

A G H

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wyszukiwanie geometryczne
KDTree i QuadTree
Dokumentacja

Mykola Haltiuk
Konrad Krzemiński
wtorek 14:40 B, grudzień 2020
Algorytmy geometryczne

Spis treści

I Wymagania techniczne	4
II Dokumentacja	5
1 Oznaczenia	5
2 Geometry	5
2.1 Point	5
2.2 Rect	6
3 Visualiser	8
3.1 VisualiserContainer	8
3.2 Visualiser	9
3.3 BuildVisualiser	10
3.4 SearchVisualiser	11
4 KDTree	12
5 QuadTree	14
III Poradnik do wykorzystania	16
1 Proste dodatkowe narzędzia	16
1.1 Mierzenie czasu	16
1.2 Testy jednostkowe	16
1.3 Testy różnorodne	16
2 Podstawowe wykorzystanie drzew	17
2.1 KDTree	17
2.2 QuadTree	17
3 Używanie testów	18
4 Wizualizacja	19
4.1 KDTree	21
4.1.1 Budowanie	21
4.1.2 Sprawdzenie czy istnieje punkt	21
4.1.3 Szukanie punktów w prostokącie	22
4.2 QuadTree	23
4.2.1 Budowanie / Wstawianie punktów	23
4.2.2 Szukanie punktów w prostokącie	24
5 Dodatkowe uwagi	24
5.1 Przepelenie stosu	24
5.2 Przepelenie pamięci	25

5.3	Duże zbiory a wizualizacja	25
IV	Sprawozdanie	26
1	Wstęp teoretyczny	26
1.1	Struktura QuadTree	26
1.2	Struktura KDTree	27
2	Testowanie dla różnych danych wejściowych	28
2.1	Zbiór o rozkładzie jednostajnym	28
2.2	Zbiór o rozkładzie normalnym	30
2.3	Siatka	32
2.4	Klastry	34
2.5	Wartości odstające (outliers)	38
2.6	Krzyż	42
2.7	Prostokąt	45
3	Testowanie dla różnych wartości <i>capacity</i>	47
4	Testowanie dla różnej ilości wymiarów	49
5	Wnioski	50

Część I

Wymagania techniczne

Struktury danych zostały zaimplementowane w języku **Python** przy użyciu wbudowanych i zainstalowanych bibliotek. Kod źródłowy znajduje się w repozytorium GitHub.

Niżej jest przedstawiona tabela bibliotek i narzędzi wraz z ich wersjami (o ile program był testowany na dwóch różnych komputerach, to niektóre punkty jak np CPU są zapisane dwukrotnie):

System operacyjny n1	Microsoft Windows 10.0.19041.685
CPU n1	Intel i7-9750H 64-bit
System operacyjny n2	macOS Catalina 10.15.7
CPU n2	Intel Core i5 1.4GHz 64-bit
Python interpreter	Python 3.7.5 64-bit
Numpy version	1.18.3
Matplotlib version	3.3.3

Tablica 1: Dane techniczne

Testy zostały wykonane na komputerze n2.

Pełny program zawiera następujące pliki:

- Pakiet **geometry**
 - **__init__.py**
 - **Point.py**
 - **Rect.py**
- Pakiet **visualiser**
 - **__init__.py**
 - **VisualiserContainer.py**
 - **Visualiser.py**
 - **BuildVisualiser.py**
 - **SearchVisualiser.py**
- **kdtree.py**
- **quadtree.py**
- **test.py**
- **tests.py**
- **utils.py**

Zalecane jest używanie **PyCharm IDE**.

Część II

Dokumentacja

Niżej są opisane klasy i metody publiczne. Prywatne klasy i metody nie zostały opisane, ponieważ nie składają **API**, czyli nie jest przewidywane korzystnie z nich wprost.

1 Oznaczenia

- **np** - skrócona nazwa biblioteki **numpy**
- **plt** - skrócona nazwa pakietu **pyplot** biblioteki **matplotlib**
- **Property** - pythonowe `@property` - dekorator, który umożliwia robienie getterów i setterów w wyglądzie zwykłego modyfikowania/odczytu wprost z atrybutu
- **Overridden** - metoda z klasy nadzędnej, która została nadpisana w opisywanej klasie

2 Geometry

Ten pakiet zapewnia podstawowe klasy elementów geometrycznych: punkt i prostokąt. Jest często używany w implementacji.

2.1 Point

Klasa Point odpowiada za reprezentację punktów. Zapewnia dużo możliwości ich przetwarzania, tworzenia nowych. Warto zauważyć, że stworzony punkt jest niemutowalny (oczywiście w stosunku do metod i properties).

- **Point(points)**

Tworzony jest punkt na podstawie dowolnego **Iterable** i **Sized**, które da się przetworzyć do **np.array**.

- **__eq__**

Zwraca **True**, jeżeli każdy wymiar obu punktów ma taką samą ilość wymiarów i wartość dla każdego wymiaru.

- **__str__ / __repr__**

Zapewnia ładne wypisywanie punktów.

- **axes_count**

Property, która trzyma ilość wymiarów punktu.

- **point**

Property, która zwraca kopię **np.array**.

- **x**

Property, która zwraca wartość dla pierwszego wymiaru.

- **y**

Property, która zwraca wartość dla drugiego wymiaru, jeżeli taki istnieje. W przypadku jednowymiarowego punktu powoduje **AssertionError** z odpowiednim komunikatem.

- **get_axis(axis)**

Metoda zwraca wartość dla wymiaru wskazanego w **axis**. **axis** jest liczbą całkowitą (**int**).

- **follows(other)**

Przyjmuje obiekt klasy **Point** i sprawdza czy wartości wszystkich wymiarów tego punktu są większe lub równe od odpowiednich w punkcie **other**.

- **precedes(other)**

Przyjmuje obiekt klasy **Point** i sprawdza czy wartości wszystkich wymiarów tego punktu są mniejsze lub równe od odpowiednich w punkcie **other**.

- **find_min(other)**

Przyjmuje obiekt klasy **Point** i zwraca nowy obiekt tej samej klasy biorąc dla każdego wymiaru minimalną wartość z tego i przekazanego punktu.

- **find_max(other)**

Przyjmuje obiekt klasy **Point** i zwraca nowy obiekt tej samej klasy biorąc dla każdego wymiaru maksymalną wartość z tego i przekazanego punktu.

2.2 Rect

Ta klasa reprezentuje prostokąt. Daje możliwość korzystania z dużej ilości metod pomocniczych. Obiekt tej klasy jest niemutowalny. Jak tu, tak i dalej opisywany jest jako 'prostokąt', chociaż nie warto go traktować jak tylko dwuwymiarową strukturę. To jest dwuwymiarowy obszar/zakres, który łatwo sobie wyobrazić w postaci prostokąta w przypadku dwóch wymiarów.

- **Rect(lowerleft, upperright)**

Przyjmuje dwa punkty, które muszą być lub obiektami **Point** lub obiektami, z których da się stworzyć obiekt klasy **Point**. Jeżeli wymiary tych punktów nie zgadzają się, to powoduje powstanie **TypeError** z odpowiednim komunikatem. Dodatkowo jeżeli wartości wszystkich wymiarów punktu **lowerleft** nie są mniejsze lub równe od wartości odpowiednich wymiarów punktu **upperright**, to powoduje **ValueError** z odpowiednim komunikatem.

- **__eq__**

Sprawdza czy punkty **lowerleft** i **upperright** obu prostokątów zgadzają się (są równe).

- **__str__ / __repr__**

Zapewnia ładne wypisywanie obiektów klasy **Rect**.

- **lowerleft**

Property, która trzyma lewy dolny punkt prostokąta.

- **upperright**

Property, która trzyma prawy górny punkt prostokąta.

- **dimensions**

Property, która trzyma ilość wymiarów dla tego prostokąta.

- **ptp_by_axis(axis)**

Przyjmuje **axis** jako **int**. Zwraca różnicę wartości prawego górnego punktu dla tego wymiaru i lewego dolnego. Czyli np. dla **axis = 0** zwróci szerokość prostokąta, a dla **axis = 1** - jego wysokość i td.

- **from_points(points)**

classmethod Metoda klasowa, która tworzy minimalny prostokąt zawierający wszystkie punkty z argumentu **points**. **points** - lista obiektów klasy **Point**.

- **overlaps(rect)**

Przyjmuje jako **rect** obiekt klasy **Rect**. Sprawdza czy ten i przekazany prostokąt mają wspólny obszar, nawet jeżeli to jest tylko wspólny brzeg.

- **contains_rect(rect)**

Przyjmuje jako **rect** obiekt klasy **Rect**. Sprawdza czy przekazany prostokąt całkiem zawiera się w danym.

- **contains_point(point)**

Przyjmuje jako **point** obiekt klasy **Point** lub taki, z którego da się stworzyć obiekt klasy **Point**. Sprawdza czy ten punkt należy do tego prostokąta.

- **divide(axis, threshold)**

Przyjmuje **axis** jako **int**, to jest wymiar, a **threshold** jako dowolną liczbę (zmiennoprzecinkową, całkowitą i in.). Jeżeli wartość **threshold** nie mieści się w zakresie wartości **lowerleft** i **upperright** dla wymiaru **axis**, to powoduje **ValueError** z odpowiednim komunikatem.

Na podstawie podanego wymiaru i wartości - dzieli dany prostokąt na dwa w podanym wymiarze i po danej wartości. W przypadku początkowego prostokąta z **lowerleft** = (1, 4), **upperright** = (15, 10), i danymi **axis** = 1, **threshold** = 8, to otrzymujemy jako wynik dwa prostokąty. Pierwszy ma **lowerleft** = (1, 4), **upperright** = (15, 8), a drugi - **lowerleft** = (1, 8), **upperright** = (15, 10).

- **intersection(rect)**

Przyjmuje **rect** jako obiekt klasy **Rect**. Tworzy nowy prostokąt, który jest częścią wspólną danego i przekazanego prostokąta. W przypadku braku części wspólnej zwrotnica jest wartość **None**.

- **add_border(border_width_ratio, preserve_type=True)**

Przyjmuje argument **border_width_ratio**, który jest dowolną wartością numeryczną. Metoda tworzy nowy prostokąt, rozszerzając go w każdą stronę. Poziom powiększenia dla pewnego wymiaru jest wyznaczony poprzez wymnożenie **border_width_ratio** przez różnicę **upperright** i **lowerleft** dla tego wymiaru, czyli to, co jest opisane w **ptp_by_axis**.

Dodatkowo argument **bool - preserve_type** decyduje czy zmieniać typ wartości. Tzn., że jak mamy początkowy typ **int**, i zwiększymy o **1.4**, to tak naprawdę dostaniemy zwiększenie tylko o **1**. A jeżeli nie zachowujemy typu, to zmieniamy go na **float**, co już powoduje pełne uwzględnienie powiększenia.

3 Visualiser

Ten pakiet zapewnia wszystko, co jest potrzebne do wizualizacji.

3.1 VisualiserContainer

Ta klasa odpowiada za jedną scenę w wizualizacji. Trzyma w sobie wszystkie obiekty, które powinny zostać narysowane.

- **VisualiserContainer()**

Tworzy obiekt tej klasy. Nie przyjmuje żadnych argumentów.

- **__copy__**

Zapewnia możliwość kopowania tego kontenera.

- **copy()**

Metoda, która wywołuje **__copy__**.

- **points_collections**

Property, która trzyma listę kolekcji punktów.

- **lines_collections**

Property, która trzyma listę kolekcji linii.

- **rects**

Property, która trzyma listę prostokątów.

- **is_empty()**

Sprawdza czy dany kontener jest pusty (nie trzyma żadnych punktów, linii i prostokątów).

- **add_points(points, **kwargs)**

Przyjmuje listę punktów (każdy punkt jest listą/krotką, nie obiektem klasy **Point**). Dodatkowo przyjmuje parametry kluczowe, które odpowiadają parametrom z biblioteki **matplotlib**. Dodaje te punkty z parametrami w postaci kolekcji do siebie.

- **add_lines(lines, **kwargs)**

Przyjmuje listę linii (każda linia jest listą/krotką punktów, które też są listami/krotkami, nie obiektami klasy **Point**). Dodatkowo przyjmuje parametry kluczowe, które odpowiadają parametrom z biblioteki **matplotlib**. Dodaje te linie z parametrami w postaci kolekcji do siebie.

- **add_rect(rect, **kwargs)**

Przyjmuje jako **rect** obiekt klasy **Rect** i dodaje do swojej kolekcji. Dodatkowo przyjmuje parametry kluczowe, które odpowiadają parametrom z biblioteki **matplotlib**.

3.2 Visualiser

Klasa, która jest klasą nadzczną **BuildVisualiser** i **SearchVisualiser**. Ona zapewnia wszystko co jest potrzebne do robienia prostej animacji.

- **Visualiser(scope_rect)**

Przyjmuje obiekt klasy **Rect**, który wyznacza prostokąt, który będzie pokazywany przez **plt**. Powoduje **ValueError** z odpowiednim komunikatem, jeżeli przekazany prostokąt nie jest dwuwymiarowym. Tworzy nową scenę i ustawia ją jako aktualną.

- **add_points(points, **kwargs)**

Przyjmuje listę punktów (każdy punkt jest listą/krotką, nie obiektem klasy **Point**). Dodatkowo przyjmuje parametry kluczowe, które odpowiadają parametrom z biblioteki **matplotlib**. Dodaje te punkty z parametrami do ostatniej sceny.

- **add_lines(lines, **kwargs)**

Przyjmuje listę linii (każda linia jest listą/krotką punktów, które też są listami/krotkami, nie obiektami klasy **Point**). Dodatkowo przyjmuje parametry kluczowe, które

odpowiadają parametrom z biblioteki **matplotlib**. Dodaje te linie z parametrami do ostatniej sceny.

- **add_rect(rect, **kwargs)**

Przyjmuje jako **rect** obiekt klasy **Rect** i dodaje do ostatniej sceny. Dodatkowo przyjmuje parametry kluczowe, które odpowiadają parametrom z biblioteki **matplotlib**.

- **next_scene()**

Tworzy nową pustą scenę i przełącza aktualną na nową.

- **clear()**

Usuwa wszystkie sceny i wywołuje **next_scene**.

- **draw()**

Rysuje wszystko co jest zapisane w scenach zaczynając od pierwszej. Przełączanie scen i używanie tego jest opisane w poradniku do korzystania. Powoduje **ValueError** z odpowiednim komunikatem, jeżeli nie został dodany żaden obiekt.

3.3 BuildVisualiser

Klasa dziedzicząca od **Visualiser**. Jest stosowana do wizualizacji tworzenia drzewa i jest odpowiednio zmodyfikowana, żeby uczynić ten proces łatwiejszym i prostszym. Polega na tym, że akumuluje przekazane punkty i linie, więc nie jest koniecznym trzymanie już skonstruowanych części, a tylko przekazywanie nowych.

- **BuildVisualiser(scope_rect)**

Przyjmuje obiekt klasy **Rect**, który określa zakres pokazywany przez **plt**, czyli jak w klasie nadzędnej.

- **add_lines(lines, **kwargs)**

Overriden. Przyjmuje listę linii (każda linia jest listą/krotką punktów, które też są listami/krotkami, nie obiektami klasy **Point**). Dodatkowo przyjmuje parametry kluczowe, które odpowiadają parametrom z biblioteki **matplotlib**. Dodaje te linie z parametrami do ostatniej sceny i do listy już zapisanych linii. Czyli przekazane w tej chwili linii zostaną pokazane we wszystkich następnych scenach, tylko z argumentami **kwargs**, które zostały ustawione przez **set_lines_kwargs**.

- **next_scene()**

Overriden. Tworzy nową pustą scenę i przełącza aktualną na nową. Do pustej sceny dodaje akumulowane punkty i linie z odpowiednio ustalonymi **kwargs** przez **set_lines_kwargs** i **set_points_kwargs**. Jeżeli nie zostały one ustawione, to przekazywane są puste słowniki. Czyli **plt** rysuje to z domyślnymi parametrami.

- **set_lines_kwarg(**kwargs)**

Przyjmuje argumenty nazwane, czyli np. `color='orange'`, `linewidth=10`. Potem akumulowane linie są rysowane z tymi argumentami. Zwraca dany obiekt, co daje możliwość robienia pipeline.

- **set_points_kwarg(**kwargs)**

Przyjmuje argumenty nazwane, czyli np. `color='navy'`, `s=2`. Potem akumulowane punkty są rysowane z tymi argumentami. Zwraca dany obiekt, co daje możliwość robienia pipeline.

- **set_points(points)**

Ustawia przekazaną listę punktów (nie obiektów `Point`) jako listę domyślnych punktów (są rysowane we wszystkich następnych scenach). Zwraca dany obiekt, co daje możliwość robienia pipeline.

- **add_default_points(points)**

Dodaje te punkty do listy już przypisanych punktów. Czyli przekazane w tej chwili punkty zostaną pokazane we wszystkich następnych scenach, tylko z argumentami `kwargs`, które zostały ustalone przez `set_points_kwarg`.

- **final_scene_container()**

Zwraca kopię ostatniej sceny, czyli obiekt klasy `VisualerContainer`. Dlatego jest sens zrobienia ostatniej sceny jako tylko domyślne punkty i linie zakumulowane, czyli 'już zbudowane drzewo'. Wtedy da się wykorzystać tę scenę jako podstawę do wizualizacji innych możliwości implementowanych struktur.

3.4 SearchVisualiser

Ta klasa dziedziczy od `Visualiser`. Jest stosowana do wizualizacji różnych operacji na implementowanych drzewach. Ma wygodne metody do tego, i zapewnia możliwość korzystania z niej wielokrotnie.

- **SearchVisualiser(scope_rect, background)**

Przyjmuje obiekt klasy `Rect`, który określa zakres pokazywany przez `plt`, czyli jak w klasie nadzędnej. Dodatkowo przyjmuje obiekt klasy `VisualiserContainer`, kopia która będzie podstawowym kontenerem (tło) do wszystkich wizualizacji pokazywanych przez ten obiekt.

- **clear(new_background=None)**

Overriden. Jeżeli zostało przekazane nowe tło, to stare zostaje usunięte i zamienione kopią nowego. Inaczej zakumulowane tło zostaje usunięte i zamienione oryginalnym (przekazanym przy tworzeniu lub jako `new_background` w tej metodzie). Wszystkie sceny zostają usunięte.

- **next_scene()**

Overriden. Tworzy nową scenę ustawiając jako kontener na zakumulowane tło.

- **add_background_points(points, **kwargs)**

Otrzymuje listę punktów (nie są obiektami klasy **Point**) i argumenty nazwane, które potem zostaną przekazane do **plt**. Dodaje te obiekty do tła (*nie dodaje do aktualnej sceny!*). Pojawiają się na rysunku tylko od następnej sceny, gdy ustawimy już zaktualizowane tło.

- **add_background_lines(lines, **kwargs)**

Otrzymuje listę linii (nie są wyznaczone obiektami klasy **Point**) i argumenty nazwane, które potem zostaną przekazane do **plt**. Dodaje te obiekty do tła (*nie dodaje do aktualnej sceny!*). Pojawiają się na rysunku tylko od następnej sceny, gdy ustawimy już zaktualizowane tło.

- **add_background_rect(rect, **kwargs)**

Otrzymuje prostokąt, który jest obiektem **Rect**, i argumenty nazwane, które potem zostaną przekazane do **plt**. Dodaje ten prostokąt do tła (*nie dodaje do aktualnej sceny!*). Pojawia się na rysunku tylko od następnej sceny, gdy ustawimy już zaktualizowane tło.

4 KDTree

Klasa znajdująca się w module **kdtree.py**. Reprezentuje KD-drzewo, czyli k-wymiarowe drzewo. Jest stosowane do wydajnego przechowywania k-wymiarowych punktów, i szybkiego sprawdzania jakie punkty należą do pewnych wielowymiarowych obszarów. Dodatkowo nadaje możliwość szybkiego zwracania najbliższego punktu do podanego, ale ta ostatnia możliwość nie została zaimplementowana, ponieważ nie jest wymagana w treści projektu.

- **KDTree(points, dimensions=2, visualise=False)**

Tworzy obiekt tej klasy. Przyjmuje następujące argumenty:

- **points** - lista punktów (nie obiekty klasy **Point**). Wszystkie punkty muszą być tego samego wymiaru. Lista nie może być pusta.
- **dimensions** - opcjonalny parametr **int**. Ustawia ile wymiarów mają punkty. Domyślna wartość - 2.
- **visualise** - opcjonalny parametr **bool**. Decyduje czy będzie tworzona wizualizacja wszystkich operacji drzewa (*silnie wpływa na wydajność!*). Domyślna wartość - False.

Mожет wyrzucić następujące wyjątki:

- **ValueError('KDTree cannot be instantiated with no points inside')** - jeżeli lista **points** przekazana do konstruktora jest pusta, to jest wyrzucany ten błąd.

- **TypeError('Point dimensions do not match the KDTree dimensions, which is "dimensions")** - jeżeli chociażby jeden punkt nie odpowiada wymiarowi wskazanemu jako **dimensions** zostaje wyrzucony ten błąd.
- **ValueError('KDTree cannot handle duplicate points')** - jeżeli wśród przekazanych punktów istnieje chociażby dwa o takich samych wartościach dla odpowiednich wymiarów, to zostaje wyrzucony ten błąd. Ta implementacja nie jest w stanie sobie poradzić z powtórzeniami punktów.

