 7
2.3.1. KdTreeNode ... 7
2.3.2. KdTree ... 9
2.4. Kd_tree_visualiser ... 9
2.5. Quad_tree ... 11
2.6. Quadtree_vis ... 12
2.7. generate_tests ... 13
2.8. quad_tree_tests ... 16
2.9. Kd_tree_test ... 18
2.10. Brute_Force ... 20
3. Opis użytkowania ... 21
3.1. Narzędzie do testowania ... 21
3.1.1. Instrukcja wykonywania testów ... 21
3.1.2. Instrukcja tworzenia dodatkowych testów poprawnościowych ... 21
3.2. Instrukcja użycia KdTree ... 21
3.2.1. Budowa ... 21
3.2.2. Przeszukiwanie obszaru ... 21
3.2.3. Sprawdzenie czy punkt jest zawarty w strukturze: ... 22
3.3. Instrukcja użycia KdTree z wizualizacją ... 22
3.3.1. Tworzenie wizualizacji tworzenia drzewa: ... 22
3.3.2. Tworzenie wizualizacji przeszukiwania drzewa ... 22
3.3.3. Wyświetlanie końcowej wizualizacji ... 23
3.3.4. Wyświetlenie gifu z krokową wizualizacją ... 23
3.4. Instrukcja użycia Quadtree ... 23
3.4.1. Budowa Quadtree ... 23
3.4.2. Przeszukiwanie drzewa ... 23
3.4.3. Sprawdzenie czy drzewo zawiera szukane punkty ... 23
3.4.4. Uwagi ... 23
3.5. Instrukcja użycia Quadtree z wizualizacją ... 23
4. Sprawozdanie ... 24
4.1. Temat i cel sprawozdania ... 24
4.2. Wstęp teoretyczny ... 24
4.2.1. KdTree ... 24
4.2.2. QuadTree ... 25

4.3. Implementacja ... 26
4.3.1. KdTree ... 26
4.3.2. QuadTree ... 26
4.4. Testy poprawności ... 27
4.5. Testy wydajności ... 27
4.5.1. Zbiór o rozkładzie jendnostajnym ... 27
4.5.2. Zbiór o rozkładzie normalnym ... 29
4.5.3. Rozkłady klastrowe ... 31
4.5.3.1. Dwa skupiska ... 31
4.5.3.2. Trzy skupiska ... 33
4.5.3.3. Dwa skupiska z punktami odstającymi ... 35
4.5.3.4. Dwa skupiska z prawie pustym obszarem wyszukiwań ... 37
4.5.4. Rozkład w kształcie okręgu ... 38
4.5.5. Rozkład krzyżowy ... 40
4.5.6. Rozkład na obwodzie prostokąta ... 42
4.5.7. Rozkład wzdłuż linii ... 44
4.5.8. Rozkład „Siatka” ... 45
4.5.9. Testy specyficzne dla KdTree ... 47
4.5.9.1. Test wydajnościowy wraz ze wzrastającą liczbą wymiarów ... 47
4.5.10. Test wydajnościowy trzymania tablicy punktów w każdym wierzchołku ... 48
5. Podsumowanie ... 50
6. Źródła i inspiracje programistyczne ... 50

1. Wstęp

1.1. Cel projektu

Ta dokumentacja ma na celu omówienie zaawansowanych metod wyszukiwania w geometrii, ze szczególnym uwzględnieniem implementacji i praktycznego zastosowania struktur KdTree oraz QuadTree, a także ich wzajemną analizę porównawczą.

1.2. Dane techniczne

1.2.1. Dane techniczne środowiska programistycznego

- Język - Python 3.10.12
- Środowisko - Jupyter Notebook
- Wykorzystane biblioteki:
  - `numpy`
  - `pandas`
  - `matplotlib`
  - `copy`
  - `bitalg`

1.2.2. Dane techniczne sprzętowe

	Komputer 1	Komputer 2
Procesor	Intel(R) Core(TM) i5 12500H 4.5 GHZ	Amd Razen 7735H 4.8 GHZ
Pamięc RAM	32 GB	16 GB
System operacyjny	Linux Mint 21.4	Windows 11

Tabela 1: Dane techniczne komputerów, na których były przeprowadzane testy

Do testów wydajnościowych został użyty komputer 1

2. Dokumentacja

Niniejsza część dokumentu opisuje procedury i funkcje zawarte w klasach używanych do implementacji struktur *KDTree* i *QuadTree*.

2.1. Point

Klasa reprezentuje niemutowalny punkt o dodatniej liczbie wymiarów, oferujący szeroki zakres możliwości związanych z jego wykorzystaniem i przetwarzaniem. Reprezentacja punktu o dowolnej liczbie wymiarów, oferująca liczne funkcje porównywania i przetwarzania współrzędnych.

- **init(cords)**
  Inicjalizuje obiekt klasy Point jako niemutowalny punkt o dodatniej liczbie wymiarów.

  Parameters:

  - cords (iterable): Iterowalny obiekt, którego wartości stanowią kolejne wymiary punktu.

Raises:

- ValueError: Jeśli obiekt cords jest pusty.

- **str()**
  Zwraca tekstową reprezentację punktu w postaci $(x_1, x_2, ..., x_n)$.

  Returns:

  - str: Tekstowa reprezentacja punktu.

- **repr()**
  Zwraca tekstową reprezentację punktu (taką samą jak metoda `__str__`).

  Returns:

  - str: Tekstowa reprezentacja punktu.

- **eq(other)**
  Porównuje dwa punkty na podstawie ich wymiarów.

  Parameters:

  - other (Point): Punkt do porównania.

  Returns:

  - bool: True, jeśli wszystkie wymiary punktów są równe; False w przeciwnym razie.

- **follow(other)**
  Sprawdza, czy wszystkie wymiary bieżącego punktu są większe lub równe wymiarom drugiego punktu.

  Parameters:

  - other (Point): Punkt do porównania.

  Returns:

  - bool: True, jeśli bieżący punkt „podąża” za drugim; False w przeciwnym razie.

  Raises:

  - ValueError: Jeśli punkty mają różne liczby wymiarów.

- **precedens(other)**
  Sprawdza, czy wszystkie wymiary bieżącego punktu są mniejsze lub równe wymiarom drugiego punktu.

  Parameters:

  - other (Point): Punkt do porównania.

  Returns:

  - bool: True, jeśli bieżący punkt „poprzedza” drugi; False w przeciwnym razie.

  Raises:

  - ValueError: Jeśli punkty mają różne liczby wymiarów.

- **lower_left(other)**
  Zwraca nowy punkt o minimalnych wartościach wymiarów z dwóch porównywanych punktów.

  Parameters:

  - other (Point): Punkt do porównania.

  Returns:

  - Point: Punkt z minimalnymi wartościami wymiarów.

  Raises:

  - ValueError: Jeśli punkty mają różne liczby wymiarów.

- **upper_right(other)**
  Zwraca nowy punkt o maksymalnych wartościach wymiarów z dwóch porównywanych punktów.

  Parameters:

  - other (Point): Punkt do porównania.

  Returns:

  - Point: Punkt z maksymalnymi wartościami wymiarów.

  Raises:

  - ValueError: Jeśli punkty mają różne liczby wymiarów.

2.2. Rectangle

Reprezentacja prostokąta w przestrzeni wielowymiarowej. Umożliwia operacje takie jak sprawdzanie przecięć, zawierania, czy określania, czy punkt znajduje się w prostokącie.

- **init(lower_left=None, upper_right=None, list_of_Point=None)**
  Inicjalizuje obiekt klasy Rectangle. Prostokąt może być zdefiniowany za pomocą dwóch punktów (dolny lewy i górny prawy) lub listy punktów, na podstawie których zostanie wyznaczony minimalny prostokąt obejmujący wszystkie te punkty.

  Parameters:

  - lower_left (Point, optional): Punkt dolnego lewego narożnika prostokąta.
  - upper_right (Point, optional): Punkt górnego prawego narożnika prostokąta.
  - list_of_Point (list[Point], optional): Lista punktów, na podstawie której zostaną wyznaczone narożniki prostokąta.

  Raises:

  - ValueError: Jeśli punkty dolny lewy i górny prawy są nieprawidłowo zdefiniowane.

- **from_Point_list(list_of_Point)**
  Wyznacza narożniki prostokąta na podstawie listy punktów.

  Parameters:

- list_of_Point (list[Point]): Lista punktów klasy Point.

Returns:

- tuple[Point, Point]: Punkt dolnego lewego oraz górnego prawego narożnika prostokąta.

- **str()**
  Zwraca tekstową reprezentację prostokąta w postaci `(lower_left, upper_right)`.

  Returns:

  - str: Tekstowa reprezentacja prostokąta.

- **repr()**
  Zwraca tekstową reprezentację prostokąta (taką samą jak metoda `__str__`).

  Returns:

  - str: Tekstowa reprezentacja prostokąta.

- **is_intersect(other)**
  Sprawdza, czy prostokąt przecina się z innym prostokątem.

  Parameters:

  - other (Rectangle): Inny prostokąt do porównania.

  Returns:

  - bool: True, jeśli prostokąty się przecinają; False w przeciwnym przypadku.

- **is_contained(other)**
  Sprawdza, czy inny prostokąt znajduje się w całości wewnątrz bieżącego prostokąta.

  Parameters:

  - other (Rectangle): Prostokąt do sprawdzenia.

  Returns:

  - bool: True, jeśli other jest w całości zawarty w self; False w przeciwnym razie.

- **intersection(other)**
  Zwraca prostokąt będący częścią wspólną dwóch prostokątów.

  Parameters:

  - other (Rectangle): Inny prostokąt do przecięcia.

  Returns:

  - Rectangle: Prostokąt będący częścią wspólną.

- **is_point_in_rectangle(point)**
  Sprawdza, czy dany punkt znajduje się wewnątrz prostokąta.

  Parameters:

  - point (Point): Punkt do sprawdzenia.

Returns:

- bool: True, jeśli punkt znajduje się w prostokącie; False w przeciwnym razie.

- **get_all_vertix_from_rectangle_on_2d()**
  Zwraca współrzędne wierzchołków prostokąta w przestrzeni dwuwymiarowej.

  Returns:

  - list[tuple[float, float]]: Lista krotek reprezentujących współrzędne wierzchołków w porządku `(lower_left, lower_right, upper_right, upper_left)`.

2.3. Kd_tree

2.3.1. KdTreeNode

Reprezentacja wierzchołka w drzewie k-d. Odpowiada za przechowywanie informacji o punkcie podziału, prostokącie obejmującym obszar oraz za rekurencyjne dzielenie przestrzeni na mniejsze podobszary.

- **init(points, dimensions_amount, depth, rectangle, is_points_in_vertix=True)** Inicjalizuje obiekt klasy KdTreeNode. Może przechowywać punkty w węźle (lub jedynie w liściach) oraz automatycznie budować poddrzewo, jeśli przekazano więcej niż jeden punkt.

  Parameters:
  - `points (list[Point])`: Lista punktów w formie krotek współrzędnych przypisanych do danego wierzchołka.
  - `dimensions_amount (int)`: Liczba wymiarów w przestrzeni punktów.
  - `depth (int)`: Głębokość węzła w drzewie.
  - `rectangle (Rectangle)`: Prostokąt ograniczający przestrzeń, do której należy ten wierzchołek.
  - `is_points_in_vertix (bool, optional)`:
    - `True`: Punkty są przechowywane w każdym węźle.
    - `False`: Punkty są przechowywane wyłącznie w liściach.

- **bsearch_right(t, index, val)** Wyszukuje skrajnie prawy indeks wartości w posortowanej liście punktów w danym wymiarze.

  Parameters:
  - `t (list[Point])`: Lista punktów.
  - `index (int)`: Numer wymiaru, względem którego dokonujemy wyszukiwania.
  - `val (int)`: Wartość, dla której szukamy skrajnie prawego wystąpienia.

  Returns:
  - `int`: Skrajnie prawy indeks punktu o wartości `val`.

- **bsearch_left(t, index, val)** Wyszukuje skrajnie lewy indeks wartości w posortowanej liście punktów w danym wymiarze.

  Parameters:
  - `t (list[Point])`: Lista punktów.

  - `index (int)`: Numer wymiaru, względem którego dokonujemy wyszukiwania.
  - `val (int)`: Wartość, dla której szukamy skrajnie lewego wystąpienia.

  Returns:
  - `int`: Skrajnie lewy indeks punktu o wartości `val`.

- **build(points)** Rekurencyjnie buduje drzewo k-d, dzieląc punkty na dwie grupy za pomocą mediany w bieżącym wymiarze.

  Parameters:
  - `points (list[Point])`: Lista punktów przypisanych do bieżącego wierzchołka.

- **_split_region(rec, dimension_number, axis)** Dzieli prostokąt na dwa mniejsze obszary wzdłuż osi podziału.

  Parameters:
  - `rec (Rectangle)`: Prostokąt do podzielenia.
  - `dimension_number (int)`: Numer wymiaru, według którego dzielimy prostokąt.
  - `axis (float)`: Współrzędna osi podziału.

  Returns:
  - `tuple[Rectangle, Rectangle]`: Dwa rozłączne prostokąty powstałe w wyniku podziału.

- **is_leaf()** Sprawdza, czy bieżący węzeł jest liściem.

  Returns:
  - `bool`: Czy węzeł jest liściem.

- **get_points()** Pobiera wszystkie punkty zapisane w bieżącym węźle i jego poddrzewach.

  Returns:
  - `list[Point]`: Lista punktów przypisanych do drzewa ukorzenionego w bieżącym wierzchołku.

- **find_points_in_region(region)** Rekurencyjnie wyszukuje wszystkie punkty znajdujące się w zadanym prostokątnym obszarze.

  Parameters:
  - `region (Rectangle)`: Prostokąt określający obszar wyszukiwania.

  Returns:
  - `list[Point]`: Lista punktów znajdujących się w regionie.

- **check_contains(point)** Sprawdza, czy dany punkt znajduje się w drzewie ukorzenionym w bieżącym wierzchołku.

  Parameters:
  - `point (Point)`: Punkt, którego przynależność do drzewa chcemy sprawdzić.

  Returns:
  - `bool`: Czy punkt znajduje się w drzewie.

2.3.2. KdTree

Struktura drzewa k-d, która umożliwia szybkie wyszukiwanie punktów w wielowymiarowej przestrzeni. Pozwala również na sprawdzanie przynależności punktów do obszaru.

- **init(points, dimensions_amount, begining_axis=0, is_points_in_vertix=False)** Inicjalizuje strukturę drzewa k-d, weryfikuje poprawność danych wejściowych i tworzy korzeń drzewa, a następnie rekurencyjnie buduje jego poddrzewa.

  Parameters:
  - `points (list[tuple])`: Lista punktów w postaci krotek współrzędnych.
  - `dimensions_amount (int)`: Liczba wymiarów punktów.
  - `begining_axis (int, optional)`: Numer współrzędnej, od której zaczyna się podział zbioru punktów.
  - `is_points_in_vertix (bool, optional)`:
    - `True`: Punkty są przechowywane w każdym wierzchołku.
    - `False`: Punkty są przechowywane tylko w liściach.

  Raises:
  - `ValueError`: Jeśli liczba współrzędnych punktu nie zgadza się z deklarowaną liczbą wymiarów.

- **find_points_in_region(region)** Wyszukuje wszystkie punkty znajdujące się w zadanym prostokącie.

  Parameters:
  - `region (Rectangle | list[tuple])`: Obszar wyszukiwania w postaci obiektu `Rectangle` lub listy dwóch krotek współrzędnych określających dolny lewy i górny prawy narożnik.

  Returns:
  - `list[tuple]`: Lista krotek współrzędnych punktów znajdujących się w prostokącie.

  Raises:
  - `ValueError`: Jeśli podany region jest niepoprawny (zła kolejność wierzchołków lub niespójne wymiary).

- **check_contains(point)** Sprawdza, czy dany punkt należy do drzewa.

  Parameters:
  - `point (Point | list)`: Punkt w postaci obiektu `Point` lub krotki współrzędnych.

  Returns:
  - `bool`: Czy punkt znajduje się w strukturze drzewa.

  Raises:
  - `ValueError`: Jeśli podany punkt jest niepoprawny (niezgodne wymiary współrzędnych).

2.4. Kd_tree_visualiser

Klasy KdTreeNode_vis i KdTree_vis są zaimplementowane na podstawie klas opisanych w punkcie 2.3.1 i 2.3.2. Zostały one dostosowane do pracy z vizualizerem od *koła naukowego*

BIT poprzez dodanie rysowania obiektów w odpowiednich miejscach funkcji pierwotnych. Do zarządzania procesem tworzenia wizualizacji została stworzona klasa Visualization, której metody są omówione poniżej:

- **give_visualization_of_create(test, draw_final=False, draw_gif=False, name=„kd_tree visualization”)** Tworzy wizualizację procesu budowy drzewa k-d na podstawie przekazanych danych.