Jeżeli wszystko jest w porządku, to zostaje utworzone kd-drzewo. W przypadku włączonej wizualizacji (**visualise=True**) po skonstruowaniu drzewa pokazuje się w odrębnym oknie animacja z możliwością sprawdzenia każdego kroku. Jest bardziej szczegółowo opisane w poradniku do korzystania.

- **__contains__(point) / contains(point, visualise=True)**

Dwie metody zapewniające to samo zachowanie. Główną metodą jest **contains**, a **__contains__** zapewnia możliwość używania konstrukcji **point in tree**. Zwraca **True** lub **False** w zależności czy istnieje ten punkt w drzewie czy go nie ma.

Przyjmuje następujące argumenty:

- **point** - punkt (obiekt klasy **Point** lub taki, który może być traktowany jako **Point**, czyli należy minimalnie do **Iterable** i **Sized**).
- **visualise** - opcjonalny argument **bool**. Należy tylko do **contains**. Nawet jeżeli dla całego drzewa zostało ustawione **visualise=True**, czyli chcemy, żeby każda operacja została wizualizowana, to ustawiając w tej metodzie **visualise=False** zapewniamy, że konkretnie to sprawdzenie nie zostanie zwizualizowane. Domyślna wartość ustawiona jako **True**, czyli jeżeli ustawiona wizualizacja w całym drzewie, to będzie domyślnie działać i dla tej operacji.

Może wyrzucić następujące wyjątki:

- **TypeError("The dimensions of the given point and KDTree do not match")** - jeżeli ilość wymiarów przekazanego punktu nie odpowiada ilości wymiarów drzewa.
- **TypeError("Passed object is not iterable or Point instance")** - jeżeli obiekt nie jest klasy **Point** lub nie może być traktowany jako taki (nie da się z niego utworzyć obiekt klasy **Point**).

- **find_points_in(rect, raw=True)**

Sprawdza które punkty z istniejących w drzewie należą do podanego prostokąta. Zwraca listę odpowiednich punktów.

Przyjmuje następujące argumenty:

- **rect** - prostokąt (obiekt klasy **Rect**). Wszystkie punkty istniejące w drzewie i należące do tego prostokąta będą zwrócone.
- **raw** - opcjonalny argument **bool**. Decyduje czy zostanie zwrócona lista punktów w postaci obiektów **Point** gdy **raw** jest ustawione na **True**, lub jako lista punktów, które są reprezentowane przez **np.array**. Domyślna wartość jest **True**.

Może wyrzucić następujące wyjątki:

- **TypeError("The dimensions of the given Rect object and KDTree do not match")** - jeżeli ilość wymiarów przekazanego prostokąta nie odpowiada ilości wymiarów drzewa.

5 QuadTree

Klasa reprezentująca quad-drzewo, czyli drzewo ćwiartkowe. Jest modyfikacją koncepcji drzewa binarnego dla dwuwymiarowej przestrzeni. Znajduje się w pliku **quadtree.py**. Zapewnia możliwość szybkiego zwrócenia wszystkich punktów należących do pewnego prostokąta (w tym kontekście 'prostokąt' będzie znaczył stricte 2d prostokąt). Także istotną zaletą jest możliwość dodawania nowych punktów do drzewa bez psucia ogólnej złożoności.

- **QuadTree(rect, capacity, points=None, visualise=False)**

Tworzy obiekt drzewa ćwiartkowego. Przyjmuje następujące argumenty:

- **rect** - prostokąt (obiekt klasy **Rect** lub taki który można traktować jako **Rect**, czyli jest kolekcją długości 2). Ten prostokąt wyznacza zakres działania drzewa. Nie można wrzucić punktu, który nie będzie należał do tego prostokąta.
- **capacity** - maksymalna ilość punktów możliwa do trzymania w jednym weźle drzewa. Musi być większa od zera.
- **points** - opcjonalny parametr. Lista punktów (nie obiekty klasy **Point**). Wszystkie punkty muszą być tego samego wymiaru.
- **visualise** - opcjonalny parametr **bool**. Decyduje czy będzie tworzona wizualizacja wszystkich operacji drzewa (*silnie wpływa na wydajność!*). Domyślna wartość - False.

Może wyrzucić następujące wyjątki:

- **AssertionError('Capacity must be a positive integer')** - jak było zaznaczono wyżej, **capacity** musi być większe od zera, więc zakładamy, że taka wartość zostanie przekazana.
- **TypeError('Passed argument is not a Rect object or cannot be treated as one')** - jeżeli **rect** nie jest prostokątem lub nie może być traktowany jako taki (nie da się z niego utworzyć prostokąta).
- **TypeError('The dimensions of the point and QuadTree do not match. Point must have 2 dimensions')** - jeżeli dowolny, wrzucany punkt ma inną ilość wymiarów niż 2, to otrzymujemy ten błąd.
- **ValueError('The point is not within the initial scope rectangle')** - jeżeli wrzucany punkt nie należy do prostokąta wskazanego przy inicjalizacji, to wyrzuca ten błąd.
- **ValueError('QuadTree cannot handle duplicate points')** - jeżeli wśród przekazanych punktów istnieją chociażby dwa o takich samych wartościach dla odpowiednich wymiarów, to zostaje wyrzucony ten błąd. Ta implementacja nie jest w stanie sobie poradzić z powtórzeniami punktów.

Jeżeli wszystko jest w porządku, to zostaje utworzone drzewo - instancja klasy **QuadTree**. W przypadku ustawionego **visualise=True** od razu po skonstruowaniu drzewa pojawi się okno z animacją. Szczegółowa informacja co do animacji jest w poradniku do wykorzystania.

- **insert(point) / insert_all(points)**

Metoda **insert** dodaje nowy punkt do drzewa, a metoda **insert_all** robi to samo, tylko dodając od razu całą listę punktów.

Przyjmuje następujące argumenty:

- **point** - punkt (nie obiekt klasy **Point**). Ten punkt zostanie dodany do drzewa. Ten argument należy do metody **insert**.
- **points** - lista punktów (nie obiekty klasy **Point**). Ten argument należy do metody **insert_all**.

Może wyrzucić następujące wyjątki:

- **TypeError('The dimensions of the point and QuadTree do not match. Point must have 2 dimensions')** - jeżeli dowolny wrzucany punkt ma inną ilość wymiarów, niż 2, to otrzymujemy ten błąd.
- **ValueError('The point is not within the initial scope rectangle')** - jeżeli wrzucany punkt nie należy do prostokąta wskazanego przy inicjalizacji, to wyrzuca ten błąd.
- **ValueError('QuadTree cannot handle duplicate points')** - jeżeli wśród przekazanych punktów istnieje chociażby dwa o takich samych wartościach dla odpowiednich wymiarów, to zostaje wyrzucony ten błąd. Ta implementacja nie jest w stanie sobie poradzić z powtórzeniami punktów.

- **find_points_in(rect, raw=True)**

Sprawdza które punkty z istniejących w drzewie należą do podanego prostokąta. Zwraca listę odpowiednich punktów.

Przyjmuje następujące argumenty:

- **rect** - prostokąt (obiekt klasy **Rect**). Wszystkie punkty istniejące w drzewie i należące do tego prostokąta będą zwrócone.
- **raw** - opcjonalny argument **bool**. Decyduje czy zostanie zwrócona lista punktów w postaci obiektów **Point** gdy **raw** jest ustawione na **True**, lub jako lista punktów, które są reprezentowane przez **np.array**. Domyślna wartość jest **True**.

Może wyrzucić następujące wyjątki:

- **TypeError("The dimensions of the given Rect object and QuadTree do not match")** - jeżeli ilość wymiarów przekazanego prostokąta nie odpowiada ilości wymiarów drzewa.

Część III

Poradnik do wykorzystania

1 Proste dodatkowe narzędzia

1.1 Mierzenie czasu

`timeit(title, precision)`

To jest dekorator przeznaczony do mierzenia czasu działania pewnej funkcji/metody. Ten dekorator przyjmuje jako argumenty **title** - czyli napis, który będzie wypisany przed czasem, **precision** - do ilu miejsc po przecinku jest zaokrąglony, wyliczony czas. Zakładamy, że długość **title** jest nie większa od 26, inaczej dostniemy **AssertionError**. Niżej jest przykład jego użycia (kod wzięty z GeeksForGeeks):

```
@timeit('Insertion sort', 7)
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >=0 and key < arr[j] :
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key

insertionSort(list(range(100, -1, -1)))
```

To będzie nasze podstawowe narzędzie do mierzenia czasu działania metod struktur danych.

1.2 Testy jednostkowe

Zostały także dodane testy jednostkowe, które sprawdzają poprawność działań obu drzew. Są bardzo proste, ale nadają możliwość szybkiego sprawdzenia poprawności przy każdej drobnej zmianie. Znajdują się w pliku **test.py**, a o ile są napisane przy pomocy biblioteki **unittest**, to w 'mądrych' IDE jest fajny sposób, żeby puszczać ich po jednym lub całe razem. Niżej jest przedstawiony przykładowy test napisany przy pomocy **unittest**:

```
class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

1.3 Testy różnorodne

Inne testy są zawarte w pliku **tests.py**. Większość z nich ma opcjonalne argumenty takie jak **count**, czyli ilość punktów, **capacity** - parametr dla **QuadTree**, niektóre mają **power** - dla pewnej wartości x ilość punktów będzie wynosiła $(2^x + 1)^2$.

2 Podstawowe wykorzystanie drzew

2.1 KDTree

API **KDTree** jest bardzo prosty i został opisany wyżej w dokumentacji. Niżej jest prosty kod tworzenia drzewa:

```
from kdtree import KDTree

points = [(11, 12), (12, 45), (18, 68), (19, 62), (20, 25),
          (23, 17), (23, 68), (24, 51), (25, 98), (28, 73)]
ktree = KDTree(points)
```

Możemy także sprawdzić czy istnieją pewne punkty w drzewie:

```
>>> (11, 12) in ktree
True
>>> ktree.contains((34, 23))
False
>>> ktree.contains((18, 68))
True
```

Żeby znaleźć wszystkie punkty, które należą do pewnego prostokąta - musimy przekazać ten prostokąt, ale zgodnie z dokumentacją to musi być obiekt **Rect**. Otóż niżej kilka przykładów tworzenia tego obiektu:

```
>>> from geometry.Rect import Rect
>>> Rect((-1, 10), (115, 203))
[-1 10] - [115 203]
>>> Rect((5.3e-4, 0.231), (1e30, 290))
[0.00053 0.231] - [1.0e+30 2.9e+02]
>>> rect = Rect((11, 23), (24, 62))
```

Otoż, spróbujmy wywołać metodę `find_points_in` drzewa:

```
>>> ktree.find_points_in(rect)
[array([20, 25]), array([12, 45]), array([24, 51]), array([19, 62])]
```

Użyjmy `raw`, żeby zwrócić listę obiektów **Point**:

```
>>> ktree.find_points_in(rect, raw=False)
[[20 25], [12 45], [24 51], [19 62]]
```

2.2 QuadTree

API **QuadTree** jest tak samo prosty jak i **KDTree**. Stwórzmy drzewo i założymy od razu, że nasze punkty mieszą się w zakresach $x \in [10, 30] \wedge y \in [10, 70]$:

```
from quadtree import QuadTree
from geometry.Rect import Rect

points = [(11, 12), (12, 45), (18, 68), (19, 62), (20, 25),
          (23, 17), (23, 68), (24, 51), (25, 28), (28, 73)]
scope = Rect((10, 10), (30, 70))
qtree = QuadTree(scope, 2, points=points)
# ValueError: The point [28 73] is not within the initial scope rectangle
```

Zmienimy ten jeden punkt i spróbujmy jeszcze raz:

```
new_points = [(11, 12), (14, 45), (12, 68), (19, 62), (13, 25),
              (23, 17), (23, 70), (24, 51), (25, 28), (28, 70)]
qtree = QuadTree((10, 10), (30, 70)), 2, points=new_points)
# successfully created
```

Spróbujmy dla podobnego jak poprzednio prostokąta zwrócić listę punktów należących do niego, dodając nowe punkty:

```
>>> rect = Rect((14, 23), (24, 69))
>>> qtree.find_points_in(rect)
[array([14, 45]), array([19, 62]), array([24, 51])]
>>> qtree.insert((10, 24))
>>> qtree.find_points_in(rect)
[array([14, 45]), array([19, 62]), array([24, 51])]
>>> qtree.insert_all([(13, 14), (21, 24)])
>>> qtree.find_points_in(rect)
[array([14, 45]), array([19, 62]), array([21, 24]), array([24, 51])]
```

3 Używanie testów

Tu skrótnie pokażemy tylko testy z **tests.py**. Spróbujmy zaimportować jeden test i puścić go:

```
>>> from tests import test_cross
>>> test_cross()
KDTree construction: 0.10423
QuadTree construction: 0.14782
-----
KDTree search: 0.001
QuadTree search: 0.00399
```

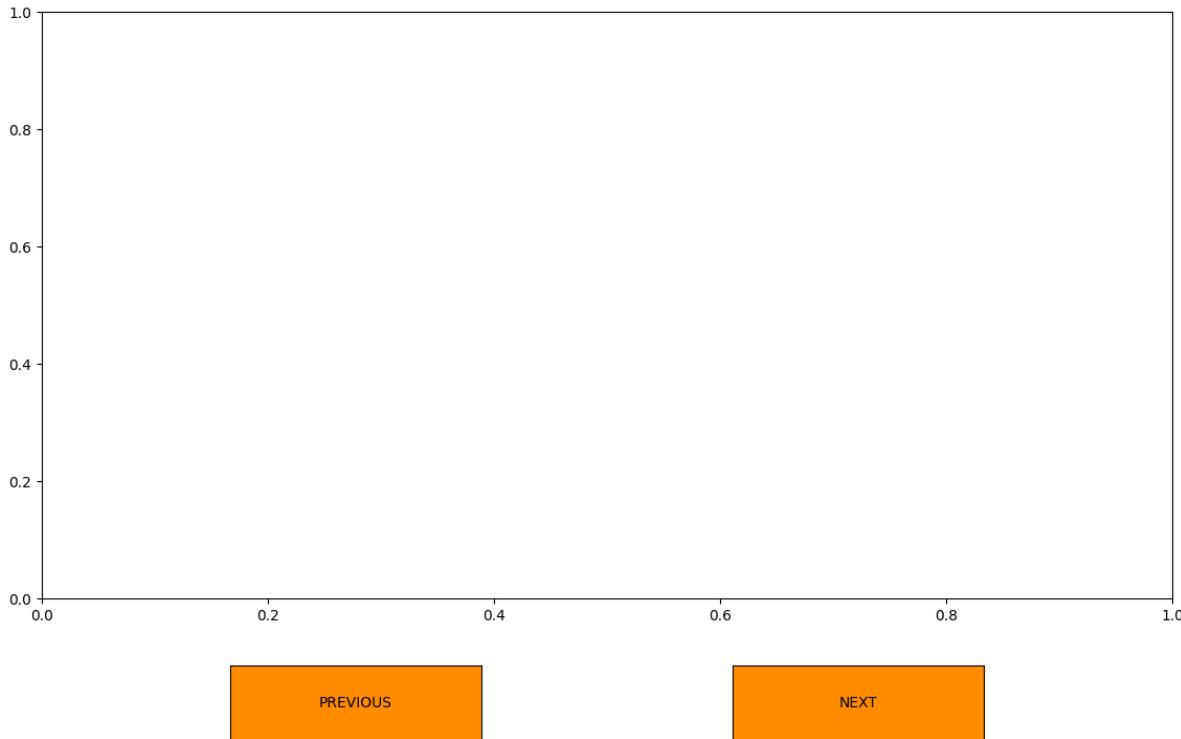
Puścimy wszystkie testy:

```
>>> from tests import test_all
>>> test_all(count=200, power=3, capacity=1)
# ...
----- test_rectangle ----- parameters - count: 200, capacity: 1
KDTree construction: 0.1038
QuadTree construction: 0.12904
-----
KDTree search: 0.00199
QuadTree search: 0.00096
----- test_grid ----- parameters - power: 3, capacity: 1
KDTree construction: 0.16176
ERROR: duplicates found
ValueError: KDTree cannot handle duplicate points
# ...
```

O ile punkty są generowane losowo, to jest szansa, że zgenerujemy dwa jednakowe punkty. Funkcja **test_all** wtedy nie wywala się, a informuje o wyjątku, wypisuje komunikat, i idzie dalej. W przypadku wyżej dla zbioru **grid** nie może trafić się taka sytuacja, to jest pokazane tylko przykładowo.

4 Wizualizacja

Doszliśmy do najciekawszej części tego poradnika - wizualizacji. Puste okno wizualizacji wygląda następująco:



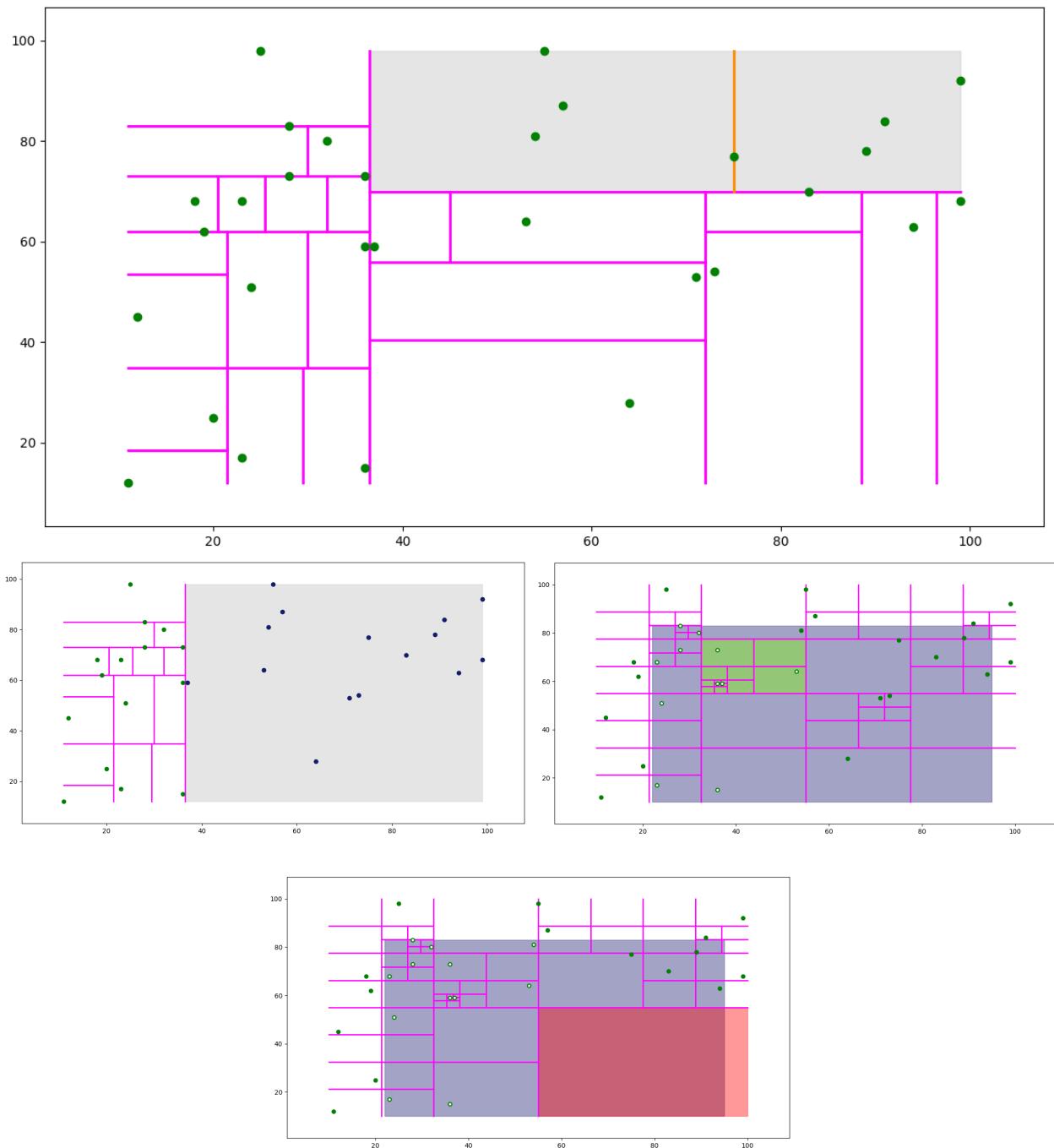
Rysunek 1: Puste okno wizualizacji

Mamy miejsce w ramce, gdzie będą rysowane figury, linie, punkty i tp. I mamy dwa przyciski 'NEXT' i 'PREVIOUS'. 'NEXT' przełącza na następną scenę, 'PREVIOUS' na poprzednią. W zależności od wersji **matplotlib** z dołu lub z góry jest podstawowa panel sterowania z możliwością przybliżania, przesuwania i tp.

Omówimy niżej wszystkie operacje, które są wizualizowane przez nasze drzewa. A mianowicie wszystkie operacje przedstawione w dokumentacji, składające **API** drzew, są wizualizowane. Mają niektóre wspólne cechy:

- Wszystkie linie stanowiące podziały w drzewach są różowe (`color='magenta'`, `line-width=2`)
- Wszystkie punkty wrzucone do drzewa są zielone (`color='green'`)
- Każdy aktualnie rozpatrywany obszar jest wydzielony szarym prostokątem (`color='silver'`, `alpha=0.4`)
- Aktualnie rozpatrywana linia podziału jest pomarańczową (`color='darkorange'`, `line-width=2`)

Na następnej stronie na rysunku 2 przedstawiona scena z wizualizacji zawierająca te wszystkie elementy:



Rysunek 2: Podstawowe elementy wizualizacji

Jeszcze kilka podstawowych elementów:

- Aktualnie dodawany do drzewa punkt jest ciemno-niebieski (color='midnightblue')
- Każdy podgląd / operacja, która zakończyła się sukcesem, rysuje zielony prostokąt (color='lawngreen', alpha=0.4) (np. znaleźliśmy punkt, który musimy potem zwrócić)
- Każdy podgląd / operacja, która zakończyła się porażką, rysuje czerwony prostokąt (color='red, alpha=0.4) (np. spróbowaliśmy dodać tu nowy punkt, ale okazało się, że ten punkt nie należy do tego obszaru)

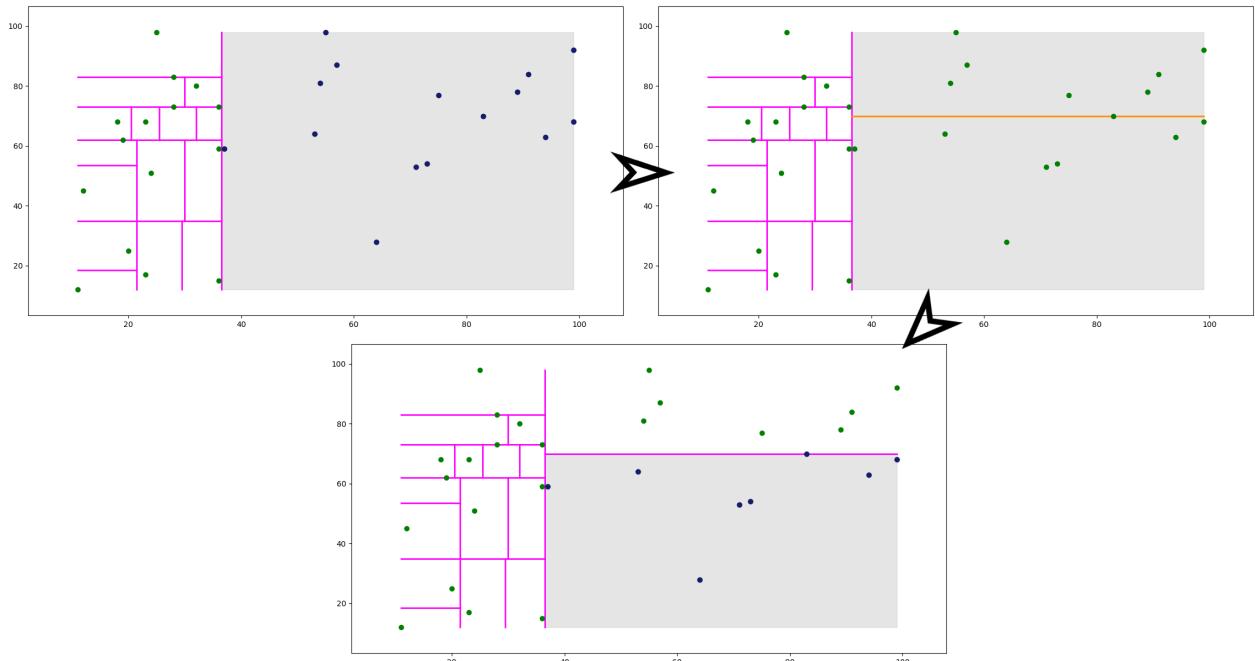
4.1 KDTree

4.1.1 Budowanie

Wszystko wygląda następująco: widzimy szary prostokąt, który wydziela aktualnie rozpatrywany obszar, i widzimy ciemno-niebieskie punkty, w tym momencie próbujemy optymalnie podzielić ten obszar.

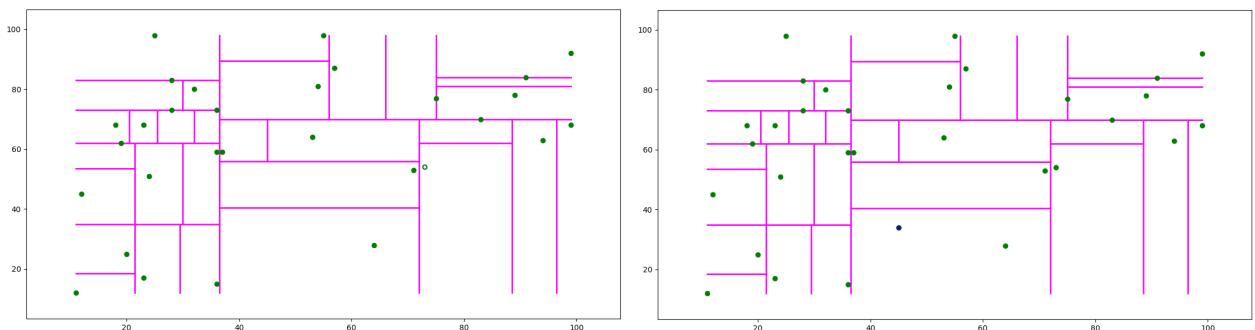
W następnym kroku pojawia się pomarańczowa linia podziału. Nasz szary prostokąt jeszcze pokazuje cały ten obszar, który podzieliliśmy.

W następnej scenie już patrzymy na jeden z dwóch otrzymanych obszarów. Nowa linia jest już pokazywana jak wszystkie na różowo. Wszystko jest jak w pierwszym kroku: tak kontynuujemy aż do końca.



Rysunek 3: Wizualizacja budowania KDTree

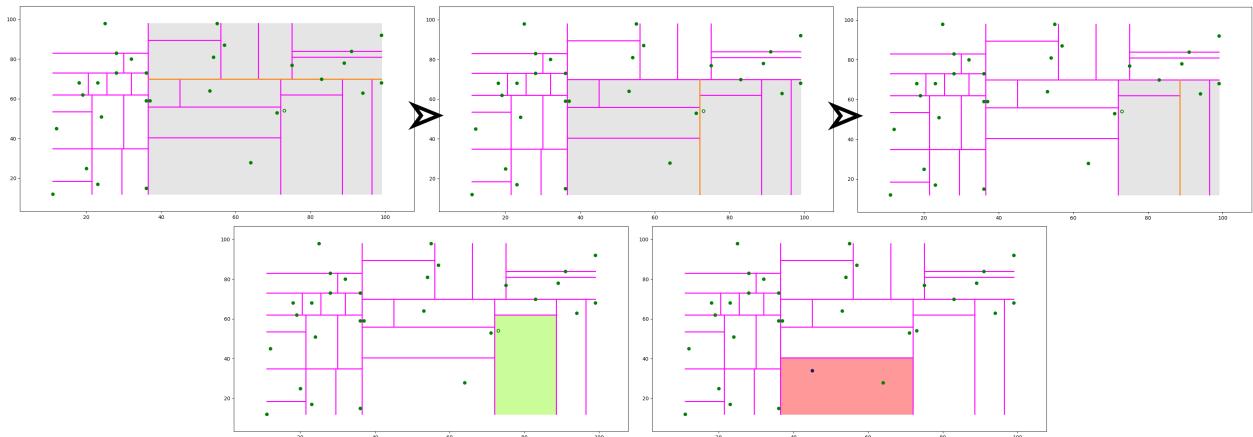
4.1.2 Sprawdzenie czy istnieje punkt



Rysunek 4: Wizualizacja sprawdzenia istnienia elementu KDTree

Na rysunku 4 zostały przedstawione 2 sytuacje. Na lewej części widzimy jeden punkt z pustym środkiem. W tym przypadku sprawdzamy czy istnieje ten punkt i już możemy od razu zobaczyć, że taki jest. Jest to zrobione, żeby nie było kolizji z punktami już istniejącymi, i przy tym było widać czego szukamy. Po prawej stronie jest już przypadek, gdy tego punktu nie istnieje, więc możemy spokojnie sobie go zaznaczyć ciemno-niebieskim kolorem.

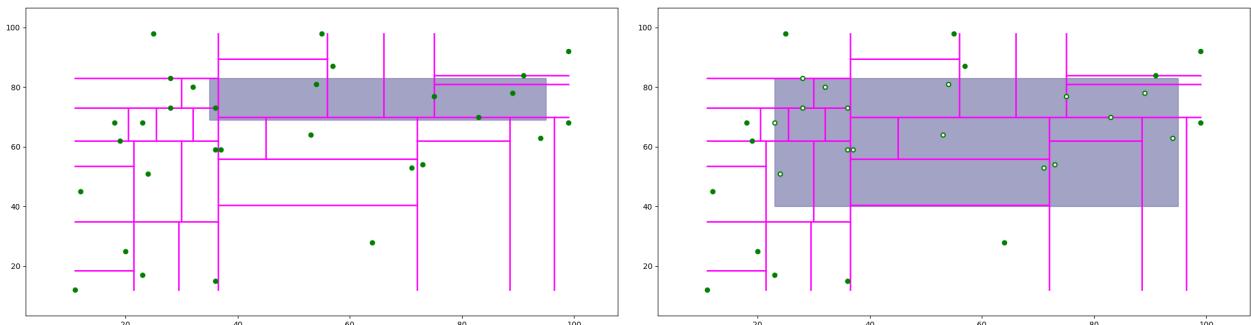
W każdym kolejnym kroku schodzimy na jeden poziom głębiej, aż w końcu (gdy dojdziemy do liścia) dostajemy lub nie dostajemy szukanego punktu. Także w każdym kroku widać obszar w którym sprawdzamy i pomarańczową linię według której porównujemy.



Rysunek 5: Wizualizacja kroków sprawzenia istnienia elementu KDTree

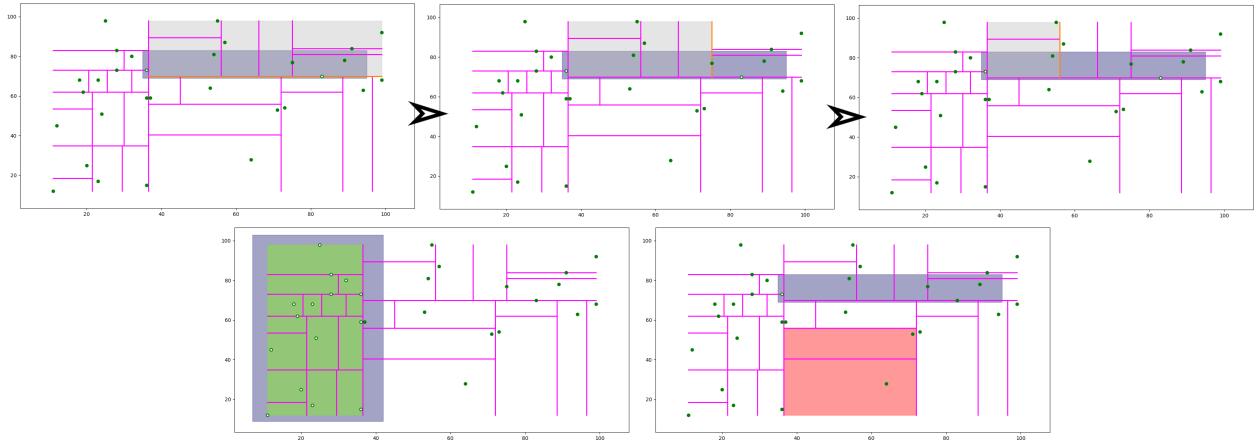
4.1.3 Szukanie punktów w prostokącie

O ile szukamy punktu w pewnym przekazanym prostokącie, to od razu sobie oznaczamy go ciemno-niebieskim kolorem. Można go zobaczyć po obu stronach na rysunku 6 niżej. Dodatkowo oznaczamy znalezione punkty (te, które znajdują się w końcowej liście) z pustym środkiem. Przykładowe oznaczenie jest po prawej stronie rysunku 6.



Rysunek 6: Wizualizacja elementów szukania punktów w prostokącie KDTree

Na następnej stronie na rysunku 7 zostały przedstawione kolejne kroki wizualizacji, a także przykład wyglądu znalezienia punktów i porażki w ich szukaniu dla pewnego obszaru.



Rysunek 7: Wizualizacja kroków szukania punktów w prostokącie KDTree

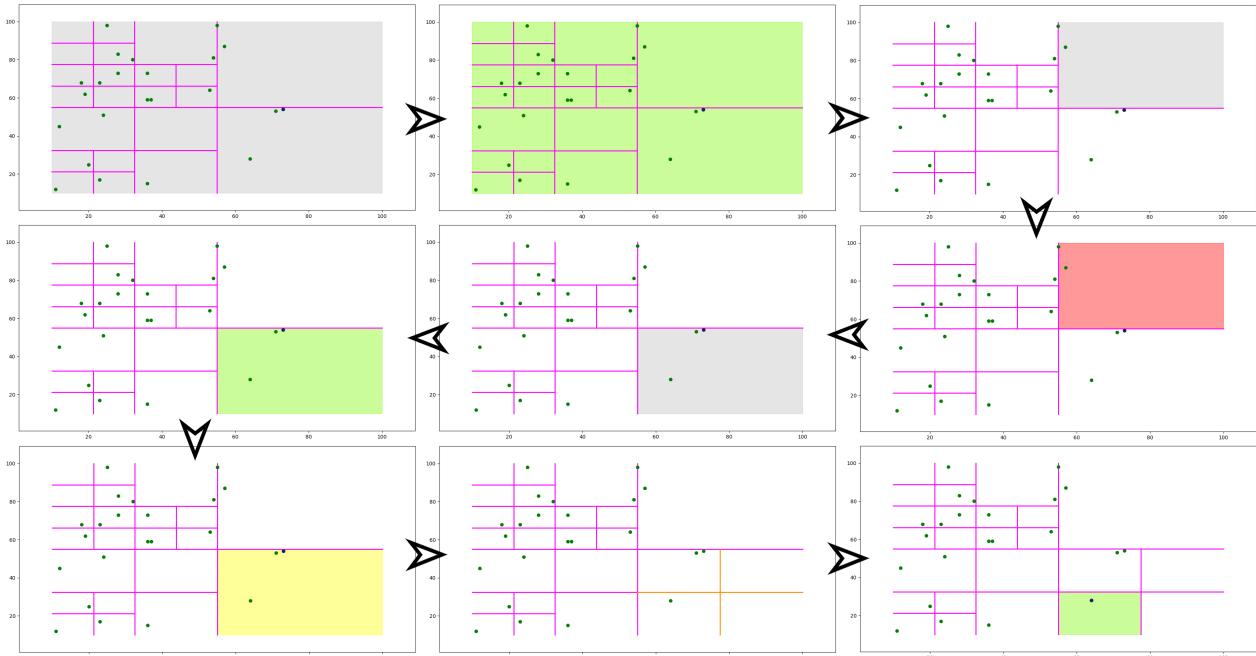
W każdym kroku pokazujemy pomarańczową linię według której poszliśmy w jedną ze stron i odpowiednio szarym prostokątem pewien obszar. Robimy to do tej pory póki ten obszar ma część wspólną z prostokątem szukanym. Jeżeli nie ma, to jest on pokazywany czerwonym. Jeżeli całkiem zawiera się, lub jest liściem i znaleźliśmy punkt, który należy do szukanego prostokąta, to rysujemy na zielono jak na rysunku 7.

4.2 QuadTree

4.2.1 Budowanie / Wstawianie punktów

Te dwie rzeczy są połączone dlatego, że budowanie korzysta ze wstawiania punktów, więc jest sens, żeby były przedstawione razem.

Wizualizacja tego jest dość skomplikowana, czego nie da się powiedzieć o algorytmie. Niżej na rysunku 8 zostało przedstawiono 9 kolejnych kroków wizualizacji.

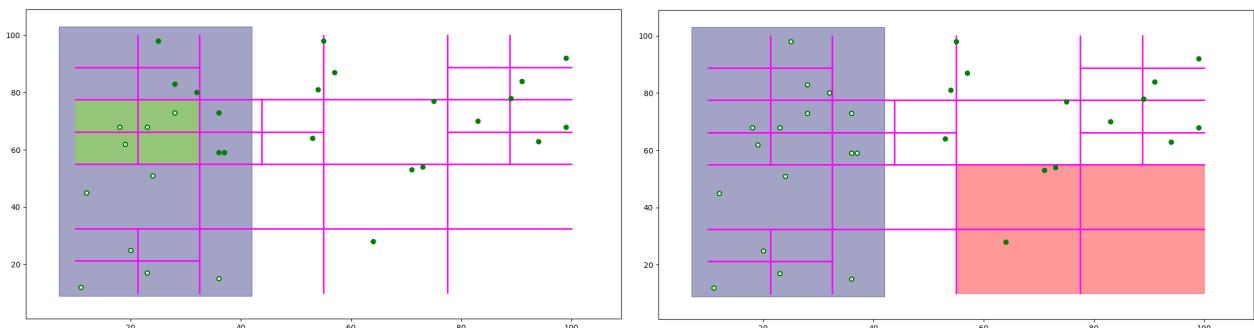


Rysunek 8: Wizualizacja kroków budowania QuadTree

Otoż, algorytm polega na tym, że jak jesteśmy na pewnym poziomie głębokości i chcemy wstawić jakiś punkt (w tym przypadku ciemno-niebieski), to sprawdzamy kolejne ćwiartki dopóki nie znajdziemy potrzebną (zaświeci się zielonym kolorem). Idziemy poziom niżej. Jak dojdziemy do liścia, to ostatecznie dodajemy nasz punkt (zaświeci się zielonym). Ale może zdarzyć się, że przepełnimy liść i trzeba będzie go podzielić. Wtedy on zaświeci się żółtym, a w następnym kroku pojawią się pomarańczowe linie dzielące ten liść. Dalej wszystkie punkty w tym liściu, w tym i nasz nowy, będą wstawiane do nowych liści podobnym sposobem jak i nowy punkt (próbując wstawić do każdego liścia i dostając czerwony lub zielony kolor jako oznaczenie porażki i sukcesu odpowiednio).

4.2.2 Szukanie punktów w prostokącie

Wizualizacja tej operacji jest bardzo podobna do wizualizacji w drzewie KDTTree. Sprawdzamy kolejne ćwiartki drzewa, i jeżeli obszar przecina się z szukanym prostokątem, to idziemy wgłąb. Jeżeli obszar całkiem zawiera się w prostokącie lub pewne punkty liścia należą do szukanego prostokąta, to ten obszar świeci się zielonym. Jeżeli obszar nie przecina się z szukanym, lub jest pustym, lub żaden punkt nie należy do szukanego, to zaświeci się na czerwono. Przykładowo jest pokazane na rysunku 9 niżej.



Rysunek 9: Wizualizacja kroków szukania punktów w prostokącie QuadTree

5 Dodatkowe uwagi

5.1 Przepelnienie stosu

Czasami dla bardzo dużych zbiorów danych i przy pewnych bug'ach, które zostały już naprawione (wciąż mogą istnieć podobne, jeszcze nie wykryte), przepęlnia się stos. Jest dwie możliwości: mamy nieskończoną rekurencję lub rekurencja jest skończona, ale bardzo głęboka. Najprostszym sposobem jest użycie modułu **sys** i **inspect**. Sprawdzić i ustawić aktualną maksymalną głębokość stosu da się następująco:

```
>>> import sys
>>> sys.getrecursionlimit()
1000
>>> sys.setrecursionlimit(7500)
>>> sys.getrecursionlimit()
7500
```

Żeby monitorować aktualną głębokość stosu, w funkcji rekurencyjnej warto użyć funkcji z modułu **inspect**.

```

>>> import inspect
>>> len(inspect.stack(0))
1 # 1 because we are calling from main
>>> def check_stack():
...     print(len(inspect.stack(0)))
...
>>> check_stack()
2 # we are 1 more level deeper (called inside check_stack() function)

```

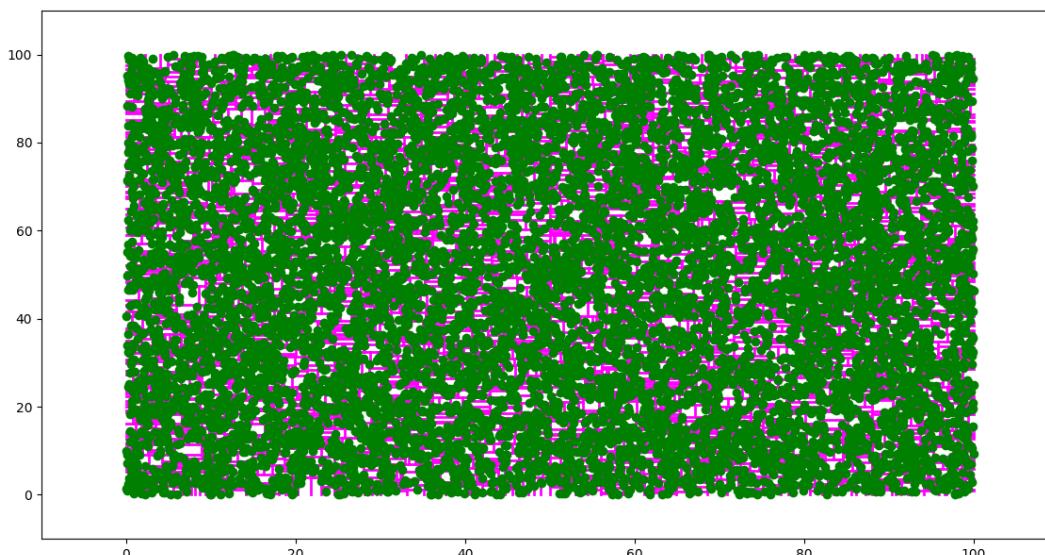
Dla **QuadTree** przy obecności bardzo dużego punktu (współrzędne rzędu 10^{20} i dwóch o bardzo małej odległości między sobą rzędu 10^{-15} , dostajemy maksymalną głębokość stosu równą 127. Czyli jest bardzo mała szansa wygenerowania danych, które by ten stos podstawowy przepełniły, ale wciąż szansa istnieje.

5.2 Przepelnienie pamięci

Jeszcze jednym problemem może okazać się przepelnienie pamięci operacyjnej, co powoduje **MemoryError**. Jest to zazwyczaj przy włączonej wizualizacji dla bardzo dużych zbiorów danych i jest największa szansa przy konstruowaniu **QuadTree**, ponieważ dla niego wizualizacja zawiera bardzo dużo kroków. Dla 10000 punktów przy włączonej wizualizacji jeden tylko rysunek **KDTree** zajmował ponad 6gb RAM. Więc warto zauważyć, że nie trzeba włączać wizualizację dla bardzo dużych zbiorów. Dla **QuadTree** rysunek zajmował ponad 10gb RAM po czym wyskoczyła **MemoryError**. Rysunek tak i nie został wyświetlony.

5.3 Duże zbiorы a wizualizacja

Jeszcze jednym argumentem, żeby nie włączać wizualizację dla dużych zbiorów jest niemożliwość odczytanie czegokolwiek z tej wizualizacji i długość przełączania scen. Jedyną możliwością odczytania czegoś z scen jest przełączenia do tyłu, żeby zobaczyć końcowy wynik, i potem używania narzędzi **matplotlib**, żeby popatrzyć na pewne części bardziej szczegółowo. Wtedy da się odczytać coś z rysunku. W ogólnych sytuacjach polecamy unikać stosowania wizualizacji dla dużych zbiorów.



Rysunek 10: Ostatnia scena konstruowania KDTree dla 10tys.-el. zbioru

Część IV

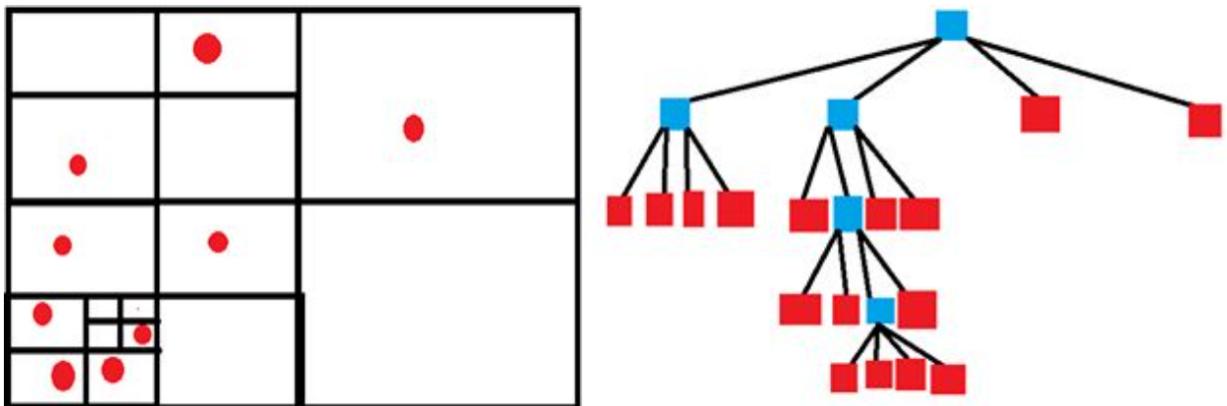
Sprawozdanie

1 Wstęp teoretyczny

W wykonywanym przez nas ćwiczeniu zajęliśmy się problemem przeszukiwania zbioru punktów w celu znalezienia tych, które należą do zadanego przez nas obszaru. Przygotowaliśmy dwa rozwiązania wspomnianego problemu. Pierwszy z nich wykorzystuje strukturę QuadTree, drugi strukturę KDTree.

1.1 Struktura QuadTree

Jest to struktura danych będąca drzewem, które służy do podziału przestrzeni dwuwymiarowej na mniejsze części, dzieląc ją na równe ćwiartki, a następnie dzieląc je na kolejne ćwiartki itd. W naszym problemie, naszą prostokątną płaszczyznę będziemy dzielić na ćwiartki wtedy, kiedy w danym polu znajdować się będzie więcej niż jeden punkt, tak aby na każde pole przypadł maksymalnie jeden punkt. Możliwym jest także przyjęcie pewnej maksymalnej głębokości drzewa, która będzie oznaczała maksymalną ilość podziałów, wówczas w danych polach znajdować się będzie mógł więcej niż jeden punkt, dla odmiany możemy też przyjąć konkretną ilość punktów inną niż 1, jako akceptowalną do zakończenia podziałów prostokąta. Nasz podzielony prostokąt reprezentowany będzie przez drzewo czwórkowe. Korzeń reprezentuje główny prostokąt, a gałęzie z niego wyrastające, jego podział. Analogicznie będzie z kolejnymi podziałami prostokątów, podział prostokąta na ćwiartki reprezentują 4 gałęzie z niego wychodzące. Liście tego drzewa będą, więc prostokątami, które nie zostały podzielone. Przykładowy podział prostokąta zgodnie z powyżej opisaną koncepcją i drzewo czwórkowe mu odpowiadające, ilustruje poniższy rysunek:



Rysunek 11: Struktura QuadTree

Złożoność czasowa, jak i pamięciowa tworzenia QuadTree zależy stricte od jego głębokości (ona zależy natomiast od rozmiesczenia punktów na płaszczyźnie) oraz ilości punktów leżących na płaszczyźnie i wynosi $O((d+1)*n)$, gdzie d – głębokość drzewa, n – liczba punktów. Mając już odpowiednio zaimplementowane QuadTree możemy je w prosty sposób wykorzystać do rozwiązania naszego problemu – znalezienia punktów znajdujących się na podanym

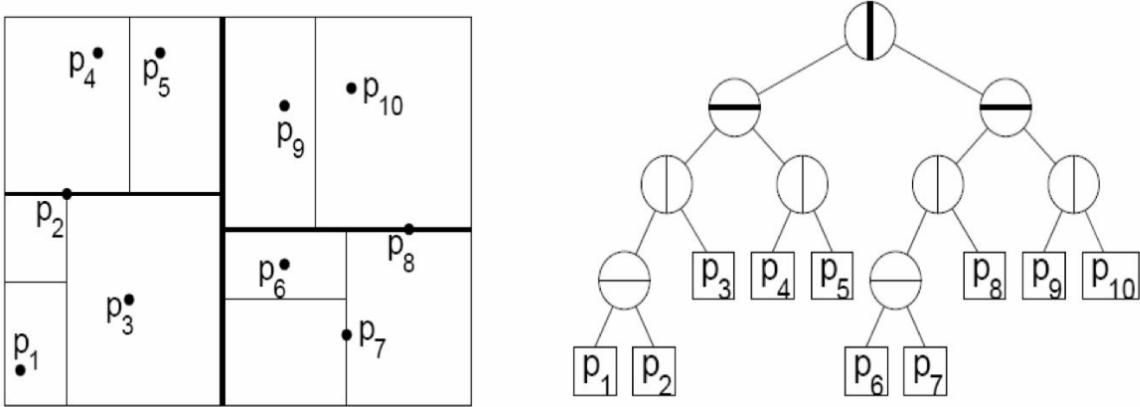
obszarze. Przeszukujemy nasze drzewo, zaczynając od korzenia, natrafiając na kolejne węzły wykonujemy następujące operacje:

- jeżeli prostokąt reprezentujący dziecko węzła zawiera się całkowicie w obszarze, w którym szukamy punktów, to wszystkie punkty należące do niego (całej gałęzi zaczynającej się od niego) zawierają się w prostokącie, więc dodajemy je do zbioru poszukiwanych punktów, i nie wchodzimy dalej w głąb gałęzi (odcinamy gałąź)
- jeżeli prostokąt reprezentujący dziecko węzła wcale nie zawiera się w obszarze, w którym szukamy punktów, to wszystkie punkty należące do niego (całej gałęzi zaczynającej się od niego) nie zawierają się w prostokącie, więc gałąź przestajemy przeszukiwać (odcinamy gałąź)
- jeżeli prostokąt reprezentujący dziecko węzła częściowo zawiera się w obszarze, w którym szukamy punktów, to wchodzimy w głąb gałęzi, jeżeli natrafiłyśmy na liść, sprawdzamy bezpośrednio czy punkty należące do tego liścia należą do przeszukiwanego obszaru, jeżeli tak to dany punkt dodajemy do zbioru poszukiwanych punktów

Złożoność obliczeniowa przeszukania takiego QuadTree wynosi $O(dl)$, gdzie d - głębokość drzewa, l – liczba liści reprezentujących prostokąty częściowo zawierające się w przeszukiwanym obszarze. Przeszukanie takiego drzewa jest, więc w praktyce znaczaco mniej kosztowne niż jego stworzenie. Struktura ta w zależności od sytuacji i zbioru punktów może okazać się bardzo optymalna, a także wręcz przeciwnie, może okazać się gorsza niż przeszukiwanie poprzez sprawdzanie każdego punktu osobno. Należy, więc korzystać z niej rozważnie i dla właściwych zbiorów punktów. Tym jakie zbioru są odpowiednie, a jakie nie, zajmiemy się w dalszej części sprawozdania testując już zaimplementowany przez nas algorytm.

1.2 Struktura KDTree

Jest to struktura danych będąca drzewem binarnym, używana do podziału przestrzeni. Podana przestrzeń k wymiarowa będzie tworzona w następujący sposób (równolegle przedstawione będzie tworzenie KDTree). Wśród punktów należących do przestrzeni wybieramy medianę przy pomocy QuickSelect, jest ona punktem podziału. Należy przed tym wybrać wymiar podziału, możemy dokonywać wyboru na dwa sposoby, na przemian lub wybierając ten gdzie różnica współrzędnych punktów jest największa. My w swojej implementacji wybraliśmy ten drugi, gdyż pozwala on uzyskać lepszą strukturę drzewa. Gdy dokonujemy danego podziału wówczas nasza obecna przestrzeń zostanie podzielona na dwie części. Odpowiada to utworzeniu dwóch nowych gałęzi z węzła, który reprezentuje właśnie podzieloną przestrzeń. W węźle posiadającym gałęzie przechowujemy informacje o wymiarze podziału przestrzeni, którą został on podzielony. Podziału przestrzeni dokonujemy wtedy, gdy dana przestrzeń zawiera więcej niż jeden punkt. Gdy zawiera dokładnie jeden, umieszczamy go jako liść naszego drzewa, wychodzący z węzła reprezentującego przestrzeń, której bezpośrednią podprzestrzenią jest ta, w której znajduje się omawiany punkt. Opisany powyżej schemat postępowania dla przykładowego zbioru punktów dwuwymiarowych ilustruje poniższy rysunek (wybór podziału przestrzeni metodą na przemian):