  Parameters:
  - `test (list[tuple])`: Lista punktów, które zostaną użyte do utworzenia drzewa.
  - `draw_final (bool, optional)`: Czy wyświetlić finalną wizualizację w formie statycznej.
  - `draw_gif (bool, optional)`: Czy wygenerować animację procesu budowy drzewa.
  - `name (str, optional)`: Tytuł wizualizacji.

  Returns:
  - `Visualizer`: Obiekt zawierający wizualizację.

- **give_visualization_of_search(find_ll, find_ur, draw_final=False, draw_gif=False, name=„kd_tree visualization”, print_output=False)** Tworzy wizualizację procesu wyszukiwania punktów w zadanym obszarze.

  Parameters:
  - `find_ll (tuple)`: Współrzędne lewego dolnego narożnika obszaru wyszukiwania.
  - `find_ur (tuple)`: Współrzędne prawego górnego narożnika obszaru wyszukiwania.
  - `draw_final (bool, optional)`: Czy wyświetlić finalną wizualizację w formie statycznej.
  - `draw_gif (bool, optional)`: Czy wygenerować animację procesu wyszukiwania.
  - `name (str, optional)`: Tytuł wizualizacji.
  - `print_output (bool, optional)`: Czy wyświetlić listę znalezionych punktów w konsoli.

  Returns:
  - `Visualizer`: Obiekt zawierający wizualizację.

- **draw_gif(name, vis)**
  Wyświetla animację wizualizacji.

  Parameters:
  - `name (str)`: Tytuł animacji.
  - `vis (Visualizer)`: Obiekt wizualizacji.

- **draw_vis(name, vis)** Wyświetla statyczną wizualizację.

  Parameters:
  - `name (str)`: Tytuł wizualizacji.
  - `vis (Visualizer)`: Obiekt wizualizacji.

Legenda obrazów:
- Wizualiazacja tworzenia Kd-drzewa
  - ***kolor czarny*** - punkty, które nie są w żadnym liściu tworzonego kd_drzewa

  - ***kolor niebieski*** - punkty należącego do, któregoś z liści kd-drzewa
  - ***kolor żółty*** - odcinki reprezentujące podział węzła
  - ***kolor szary*** - obszar aktualnie przetwarzany.
- Wizualizacja wyszukiwania w Kd-drzewie:
  - ***kolor niebieski*** - punkty należące do struktury Kd-drzewa
  - ***kolor pomarańczowy*** - odcinki reprezentujące podział węzła
  - ***kolor szary*** - obszar aktualnie przetwarzany.
  - ***kolor zielony*** - punkt, który został znaleziony oraz kolor obszaru, w którym wykryto wyszukiwane punkty
  - ***kolor czerwony*** - obszar, w którym nie wykryto punktów, których poszukujemy

2.5. Quad_tree

- **init(rectangle, max_points = 3, depth = 0)**
  inicjalizuje strukture Quad Tree
  Parameters:
  - rectangle - prostokąt który ma ograniczać dany obszar quadtree
  - max_points - maksymalna liczba punktów która może znajdować się w każdym węźle quadtree
  - depth - na jakiej głębokości ma znajdować się dany węzeł quadtree

  Atrubutes:
  - nw - węzeł quadtree znajdujący się w prawej górnej ćwiartce prostokątu rodzica
  - ne - węzeł quadtree znajdujący się w lewej górnej ćwiartce prostokątu rodzica
  - sw - węzeł quadtree znajdujący się w prawej dolnej ćwiartce prostokątu rodzica
  - se - węzeł quadtree znajdujący się w lewej dolnej ćwiartce prostokątu rodzica
  - rectangle - obszar na którym są punkty w danym węźle quadtree (w każdym węźle znajduje podprostokąt rodzica); używana jest do określenia tego obszaru klasa rectangle
  - max_points - liczba punktów które maksymalnie może mieć węzeł drzewa przed koniecznością jego rozdzielenia
  - points - punkty które znajdują się w danym węźle quadtree przechowywane w liście; wszystkie punkty znajdujące się w niej są klasy rectangle
  - depth - atrybut określający ile węzłów odległości jest od korzenia quadtree
  - divided - parametr typu bool (domyślnie ustawiony na False) za pomocą którego jest sprawdzane czy prostokąt jest liściem, to jest czy nie jest podzielony (nie ma dzieci)
- **divide()**
  metoda dzieląca prostokąt węzłu quadtree na cztery ćwiartki. Metoda dzieli czworokąt węzłu dzieląc jego boki na pół tworzy przez to dzieci węzłu (nw, ne, sw, se), które są mają parametry quadtree. Wywołanie tej metody zmienia parametr divided na True.
- **insert(point)** metoda wsadzająca punkty do struktury quadtree Parameters:
  - point - punkt który wkładany jest do quadtree

  Returns:
  - True - jeżeli możemy umieścić punkt w danym węźle
  - False - jeżeli nie możemy umieścić punktu w danym węźle

  - self.se.insert(point) or self.ne.insert(point) or self.sw.insert(point) or self.nw.insert(point) - jeżeli nie jest liściem odwiedzamy jego dzieci
- **search(boundary, found_points)** metoda służąca do przeszukiwania danego czworokąta zadanego za pomocą boundary. Ta metoda zwraca wszystkie punkty, które znajdują się w quadtree na danym obszarze Parameters:
  - boundary - obszar na którym poszukiwane są punkty w quadtree
  - found_points - punkty znalezione przez quadtree w całym drzewie

  Returns:
  - found_points - lista znalezionych punktów w postaci krotek
- **contains(self, point)** metoda sprawdzająca czy dany punkt znajduje się w drzewie Paramiters:
  - point - szukany punkt w postaci krotki lub struktury Point

  Returns:
  - True - jeżeli szukany punkt znajduje się w drzewie
  - False - jeżeli szukany punkt nieznajduje się w drzewie
  - **build_quadtree(points_tuples, max_points = 3)**
    funkcja która buduje quadtree za pomocą listy ktotek punktów Arguments:
    - point_tuples - lista krotek punktów (punkty muszą być dwuwymiarowe), które mają znaleźć się w quadtree
    - max_points - wskazuje na to ile punktów może maksymalnie znajdować się w węzłach quadtree

    -**find_rectangle_conv_to_point(points)** funkcja szukająca minimalnego prostokątu zawierającego wszystkie punkty (dwuwymiarowe) zadane za pomocą parametru **points** oraz przekształcająca listae dwuelementowych punktów zawierających współrzędne (x, y) ma lista punktów (res). Funkcja zwraca lista punktów (res) i minimalny prostokąt zawierający wszystkie zadane punkty. Arguments:
    - points - lista krotek w postaci (x, y) do przekształcenia do punktów o współrzędnych (x, y) i znalezienia minimalnego prostokątu który wszystkie punkty zawiera

2.6. Quadtree_vis

Jest to opisana struktura w punkcie **2.5** w opisie jej będę poruszał jedynie dodane elementy do już tam opisanych. W programie założone jest korzystanie z wizualizatora wykonanego przez AGH BIT jako obserwatora.

- **init(self, rectangle, max_points=3, depth=0, visualizer=None)**
  patrz **2.5**. Zostało dodanie tworzenie wizualizacji prostokątu który przechowuje punkty w strukturze quadtree
  Parameters:
  - visualizer - dodany parametr do odpowiednika z **2.5** który przekazuje oobserwatora do struktury quadtree

  Atributes:

  - visualizer - dodany atrybut który wskazuje na obserwatora quadtree

- **divide(self)**
  parz **divide 2.5**
- **insert(self, point)** patrz **insert 2.5**. Zostało do niego powiadomienie obserwatora o dodaniu danego punktu do struktury
- **search(self, boundary, found_points)** patrz **search 2.5**. Zostało dodane do tej metody powiadomienie obserwatora o obszarze w który jest do przeszukania, obszarze w którym aktualnie znajduje się algorytm poszukiwania punktów oraz wszystkich znalezionych do tej pory punktach z zadanego obszaru
- **contains(self, point)**
  patrz **contains 2.5**
- **build_quadtree(points_tuples, max_points=3, visualizer=None)**
  patrz **build_quadtree 2.5** Dodane argumenty:
  - visualizer - obserwator który ma zostać dodany do quadtree (narzędzie do wizualizacji od kołą naukowego BIT)
- **find_rectangle_conv_to_point**
  patrz **find_rectangle_conv_to_point 2.5**

  Legenda obrazów:
- Wizualiazacja tworzenia Kd-drzewa
  - ***kolor niebieski*** - punkty należącego do, któregoś z liści kd-drzewa
  - ***kolor żółty*** - prostokąty reprezentujące podział węzła
- Wizualizacja wyszukiwania w Kd-drzewie:
  - ***kolor niebieski*** - punkty należące do struktury Kd-drzewa
  - ***kolor pomarańczowy*** - odcinki reprezentujące podział węzła
  - ***kolor szary*** - obszar aktualnie przetwarzany.

2.7. generate_tests

-generate_random_on_circle(n,R,r)
funkcja generująca n punktów na pierścieniu o środku w punkcie (0, 0) oddalone od środka o co najmniej r i co najwyżej R.
Arguments:
- n - liczba punktów do wygenerowania
- R - maksymalne oddalenie od środka okręgu (0, 0)
- r - minimalne oddalenie od środka okręgu (0, 0)

Returns:
- ans - lista wygenerowanych krotek współrzędnych punktów wygenerowanych na pierścieniu o środku w punkcie (0, 0) oddalone od środka o co najmniej r i co najwyżej R

- **generate_random_on_line(n,a,b,min_x,max_x,thick)**
  funkcja generująca punkty w co najwyżej thick danej odległości od prostej.
  Arguments:
  - n - liczba punktów do wygenerowania
  - a - współczynnik kierunkowy prostej
  - b - wyraz wolny prostej
  - min_x - dolna granica dziedziny punktów
  - max_y - górna granica dziedziny punktów
  - thick - maksymalne oddalenie punktów od prostej

  Returns:
  - ans - lista wygenerowanych krotek współrzędnych punktów znajdujących się w odległości do najwyżej thick od danej prostej

- **generate_grid_normal(n, min_x,max_x,min_y,max_y)**
  funkcja generująca n^2 punktów które tworzą siatkę. w każdym rzędzie i każdej kolumnie jest n punktów równomiernie od siebie oddalonych
  Arguments:
  - n - liczba punktów w rzędzie w siatce
  - min_x - dolna granica generowania x-owych współrzędnych punktów
  - max_x - górna granica generowania x-owych współrzędnych punktów
  - min_x - dolna granica generowania y-owych współrzędnych punktów
  - max_x - górna granica generowania y-owych współrzędnych punktów

  Returns:
  - ans - lista wygenerowanych krotek współrzędnych punktów w siatce

- **generate_cross_on_axes(n,min_x,max_x,min_y,max_y)**
  funkcja generująca n punktów w dwóch prostych równoległych odpowiednio do osi OX i osi OY
  Arguments:
  - n - liczba punktów do wygenerowania
  - min_x - dolna granica generowania x-owych współrzędnych punktów
  - max_x - górna granica generowania x-owych współrzędnych punktów
  - min_x - dolna granica generowania y-owych współrzędnych punktów
  - max_x - górna granica generowania y-owych współrzędnych punktów

  Returns:
  - ans - lista wygenerowanych krotek współrzędnych punktów na prostych

- **generate_uniform_distribution(n,min_x,max_x,min_y,max_y)**
  funkcja generująca n punktów w rozkładzie ciągłym
  Arguments:
  - n - liczba punktów do wygenerowania
  - min_x - dolna granica generowania x-owych współrzędnych punktów
  - max_x - górna granica generowania x-owych współrzędnych punktów
  - min_x - dolna granica generowania y-owych współrzędnych punktów
  - max_x - górna granica generowania y-owych współrzędnych punktów

Returns:
- ans - lista wygenerowanych krotek współrzędnych punktów w rozkładzie ciągłym

- **generate_clasters(n, m, min_x, max_x, min_y, max_y)**
  funkcja generująca n punktów w m skupiskach
  Arguments:
  - n - liczba punktów do wygenerowania
  - m - liczba skupisk
  - min_x - dolna granica generowania x-owych współrzędnych punktów
  - max_x - górna granica generowania x-owych współrzędnych punktów
  - min_x - dolna granica generowania y-owych współrzędnych punktów
  - max_x - górna granica generowania y-owych współrzędnych punktów

  Returns:
  - points - lista krotek współżędnych punktów wygenerowanych w skupiskach

- **generate_random_points_on_rectangle(n, min_x, max_x, min_y, max_y)**
  funkcja generująca n punktów na bokach prostokątu
  Arguments:
  - n - liczba punktów do wygenerowania
  - min_x - dolna granica generowania x-owych współrzędnych punktów
  - max_x - górna granica generowania x-owych współrzędnych punktów
  - min_x - dolna granica generowania y-owych współrzędnych punktów
  - max_x - górna granica generowania y-owych współrzędnych punktów

  Returns:
  - points - lista krotek współżędnych punktów wygenerowanych na bokach prostokątu

- **add_outliners(n,list,min_x,max_x,min_y,max_y)**
  funkcja dodająca do listy n punktów wygenerowanych w rozkładzie ciągłym, które mają być w założeniu odstające od rozkładu w który zostały wygenerowane punkty z listy list
  Arguments:
  - n - liczba punktów do wygenerowania
  - min_x - dolna granica generowania x-owych współrzędnych punktów
  - list - lista punktów do których dodawane są punkty odstające
  - max_x - górna granica generowania x-owych współrzędnych punktów
  - min_x - dolna granica generowania y-owych współrzędnych punktów
  - max_x - górna granica generowania y-owych współrzędnych punktów

  Returns:
  - list + outliners - konkatenacja listy krotek współrzędnych punktów oraz listy punktów z rozkładu ciągłego

- **generate_standard_distribution(n,mean_x, mean_y, std_x, std_y)**
  funkcja generująca punkty w rozkładzie normalnym
  Arguments:
  - n - liczba punktów do wygenerowania
  - mean_x - średnia wartość współrzędnej x-owej
  - maen_y - średnia wartość współrzędnej y-owej

- std_x - odchylenie standardowe dla współżędnej x
- std_y - odchylenie standardowe dla współżędnej y

Returns:

- list(zip(x_coords, y_coords)) - lista krotek współrzędnych wygenerowanych według rozkładu normalnego

- **generate_multidimensional_cluster(mean, std, n, k)**
funkcja generująca n punktów zgodnie z rozkładem normalnym Arguments:
  - mean - średnia dla każdego wymiaru
  - std - odchylenie standardowe dla każdego wymiaru
  - n - liczba punktów do wygenerowania
  - k - liczba wymiarów

  Returns:
  - points - Macierz o wymiarach (n, k), gdzie każda kolumna to wymiar, a każdy wiersz to punkt.

2.8. quad_tree_tests

- **get_sets_from_array(quad_ans, brute_ans)**
Przekształca listy punktów na zbiory krotek, co umożliwia porównanie wyników dwóch metod.

  Parameters:
  - `quad_ans (list[list[int]])`: Lista punktów zwróconych przez QuadTree.
  - `brute_ans (list[list[int]])`: Lista punktów zwróconych przez algorytm brutalnej siły.

  Returns:
  - `tuple[set[tuple[int, ...]], set[tuple[int, ...]]]`: Dwa zbiory punktów, odpowiednio dla QuadTree i algorytmu brutalnej siły.

- **_run_test_rectangle(points, ll, ur, k=2)**
Testuje wyszukiwanie punktów w zdefiniowanym prostokącie za pomocą QuadTree i algorytmu brutalnej siły.

  Parameters:
  - `points (list[tuple[int, ...]])`: Lista punktów w przestrzeni.
  - `ll (tuple[int, ...])`: Dolny lewy narożnik prostokąta.
  - `ur (tuple[int, ...])`: Górny prawy narożnik prostokąta.
  - `k (int)`: Liczba wymiarów przestrzeni (domyślnie 2).

  Returns:
  - `bool`: `True`, jeśli wyniki obu metod są zgodne, w przeciwnym razie `False`.

- **_run_test_contains(points)**
Sprawdza, czy wszystkie punkty są zawarte w strukturze QuadTree.

  Parameters:
  - `points (list[tuple[int, ...]])`: Lista punktów w przestrzeni.

Returns:
- `bool`: `True`, jeśli wszystkie punkty są poprawnie zawarte, w przeciwnym razie `False`.

- **run_circle_test()**
  Testuje QuadTree dla punktów rozmieszczonych na okręgu.

  Region testowy:
  - Prostokąt od `(0, 0)` do `(13, 13)`.