Rysunek 12: Struktura KDTTree

Złożoność obliczeniowa tworzenia KDTTree wynosi $O(kn\log n)$, dla pewnych implementacji $O(n\log n)$, gdzie n to ilość punktów, a k - ilość wymiarów, wynika ona bezpośrednio z głębokości drzewa, która wynosi $\log n$, jeżeli zakładamy, że drzewo jest zbalansowane, a takie właśnie tworzymy. Algorytm poszukiwania punktów należących do konkretnego obszaru jest niemal identyczny jak ten przy wykorzystaniu quadtree. Wykorzystujemy natomiast do tego strukturę KDTTree co wpływa na poprawę złożoności dla niektórych przypadków. Dla drzewa zrównoważonego złożoność czasowa zliczania wynosi $O(\sqrt{n})$, złożoność czasowa wyszukiwania $O(\sqrt{n} + k)$, gdzie k - jest ilością punktów wynikowych, a złożoność pamięciowa $O(n)$. Widać, więc że tak jak i w quadtree, przeszukiwanie drzewa jest znacznie mniej kosztowne niż jego stworzenie. Porównywaniem QuadTree oraz KDTTree zajmiemy się w dalszej części sprawozdania na podstawie konkretnych przykładów, dzięki którym będziemy mogli dostrzec wady i zalety, a także podobieństwa i różnice tych dwóch podejść rozwiązywających problem zaklasyfikowania punktów z danego zbioru do konkretnego obszaru.

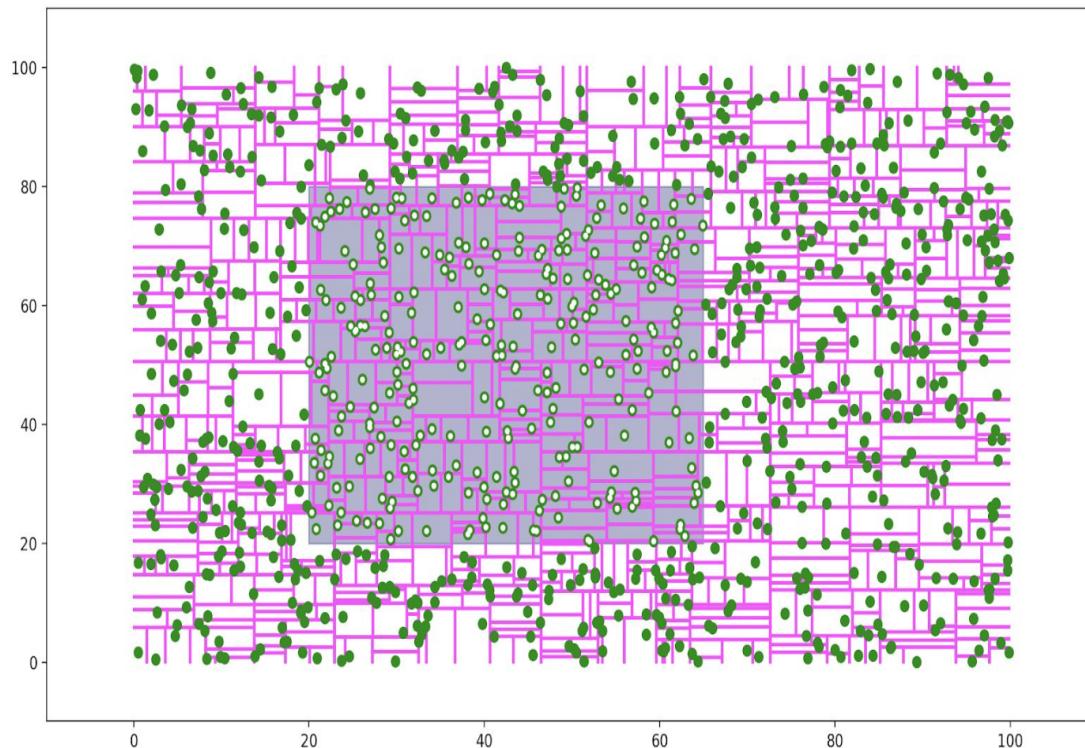
2 Testowanie dla różnych danych wejściowych

Przygotowaliśmy dużo zbiorów testowych, mających na celu ustalić poprawność działania zaimplementowanych struktur oraz metod poszukujących punktów należących do danego prostokąta. Porównamy także czas działania tych metod oraz samej inicjalizacji struktur.

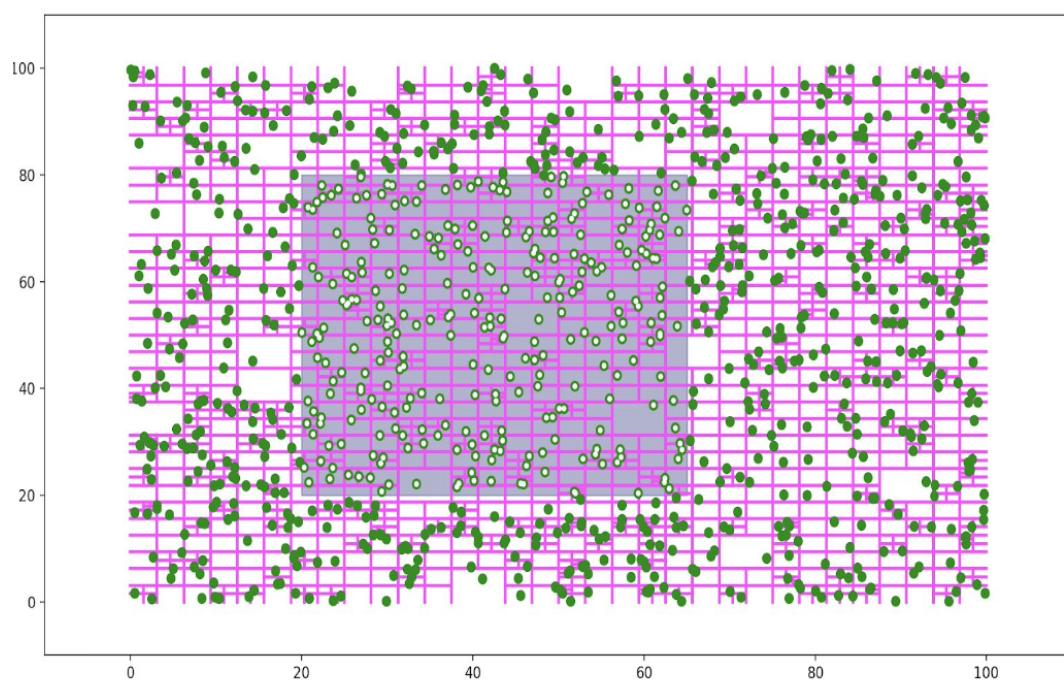
2.1 Zbiór o rozkładzie jednostajnym

Jest to zbiór losowo wybranych punktów należących do danego prostokąta. Natomiast obszar, w którym mamy odnaleźć punkty to prostokąt częściowo przecinający nasz zbiór punktów. W związku z losowym położeniem punktów na płaszczyźnie są one na niej równomiernie rozłożone, jest to więc przypadek dość powszechny, pozabawiony ekscesów. Dlatego został on wybrany przez nas jako pierwszy zbiór do przeanalizowania, aby sprawdzić działanie programów dla tak podstawowego zbioru punktów.

Poniższe ilustracje demonstrują wyniki działania programów dla 1000 punktów:



Rysunek 13: Zbiór o rozkładzie jednostajnym KDTreer



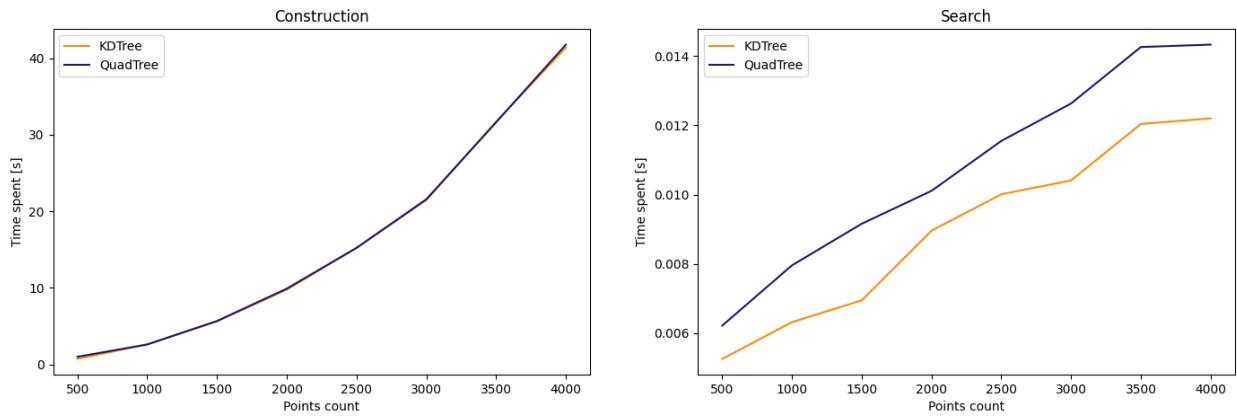
Rysunek 14: Zbiór o rozkładzie jednostajnym QuadTree

Jak widać na zamieszczonych ilustracjach, program działa prawidłowo zarówno przy użyciu QuadTree, jak i KDTreer.

Poniższa tabela ilustruje czasy działania programów dla podanego zbioru, przy danej ilości punktów:

Liczba punktów	Czas tworzenia KDTree [s]	Czas przeszukiwania KDTree [s]	Czas tworzenia QuadTree [s]	Czas przeszukiwania QuadTree [s]
500	0.73491	0.00526	1.01075	0.00622
1000	2.63573	0.00632	2.6174	0.00796
1500	5.64069	0.00695	5.68459	0.00916
2000	9.7806	0.00896	9.91145	0.01011
2500	15.21827	0.01001	15.23907	0.01155
3000	21.47649	0.01041	21.5799	0.01263
3500	31.72233	0.01204	31.63304	0.01426
4000	41.39933	0.0122	41.77491	0.01433

Tablica 2: Wyniki pomiarów czasu dla zbioru o rozkładzie jednostajnym



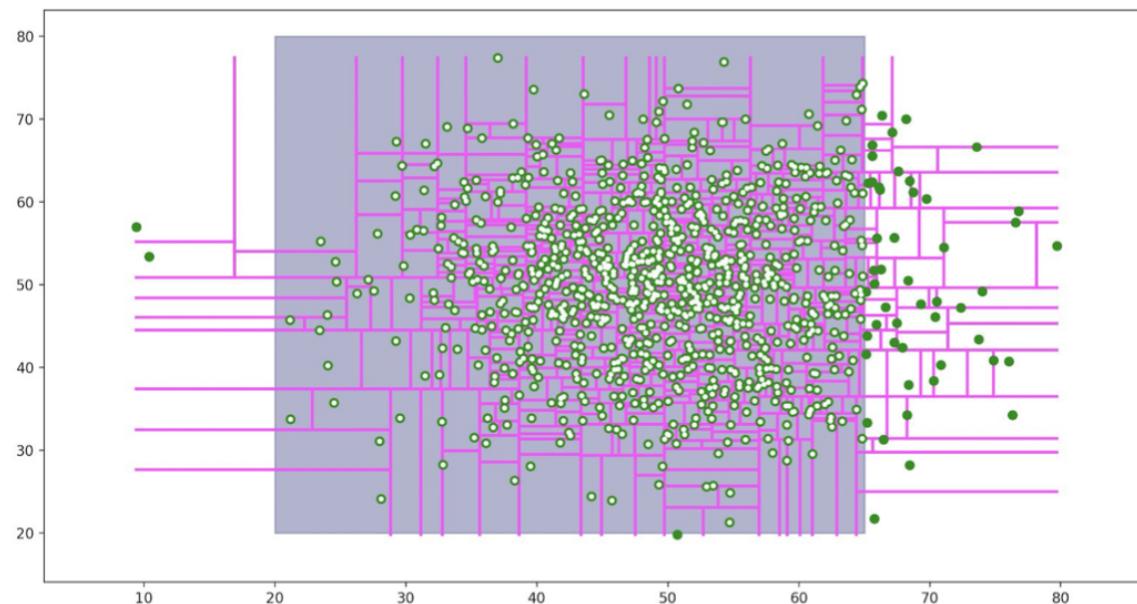
Rysunek 15: Wyniki pomiarów czasu dla zbioru o rozkładzie jednostajnym

Na podstawie powyższej tabeli i wykresów możemy stwierdzić, że dla zbioru 1 tworzenie QuadTree i KDTree zajmuje jednakową ilość czasu. Natomiast przeszukiwanie QuadTree dla tego zbioru zajmuje średnio 20% więcej czasu niż przeszukiwanie KDTree.

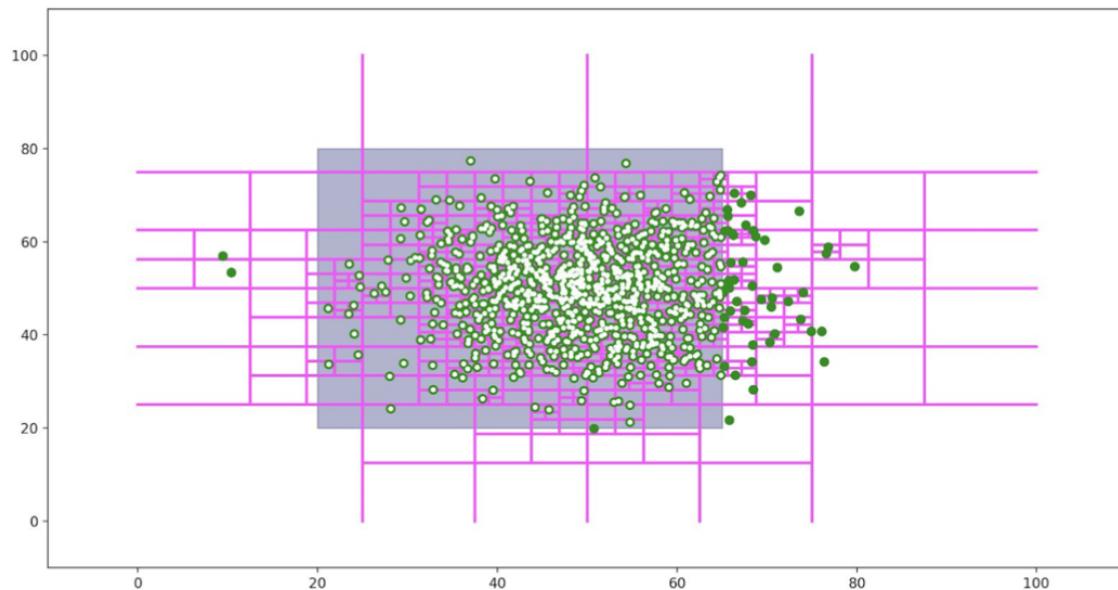
2.2 Zbiór o rozkładzie normalnym

Jest to zbiór punktów rozlokowany na płaszczyźnie zgodnie z rozkładem normalnym. Natomiast obszar, w którym mamy odnaleźć punkty to prostokąt zawierający część punktów należących do tego zbioru. Zbiór ten został wybrany ze względu na to, że jest on rozkładem bardzo powszechnym, więc uznaliśmy, że należy sprawdzić działanie naszych algorytmów dla tego zbioru.

Poniższe ilustracje demonstrują wyniki działania programów dla 1000 punktów:



Rysunek 16: Zbiór o rozkładzie normalnym KDTree



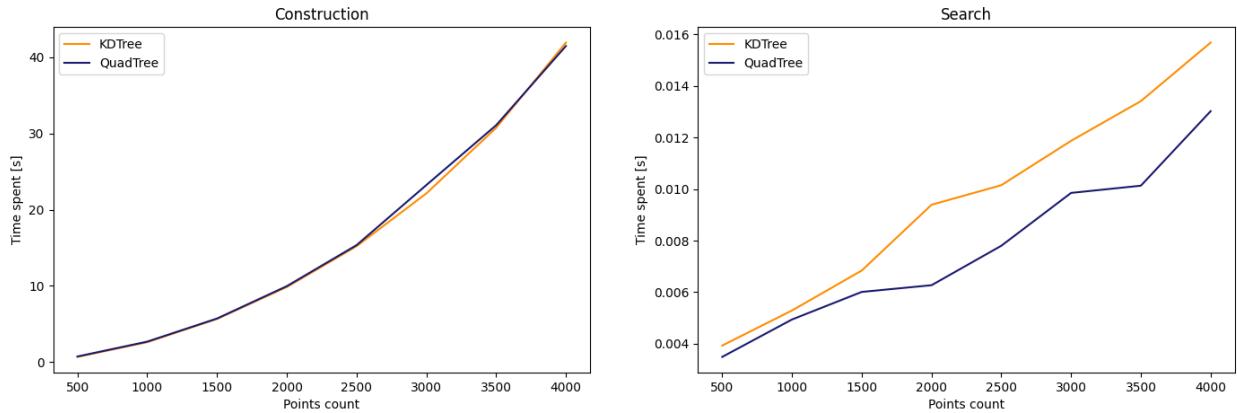
Rysunek 17: Zbiór o rozkładzie normalnym QuadTree

Jak widać na zamieszczonych ilustracjach, program działa prawidłowo zarówno przy użyciu QuadTree, jak i KDTree.

Poniższa tabela ilustruje czasy działania programów dla podanego zbioru, przy danej ilości punktów:

Liczba punktów	Czas tworzenia KDTree [s]	Czas przeszukiwania KDTree [s]	Czas tworzenia QuadTree [s]	Czas przeszukiwania QuadTree [s]
500	0.65803	0.00393	0.71088	0.00349
1000	2.59145	0.00529	2.66477	0.00494
1500	5.64883	0.00684	5.69806	0.00601
2000	9.8328	0.00939	9.95165	0.00627
2500	15.19954	0.01015	15.3401	0.0078
3000	22.13449	0.01187	23.23205	0.00985
3500	30.74997	0.01341	31.08935	0.01013
4000	41.94612	0.01568	41.49548	0.01302

Tablica 3: Wyniki pomiarów czasu dla zbioru o rozkładzie normalnym



Rysunek 18: Wyniki pomiarów czasu dla zbioru o rozkładzie normalnym

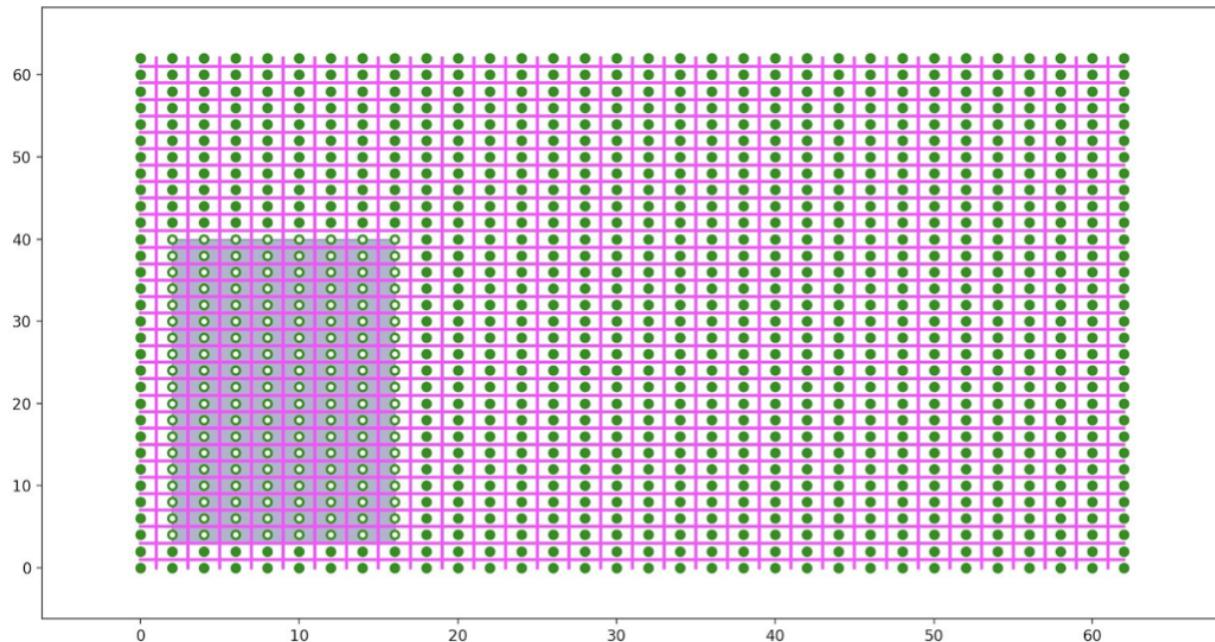
Na podstawie powyższej tabeli i wykresów możemy stwierdzić, że dla zbioru 2 tworzenie QuadTree i KDTree zajmuje jednakową ilość czasu. Gdy liczba punktów w zbiorze nie przekracza 1500, przeszukiwanie KDTree dla tego zbioru zajmuje średnio 10% więcej czasu niż przeszukiwanie quadtree, w przeciwnym wypadku zajmuje ono ok. 25% więcej czasu.

2.3 Siatka

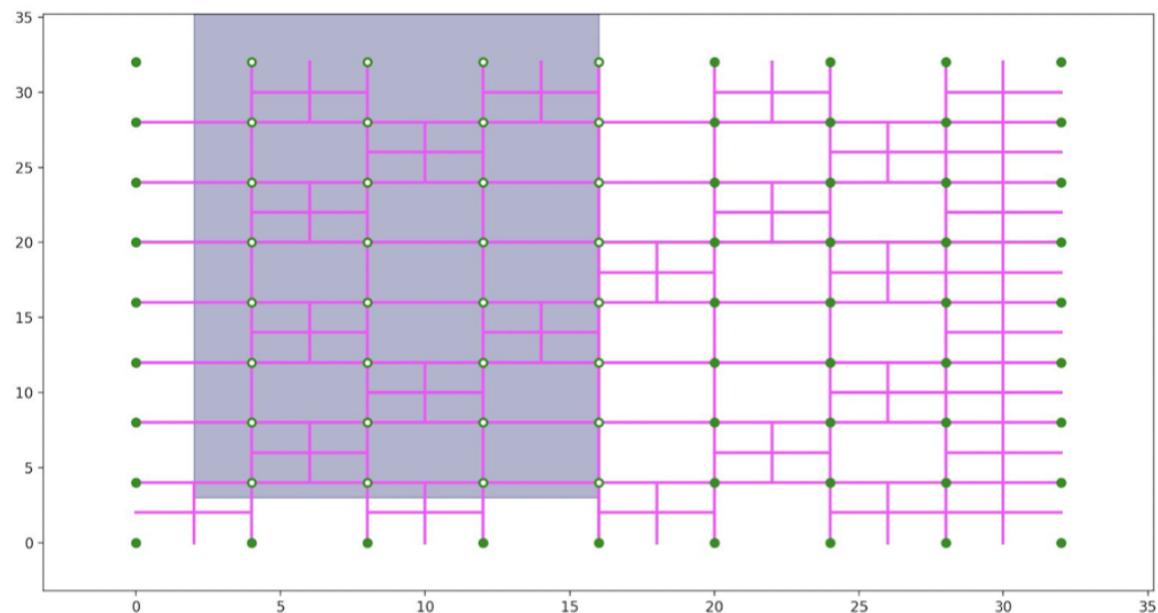
Jest to zbiór punktów należących do zadanego prostokąta, które odległe są od swoich sąsiadów o taką samą odległość. Punkty należące do zadanego zbioru są więc jednostajnie rozłożone na płaszczyźnie. Do tego prostokąt ten jest kwadratem o szerokości będącej potęgą dwójki (dlatego odległość pomiędzy sąsiadami w danej płaszczyźnie też jest potęgą dwójki). Natomiast obszar, w którym mamy odnaleźć punkty to prostokąt, częściowo przecinający nasz zbiór punktów. Warto zauważyć także, że poziomo jak i pionowo punkty te będą rozłożone na prostych zawierających identyczną ilość punktów, co mogłoby okazać się dla programu problematyczne. Należy też zwrócić uwagę, że w wypadku struktury QuadTree praktycznie każdy punkt będzie leżał na ramię prostokąta (ze względu na tak zadany zbiór punktów), co przy błędnej implementacji mogłoby spowodować przydzielenie danego punktu do więcej niż jednej części podziału, obecnie dzielonego prostokąta. Z powyższych powodów