  Returns:
  - `bool`: Wynik testu.

- **run_on_line_test()**
  Testuje QuadTree dla punktów rozmieszczonych na prostej.

  Region testowy:
  - Prostokąt od `(4, 3)` do `(13, 20)`.

  Returns:
  - `bool`: Wynik testu.

- **run_grid_normal_test()**
  Testuje QuadTree dla punktów rozmieszczonych w regularnej siatce.

  Region testowy:
  - Prostokąt od `(4, 3)` do `(13, 20)`.

  Returns:
  - `bool`: Wynik testu.

- **run_cross_on_axes_test()**
  Testuje QuadTree dla punktów rozmieszczonych w kształcie krzyża wzdłuż osi współrzędnych.

  Region testowy:
  - Prostokąt od `(-4, -3)` do `(4, 8)`.

  Returns:
  - `bool`: Wynik testu.

- **run_uniform_distribution_test()**
  Testuje QuadTree dla punktów rozmieszczonych równomiernie w przestrzeni.

  Region testowy:
  - Prostokąt od `(-4, -3)` do `(13, 20)`.

  Returns:
  - `bool`: Wynik testu.

- **run_clasters_test()**
  Testuje QuadTree dla punktów zgrupowanych w klastry.

  Region testowy:

- Prostokąt od `(-4, -3)` do `(13, 20)`.

Returns:
- `bool`: Wynik testu.

- **run_random_points_on_rectangle_test()**
Testuje QuadTree dla punktów rozmieszczonych losowo na granicach prostokąta.

Region testowy:
- Prostokąt od `(-15, -3)` do `(13, 20)`.

Returns:
- `bool`: Wynik testu.

- **run_standard_distribution_test()**
Testuje QuadTree dla punktów rozmieszczonych według rozkładu normalnego.

Region testowy:
- Prostokąt od `(-15, -3)` do `(13, 20)`.

Returns:
- `bool`: Wynik testu.

- **start_generate_test()**
Uruchamia wszystkie funkcje testujące i podsumowuje wyniki.

Returns:
- `None`: Wyświetla wyniki testów w konsoli.

2.9. Kd_tree_test

- **get_sets_from_array(kd_ans, brute_ans)**
Przekształca listy punktów na zbiory krotek, co umożliwia porównanie wyników dwóch metod.

Parameters:
- `kd_ans (list[list[int]])`: Lista punktów zwróconych przez KDTree.
- `brute_ans (list[list[int]])`: Lista punktów zwróconych przez algorytm brutalnej siły.

Returns:
- `tuple[set[tuple[int, ...]], set[tuple[int, ...]]]`: Dwa zbiory punktów, odpowiednio dla KDTree i algorytmu brutalnej siły.

- **_run_test_rectangle(points, ll, ur, k=2)**
Testuje wyszukiwanie punktów w zdefiniowanym prostokącie za pomocą KDTree i algorytmu brutalnej siły.

Parameters:
- `points (list[tuple[int, ...]])`: Lista punktów w przestrzeni.
- `ll (tuple[int, ...])`: Dolny lewy narożnik prostokąta.
- `ur (tuple[int, ...])`: Górny prawy narożnik prostokąta.
- `k (int)`: Liczba wymiarów przestrzeni (domyślnie 2).

Returns:
- `bool`: `True`, jeśli wyniki obu metod są zgodne, w przeciwnym razie `False`.

- **_run_test_contains(points)**
  Sprawdza, czy wszystkie punkty są zawarte w strukturze KDTree.

  Parameters:
  - `points (list[tuple[int, ...]])`: Lista punktów w przestrzeni.

  Returns:
  - `bool`: `True`, jeśli wszystkie punkty są poprawnie zawarte, w przeciwnym razie `False`.

- **run_circle_test()**
  Testuje KDTree dla punktów rozmieszczonych na okręgu.

  Region testowy:
  - Prostokąt od `(0, 0)` do `(13, 13)`.

  Returns:
  - `bool`: Wynik testu.

- **run_on_line_test()**
  Testuje KDTree dla punktów rozmieszczonych na prostej.

  Region testowy:
  - Prostokąt od `(4, 3)` do `(13, 20)`.

  Returns:
  - `bool`: Wynik testu.

- **run_grid_normal_test()**
  Testuje KDTree dla punktów rozmieszczonych w regularnej siatce.

  Region testowy:
  - Prostokąt od `(4, 3)` do `(13, 20)`.

  Returns:
  - `bool`: Wynik testu.

- **run_cross_on_axes_test()**
  Testuje KDTree dla punktów rozmieszczonych w kształcie krzyża wzdłuż osi współrzędnych.

  Region testowy:
  - Prostokąt od `(-4, -3)` do `(4, 8)`.

  Returns:
  - `bool`: Wynik testu.

- **run_uniform_distribution_test()**
  Testuje KDTree dla punktów rozmieszczonych równomiernie w przestrzeni.

  Region testowy:

- Prostokąt od `(-4, -3)` do `(13, 20)`.

Returns:
- `bool`: Wynik testu.

- **run_clasters_test()**
Testuje KDTree dla punktów zgrupowanych w klastry.

Region testowy:
- Prostokąt od `(-4, -3)` do `(13, 20)`.

Returns:
- `bool`: Wynik testu.

- **run_random_points_on_rectangle_test()**
Testuje KDTree dla punktów rozmieszczonych losowo na granicach prostokąta.

Region testowy:
- Prostokąt od `(-15, -3)` do `(13, 20)`.

Returns:
- `bool`: Wynik testu.

- **run_standard_distribution_test()**
Testuje KDTree dla punktów rozmieszczonych według rozkładu normalnego.

Region testowy:
- Prostokąt od `(-15, -3)` do `(13, 20)`.

Returns:
- `bool`: Wynik testu.

- **run_multidimensional_cluster_test()**
Testuje KDTree dla punktów w klastrach w przestrzeni wielowymiarowej.

Region testowy:
- Hiperprostokąt od `(-3, 0, 0, 0)` do `(10, 10, 10, 10)`.

Returns:
- `bool`: Wynik testu.

- **start_generate_test()**
Uruchamia wszystkie funkcje testujące i podsumowuje wyniki.

Returns:
- `None`: Wyświetla wyniki testów w konsoli.

2.10. Brute_Force

Klasa ta jest używana w trakcie testów. Algorytmem brutalnej siły znajduje punkty zawarte w danym obszarze

- **brute_force_rectangle(points,lower_left,upper_right)**: Funkcje sprawdza jakie, ktore z punktów zawierają sie w prostokącie
  - points - tablica krotek współrzędnych punktów

- lower_left - krotka współrzędnych lewego dolnego wierzchołka prostokąta
- upper_right - krotka współrzędnych prawego górnego wierzchołka prostokąta

3. Opis użytkowania

3.1. Narzędzie do testowania

Dla zarówno KdTree jak i QuadTree testy poprawności przechowywane są w pliku **quad_tree_test.py** w przypadku QuadTree oraz w pliku **kd_tree_test.py** w przypadku KdTree

3.1.1. Instrukcja wykonywania testów

Aby uruchomić wywołanie testów należy uruchomić funkcje:
`start_generate_test()`
W obu strukturach funkcja ta wypisuje do konsoli - ile testów jest zaliczonych oraz stan zaliczenia danego testu.

3.1.2. Instrukcja tworzenia dodatkowych testów poprawnościowych

Testy są budowane według konwencji:

```
def run_nazwa_testu():
  points = nazwa_nazwa_generatora_tekstu
  ll = (x, y) współwrzędne lewego dolnego rogu prostokontu
  ur = (x, y) współwrzędne prawego górnego rogu prostokontu
  return _run_test_rectangle(points,ll,ur) and _run_test_contains(points)
```

ostatnia linijka zwraca czy testy są zaliczone czyli czy struktura zawiera wszystkie punkty które do niej miały zostać włożone

3.2. Instrukcja użycia KdTree

3.2.1. Budowa

Aby zbudować nowe KdTree trzeba zainicjalizować to drzewo używając:
`KdTree(points, k)`
gdzie:

- points - lista krotek współrzędnych punktów wszystkie punkty muszą mieć ten sam wymiar
- k - wymiar tych punktów musi się pokrywać z wymiarem podanych w poprzednim parametrze punktów

3.2.2. Przeszukiwanie obszaru

przeszukiwanie może odbywać się wyłącznie w prostokącie za pomocą metody:
`find_points_in_region(self, region)`
gdzie:

- region - obszar poszukiwania punktów może to być wyłącznie prostokąt zadany przez strukturę Rectangle lub przez tablice krotek współrzędnych lewego dolnego i prawego górnego rogu

metoda ta zwraca wszystkie punkty które są w szukanym obszarze za pomocą listy krotek współrzędnych punktów.

3.2.3. Sprawdzenie czy punkt jest zawarty w strukturze:

sprawdzenie czy punkty są w prostokącie można wykonać za pomocą metody:
`check_contains(self,point)`
gdzie:

- points - punkt zadanych przez strukturę Point lub tablice krotek współrzędnych punktu

metoda ta zwraca czy punkt zawiera się w strukturze danych nie wykonując żadnych zmian w drzewie.

3.3. Instrukcja użycia KdTree z wizualizacją

aby zacząć wizualizacje KdDrzewa należy zainicjować najpierw klasę **Visualization** z pliku **kd_tree_vis.py** za pomocą
`visualization()`
która tworzy obiekt Visualization, który posiada metody:

- give_visualization_of_create()
- give_visualization_of_search()
- draw_gif()
- draw_vis()

kolory w wizualizacji są opisane w podpunkcie **2.4 legenda**

3.3.1. Tworzenie wizualizacji tworzenia drzewa:

aby stworzyć wizualizacje tworzenia drzewa należy wywołać metodę
`give_visualization_of_create(self,test, draw_final=False, draw_gif=False, name = "kd_tree visualization")` gdzie:

- test - lista krotek współrzędnych punktów do dodania do KdTree
- draw_final - wartość która mówi o tym czy wyświetlona zostanie ostateczna wizualizacja czyli obraz z całą strukturą. Domyślnie ustawiona na False
- draw_gif - wartość która mówi o tym czy wyświetlony zostanie gif z krokową budową struktury drzewa. Domyślnie ustawiona na False
- name - tytuł wykresu

3.3.2. Tworzenie wizualizacji przeszukiwania drzewa

aby stworzyć wizualizacje przeszukiwania drzewa kd należy wywołać metodę
`give_visualization_of_search(self,region,draw_final=False, draw_gif=False, name = "kd_tree visualization",print_output = False)` gdzie:

- region - przeszukiwany prostokąt przedstawiony za pomocą krotki lewego dolnego rogu prostokąta oraz prawego górnego rogu obszaru wyszukiwania
- test - lista krotek współrzędnych punktów do dodania do KdTree
- draw_final - wartość która mówi o tym czy wyświetlona zostanie ostateczna wizualizacja czyli obraz z całą strukturą. Domyślnie ustawiona na False
- draw_gif - wartość która mówi o tym czy wyświetlony zostanie gif z krokową budową struktury drzewa. Domyślnie ustawiona na False

- name - tytuł wykresu

3.3.3. Wyświetlanie końcowej wizualizacji

aby wyświetlić końcową wizualizacje należy użyć do tego metody
`draw_vis()`
która wyświetla obrazek z końcową wizualizacją

3.3.4. Wyświetlenie gifu z krokową wizualizacją

aby wyświetlić gif z krokową wizualizacją należy użyć do tego metody
`show_gif()`
która wyświetla gif z wizualizacją

3.4. Instrukcja użycia Quadtree

struktura dostępna jest w pliku **quad_tree.py**

3.4.1. Budowa Quadtree

aby wybudować QuadTree należy użyć ją zainicjalizować za pomocą
`QuadTree()`
by dodać punkty do struktury należy użyć metody
`insert(point)`
gdzie: point - punkt który chcemy dodać do struktury wyrażony za pomocą struktury Point lub z użyciem funkcji
`build_quadtree(points_tuples, max_points = 3)`
gdzie:

- points_tuples - tablica krotek współrzędnych punktów
- max_points - maksymalna liczba punktów w węźle drzewa

3.4.2. Przeszukiwanie drzewa

aby przeszukać QuadTree należy użyć metody
`search(boundary, found_points)` gdzie:

- boundary - obszar który chcemy przeszukać wyrażony za pomocą struktury Rectangle

3.4.3. Sprawdzenie czy drzewo zawiera szukane punkty

aby sprawdzić czy drzewo zawiera wszystkie szukane punkty trzeba użyć do tego metody
`contains(point)`
gdzie: -point - zadany punkt za pomocą struktury Point lub krotki współrzędnych

3.4.4. Uwagi

QuadTree może z założenia przyjmować punkty, które mają współrzędne dwuwymiarowe. W przypadku próby dodania do struktury punktów o innej liczbie wymiarów struktura zwróci błąd ValueError.

3.5. Instrukcja użycia Quadtree z wizualizacją

Implementacja dostępna jest w pliku **quad_tree_vis.py**. Drzewa z wizualizacją używa się dokładnie tak samo jak w przypadku użycia bez jej użycia. Dodany został do struktury tylko:

- parametr vis, który przechowuje informacje o wizualizerze
- w funkcji build_quadtree() został dodany atrybut visualizer który wskazuje jakiego obserwatora dodać do struktury.

4. Sprawozdanie

4.1. Temat i cel sprawozdania

Dane: Zbiór punktów P na płaszczyźnie.
Zapytanie: Dla zadanych współrzędnych punktów lewego dolnego rogu (x_1, y_1) i oraz prawego górnego rogu (x_2, y_2), należy znaleźć punkty q ze zbioru P, takie że spełniają one warunki: $x_1 \leq q_x \leq x_2$ oraz $y_1 \leq q_y \leq y_2$.

Celem niniejszego sprawozdania jest rozwiązanie zagadnienia dotyczącego wyszukiwania punktów w zbiorze, które znajdują się w określonym obszarze. Zostaną zaprezentowane kluczowe informacje na temat struktur danych QuadTree oraz KdTree, w tym szczegółowy opis ich implementacji. Dodatkowo, przeprowadzona zostanie analiza porównawcza ich efektywności w kontekście różnych typów danych wejściowych.

4.2. Wstęp teoretyczny

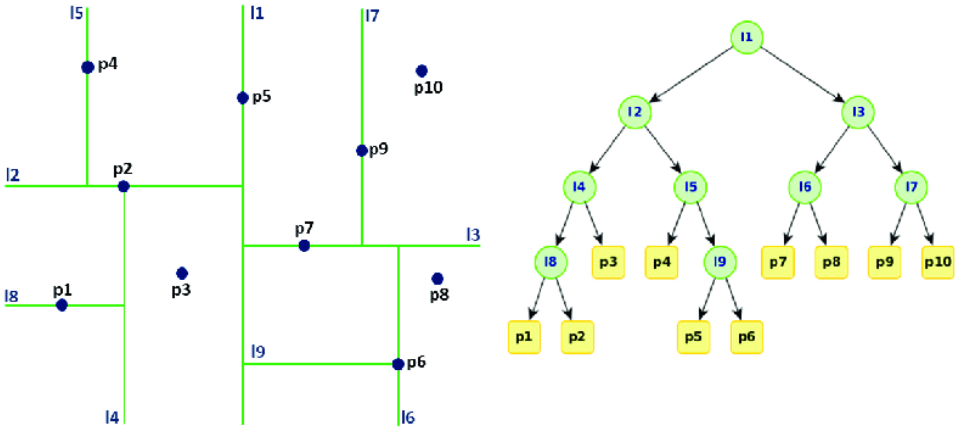
4.2.1. KdTree

KdTree (ang. k-dimensional tree) to hierarchiczna struktura danych umożliwiająca efektywne przechowywanie i przeszukiwanie punktów w przestrzeniach wielowymiarowych. Jest szczególnie użyteczna w problemach takich jak wyszukiwanie najbliższego sąsiada, znajdowanie punktów w określonym regionie, czy sortowanie wielowymiarowe.

Struktura ta organizuje punkty w formie binarnego drzewa, gdzie każdy węzeł reprezentuje podział przestrzeni wzdłuż jednej z osi. Przy każdej iteracji podział odbywa się na przemian względem kolejnych wymiarów. Dzięki temu KdTree jest w stanie ograniczyć liczbę porównań potrzebnych do znalezienia wyników, co czyni ją znacznie bardziej efektywną od metod brutalnych, szczególnie w dużych zbiorach danych.

Węzły drzewa zawierają informacje o współrzędnych punktu, a także podregionach przestrzeni, które obejmuje lewy i prawy poddrzewo. Dzięki tej organizacji możliwe jest szybkie odrzucenie podregionów, które nie mogą zawierać poszukiwanych punktów.

Złożoność obliczeniowa budowy KdTree wynosi O($kn \log^2 n$), gdzie n to liczba punktów w drzewie, a k to liczba wymiarów. Dla zrównoważonego drzewa, złożoność wyszukiwania punktów w obrębie drzewa wynosi O($\sqrt{n} + k$), gdzie n to liczba punktów w drzewie, a k to liczba punktów spełniających kryteria wyszukiwania.

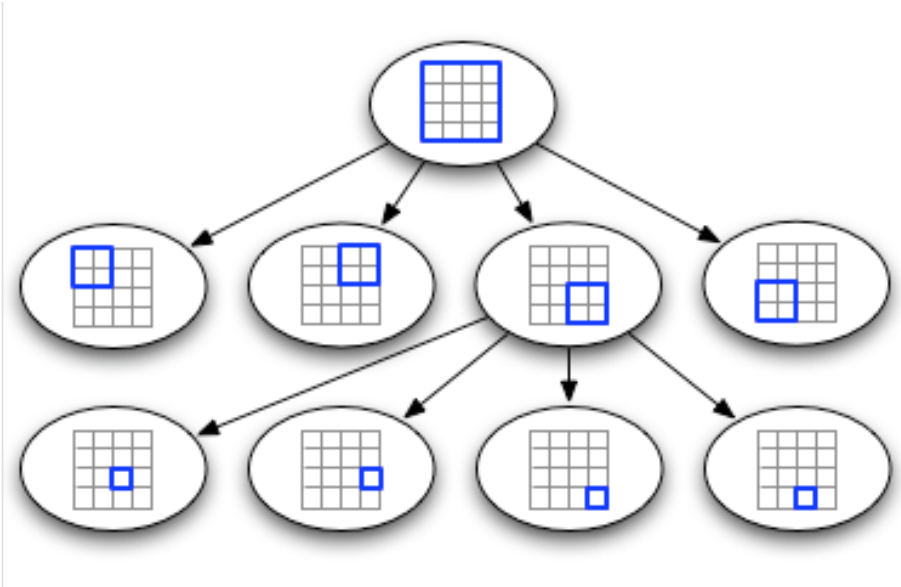


Rysunek 1: Przykładowe drzewo Kd wraz z jego reprezentacją na płaszczyźnie

4.2.2. QuadTree

Drzewo czwórkowe (ang. quadtree) – struktura danych będąca drzewem, używana do podziału dwuwymiarowej przestrzeni na mniejsze części, dzieląc ją na cztery równe ćwiartki, a następnie każdą z tych ćwiartek na cztery kolejne itd. Przez to jest w stanie bardzo skutecznie ogranicza obszar drzewa które trzeba odwiedzić aby przeszukać obszar płaszczyzny. Jej ograniczeniem natomiast jest to, że można używać jej jedynie do przeszukiwania płaszczyzn.

Jest używana na przykład w procesie wykrywania kolizji w dwóch wymiarach. Umożliwia szybkie odrzucenie dużych przestrzeni – gdy zostanie stwierdzone, że któraś ćwiartka nie ma kolizji z danym obiektem, jej pod ćwiartki też nie mają z nim kolizji. Drzewa czwórkowe znalazły również zastosowanie w kompresji bitmap dwukolorowych (czarno-białych), gdzie obraz dzielony jest na mniejsze części dopóki nie będą one jednokolorowe, a wtedy wystarczy tylko zapisać kolor tego kwadratu, na co wystarcza pojedynczy bit.



Rysunek 2: Przykładowe drzewo czwórkowe

Złożoność czasowa, jak i pamięciowa tworzenia QuadTree zależy stricte od jego głębokości (ona zależy natomiast od rozmieszczenia punktów na płaszczyźnie) oraz ilości punktów leżących na płaszczyźnie i wynosi O((d+ 1) ∗n), gdzie d – głębokość drzewa, n – liczba punktów.

Złożoność obliczeniowa przeszukania takiego QuadTree wynosi O(dl), gdzie d - głębokość drzewa, l – liczba ilści reprezentujących prostokąty częściowo zawierające się w przeszukiwanym obszarze.

4.3. Implementacja

Do wykonania obu struktur użyliśmy dwóch klas pomocniczych Point i Rectangle, które reprezentowały odpowiednio punkt w przestrzeni k wymiarowej oraz obszar przeszukiwań. Każda klasa posiada odpowiednie metody, które ułatwiły implementacje poniższych struktur i poprawiły ich czytelność, zostały one opisane w sekcji 2.1 i 2.2.

4.3.1. KdTree

Budowa KD-Drzewa odbywa się rekurencyjnie, poprzez sukcesywne dzielenie przestrzeni na coraz mniejsze podobszary. Na każdym poziomie drzewa punkty są sortowane względem współrzędnych w wybranej osi, która zmienia się cyklicznie, co pozwala na równomierne uwzględnienie wszystkich wymiarów przestrzeni. Wartość mediany w danej osi wyznacza punkt podziału, równoważąc liczbę punktów w obu podobszarach. Wykorzystanie wyszukiwania binarnego optymalizuje wybór osi dzielącej, poprawiając jakość podziału przestrzeni oraz skracając czas przetwarzania w wielu przypadkach testowych. Opis optymalizacji jest zawarty w podpunkcie 4.5.5 wraz z przedstawieniem jego efektywności. Każdy wierzchołek reprezentuje prostokątny obszar przestrzeni, który jest dzielony na dwa mniejsze regiony, przypisane do jego dzieci. W liściach drzewa przechowywane są pojedyncze punkty.

Przeszukiwanie drzewa również zostało zaimplementowane rekurencyjnie. Algorytm eksploruje wierzchołki wzdłuż zadanego obszaru poszukiwań. Jeśli prostokąt wierzchołka w całości mieści się w tym obszarze, zwracane są wszystkie punkty przechowywane w tym wierzchołku. W przypadku liści sprawdzana jest przynależność przechowywanego punktu do poszukiwanego obszaru, a w przypadku braku pełnej zgodności algorytm kontynuuje poszukiwania w wierzchołkach potomnych.

Sprawdzanie zawierania punktu w strukturze KD-Drzewa również odbywa się rekurencyjnie, poprzez przeszukiwanie drzewa w głąb. Algorytm sprawdza kolejne wierzchołki, przechodząc w odpowiednie poddrzewo w zależności od pozycji punktu względem osi podziału w danym węźle. W przypadku liści porównywany jest przechowywany punkt z punktem poszukiwanym. Taka metoda okazała się szczególnie przydatna podczas testów, gdy błędny podział prostokąta na etapie budowy wierzchołków prowadził do problemów z poprawnym wyszukiwaniem punktów w strukturze.

4.3.2. QuadTree

Budowa drzewa czwórkowego odbywa się rekurencyjnie. Poprzez sukcesywne dzielenie czworokąty na cztery mniejsze, które są jego dziećmi. Jako, że ta struktura będzie używana głównie do przeszukiwania płaszczyzny i przyspieszyć czas budowy drzewa to zdecydowałem się podczas dzielenia czworokąta na mniejsze figury nie przenosić do nich wszystkich punktów, które są przechowywane w węźle rodzica, przez to budowa tego drzewa jest znaczne szybsza i drzewo ma mniej węzłów. Maksymalna ilość punktów w węźle przed koniecznością dokonania podziału jest ustawiona domyślnie na 3.

Przeszukiwanie drzewa również zostało zaimplementowane rekurencyjnie. Algorytm eksploruje wierzchołki wzdłuż zadanego obszaru poszukiwań. Jeśli prostokąt mieści się w tym obszarze to sprawdzane jest czy punkty w węźle się w nim mieszczą, a następnie odwiedzane są węzły dzieci tego czworokąta. Jeżeli natomiast jeżeli obszar przeszukiwania nie zawiera się w prostokącie węzła to sprawdzanie jego punktów i dzieci jest pomijana.

Sprawdzanie zawierania punktu w strukturze drzewa ćwiartkowego również odbywa się rekurencyjnie, poprzez przeszukiwanie drzewa w głąb. Algorytm sprawdza kolejne wierzchołki pod kątem zawierania się w nich szukanego punktu. Ta metoda była pomocna w sprawdzaniu poprawności budowania tego drzewa.

4.4. Testy poprawności

Testy poprawności zostały przeprowadzone na tych samych zbiorach, które wykorzystano do testowania wydajności algorytmów. Zbiory te zapewniały wystarczającą różnorodność przypadków, umożliwiając rzetelną weryfikację działania zaimplementowanych struktur. Wyniki uzyskiwane przez nasze algorytmy były porównywane z rezultatami algorytmu brutalnej siły, co pozwoliło na szybkie i jednoznaczne sprawdzenie poprawności zwracanych zbiorów punktów.

Dodatkowo przeprowadzono testy weryfikujące, czy wszystkie punkty przechowywane w strukturach są poprawnie znajdowane przez algorytmy wyszukiwania. Analiza ta umożliwiła upewnienie się, że obszary dzielone podczas budowy drzew są właściwie zarządzane. Tym samym testy potwierdziły poprawność implementacji oraz zgodność wyników z założeniami teoretycznymi obu algorytmów.

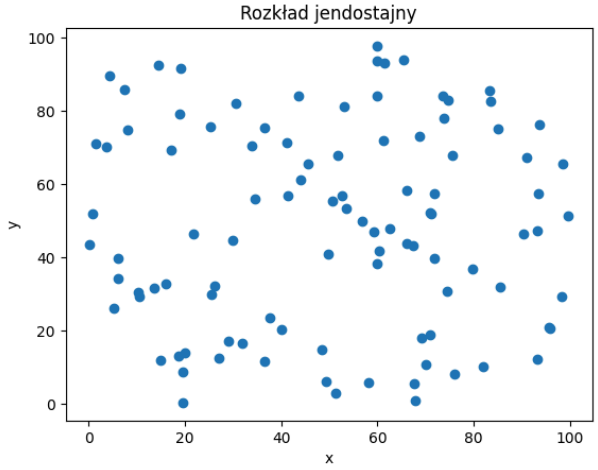
4.5. Testy wydajności

Testy wydajnościowe zostały przeprowadzone w pliku main.ipynb. Znajdują się tam także animacje budowania i wyszukiwania w obu drzewach.

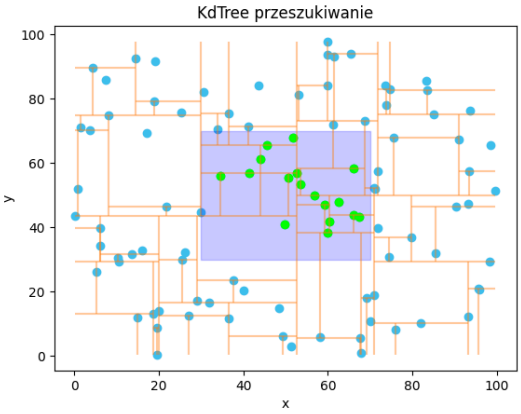
Przypadki testowe posiadały ilość punktów z zakresu 10 000 do 100 000 ze skokiem co 10 000. Taka wielość zbiorów oraz odpowiedni skok pozwala ładnie trend wzrostu czasu przeszukiwania i budowania drzewa względem wzrastającej ilości punktów. Porównane zostały czasy inicjalizacji struktury oraz czasy wyszukiwania punktów w strukturze.

4.5.1. Zbiór o rozkładzie jendnostajnym

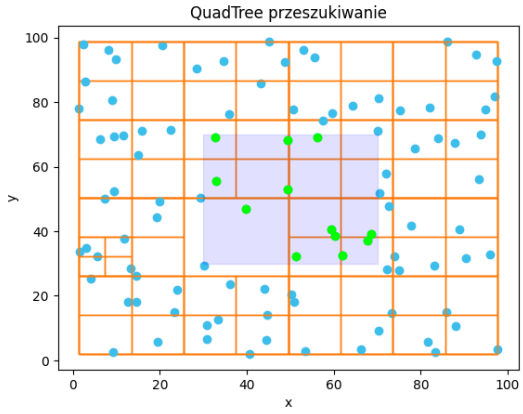
Ten zbiór cechuje się w miarę równomiernym rozkładem punktów na kwadracie o wierzchołkach w punktach (0,0), (100,0), (100,100), (0,100) (co widzimy na Rysunek 3) i stanowi przypadek podstawowy rozważań na temat szybkości działania algorytmów. Prostokąt, w którym szukamy punktów ma wierzchołki o współrzędnych (25,25), (75,25), (75,75), (25,75).



Rysunek 3: Wizualizacja zbioru o rozkładzie jednostajnym

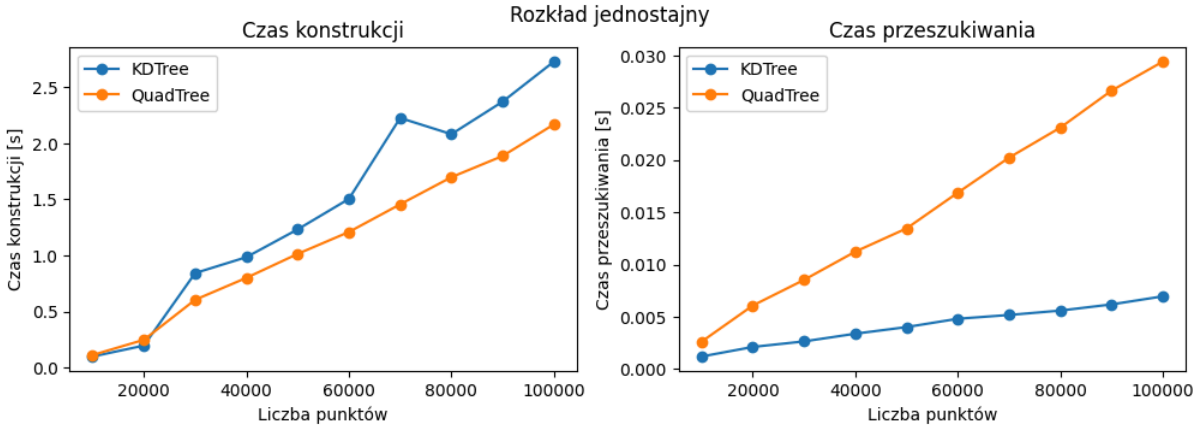


Rysunek 4: Siatka podziału obszaru dla KdTree dla obszaru jednostajnego



Rysunek 5: Siatka podziału obszaru dla QuadTree dla obszaru jednostajnego

Wizualizacje ujęte na Rysunek 4 i Rysunek 5 pokazują poprawność budowy oraz wyszukiwania dla obu struktur.



Rysunek 6: Porównanie czasów budowy i przeszukiwania obu struktur dla rozkładu jednostajnego

Liczba punktów	Liczba znalezionych	KDTree Build [s]	QuadTree Build [s]	KDTree Search [s]	QuadTree Search [s]
10000	1584	0.09970	0.11311	0.00121	0.00262
20000	3153	0.19713	0.24831	0.00212	0.00608
30000	4878	0.84256	0.60500	0.00265	0.00854
40000	6376	0.98466	0.80027	0.00339	0.01123
50000	8122	1.23196	1.01365	0.00402	0.01346
60000	9698	1.50340	1.20880	0.00482	0.01688
70000	11109	2.22418	1.45510	0.00518	0.02023
80000	12799	2.07997	1.69697	0.00560	0.02310
90000	14622	2.36995	1.88599	0.00619	0.02663
100000	15947	2.72638	2.16719	0.00696	0.02941

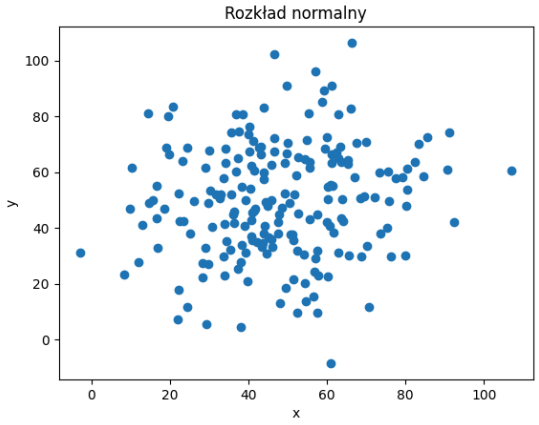
Tabela 2: Czasy budowy i przeszukiwania obu struktur dla rozkładu jednostajnego

Jak wynika z analizy przedstawionej na wykresie Rysunek 6 oraz w tabeli Tabela 2, struktura QuadTree charakteryzuje się szybszym czasem budowy w porównaniu do KdTree. Jednakże, czas wyszukiwania w QuadTree jest przeszło czterokrotnie dłuższy niż w przypadku KdTree. Dłuższy czas inicjalizacji drzewa KdTree jest konsekwencją zastosowanej optymalizacji, która, choć skutkuje poprawą jakości podziału przestrzeni, wprowadza dodatkową złożoność obliczeniową. Wiąże się to z koniecznością wykonania większej liczby operacji podczas procesu budowy drzewa, co negatywnie wpływa na czas jego inicjalizacji.