postanowiliśmy przetestować działanie naszych struktur także dla tego konkretnego zbioru punktów.

Poniższe ilustracje demonstrują wyniki działania programów dla punktów odległych od sąsiadów o 2 jednostki:



Rysunek 19: Siatka KDTree



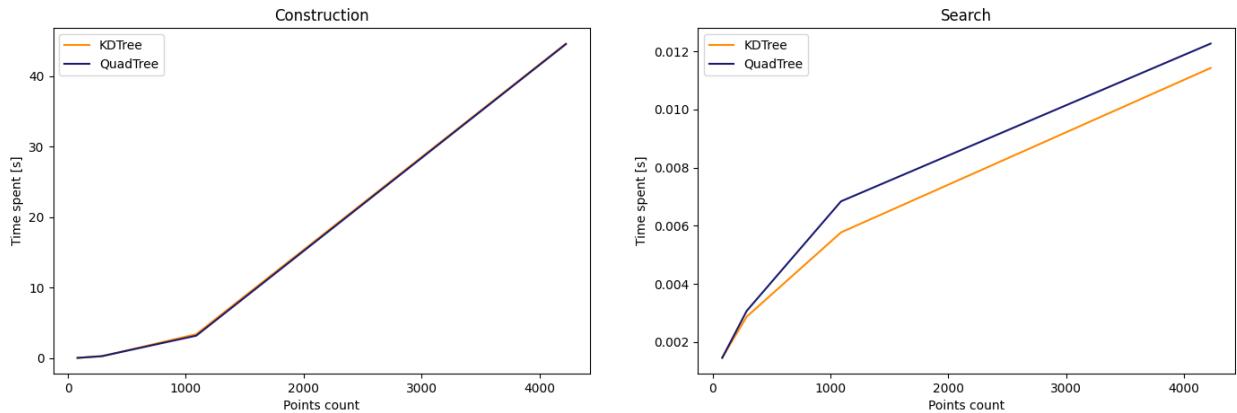
Rysunek 20: Siatka QuadTree

Jak widać na zamieszczonych ilustracjach, program działa prawidłowo zarówno przy użyciu QuadTree, jak i KDTree.

Poniższa tabela ilustruje czasy działania programów dla podanego zbioru, przy danej ilości punktów:

Liczba punktów	Czas tworzenia KDTree [s]	Czas przeszukiwania KDTree [s]	Czas tworzenia QuadTree [s]	Czas przeszukiwania QuadTree [s]
81	0.0259	0.00146	0.02375	0.00145
289	0.2503	0.00287	0.27596	0.00307
1089	3.39618	0.00577	3.18027	0.00684
4225	44.64996	0.01143	44.58075	0.01227

Tablica 4: Wyniki pomiarów czasu dla siatki



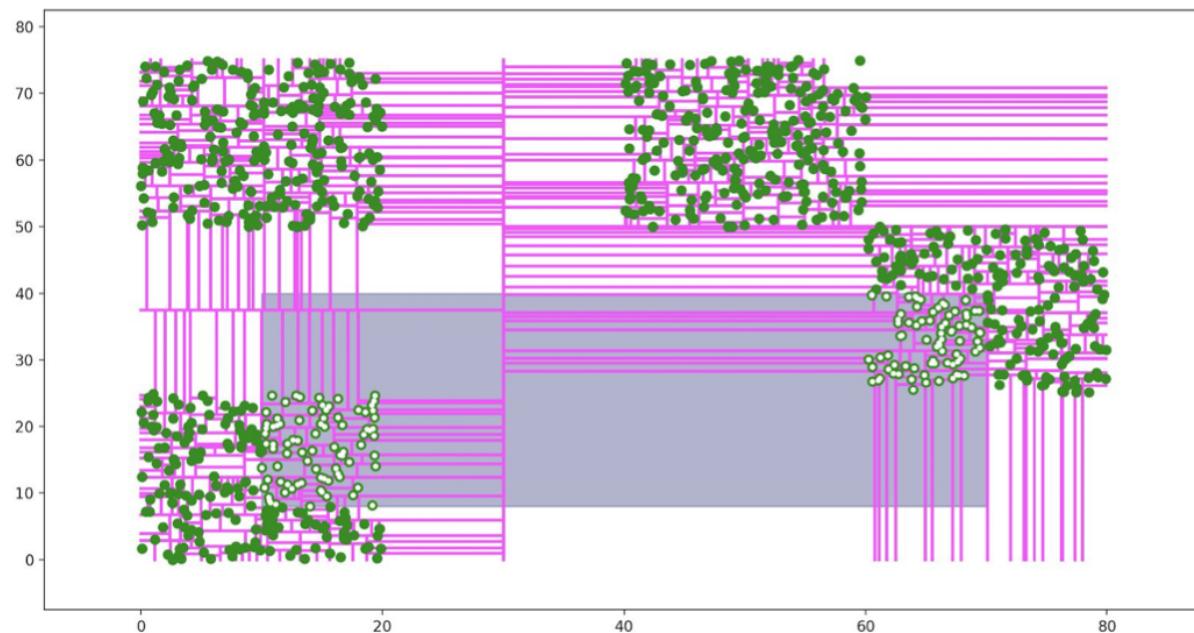
Rysunek 21: Wyniki pomiarów czasu dla siatki

Na podstawie powyższej tabeli i wykresów możemy stwierdzić, że dla zbioru 3 tworzenie QuadTree zajmuje średnio tyle samo czasu co tworzenie KDTree. Natomiast przeszukiwanie QuadTree dla tego zbioru zajmuje średnio 15% więcej czasu niż przeszukiwanie KDTree.

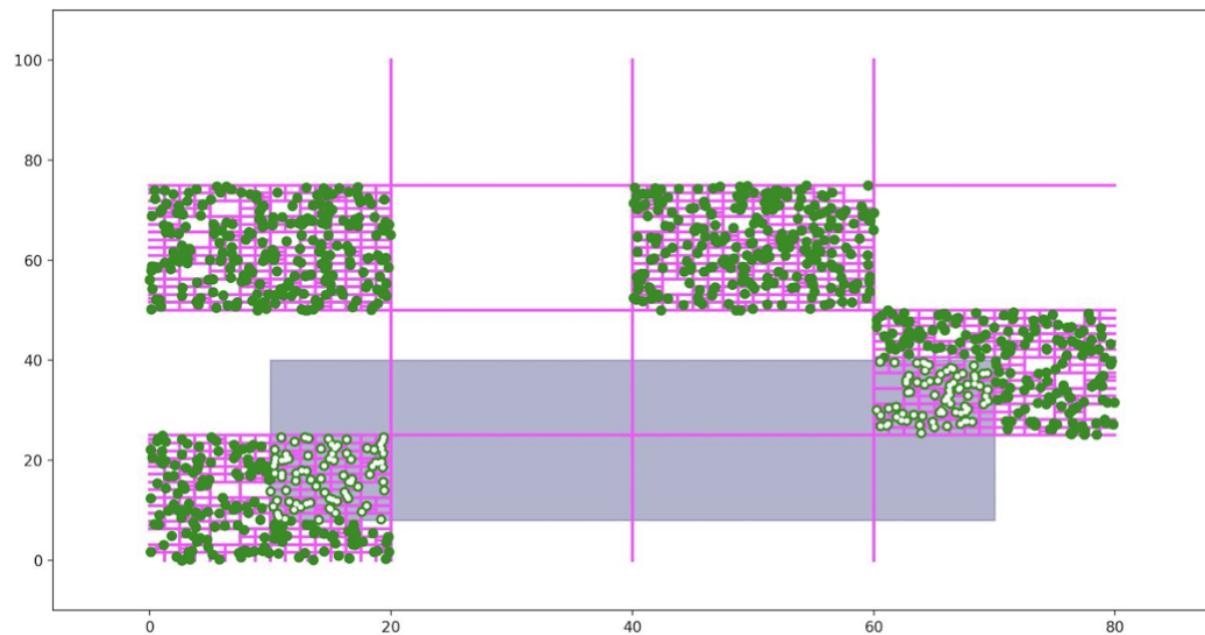
2.4 Klastry

Jest to zbiór punktów złożony z czterech gęsto skoncentrowanych skupisk punktów. Zbiór ten sprawdza działanie programu, gdy punkty zamiast być jednostajnie rozłożone po całej płaszczyźnie, są skupione tylko w poszczególnych częściach prostokątnej płaszczyzny. Może to uwydacnić różnice w czasie działania, badanych przez nas struktur. Dla tego zbioru przetestujemy poszukiwanie punktów dla dwóch różnie zadanych prostokątów. Jednego, przecinającego skupiska punktów, drugiego, który zawiera w sobie wszystkie lub żadne punkty z danego skupiska. Zabieg ten ma na celu weryfikację czy program optymalnie wyszukuje punkty należące do danego prostokąta.

Poniższe ilustracje demonstrują wyniki działania programów dla 1000 punktów:

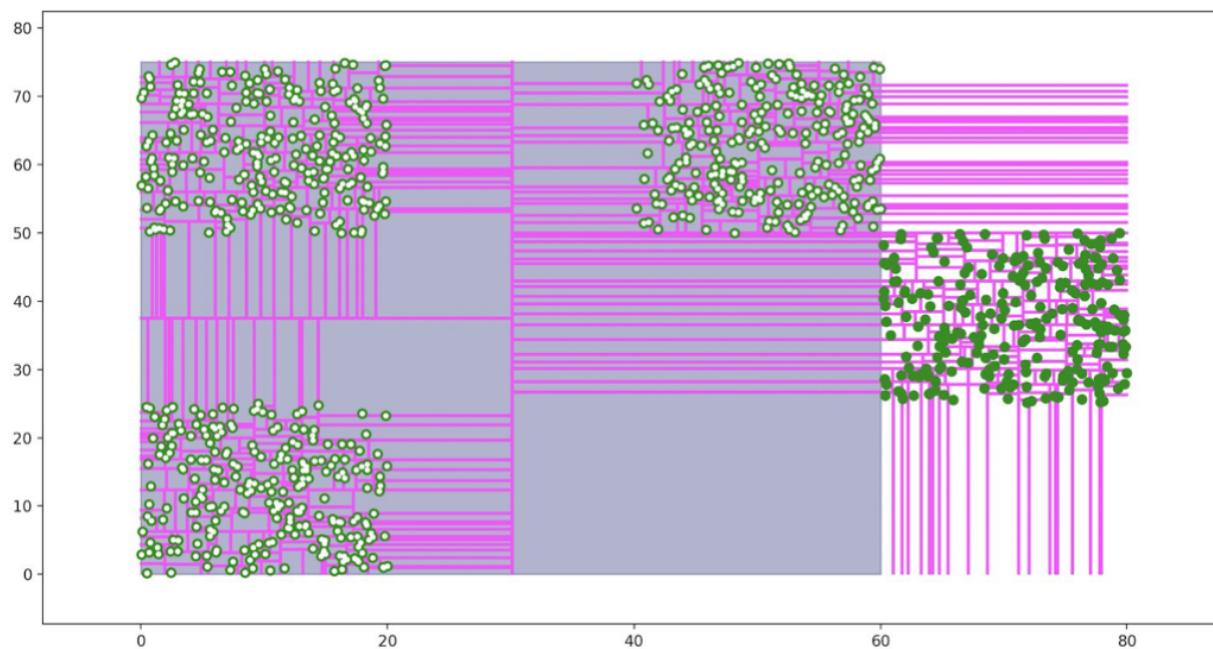


Rysunek 22: Klastry n1 KDTree

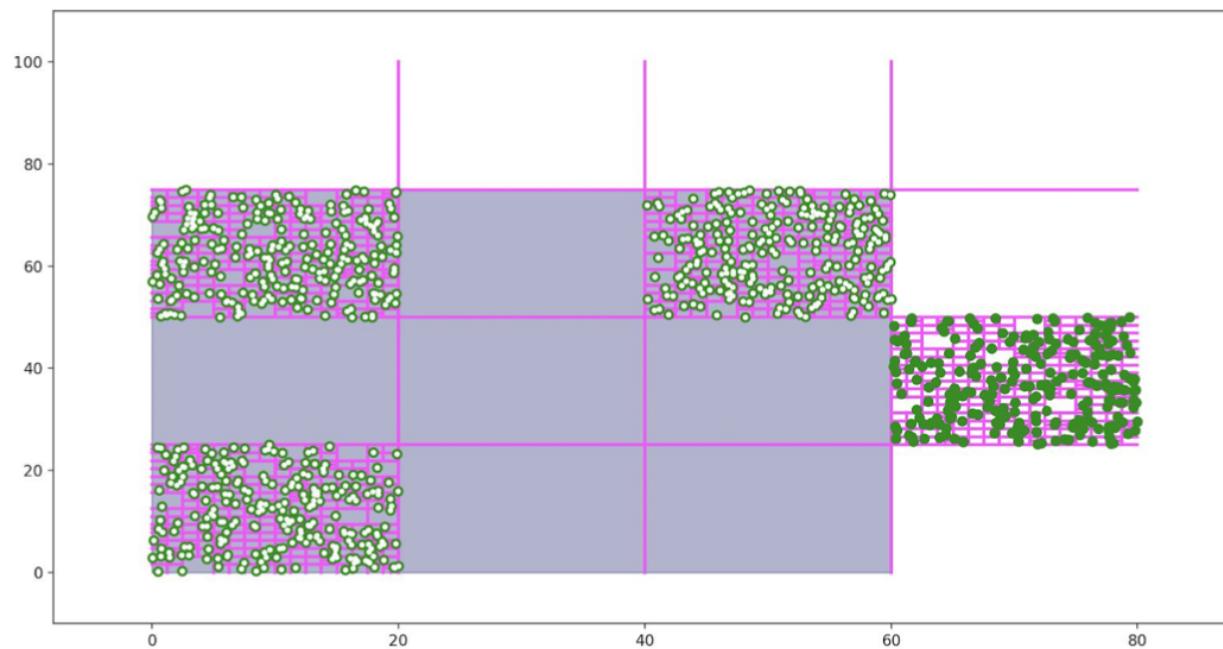


Rysunek 23: Klastry n1 QuadTree

Jak widać na zamieszczonych powyżej i poniżej ilustracjach, program działa prawidłowo zarówno przy użyciu QuadTree, jak i KDTree.



Rysunek 24: Klastry n2 KDTree

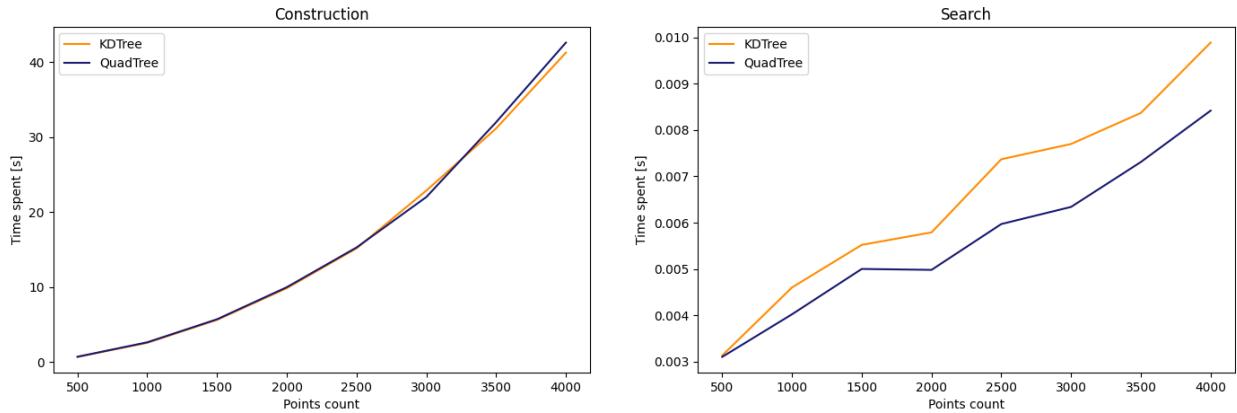


Rysunek 25: Klastry n2 QuadTree

Poniższa tabela ilustruje czasy działania programów dla podanego zbioru, przy danej ilości punktów (odpowiednio niżej są przedstawione wykresy):

Liczba punktów	Czas tworzenia KDTree [s]	Czas przeszukiwania KDTree [s]	Czas tworzenia QuadTree [s]	Czas przeszukiwania QuadTree [s]
500	0.68132	0.00313	0.6991	0.0031
1000	2.56068	0.0046	2.62753	0.00402
1500	5.61545	0.00552	5.70088	0.005
2000	9.84998	0.00579	9.98057	0.00498
2500	15.16315	0.00737	15.28278	0.00597
3000	22.87313	0.0077	22.01085	0.00634
3500	31.16272	0.00837	31.99417	0.00731
4000	41.27115	0.00989	42.60962	0.00842

Tablica 5: Wyniki pomiarów czasu dla klaster n1

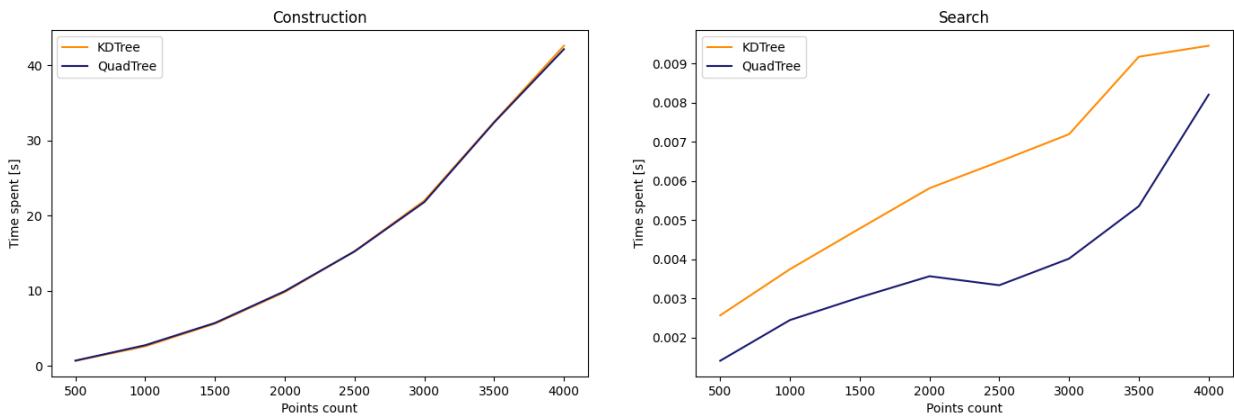


Rysunek 26: Wyniki pomiarów czasu dla klaster n1

Liczba punktów	Czas tworzenia KDTree [s]	Czas przeszukiwania KDTree [s]	Czas tworzenia QuadTree [s]	Czas przeszukiwania QuadTree [s]
500	0.67177	0.00257	0.68885	0.00141
1000	2.59746	0.00375	2.74539	0.00245
1500	5.61685	0.00479	5.7088	0.00303
2000	9.82799	0.00582	9.94246	0.00357
2500	15.26792	0.0065	15.23768	0.00334
3000	22.00586	0.0072	21.81148	0.00402
3500	32.48849	0.00918	32.4198	0.00536
4000	42.63721	0.00946	42.17839	0.00821

Tablica 6: Wyniki pomiarów czasu dla klaster n2

Na podstawie powyższych tabel i wykresów (ostatni jest na nast. stronie) możemy stwierdzić, że dla zbioru 4 tworzenie QuadTree zajmuje średnio tyle samo czasu co KDTree. Gdy



Rysunek 27: Wyniki pomiarów czasu dla klaster n2

liczba punktów w zbiorze nie przekracza 2000, przeszukiwanie KDTree dla obszaru przeszukiwanego nr 1 zajmuje średnio 10% więcej czasu niż przeszukiwanie QuadTree, w przeciwnym wypadku zajmuje ono ok. 20% więcej czasu. Natomiast dla obszaru przeszukiwanego nr 2, przeszukiwanie KDTree zajmuje 60-80% więcej czasu niż przeszukiwanie QuadTree. Możemy zauważać, że QuadTree działa w drugim przypadku zdecydowanie szybciej od KDTree, dzieje się to na skutek tego, że w tym wypadku QuadTree odcina węzły znacznie szybciej (“pływcej”) od KDTree, co bezpośrednio wpływa na szybkość przeszukiwania drzewa .

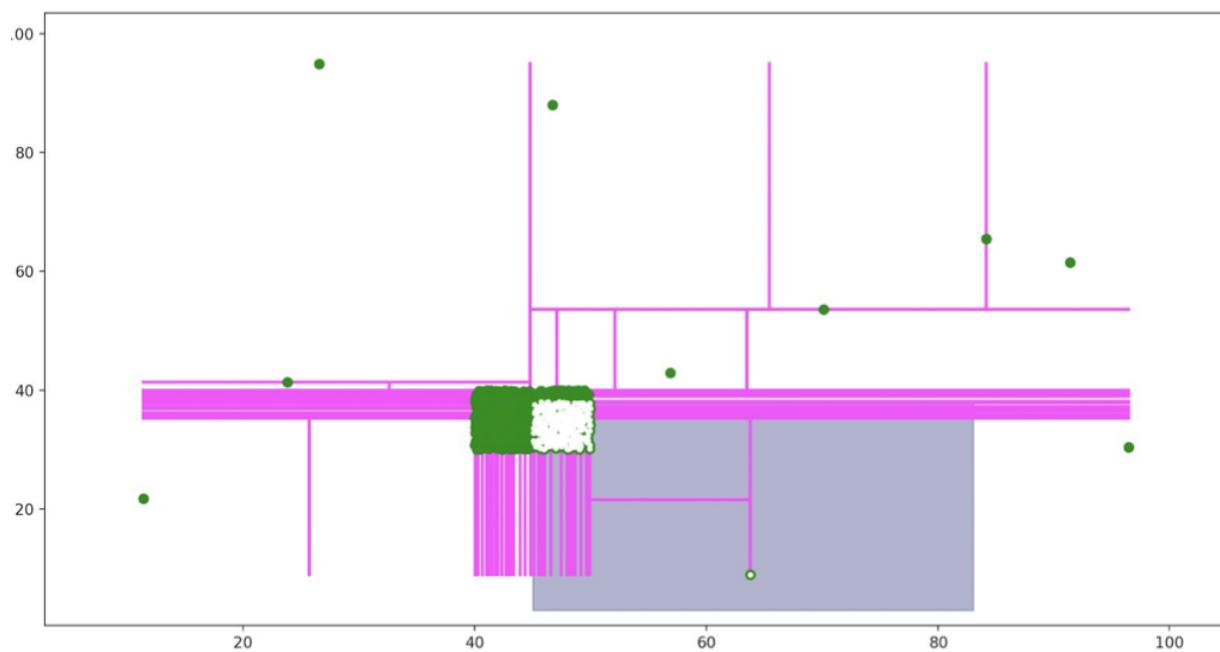
2.5 Wartości odstające (outliers)

Mamy dwa różne zbiory tutaj:

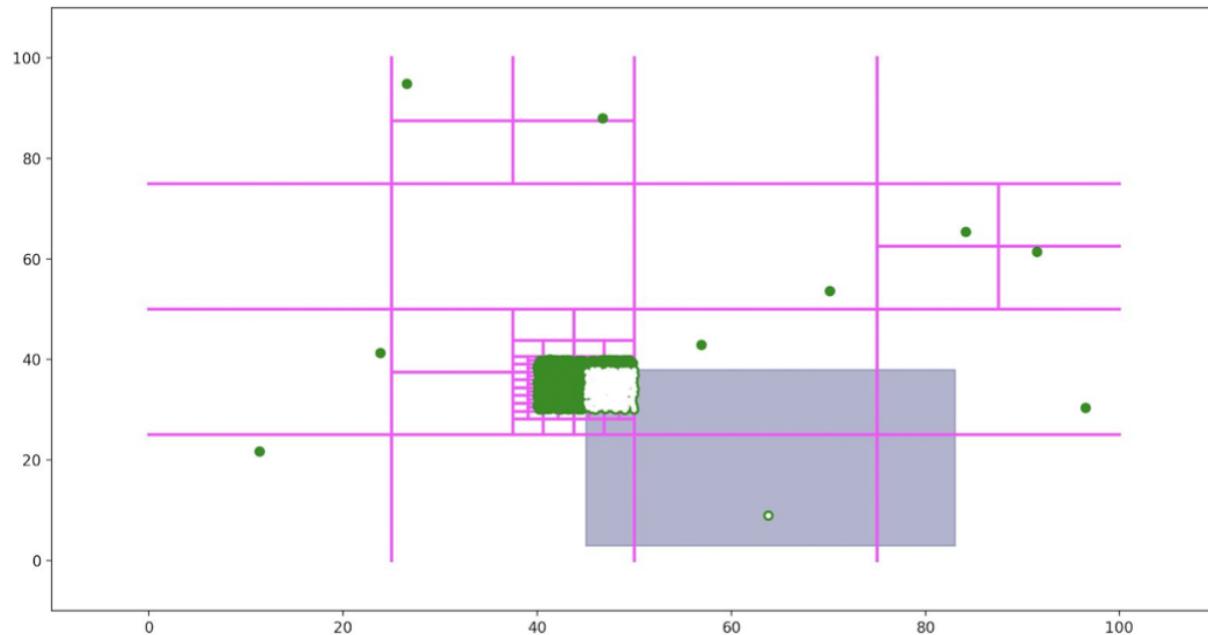
- **Zbiór 5.1** Jest to zbiór, którego 99% punktów należy do konkretnego skupiska punktów (niewielkiego obszaru będącego częścią zadanego zbioru), a pozostałe punkty są losowo rozmieszczone na zadanym zbiorze. Natomiast obszar, w którym mamy odnaleźć punkty to prostokąt, częściowo przecinający skupisko, w którym znajduje się 99% punktów naszego zbioru.
- **Zbiór 5.2** Jest to zbiór gdzie 3 jego punkty leżą na wierzchołkach zadanego prostokąta o bardzo dużych wymiarach w porównaniu do wymiarów prostokąta z resztą, a właśnie reszta punktów leży skupiona w niewielkim obszarze obok czwartego wierzchołka tego prostokąta. Natomiast obszar, w którym mamy odnaleźć punkty to prostokąt częściowo przecinający skupisko punktów i nie przecinający punktów leżących na pozostałych wierzchołkach.

Zbiory te zostały wybrane ze względu na to, że mogą być one problematyczne dla QuadTree, warto więc przeanalizować ten charakterystyczny rozkład punktów, szczególnie zwierając uwagę na różnice w czasie działania QuadTree i KDTree.

Poniższe ilustracje demonstrują wyniki działania programów dla 1000 punktów:

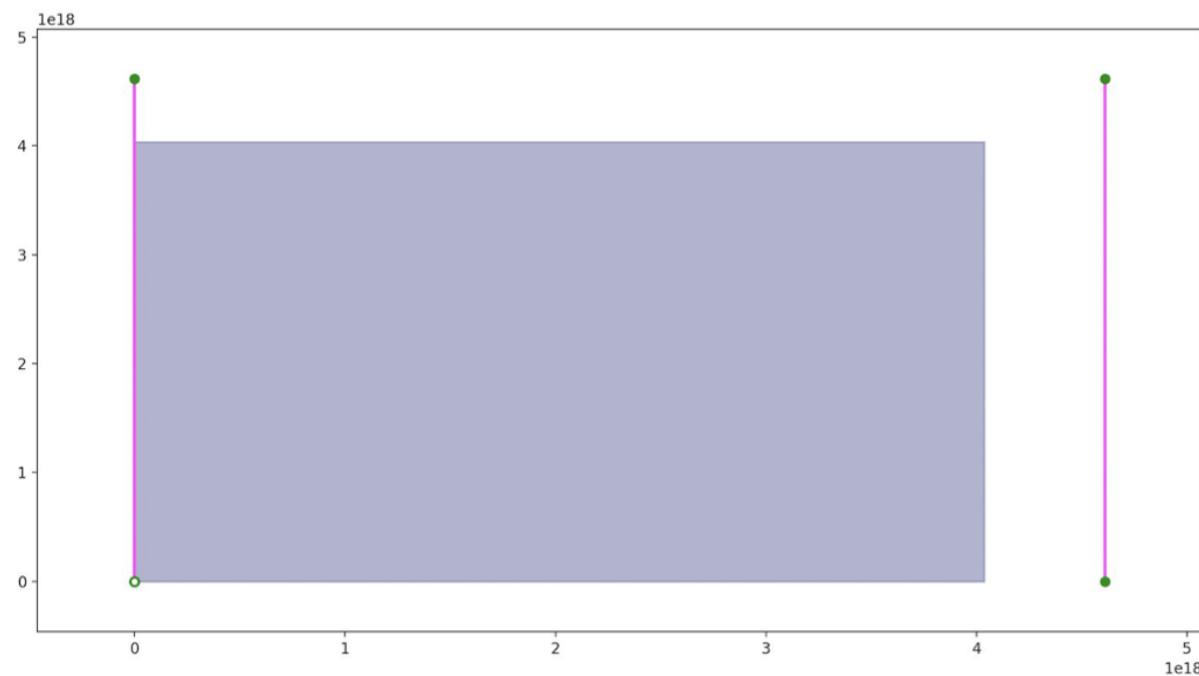


Rysunek 28: Wartości odstające n1 KDTree

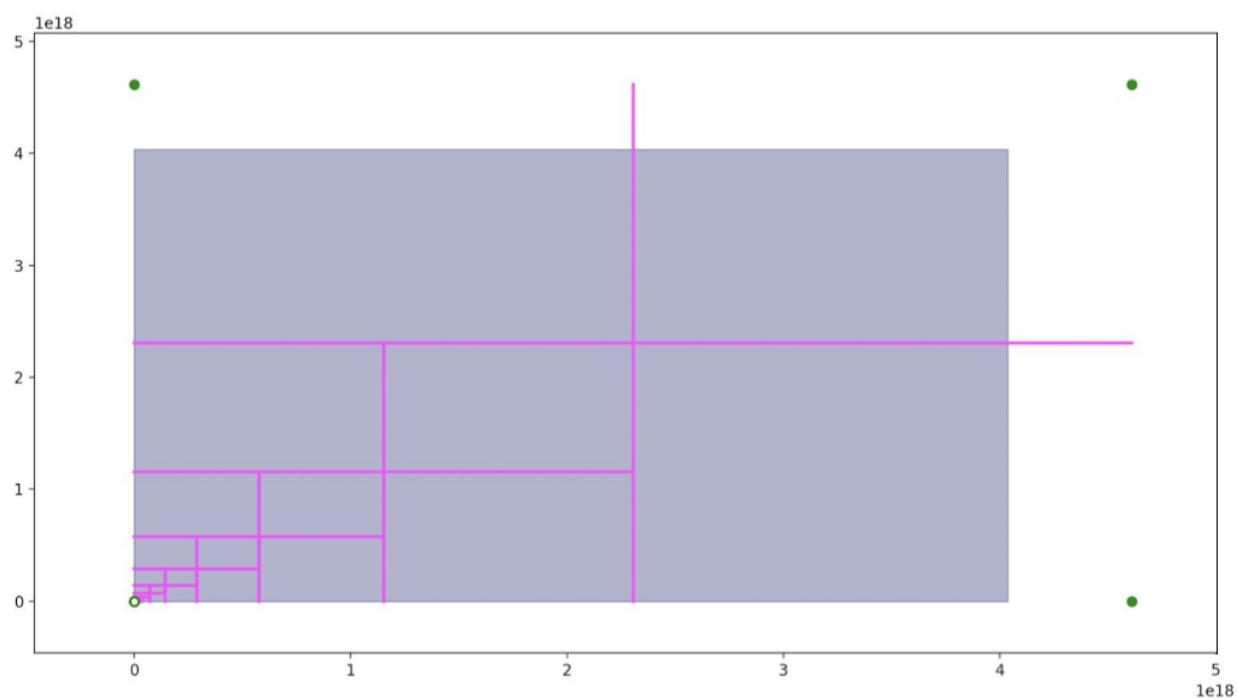


Rysunek 29: Wartości odstające n1 QuadTree

Jak widać na zamieszczonych powyżej i poniżej ilustracjach, program działa prawidłowo zarówno przy użyciu QuadTree, jak i KDTree.