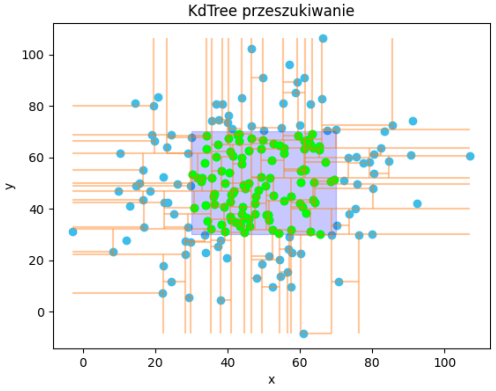
Odchylenie pojedynczego punktu od prostej na wykresie czasu budowania struktur może być rezultatem niekorzystnego rozmieszczenia punktów, co negatywnie wpływa na konstrukcję. Jednakże, to zjawisko może również wynikać z czynników technicznych, takich jak problemy z utrzymaniem wysokiego taktowania procesora w laptopie podczas wykonywania zasobożernych operacji.

4.5.2. Zbiór o rozkładzie normalnym

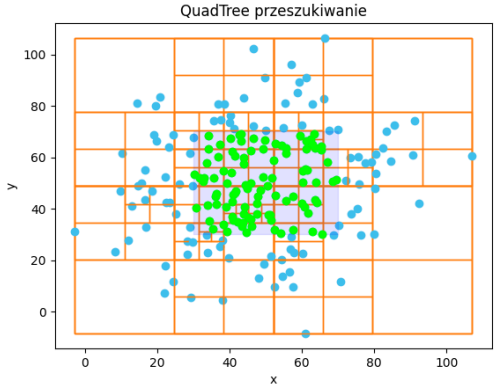
Ten zbiór cechuje się większym zagęszczeniem punktów w obrębie jednego obszaru (Rysunek 7). Wraz z oddalaniem się od punktu środka ilość punktów robi się coraz mniejsza. Ten zbiór pozwala sprawdzić działanie struktur w bardziej wymagającym przypadku. Prostokąt, w którym szukamy punktów ma wierzchołki o współrzędnych (25,25), (75,25), (75,75), (25,75).



Rysunek 7: Wizualizacja zbioru o rozkładzie normalnym

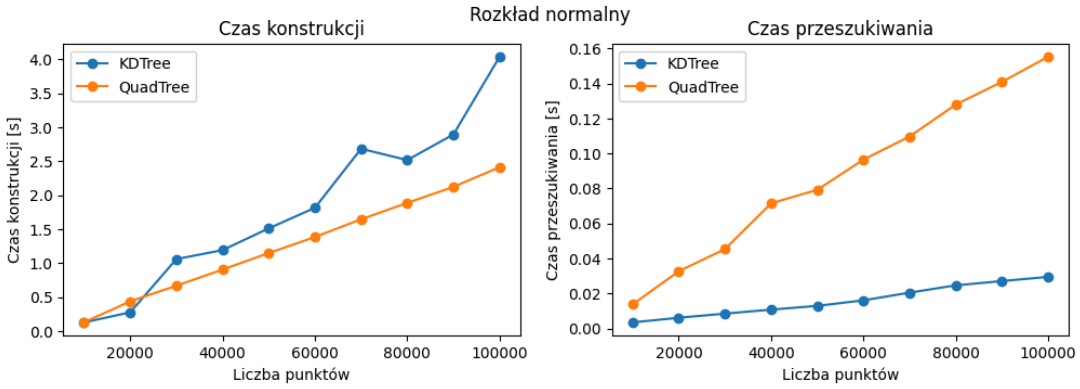


Rysunek 8: Siatka podziału obszaru dla KdTree dla rozkładu normalnego



Rysunek 9: Siatka podziału obszaru dla QuadTree dla rozkładu normalnego

Wizualizacje ujęte na Rysunek 8 i Rysunek 9 pokazują poprawność budowy oraz wyszukiwania dla obu struktur.



Rysunek 10: Porównanie czasów budowy i przeszukiwania obu struktur dla rozkładu normalnego

Liczba punktów	Liczba znalezionych	KDTree Build [s]	Quad-Tree Build [s]	KDTree Search [s]	Quad-Tree Search [s]
10000	9065	0.12691	0.12865	0.00344	0.01371
20000	18265	0.27495	0.43675	0.00611	0.03268
30000	27325	1.05869	0.66678	0.00846	0.04533
40000	36565	1.19051	0.90432	0.01074	0.07148
50000	45502	1.51053	1.14712	0.01289	0.07921
60000	54761	1.81284	1.38338	0.01606	0.09648
70000	63749	2.68174	1.64420	0.02045	0.10968
80000	72821	2.51551	1.88528	0.02462	0.12795
90000	81919	2.89194	2.12116	0.02708	0.14091
100000	90945	4.03516	2.41045	0.02947	0.15525

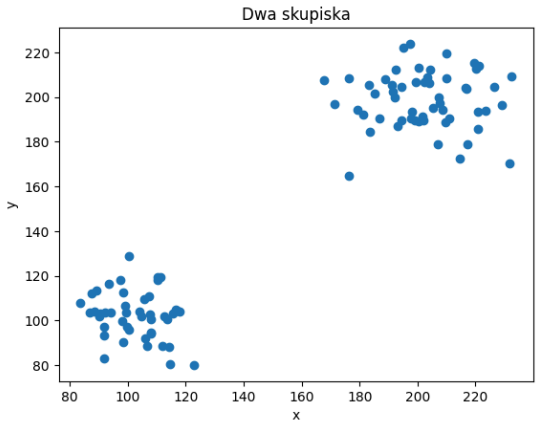
Tabela 3: Czasy budowy i przeszukiwania obu struktur dla rozkładu normalnego

Z analizy przedstawionej na wykresie Rysunek 10 oraz w tabeli Tabela 3 wynika, że obie struktury wykazały podobne zachowanie dla rozkładu normalnego. Warto zauważyć, że Kd--drzewo nadal utrzymuje przewagę, której wartość pozostaje na poziomie zbliżonym do obserwowanego w przypadku zbioru jednostajnego. Z przeprowadzonych testów można wyciągnąć wniosek, że dla losowego rozkładu punktów KdTree jest bardziej efektywne. Jednakże rodzi się pytanie, czy ta tendencja utrzyma się w innych przypadkach?

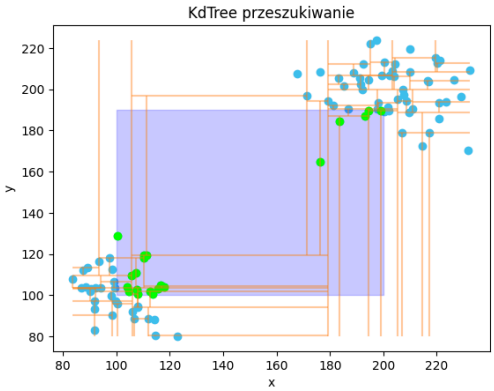
4.5.3. Rozkłady klastrowe

4.5.3.1. Dwa skupiska

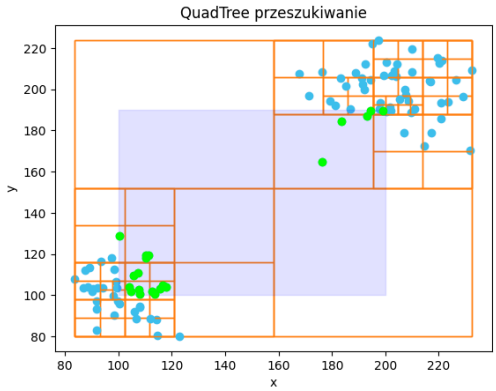
Zbiór ten wykazuje wyraźne zagęszczenie punktów w dwóch specyficznych lokalizacjach na płaszczyźnie, co można zaobserwować na Rysunek 11. Zjawisko to prowadzi do znacznej zmienności w wielkości obszarów w wierzchołkach, co w konsekwencji utrudnia konstrukcję struktur.



Rysunek 11: Wizualizacja zbioru o dwóch skupiskach punktów

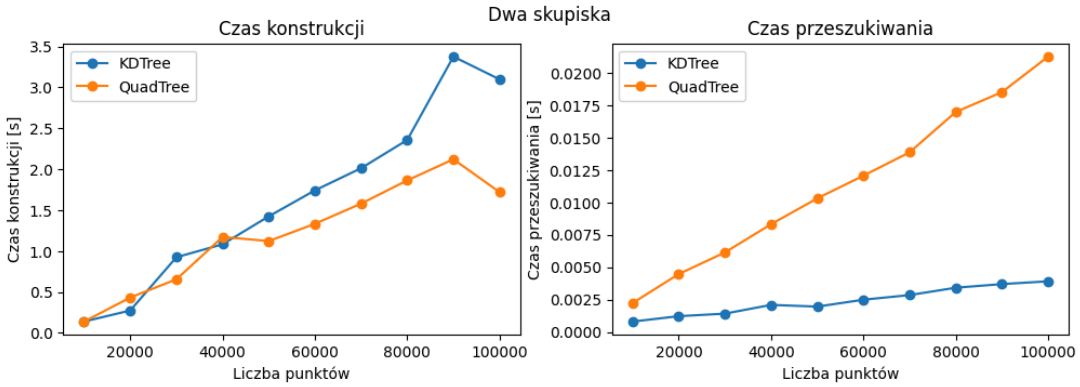


Rysunek 12: Siatka podziału obszaru dla KdTree dla rozkładu o dwóch skupiskach punktów



Rysunek 13: Siatka podziału obszaru dla QuadTree dla rozkładu o dwóch skupiskach punktów

Wizualizacje ujęte na Rysunek 12 i Rysunek 13 pokazują poprawność budowy oraz wyszukiwania dla obu struktur.



Rysunek 14: Porównanie czasów budowy i przeszukiwania dla obu struktur rozkładu o dwóch skupiskach punktów

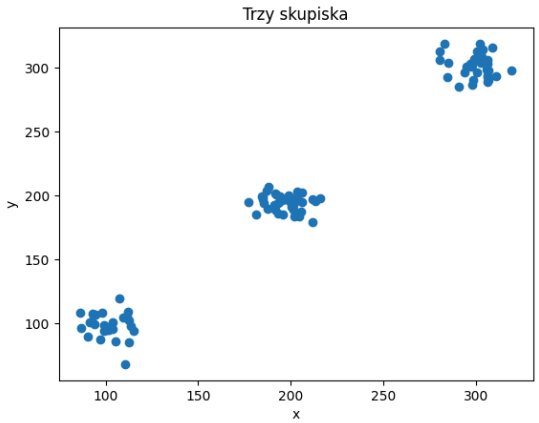
Liczba punktów	Liczba znalezionych	KDTree Build [s]	Quad-Tree Build [s]	KDTree Search [s]	Quad-Tree Search [s]
10000	1282	0.13981	0.13901	0.00080	0.00223
20000	2509	0.27432	0.43278	0.00122	0.00447
30000	3726	0.92486	0.65490	0.00141	0.00614
40000	4983	1.08302	1.17275	0.00209	0.00832
50000	6211	1.42406	1.12220	0.00196	0.01031
60000	7385	1.74145	1.33321	0.00249	0.01206
70000	8498	2.01202	1.57966	0.00285	0.01386
80000	10055	2.35710	1.86330	0.00341	0.01698
90000	11417	3.37494	2.12122	0.00369	0.01851
100000	12571	3.09705	1.72011	0.00391	0.02124

Tabela 4: Czasy budowy i przeszukiwania obu struktur dla rozkładu o dwóch skupiskach punktów

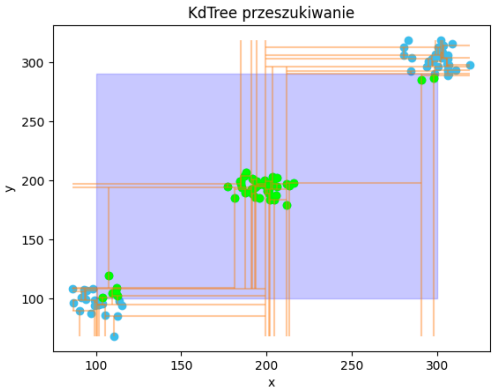
Na podstawie analizy przedstawionej w tabeli Tabela 4 oraz wykresie Rysunek 14, ponownie obserwujemy znaczną przewagę KDTree. Jest to wynikiem relatywnie dużego obszaru poszukiwań, który umożliwia KDTree szybkie i efektywne lokalizowanie punktów w zadanym zakresie. Z kolei QuadTree napotyka trudności w tym zakresie, mając problem z optymalnym znalezieniem punktów, co wpływa negatywnie na jego wydajność w porównaniu do KDTree.

4.5.3.2. Trzy skupiska

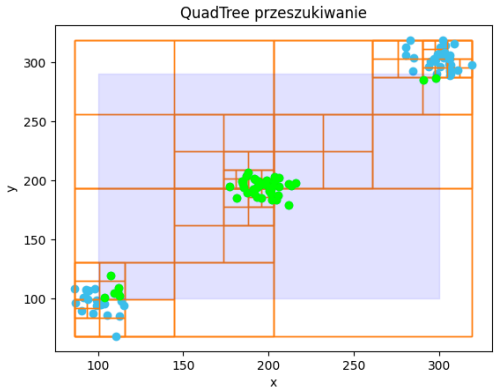
Zbiór ten wykazuje wyraźne zagęszczenie punktów w trzech specyficznych lokalizacjach na płaszczyźnie, co można zaobserwować na Rysunek 15. Sprawia to, ze punkty są jeszcze bardziej nierównomiernie rozłożone.



Rysunek 15: Wizualizacja zbioru o trzech skupiskach punktów

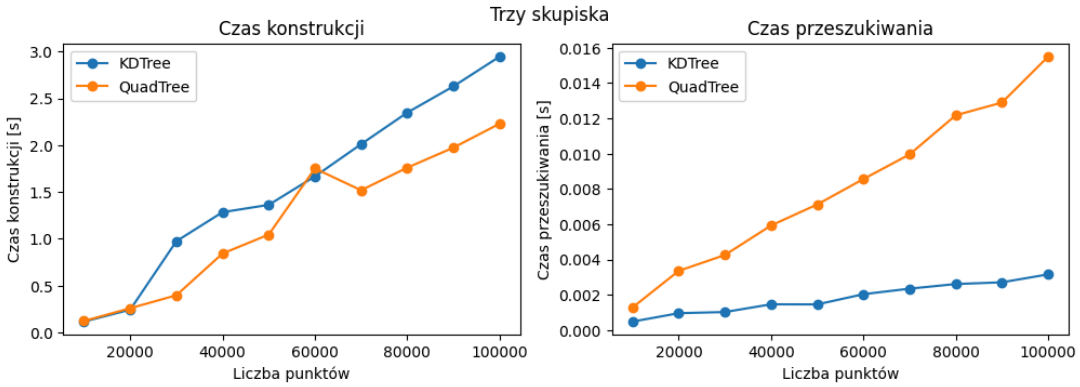


Rysunek 16: Siatka podziału obszaru dla KdTree dla rozkładu o trzech skupiskach punktów



Rysunek 17: Siatka podziału obszaru dla QuadTree dla rozkładu o trzech skupiskach punktów

Wizualizacje ujęte na Rysunek 16 i Rysunek 17 pokazują poprawność budowy oraz wyszukiwania dla obu struktur.



Rysunek 18: Porównanie czasów budowy i przeszukiwania obu struktur dla rozkładu o trzech skupiskach punktów

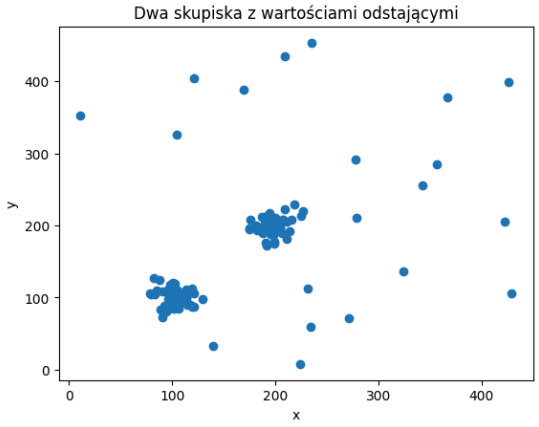
Liczba punktów	Liczba znalezionych	KDTree Build [s]	Quad-Tree Build [s]	KDTree Search [s]	Quad-Tree Search [s]
10000	786	0.11610	0.12284	0.00049	0.00129
20000	1720	0.24333	0.25877	0.00097	0.00336
30000	2450	0.97482	0.39729	0.00103	0.00426
40000	3305	1.28461	0.84312	0.00147	0.00593
50000	4142	1.36253	1.04503	0.00146	0.00713
60000	5002	1.66643	1.75091	0.00204	0.00856
70000	5858	2.01058	1.51872	0.00236	0.00996
80000	6699	2.34798	1.76000	0.00262	0.01218
90000	7483	2.62838	1.97355	0.00271	0.01290
100000	8434	2.94574	2.22944	0.00316	0.01550

Tabela 5: Czasy budowy i przeszukiwania obu struktur dla rozkładu o trzech skupiskach punktów

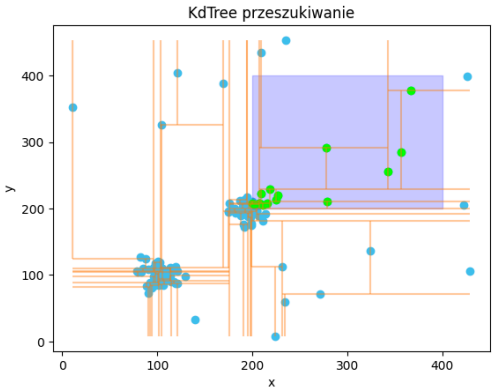
Ponownie obserwujemy, że KDTree utrzymuje przewagę pod względem czasu wyszukiwania, osiągając najlepsze wyniki. Choć zmniejszenie liczby punktów oraz obecność dodatkowych skupisk w obrębie poszukiwanego obszaru nieco osłabiły tę przewagę, wciąż wynosi ona aż 400%. Wynik ten podkreśla kluczowe znaczenie liczby punktów w danym obszarze, szczególnie w kontekście algorytmu QuadTree. Zwiększona liczba punktów w regionie poszukiwań powoduje, że QuadTree traci na efektywności, co wyraźnie widać w analizowanych wynikach.

4.5.3.3. Dwa skupiska z punktami odstającymi

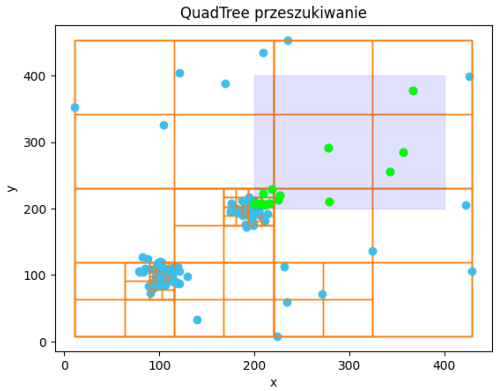
Analizowany zbiór danych jest modyfikacją zbioru opisanego w podpunkcie 4.5.3, wzbogaconą o punkty wyraźnie odstające od głównych skupisk . Tego rodzaju adaptacja pozwala na bardziej szczegółowe zbadanie wpływu obecności odległych punktów na efektywność działania struktur QuadTree i KdTree.



Rysunek 19: Wizualizacja zbioru o dwóch skupiskach z punktami odstającymi

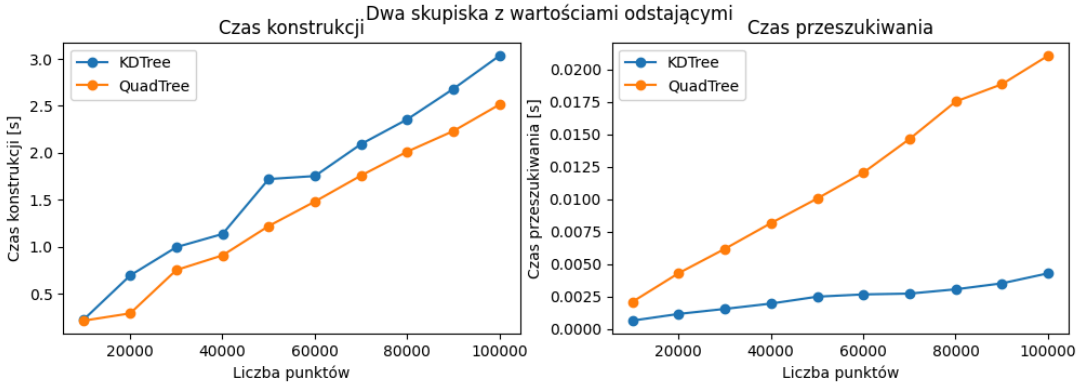


Rysunek 20: Siatka podziału obszaru dla KdTree dla rozkładu o dwóch skupiskach z punktami odstającymi



Rysunek 21: Siatka podziału obszaru dla QuadTree dla rozkładu o dwóch skupiskach z punktami odstającymi

Wizualizacje ujęte na Rysunek 20 i Rysunek 21 pokazują poprawność budowy oraz wyszukiwania dla obu struktur.



Rysunek 22: Porównanie czasów budowy i przeszukiwania obu struktur dla rozkładu o dwóch skupiskach z punktami odstającymi

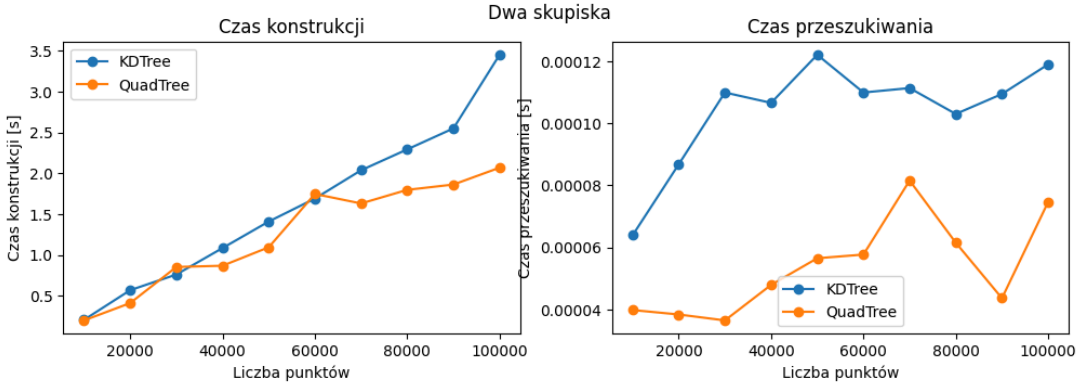
Liczba punktów	Liczba znalezionych	KDTree Build [s]	Quad-Tree Build [s]	KDTree Search [s]	Quad-Tree Search [s]
10050	1207	0.22666	0.21215	0.00064	0.00210
20050	2564	0.69287	0.28929	0.00116	0.00430
30050	3687	0.99403	0.75074	0.00155	0.00617
40050	4924	1.13402	0.90660	0.00196	0.00816
50050	6361	1.71894	1.21897	0.00249	0.01004
60050	7467	1.75085	1.47940	0.00266	0.01204
70050	8737	2.09096	1.75642	0.00272	0.01463
80050	10113	2.35459	2.01099	0.00305	0.01752
90050	11271	2.67937	2.23012	0.00351	0.01885
100050	12529	3.03277	2.51218	0.00429	0.02105

Tabela 6: Czasy budowy i przeszukiwania obu struktur dla rozkładu o dwóch skupiskach z punktami odstającymi

Wyniki przedstawione w Tabela 6 oraz Rysunek 22 ponownie nie wnoszą większego zaskoczenia. Obecność dodatkowych punktów odstających miała niewielki wpływ na zmniejszenie różnicy między strukturami danych. Z analizy można wywnioskować, że w przypadkach, gdzie zadany prostokąt obejmuje dużą liczbę punktów, algorytm KDTree okazuje się bardziej efektywny. Warto jednak postawić pytanie, czy ta przewaga utrzyma się również w sytuacji, gdy liczba punktów w obszarze poszukiwań jest niewielka.

4.5.3.4. Dwa skupiska z prawie pustym obszarem wyszukiwań

Zbiór identyczny charakterystyką jak w podpunkcie 4.5.3, tylko obszar poszukiwań został tak dobrany, aby zminimalizować ilość punktów w nim, jednocześnie pozostając względnie dużym prostokątem.



Rysunek 23: Porównanie czasów budowy i przeszukiwania dla obu struktur dla rozkładu o dwóch skupiskach z prawie pustym polem wyszukiwań

Liczba punktów	Liczba znalezionych	KDTree Build [s]	QuadTree Build [s]	KDTree Search [s]	QuadTree Search [s]
10000	0	0.20641	0.19904	0.00006	0.00004
20000	1	0.56917	0.41150	0.00009	0.00004
30000	3	0.76108	0.85316	0.00011	0.00004
40000	0	1.08495	0.86833	0.00011	0.00005
50000	5	1.40935	1.09438	0.00012	0.00006
60000	0	1.69060	1.74658	0.00011	0.00006
70000	3	2.03700	1.63059	0.00011	0.00008
80000	2	2.29262	1.79724	0.00010	0.00006
90000	3	2.54953	1.86216	0.00011	0.00004
100000	6	3.45063	2.06853	0.00012	0.00007

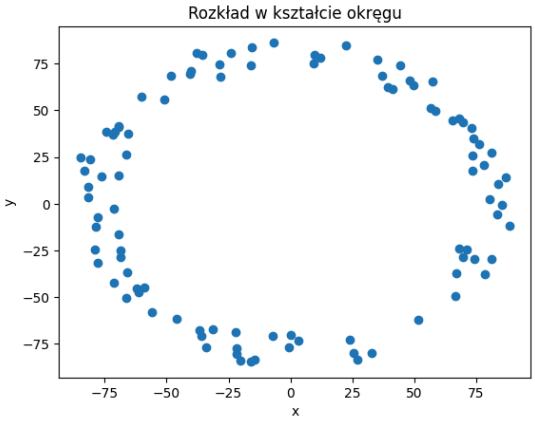
Tabela 7: Czasy budowy i przeszukiwania dla obu struktur dla rozkładu o dwóch skupiskach z prawie pustym polem wyszukiwań

Wyniki przedstawione w tabeli Tabela 7 oraz na wykresie Rysunek 23 wskazują, że struktura QuadTree przewyższa KdTree pod względem efektywności w kontekście przeszukiwania przestrzeni. Przewaga QuadTree wynika przede wszystkim z jego skuteczniejszego zarządzania podziałem pustych obszarów. Podczas gdy KdTree dzieli przestrzeń wzdłuż określonych osi, co w przypadku pustych obszarów może prowadzić do niepotrzebnych operacji przeszukiwania, QuadTree skutecznie ignoruje puste regiony poprzez hierarchiczne dzielenie przestrzeni na cztery części. Dzięki temu proces wyszukiwania w obszarach o niewielkiej liczbie punktów jest znacznie szybszy w przypadku QuadTree.

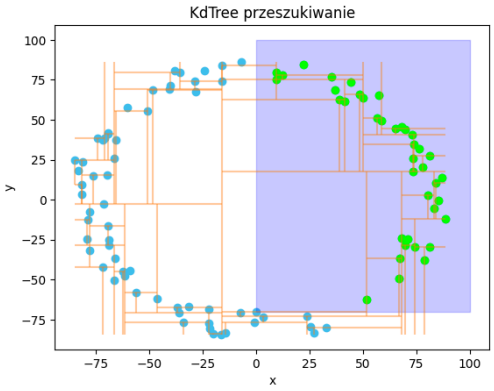
Wyniki te sugerują, że dla obszarów, w których spodziewamy się niskiej gęstości punktów, QuadTree może być bardziej odpowiednią strukturą danych.

4.5.4. Rozkład w kształcie okręgu

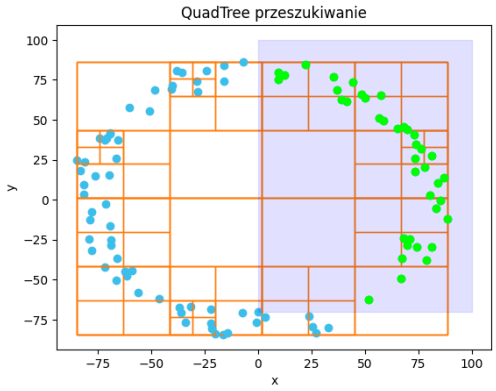
Punkty w tym zbiorze są rozmieszczone w sposób zbliżony do równomiernego na okręgu (Rysunek 24), co stanowi wyzwanie dla algorytmów w kontekście rozmiarów prostokątów w wierzchołkach. Problematyczne jest zarządzanie dużymi różnicami w rozmiarze tych prostokątów, co może prowadzić do nieoptymalnych wyników w procesie budowania i wyszukiwania struktur przestrzennych, takich jak KDTree czy QuadTree.



Rysunek 24: Wizualizacja zbioru dla rozkładu na okręgu

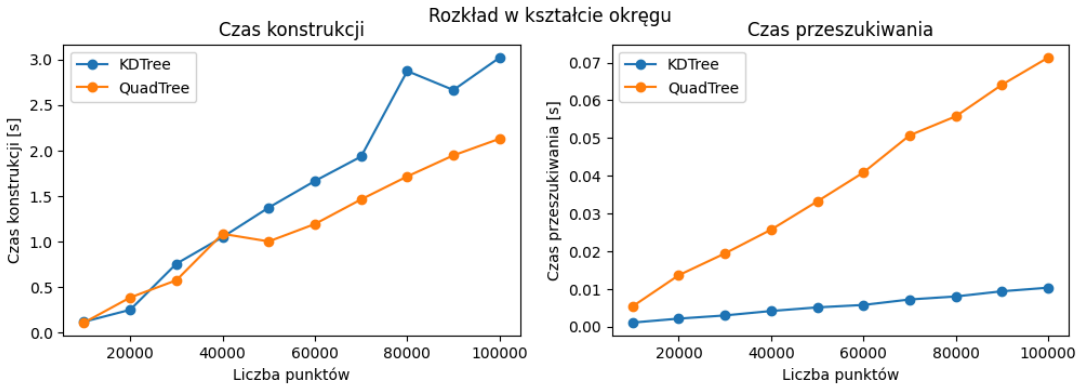


Rysunek 25: Siatka podziału obszaru dla KdTree dla rozkładu na okręgu



Rysunek 26: Siatka podziału obszaru dla QuadTree dla rozkładu na okręgu

Wizualizacje ujęte na Rysunek 25 i Rysunek 26 pokazują poprawność budowy oraz wyszukiwania dla obu struktur.



Rysunek 27: Porównanie czasów budowy i przeszukiwania dla obu struktur dla rozkładu na okręgu

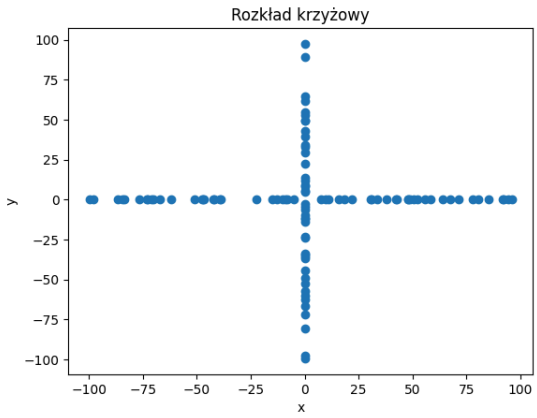
Liczba punktów	Liczba znalezionych	KDTree Build [s]	Quad-Tree Build [s]	KDTree Search [s]	Quad-Tree Search [s]
10000	4194	0.11993	0.11042	0.00107	0.00537
20000	8562	0.25099	0.38605	0.00212	0.01363
30000	12844	0.75636	0.57505	0.00296	0.01941
40000	16840	1.04875	1.08477	0.00414	0.02571
50000	21196	1.37287	1.00138	0.00510	0.03320
60000	25624	1.66477	1.19226	0.00575	0.04088
70000	29795	1.93247	1.46225	0.00720	0.05072
80000	34058	2.87125	1.71477	0.00800	0.05577
90000	38217	2.66314	1.94581	0.00937	0.06415
100000	42577	3.01868	2.12836	0.01032	0.07133

Tabela 8: Czasy budowy i przeszukiwania dla struktur dla rozkładu na okręgu

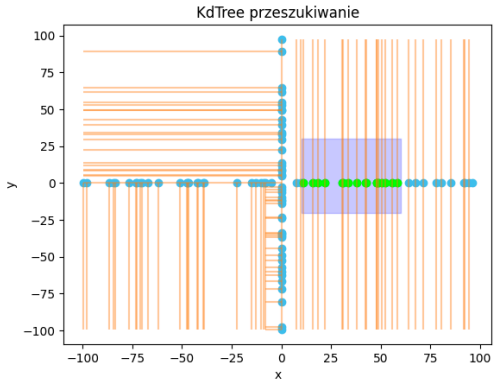
Jak wynika z analizy przedstawionej w tabeli Tabela 8 oraz na wykresie Rysunek 27, struktura KDTree ponownie wykazuje lepszą wydajność. Duża liczba punktów w wyszukiwanym obszarze sprzyja tej strukturze danych, ponieważ jej sposób dzielenia przestrzeni na mniejsze części sprawia, że jest w stanie skutecznie zarządzać dużymi zbiorami punktów. Dodatkowo, równomierne rozmieszczenie punktów na okręgu pozwala na optymalne rozłożenie struktury, co sprzyja szybszemu wyszukiwaniu. Czas budowania dla obu struktur pozostaje zbliżony z lekką przewagą dla QuadTree

4.5.5. Rozkład krzyżowy

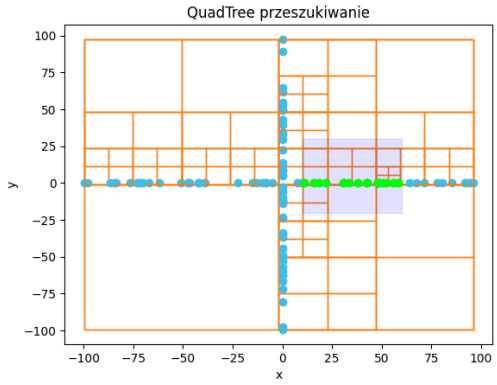
Punkty w tym zbiorze są rozmieszczone w sposób zbliżony do równomiernego na osiach układu współrzędnych (Rysunek 28), co wpływa na charakter podziału przestrzeni przez struktury danych. W przypadku takiego rozmieszczenia, duża liczba punktów współliniowych może prowadzić do nieoptymalnego dzielenia obszarów, w wyniku czego struktura danych może nieefektywnie zarządzać przestrzenią.