Rysunek 30: Wartości odstające n2 KDTree

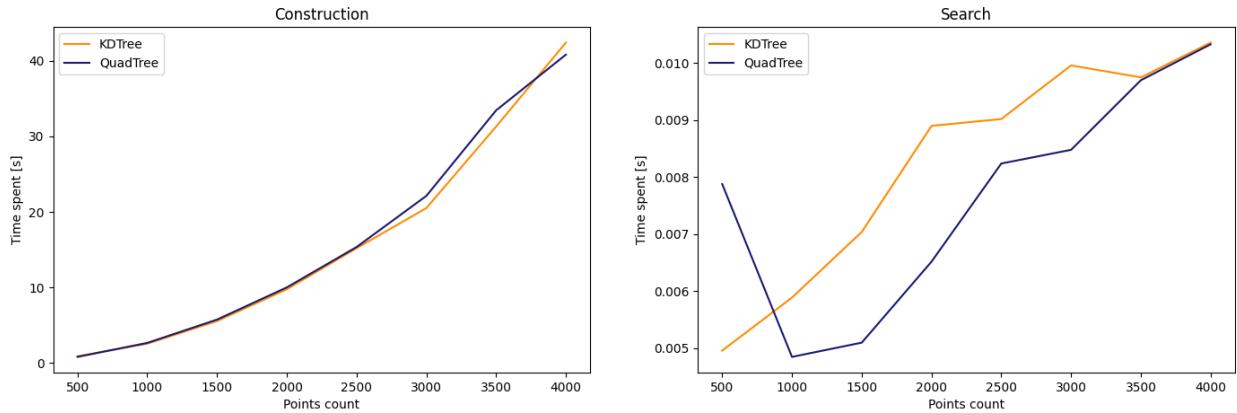


Rysunek 31: Wartości odstające n2 QuadTree

Poniższa tabela i wykresy ilustruje czasy działania programów dla podanego zbioru, przy danej ilości punktów:

Liczba punktów	Czas tworzenia KDTree [s]	Czas przeszukiwania KDTree [s]	Czas tworzenia QuadTree [s]	Czas przeszukiwania QuadTree [s]
500	0.91441	0.00496	0.83041	0.00788
1000	2.57154	0.00589	2.68101	0.00485
1500	5.56844	0.00704	5.76521	0.0051
2000	9.77039	0.0089	10.02134	0.00652
2500	15.20645	0.00902	15.35985	0.00824
3000	20.51018	0.00996	22.12177	0.00848
3500	31.29155	0.00975	33.43466	0.0097
4000	42.39134	0.01036	40.8002	0.01033

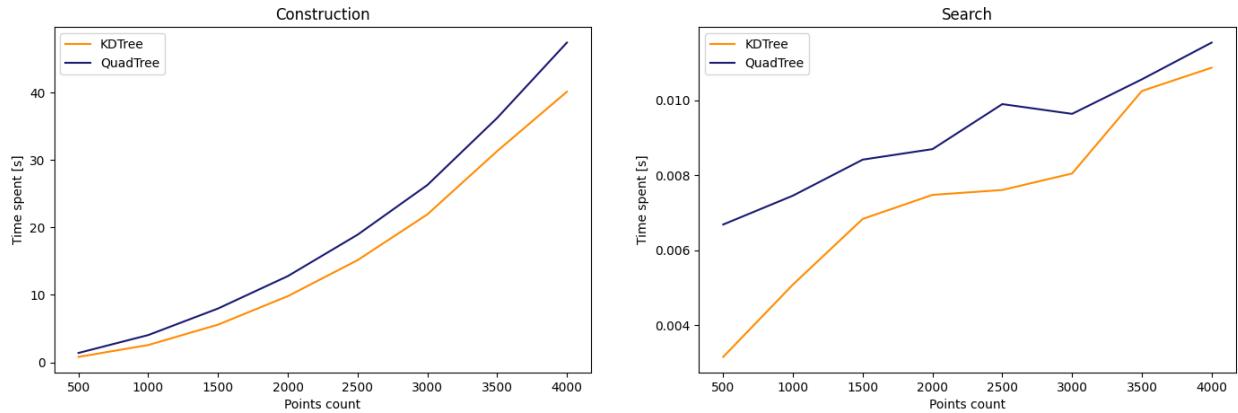
Tablica 7: Wyniki pomiarów czasu dla wartości odstających n1



Rysunek 32: Wyniki pomiarów czasu dla wartości odstających n1

Zbiory o liczbie 500 i 1000 punktów nie są brane pod uwagę w poniższej analizie, ponieważ ze względu na charakterystykę tego zbioru jest to za mała liczba punktów, aby uzyskać rzetelne wyniki odpowiednie do wyciągania z nich wniosków. Na podstawie powyższej tabeli i wykresów możemy stwierdzić, że dla zbioru 5.1 tworzenie QuadTree zajmuje tyle samo czasu co KDTree. Natomiast różnica czasów przeszukiwania tych struktur zależy od ilości punktów. Im więcej punktów tym lepiej (pod względem szybkości) wypada KDTree. Dla 1500 punktów KDTree jest przeszukiwane 40% dłużej od QuadTree, natomiast dla 4000 punktów czas przeszukiwania tych dwóch struktur jest niemal identyczny, możemy przewidywać, że dla większej ilości punktów KDTree będzie przeszukiwane szybciej niż QuadTree.

Liczba punktów	Czas tworzenia KDTree [s]	Czas przeszukiwania KDTree [s]	Czas tworzenia QuadTree [s]	Czas przeszukiwania QuadTree [s]
500	0.8176	0.00316	1.40296	0.00669
1000	2.57741	0.00509	4.05358	0.00746
1500	5.59511	0.00684	7.9869	0.00842
2000	9.8141	0.00748	12.78849	0.0087
2500	15.1543	0.00761	18.92475	0.0099
3000	21.93263	0.00805	26.28272	0.00964
3500	31.30687	0.01025	36.22217	0.01056
4000	40.10829	0.01087	47.38874	0.01154

Tablica 8: Wyniki pomiarów czasu dla wartości odstających n²Rysunek 33: Wyniki pomiarów czasu dla wartości odstających n²

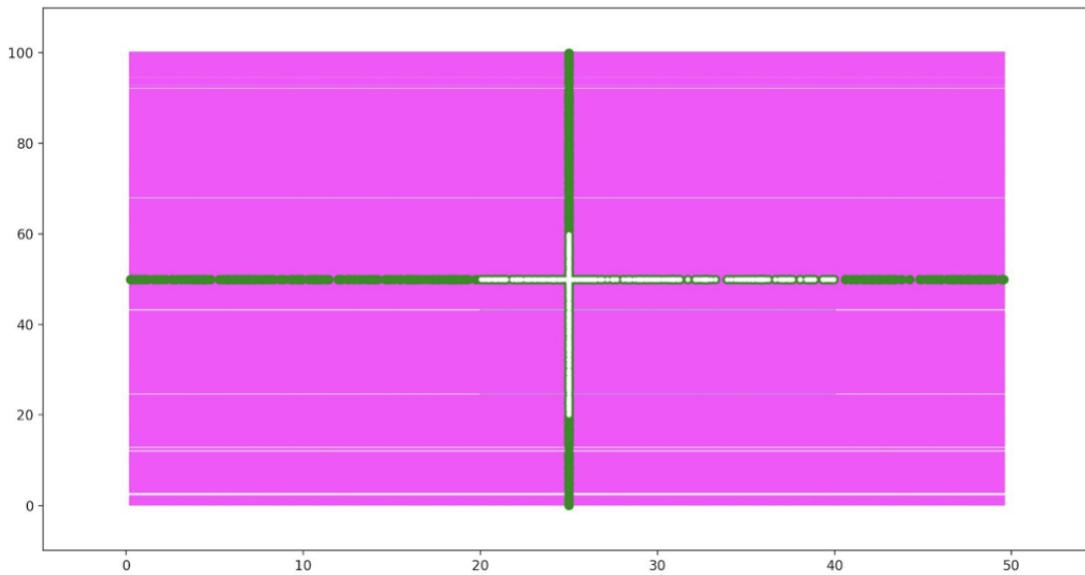
Na podstawie powyższej tabeli i wykresów możemy stwierdzić, że dla zbioru 5.2 tworzenie QuadTree zajmuje 20% dłużej niż KDTree. Natomiast różnica czasów przeszukiwania tych struktur zależy od ilości punktów. Im więcej punktów tym lepiej (pod względem szybkości) wypada QuadTree. Dla 500 punktów QuadTree jest przeszukiwane 2 razy dłużej od KDTree (dzieje się to dlatego, że punkty na kątach prostokąta są rzędu 2^{62} , więc QuadTree musi zejść bardzo głęboko, żeby dostać się do reszty punktów zawartych w obszarze $[0, 100]^2$). Dla 4000 punktów czas przeszukiwania QuadTree jest już tylko niecałe 10% dłuższy niż czas przeszukiwania KDTree. Możemy przewidywać, że dla większej ilości punktów QuadTree będzie przeszukiwane szybciej niż KDTree. Jest to dlatego, że KDTree tworzy duże rozgałęzienie pionowe i poziomowe, które zaczepia praktycznie całą stronę prostokąta, który jest bardzo duży. Potwierdzeniem jest pionowa linia podziału na rysunku 30. QuadTree w tym czasie nie komplikuje swojej struktury zewnętrz prostokąta zawierającego obszar $[0, 100]^2$. Więc zmienia prędkość tak, jakby dla zbioru o rozkładzie jednostajnym.

2.6 Krzyż

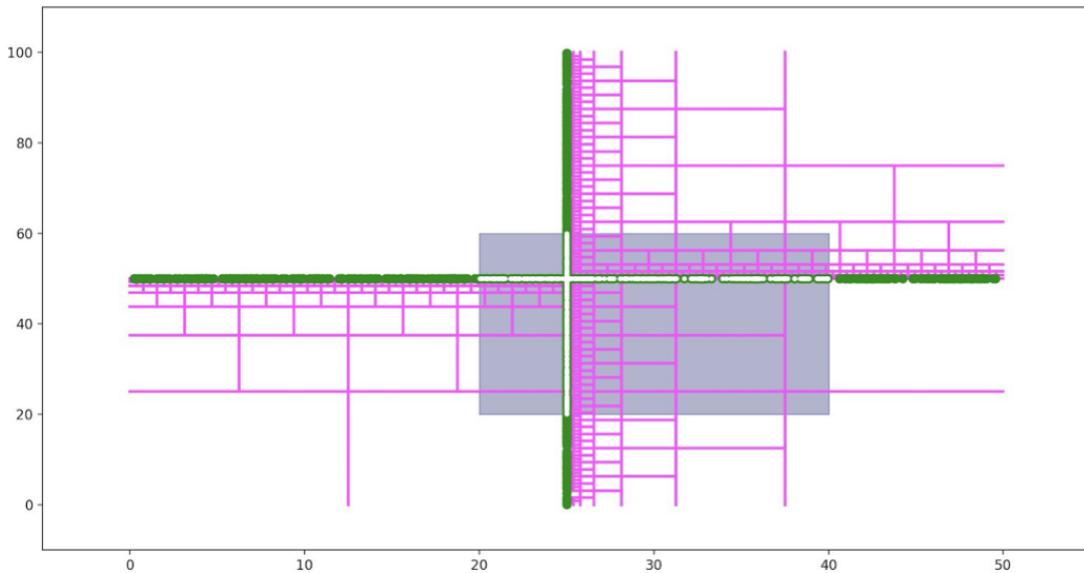
Jest to zbiór punktów należących do dwóch prostych będących osiami symetrii zadanego prostokąta. Przecinają się one w środku symetrii prostokąta. Natomiast obszar, w którym

mamy odnaleźć punkty to prostokąt zawierający fragmenty obydwóch prostych (więc i środek symetrii). Wybrany został ten zbiór punktów ze względu na to, że punkty należące do niego znajdują się tylko na dwóch prostych i to tych dzielących prostokąt na dwie równe części, co może być problematyczne dla działania programów, w związku z tym istnieje duże prawdopodobieństwo wykrycia błędów jeżeli takie program posiada. Z tego powodu postanowiliśmy przeanalizować także ten szczególny zbiór punktów.

Poniższe ilustracje demonstrują wyniki działania programów dla 1000 punktów:



Rysunek 34: Krzyż KDTree



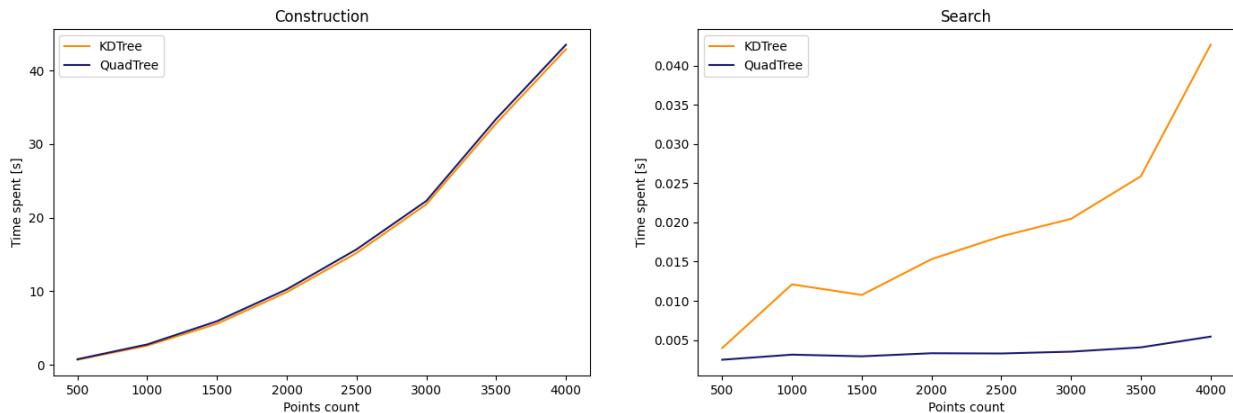
Rysunek 35: Krzyż QuadTree

Jak widać na zamieszczonych ilustracjach, program działa prawidłowo zarówno przy użyciu QuadTree, jak i KDTree.

Poniższa tabela i wykresy ilustrują czasy działania programów dla podanego zbioru, przy danej ilości punktów:

Liczba punktów	Czas tworzenia KDTree [s]	Czas przeszukiwania KDTree [s]	Czas tworzenia QuadTree [s]	Czas przeszukiwania QuadTree [s]
500	0.67863	0.00398	0.74997	0.00249
1000	2.59964	0.0121	2.7639	0.00314
1500	5.57651	0.01075	5.92202	0.00293
2000	9.86725	0.01531	10.25673	0.00332
2500	15.20696	0.01822	15.69262	0.00329
3000	21.8318	0.02045	22.28464	0.00352
3500	32.82205	0.02588	33.4228	0.00407
4000	42.91555	0.04264	43.52262	0.00544

Tablica 9: Wyniki pomiarów czasu dla krzyża



Rysunek 36: Wyniki pomiarów czasu dla krzyża

Na podstawie powyższej tabeli i wykresów możemy stwierdzić, że dla zbioru 6 tworzenie QuadTree zajmuje średnio tyle samo czasu co KDTree. Natomiast różnica czasów przeszukiwania tych struktur zależy od ilości punktów. Dla 500 punktów przeszukiwanie KDTree działa 60% dłużej od przeszukiwania QuadTree, dla 1000 punktów działa 4 razy dłużej, a dla 4000 punktów działa 8 razy dłużej. Oznacza to, że im bardziej konsystentnie punkty tworzą proste, tym różnica czasów jest coraz większa. Możemy, więc stwierdzić, że dla tak zadanego zbioru przeszukiwanie QuadTree działa znacznie szybciej od przeszukiwania KDTree.

Właśnie dlatego ten zbiór i został wybrany, że ilustruje jeden duży problem KDTree - linie równoległe do osi. Dla przypadku gdy mamy 100 punktów z prostej równoległej do OX, i 101 punktów z prostej równoległej do OY, i założymy, że rozstęp punktów z osi równoległą do OX (dalej 'x') jest większy, to podzielimy według x. O ile 101 punkt jest z prostej równoległą do OY (dalej 'y'), to wybierzymy ten punkt podziału, a jeżeli założymy, że wartości x dla wszystkich punktów z prostej 'x' oprócz jednego są mniejsze od wartości x dla 'y', to odcinamy tylko jeden punkt i zostawiamy 200, co już nie wygląda na bardzo wydajną strukturę.

Jeszcze większy problem będzie, jeżeli wartości x wszystkich punktów z prostej 'x' będą mniejsze od punktów z 'y'. Wtedy wszystkie punkty zakwalifikują do lewego poddrzewa, i tak