Rysunek 28: Wizualizacja zbioru o rozkładzie krzyżowym

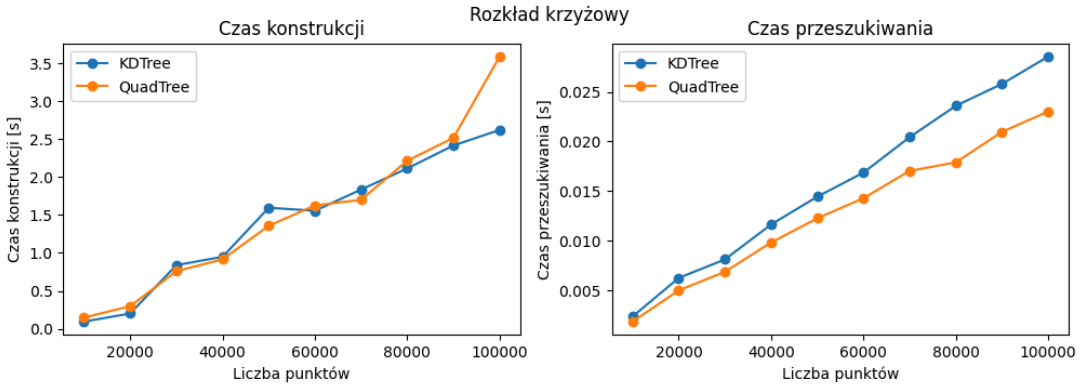


Rysunek 29: Siatka podziału obszaru dla KdTree dla rozkładu krzyżowego



Rysunek 30: Siatka podziału obszaru dla QuadTree dla rozkładu krzyżowego

Wizualizacje ujęte na Rysunek 29 i Rysunek 30 pokazują poprawność budowy oraz wyszukiwania dla obu struktur.



Rysunek 31: Porównanie czasów budowy i przeszukiwania dla obu struktur dla rozkładu krzyżowego

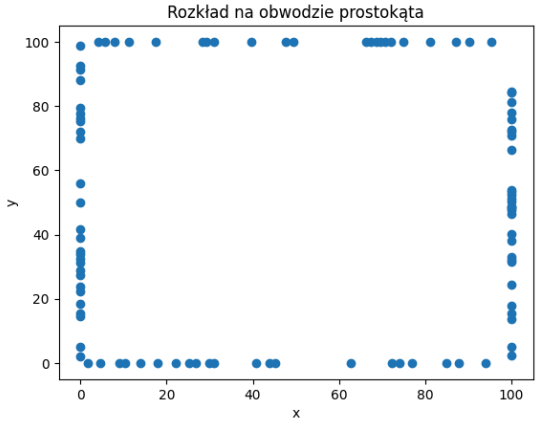
Liczba punktów	Liczba znalezionych	KDTree Build [s]	QuadTree Build [s]	KDTree Search [s]	QuadTree Search [s]
10000	1269	0.09490	0.14799	0.00232	0.00181
20000	2546	0.20136	0.29764	0.00622	0.00496
30000	3751	0.83806	0.75862	0.00809	0.00684
40000	5108	0.94713	0.91360	0.01161	0.00979
50000	6322	1.59325	1.35490	0.01442	0.01224
60000	7576	1.55310	1.62316	0.01686	0.01427
70000	8874	1.83239	1.69800	0.02044	0.01703
80000	9906	2.11042	2.21143	0.02360	0.01789
90000	11312	2.41252	2.51288	0.02581	0.02097
100000	12451	2.61671	3.58213	0.02857	0.02303

Tabela 9: Czasy budowy i przeszukiwania dla struktur dla rozkładu krzyżowego

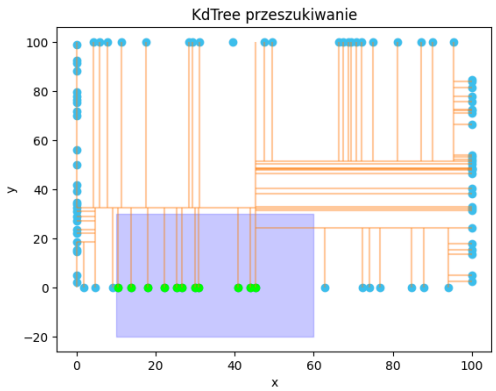
Z analizy przedstawionej w Tabela 9 oraz na Rysunek 31 wynika, że czasy zarówno budowy, jak i wyszukiwania dla obu struktur danych są zbliżone. Takie zjawisko jest efektem zastosowanej optymalizacji w algorytmie KDTree. Podczas procesu wyszukiwania odpowiednich osi podziału dla prostokąta, w celu utworzenia kolejnych wierzchołków w KDTree, weryfikuję, czy liczba punktów o tej samej współrzędnej względem osi podziału jest zgodna z medianą. Jeżeli liczba punktów przekroczy 75% całkowitej liczności zbioru punktów w danym wierzchołku, następuje zmiana osi podziału na kolejną. Takie podejście pozwala na bardziej efektywne tworzenie struktury, minimalizując konieczność przeszukiwania zbyt dużych obszarów. Niemniej jednak, optymalizacja ta może nie przynieść oczekiwanych rezultatów w przypadku, gdy punkty w zbiorze mają bardzo zbliżone współrzędne (choć nie identyczne), co prowadzi do trudności w efektywnym podziale przestrzeni. W takich przypadkach może wystąpić utrata wydajności, gdyż podział nie będzie dostatecznie rozdzielał punktów, co może spowodować spadek efektywności algorytmu.

4.5.6. Rozkład na obwodzie prostokąta

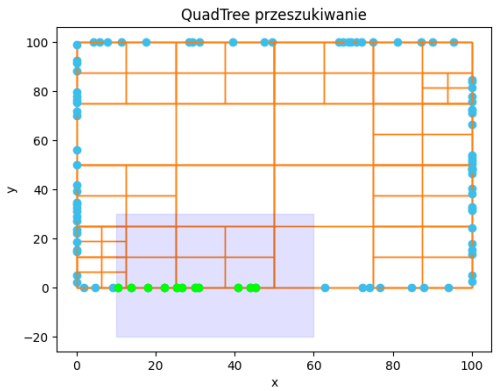
Punkty w tym zbiorze są rozmieszczone w sposób zbliżony do równomiernego na bokach prostokąta (Rysunek 32). taki rozkład jest rozszerzeniem koncepcji rozkładu krzyżowego. Algorytmy ponownie będą musiały się zmierzyć z problematycznym dzieleniem obszarów.



Rysunek 32: Wizualizacja zbioru o rozkładzie na obwodzie prostokąta

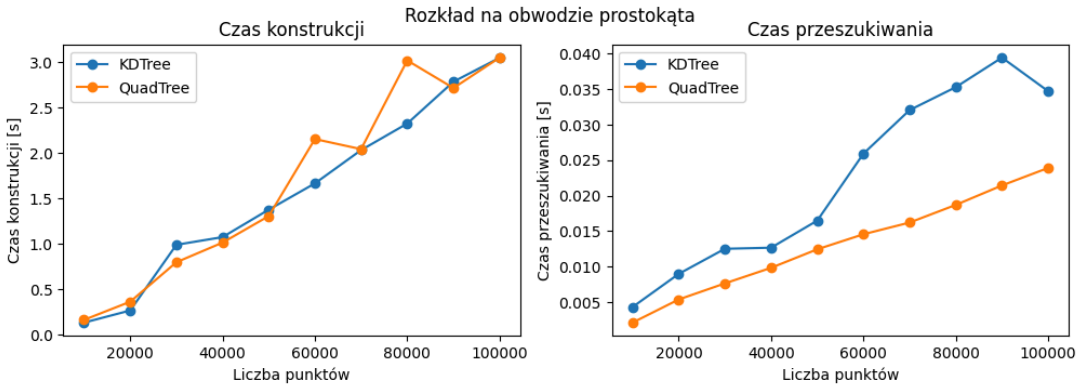


Rysunek 33: Siatka podziału obszaru dla KdTree dla rozkładu na obwodzie prostokąta



Rysunek 34: Siatka podziału obszaru dla QuadTree dla rozkładu na obwodzie prostokąta

Wizualizacje ujęte na Rysunek 33 i Rysunek 34 pokazują poprawność budowy oraz wyszukiwania dla obu struktur.



Rysunek 35: Porównanie czasów budowy i przeszukiwania dla obu struktur dla rozkładu na obwodzie prostokąta

Liczba punktów	Liczba znalezionych	KDTree Build [s]	QuadTree Build [s]	KDTree Search [s]	QuadTree Search [s]
10000	1281	0.13124	0.16121	0.00434	0.00215
20000	2582	0.26449	0.36084	0.00900	0.00540
30000	3871	0.98713	0.79522	0.01253	0.00767
40000	4964	1.07167	1.01128	0.01268	0.00985
50000	6336	1.37368	1.30295	0.01649	0.01245
60000	7569	1.66263	2.15050	0.02590	0.01457
70000	8666	2.02893	2.03961	0.03202	0.01621
80000	9878	2.32102	3.01552	0.03524	0.01789
90000	11268	2.78150	2.71184	0.03939	0.02142
100000	12543	3.04474	3.04651	0.03468	0.02389

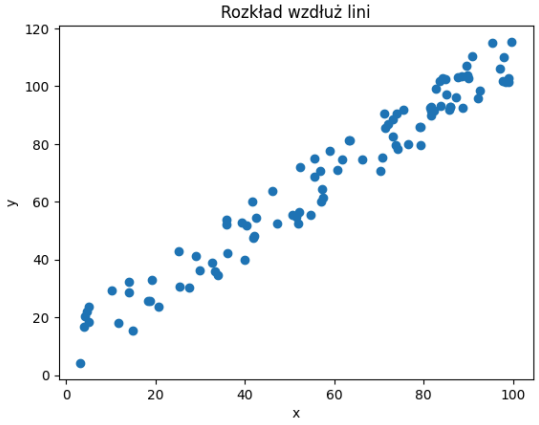
Tabela 10: Czasy budowy i przeszukiwania dla struktur dla rozkładu na obwodzie prostokąta

Z analizy przedstawionej w Tabela 10 oraz na Rysunek 35 wynika, że mimo zastosowanej optymalizacji, struktura KDTree nie radzi sobie lepiej niż QuadTree. Wynika to z bardziej naturalnego sposobu dzielenia przestrzeni przez QuadTree, które lepiej radzi sobie z równomiernym podziałem obszaru, zwłaszcza w kontekście punktów rozmieszczonych współliniowo. W przypadku KDTree, nierównomierne dzielenie obszarów do synów wierzchołka może prowadzić do suboptymalnych wyników, szczególnie w obliczu dużej liczby punktów współliniowych, co skutkuje nieefektywnym podziałem przestrzeni.

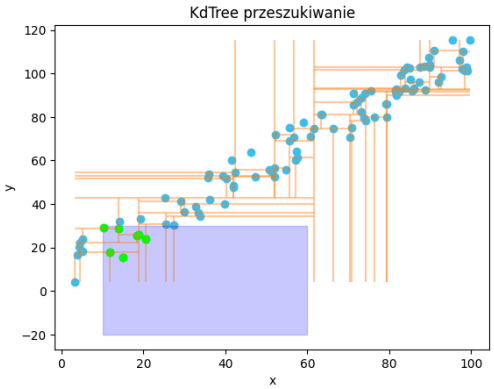
Wnioskiem płynącym z analizy rozkładu punktów w układzie krzyżowym oraz w przestrzeni prostokątnej jest to, że QuadTree lepiej obsługuje punkty współliniowe względem osi. W takich przypadkach struktura ta okazuje się zdecydowanie bardziej efektywna, wykazując przewagę nad KDTree. Optymalizacja w KDTree, choć skuteczna w wielu scenariuszach, nie zapewnia wystarczającej elastyczności w rozwiązywaniu problemów związanych z punktami, które mają bardzo zbliżone współrzędne, co czyni QuadTree bardziej odpowiednią opcją w takich przypadkach.

4.5.7. Rozkład wzdłuż linii

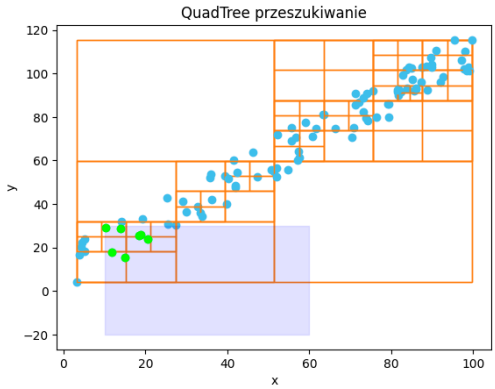
Punkty w tym zbiorze są rozmieszczone w sposób zbliżony do równomiernego wzdłuż prostej o niewielkiej grubości względem jej długości (Rysunek 36). Taki układ powoduje nagromadzenie punktów w specyficzny sposób, zwłaszcza wzdłuż przekątnych prostokątów, co może prowadzić do trudności podczas wyszukiwania. Z uwagi na skupisko punktów w wąskim obszarze, struktury przestrzenne, takie jak KDTree czy QuadTree, mogą napotkać problemy związane z optymalnym podziałem przestrzeni, co skutkuje zwiększeniem czasów wyszukiwania i obniżeniem efektywności algorytmu.



Rysunek 36: Wizualizacja zbioru o rozkładzie wzdłuż linii

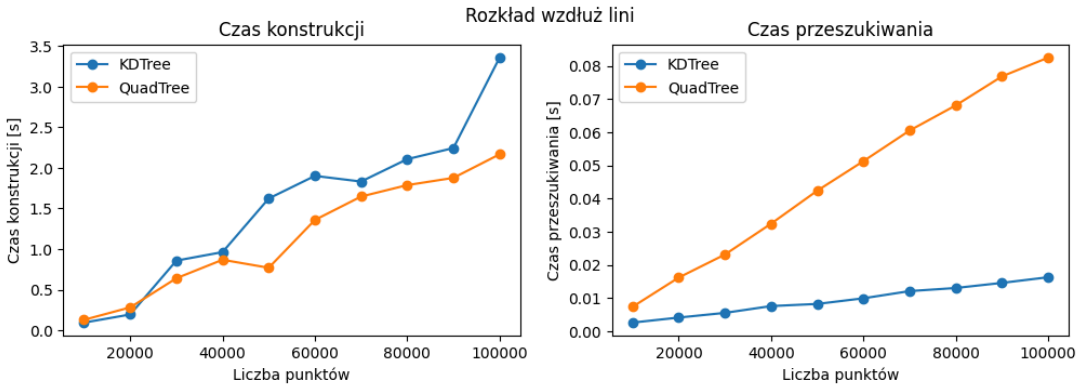


Rysunek 37: Siatka podziału obszaru dla KdTree dla rozkładu wzdłuż linii



Rysunek 38: Siatka podziału obszaru dla QuadTree dla rozkładu wzdłuż linii

Wizualizacje ujęte na Rysunek 37 i Rysunek 38 pokazują poprawność budowy oraz wyszukiwania dla obu struktur.



Rysunek 39: Porównanie czasów budowy i przeszukiwania dla obu struktur dla rozkładu wzdłuż linii

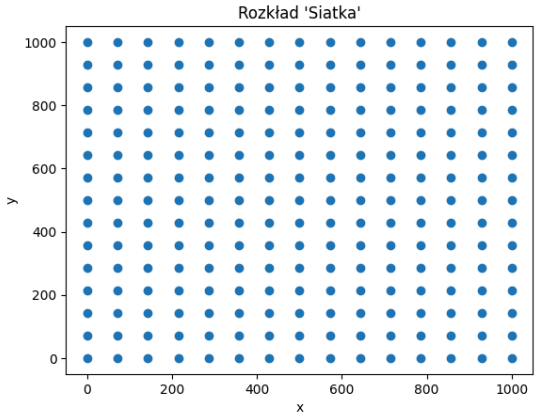
Liczba punktów	Liczba znalezionych	KDTree Build [s]	Quad-Tree Build [s]	KDTree Search [s]	Quad-Tree Search [s]
10000	4989	0.09443	0.12879	0.00256	0.00732
20000	9906	0.19426	0.28169	0.00407	0.01614
30000	14958	0.85764	0.64067	0.00549	0.02309
40000	20017	0.96251	0.86806	0.00754	0.03240
50000	24824	1.62278	0.76961	0.00819	0.04235
60000	30043	1.90130	1.35858	0.00989	0.05127
70000	35092	1.83166	1.64676	0.01206	0.06050
80000	39853	2.10740	1.78769	0.01300	0.06810
90000	45236	2.24434	1.87700	0.01454	0.07683
100000	50055	3.35800	2.16725	0.01624	0.08250

Tabela 11: Czasy budowy i przeszukiwania dla struktur dla rozkładu wzdłuż linii

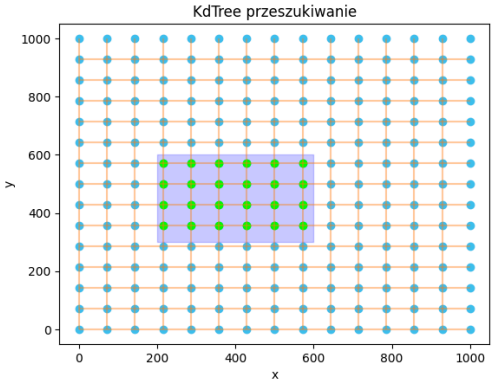
Z analizy przedstawionej w Tabela 11 oraz na Rysunek 39 wynika, że po raz kolejny struktura kdTree wypada korzystniej od QuadTree. Taki rezultat jest efektem podobnych czynników, jak w przypadku zbiorów z dwoma skupiskami punktów. Duża liczba punktów w przeszukiwanym obszarze sprzyja kdTree, co znajduje odzwierciedlenie w czasach wyszukiwania. Natomiast czas budowy obu struktur jest zbliżony, choć z niewielką przewagą na korzyść QuadTree.

4.5.8. Rozkład „Siatka”

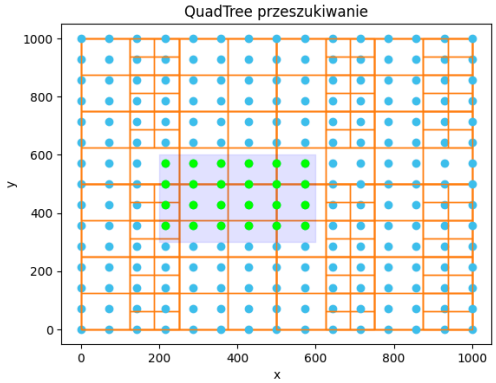
Zbiór składa się z punktów ułożonych w równych odległościach od siebie (Rysunek 40). Takie rozmieszczenie przypomina rozkład jednostajny, lecz dokłada problem współliniowych punktów względem osi ukladu współrzędnych.



Rysunek 40: Wizualizacja zbioru o rozkładzie „Siatka”

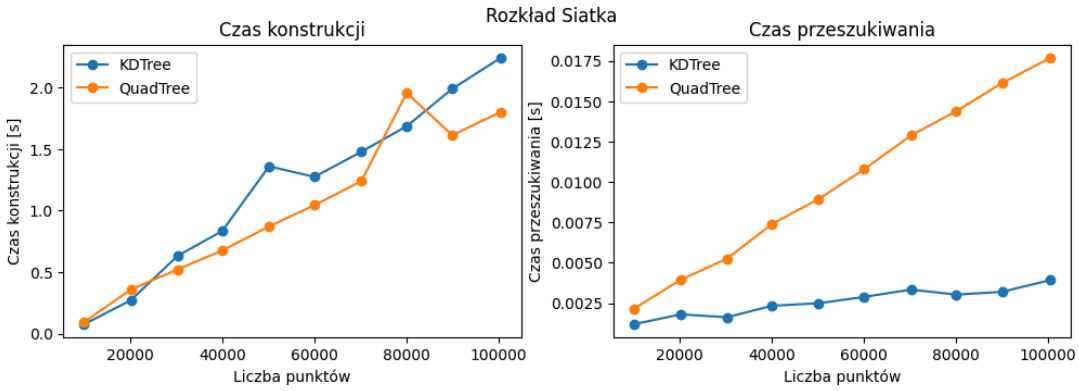


Rysunek 41: Siatka podziału obszaru dla KdTree dla rozkładu „Siatka”



Rysunek 42: Siatka podziału obszaru dla QuadTree dla rozkładu „Siatka”

Wizualizacje ujęte na Rysunek 41 i Rysunek 42 pokazują poprawność budowy oraz wyszukiwania dla obu struktur.



Rysunek 43: Porównanie czasów budowy i przeszukiwania dla obu struktur dla rozkładu „Siatka”

Liczba punktów	Liczba znalezionych	KDTree Build [s]	Quad-Tree Build [s]	KDTree Search [s]	Quad-Tree Search [s]
10000	1600	0.07683	0.09583	0.00119	0.00215
20164	3136	0.27291	0.35871	0.00180	0.00394
30276	4900	0.63179	0.52247	0.00162	0.00526
40000	6400	0.83482	0.67591	0.00234	0.00739
50176	8100	1.36052	0.87163	0.00249	0.00895
60025	9409	1.27411	1.04431	0.00288	0.01078
70225	11025	1.47824	1.23922	0.00334	0.01288
80089	12769	1.68649	1.95646	0.00303	0.01439
90000	14400	1.99041	1.61232	0.00320	0.01612
100489	16129	2.24103	1.80197	0.00392	0.01768

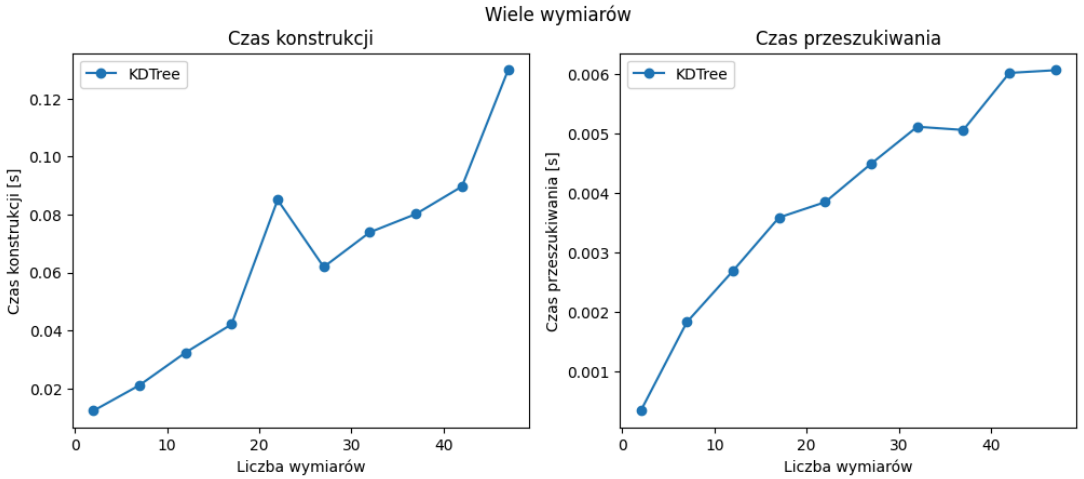
Tabela 12: Czasy budowy i przeszukiwania dla struktur dla rozkładu „Siatka”

Z analizy przedstawionej w Tabela 12 oraz na Rysunek 43 wynika, że pomimo problematycznego rozkładu punktów, struktura kdTree nadal okazuje się lepszym wyborem. Choć czas budowy kdTree jest dłuższy, to czas wyszukiwania w przypadku QuadTree, mimo sprzyjającego dla tej struktury rozkładu punktów, okazuje się niewystarczająco szybki. Wskazuje to na fakt, że pomimo potencjalnych trudności związanych z rozkładem punktów, kdTree lepiej radzi sobie w przypadku większych zbiorów danych, oferując lepszą efektywność podczas wyszukiwania.

4.5.9. Testy specyficzne dla KdTree

4.5.9.1. Test wydajnościowy wraz ze wzrastającą liczbą wymiarów

KDTree, w przeciwieństwie do QuadTree, umożliwia operowanie na zbiorach punktów w przestrzeniach o wymiarach większych niż dwa. Warto zatem przeanalizować, jak zmieniają się czasy budowy i wyszukiwania w KDTree w przypadku danych wielowymiarowych. Badanie to pozwoli ocenić efektywność tej struktury w bardziej złożonych przestrzeniach.



Rysunek 44: Porównanie czasów budowy i przeszukiwania dla KdTree w zależności od ilości wymiarów punktów

Liczba punktów	Liczba wymiarów	KDTree Build [s]	KDTree Search [s]
1000	2	0.01228	0.00034
1000	7	0.02109	0.00183
1000	12	0.03237	0.00269
1000	17	0.04214	0.00359
1000	22	0.08508	0.00385
1000	27	0.06192	0.00450
1000	32	0.07390	0.00512
1000	37	0.08020	0.00506
1000	42	0.08971	0.00602
1000	47	0.12983	0.00607

Tabela 13: Czasy budowy i przeszukiwania dla KdTree w zależności od ilości wymiarów punktów

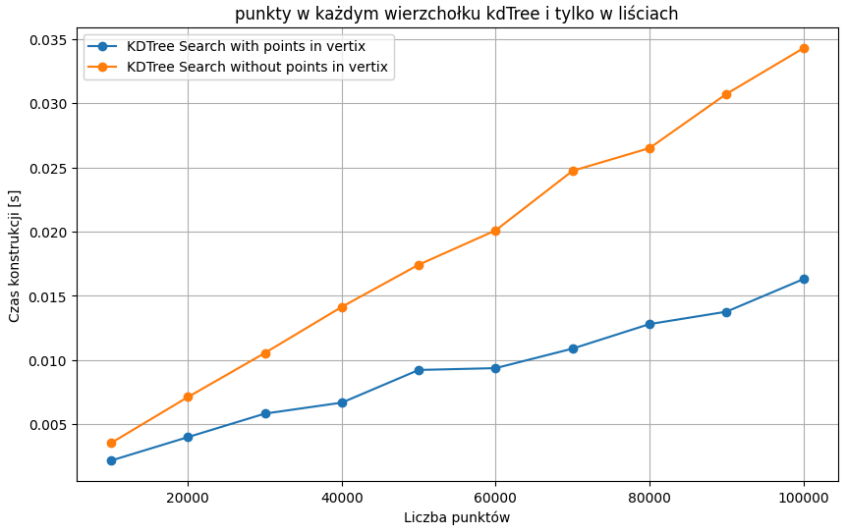
Jak przedstawiono w Tabela 13 oraz na Rysunek 44, czas konstrukcji KdTree pozostaje względnie liniowy, co potwierdza stabilność tej struktury w kontekście wzrastającej liczby wymiarów. Wyjątkowe odchylenie dla 22 wymiarów można przypisać specyficznemu rozkładowi danych, który mógł skomplikować proces budowy KdTree.

Czas przeszukiwania również wykazuje wzrost bliski liniowemu. Niemniej, wraz ze zwiększaniem liczby wymiarów, różnice w czasie wyszukiwania pomiędzy kolejnymi wymiarami stają się coraz mniejsze. Sugeruje to, że wzrost liczby wymiarów wpływa na wyszukiwanie w KdTree w sposób mniej znaczący niż na jego konstrukcję, co może wynikać z optymalnego podziału przestrzeni nawet w wielowymiarowych przypadkach.

4.5.10. Test wydajnościowy trzymania tablicy punktów w każdym wierzchołku

KDTree używane w porównaniach z QuadTree było zbudowane w sposób, w którym każdy wierzchołek zawierał listę punktów przypisanych do danego obszaru. Alternatywnym podejściem jest umieszczanie punktów wyłącznie w liściach drzewa. Taka konstrukcja pozwala na znaczne zmniejszenie wykorzystania pamięci przez strukturę, jednak może prowadzić do wydłużenia czasu wyszukiwania.

Aby lepiej zrozumieć różnice wynikające z tych podejść, warto przeprowadzić testy porównawcze, analizując wpływ struktury węzłów na czas budowy oraz przeszukiwania KDTree. Wyniki mogą dostarczyć istotnych wskazówek dotyczących wyboru optymalnej implementacji w zależności od wymagań dotyczących pamięci i szybkości operacji.



Rysunek 45: Porównanie czasów przeszukiwania dla KdTree w zależności od sposobu przechowywania punktów w strukturze

Liczba punktów	KDTree Search with points in vertex [s]	KDTree Search without points in vertex [s]
10000	0.00217	0.00354
20000	0.00400	0.00712
30000	0.00583	0.01056
40000	0.00669	0.01415
50000	0.00923	0.01743
60000	0.00938	0.02011
70000	0.01089	0.02474
80000	0.01281	0.02651
90000	0.01377	0.03073
100000	0.01631	0.03429

Tabela 14: Czasy budowy i przeszukiwania dla KdTree w zależności od sposobu przechowywania punktów w strukturze

Jak zaprezentowano w Tabela 14 oraz na Rysunek 45, implementacja KDTree oparta na przechowywaniu listy punktów w każdym wierzchołku wykazała się wyraźnie wyższą wydajnością w porównaniu do alternatywnego rozwiązania. Przewaga tej metody wynika z możliwości natychmiastowego zwrócenia tablicy punktów dla wierzchołków, których prostokąt w całości mieści się w obszarze poszukiwań. W przypadku braku takiej tablicy konieczne jest przeprowadzenie rekurencyjnej agregacji punktów z potomków, co zwiększa złożoność operacji wyszukiwania.

Pomimo tej różnicy, rozwiązanie bez list punktów w wierzchołkach nie charakteryzuje się drastycznie niższą wydajnością. Wyniki są w znacznym stopniu zależne od specyfiki testowanego przypadku, w tym rozkładu punktów oraz wymiarów prostokąta, w którym przeprowadzane jest wyszukiwanie. Warto zauważyć, że wybór odpowiedniego rozwiązania powinien uwzględniać nie tylko czas wyszukiwania, ale również ograniczenia pamięciowe oraz wymagania konkretnej aplikacji.

5. Podsumowanie

Na podstawie przeprowadzonych testów dla wcześniej omówionych zbiorów danych możemy stwierdzić, że zaimplementowane przez nas struktury działają prawidłowo i poprawnie identyfikują podzbiór punktów znajdujących się na zadanej płaszczyźnie.

W każdym z testowanych przypadków czas budowy struktur danych był porównywalny, co wskazuje na poprawność implementacji obu algorytmów.

Jeśli chodzi o wyszukiwanie punktów, można zauważyć, że dla zbiorów o dużym i średnim zagęszczeniu, takich jak rozkłady jednolite lub normalne, drzewo KD okazuje się lepszym rozwiązaniem.

Z kolei w przypadku zbiorów rzadkich lub takich, w których wiele punktów leży na jednej linii, przewagę zyskuje drzewo ćwiartkowe. Jego struktura pozwala szybciej eliminować puste węzły, co przekłada się na krótszy czas wyszukiwania.

Jednak w sytuacji, gdy nie znamy charakterystyki rozkładu danych, bardziej uniwersalnym wyborem będzie drzewo KD.

6. Źródła i inspiracje programistyczne

- https://medium.com/@isurangawarnasooriya/exploring-kd-trees-a-comprehensive-guide-to-implementation-and-applications-in-python-3385fd56a246
- https://scipython.com/blog/quadtree-1-background/
- https://github.com/aghbit/Algorytmy-Geometryczne
- 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://en.wikipedia.org/wiki/Range_searching
- Wykłady z Algorytmów Geometrycznych, prowadzone przez dr inż. Barbarę Głut, na 3 semestrze Informatyki AGH WI.

Podział pracy

- Dariusz Marecik - Kd-drzewa
- Piotr Sękulski - QuadTree