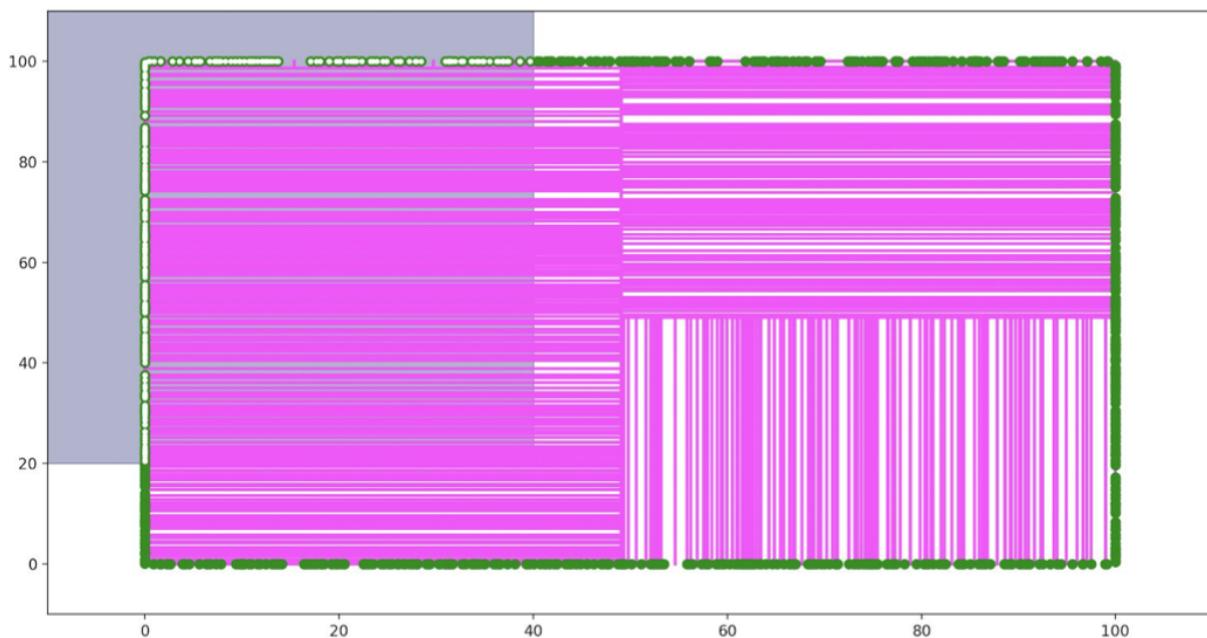
wejdziemy w nieskończoną rekurencję, bo zawsze będziemy próbować rozbić ten sam zbiór punktów, który pozostaje niezmienny za każdym razem. Jest to rozwiązane w taki sposób, że jak ilość punktów nie zmienia się, to warunek ' \leq ' staje się ' $<$ '. A o ile korzystamy z funkcji anonimowych, które zwracają nam **True**, jeżeli idziemy do lewego poddrzewa i **False**, jeżeli idziemy do prawego, to nie będzie z tym żadnym problemów przy korzystaniu z metody **contains**.

Poniższy zbiór punktów jest jeszcze jedną reprezentacją tego problemu. Tutaj właśnie w kątach i mamy tą najgorszą sytuację z możliwością nieskończonej rekurencji.

2.7 Prostokąt

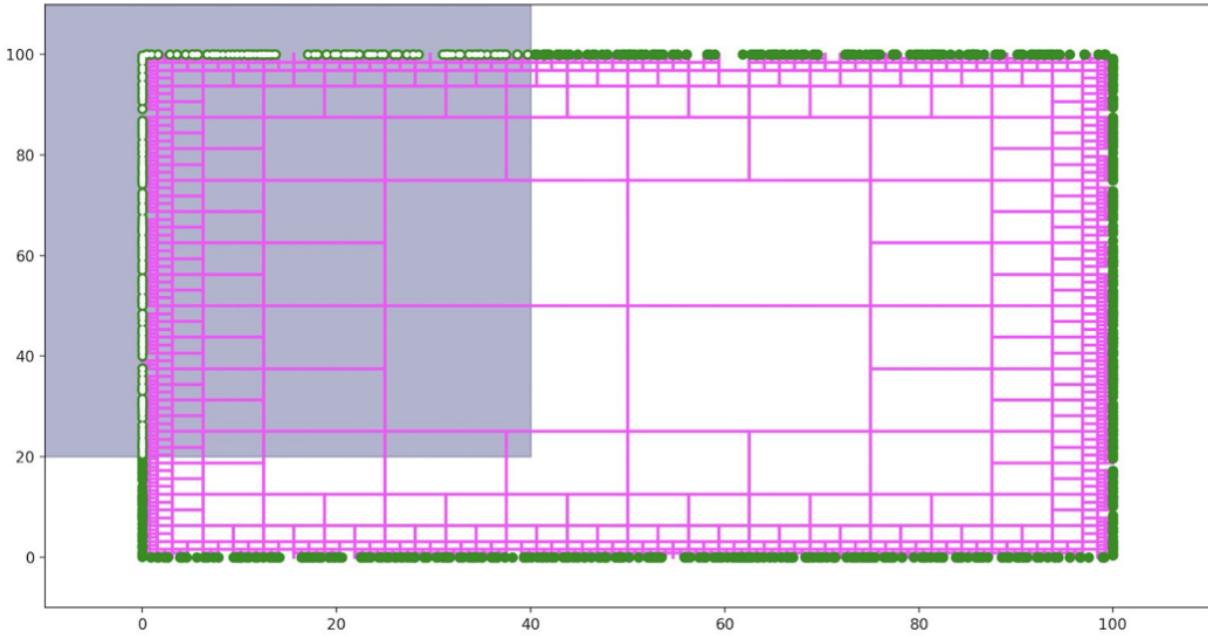
Jest to zbiór punktów należących do czterech prostych będących krawędziami zadanego prostokąta. Proste te przecinają się w wierzchołkach prostokąta. Natomiast obszar, w którym mamy odnaleźć punkty, to prostokąt zawierający fragmenty dwóch prostych, wraz z jednym wierzchołkiem. Zbiór ten został wybrany z powodów analogicznych do argumentacji analizy zbioru 6-ego.

Poniższe ilustracje demonstrują wyniki działania programów dla 1000 punktów:



Rysunek 37: Prostokąt KDTree

Jak widać na zamieszczonych ilustracjach, program działa prawidłowo zarówno przy użyciu QuadTree, jak i KDTree.



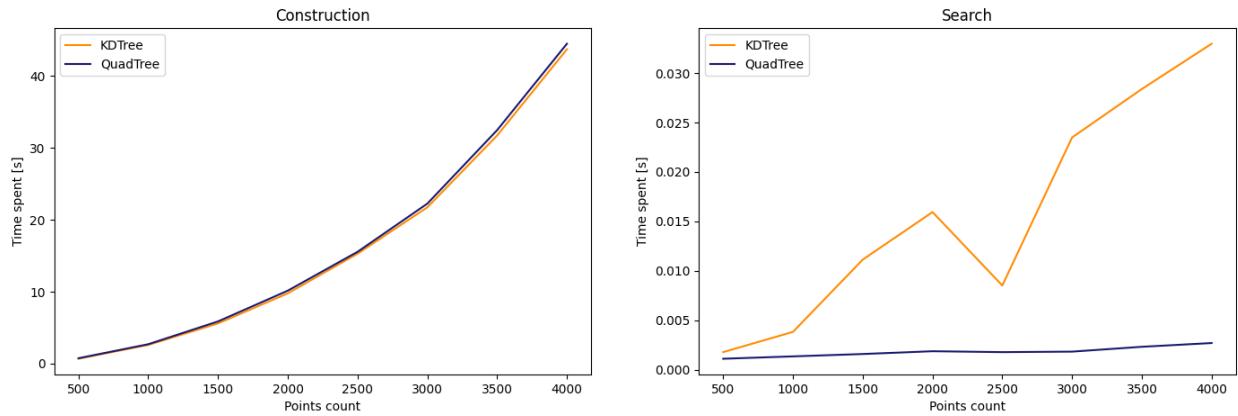
Rysunek 38: Prostokąt QuadTree

Poniższa tabela i wykresy ilustrują czasy działania programów dla podanego zbioru, przy danej ilości punktów:

Liczba punktów	Czas tworzenia KDTree [s]	Czas przeszukiwania KDTree [s]	Czas tworzenia QuadTree [s]	Czas przeszukiwania QuadTree [s]
500	0.67824	0.00177	0.75023	0.0011
1000	2.60074	0.00382	2.69825	0.00134
1500	5.59026	0.01113	5.86845	0.00158
2000	9.77206	0.01594	10.14577	0.00186
2500	15.28948	0.00851	15.5487	0.00177
3000	21.73445	0.02349	22.24732	0.00182
3500	31.71987	0.02839	32.46397	0.00231
4000	43.70219	0.03297	44.47277	0.00269

Tablica 10: Wyniki pomiarów czasu dla prostokąta

Na podstawie powyższej tabeli i wykresów z następnej strony możemy stwierdzić, że dla zbioru 7 tworzenie QuadTree zajmuje średnio tyle samo czasu co KDTree. Natomiast różnica czasów przeszukiwania tych struktur zależy od ilości punktów. Dla 500 punktów przeszukiwanie KDTree działa 70% dłużej od przeszukiwania quadtree, dla 1500 punktów działa 7 razy dłużej, a dla 4000 punktów działa 12 razy dłużej. Oznacza to, że im bardziej konsistentnie punkty tworzą prosty, tym różnica czasów jest coraz większa. Możemy, więc stwierdzić, że dla tak zadanego zbioru przeszukiwanie QuadTree działa znacznie szybciej od przeszukiwania KDTree. Możemy, więc zauważać, biorąc pod uwagę także analizę zbioru 6, że pojawienie się punktów tworzących prostą równoległą do osi OX lub OY powoduje znaczne spowolnienie działania przeszukiwania KDTree, więc możemy stwierdzić, że dla takiego typu zbiorów należy zdecydowanie korzystać ze struktury QuadTree nad KDTree.

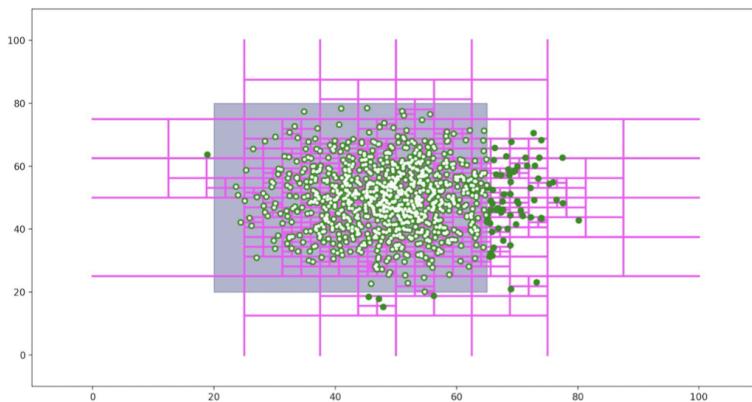
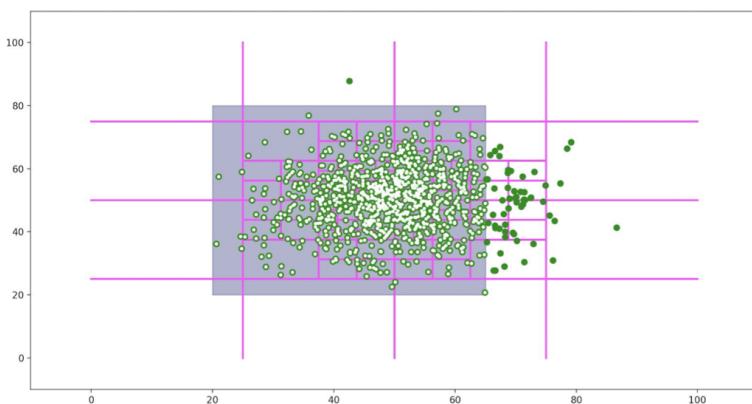


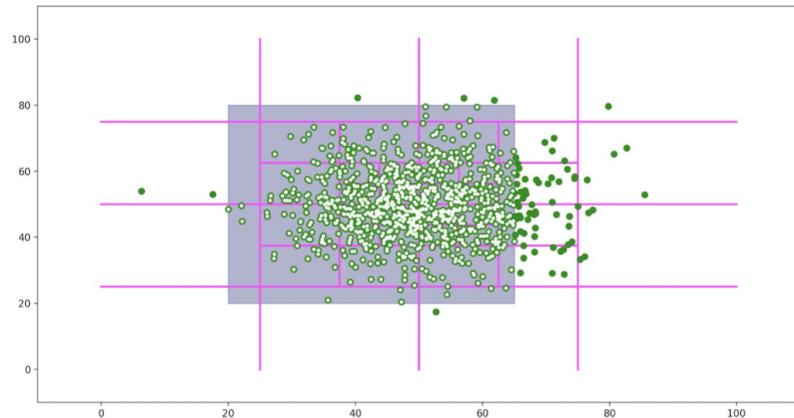
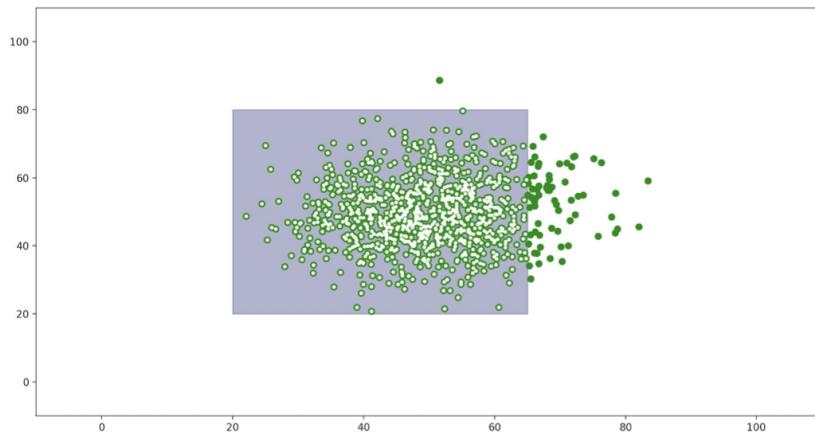
Rysunek 39: Wyniki pomiarów czasu dla prostokąta

3 Testowanie dla różnych wartości *capacity*

Przeanalizowane zostanie działanie programu dla zbioru 2, dla 1000 punktów, przy różnym capacity. W szczególności przyjrzymy się poprawności oraz czasowi działania programu.

Poniższe ilustracje demonstrują działanie programu przy różnych capacity:

Rysunek 40: QuadTree : $capacity = 1$ Rysunek 41: QuadTree : $capacity = 10$

Rysunek 42: QuadTree : $capacity = 100$ Rysunek 43: QuadTree : $capacity = 1000$

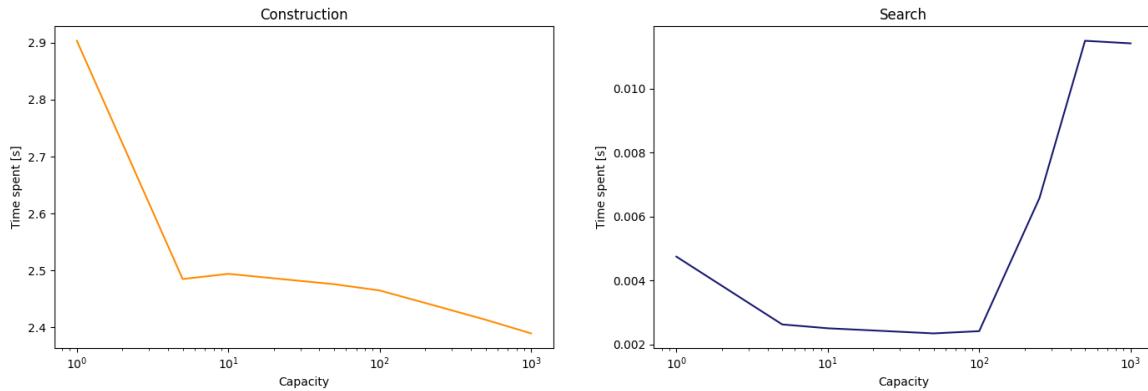
Na podstawie powyższych ilustracji możemy stwierdzić, że program działa prawidłowo również dla capacity większym niż 1.

Poniższa tabela przedstawia czas konstrukcji i przeszukiwania QuadTree dla różnych capacity, dla zbioru 2:

Capacity	Czas konstrukcji QuadTree [s]	Czas przeszukiwania QuadTree [s]
1	2.90348	0.00475
5	2.48504	0.00263
10	2.49417	0.00251
50	2.47602	0.00235
100	2.46502	0.00242
250	2.43592	0.00658
500	2.41363	0.0115
1000	2.38961	0.01142

Tablica 11: Wyniki pomiarów czasu dla różnych wartości $capacity$ dla QuadTree

Zauważmy, że tworzenie QuadTree jest najdłuższe dla capacity = 1, a dla reszty badanych capacity czas konstrukcji mieści się w przedziale 2.38-2.49 s. Możemy, więc powiedzieć, że

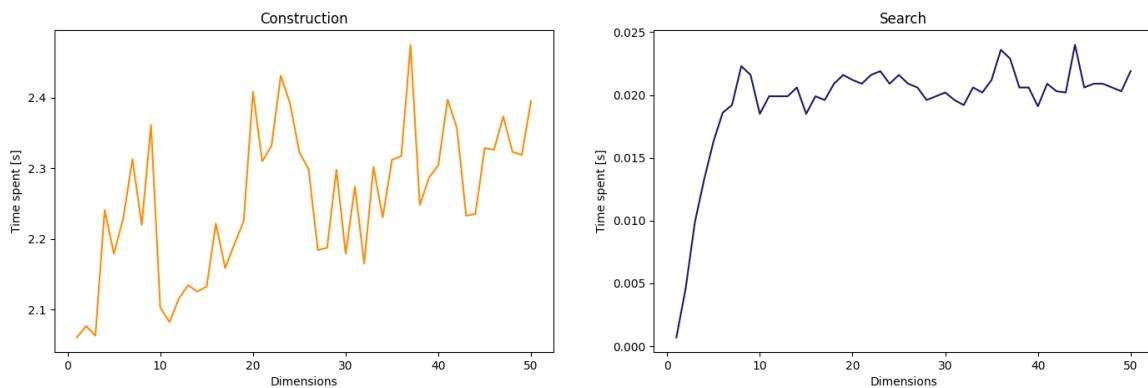
Rysunek 44: Wyniki pomiarów czasu dla różnych wartości *capacity* dla QuadTree

wybór capacity nie wpływa znacząco na czas konstrukcji QuadTree. Czas przeszukiwania jest najkrótszy dla capacity = 50 (wśród badanych), jednakże dla capacity 5-100 jest on bardzo zbliżony. Dla capacity większego od 100 czas przeszukiwania zbliża się do liniowego, dla capacity 1000 mamy po prostu poszukiwanie liniowe, możemy na tym przykładzie zobaczyć jego kosztowność i różnice pomiędzy capacity 1, z którego korzystaliśmy, a także capacity 50, które jest najbardziej optymalne wśród badanych capacity. Przeszukiwanie liniowe działa 2.5-krotnie dłużej niż przy capacity 1 i 5 razy dłużej niż przy capacity 50. Udowadnia to optymalność i prawidłowość działania naszego programu (osiągamy oczekiwana złożoność czasową, lepszą niż liniową).

4 Testowanie dla różnej ilości wymiarów

Struktura KDTTree zapewnia możliwość przechowywania punktów dla różnych wymiarów. Stąd właśnie i pochodzi nazwa - KD od 'k dimensions'. Chcieliśmy zobaczyć jak zmienia się wydajność tej struktury dla różnej ilości wymiarów. Warto zauważyć, że zgodnie z teorią jest sens używania tej struktury, gdy $n >> 2^k$, gdzie n - liczba punktów, k - liczba wymiarów. Przetestowaliśmy dla ilości wymiarów w zakresie $[1, 50]$. O ile testowana ilość punktów jest równa 1000, to już dla $k \geq 10 - 2^k$ staje się większym od n , co już całkiem nie spełnia założeń dla wydajnego stosowania.

O ile ilość wyników jest bardzo duża, to przedstawiamy tylko wykresy:



Rysunek 45: Czas budowania i przeszukiwania KDTTree dla różnej ilości wymiarów

Jak widzimy z wykresów, dla $k \geq 10$ wydajność przeszukiwania jest bardzo niewydajna, co właśnie i stwierdziliśmy wcześniej. Dla $k \in [1, 10]$ czas gwałtownie rośnie. Czyli widzimy, że ilość wymiarów wpływa wprost na czas przeszukiwania. I dla małej ilości punktów przy dużej ilości wymiarów nie warto używać struktury KDTree.

Co do budowania, to możemy stwierdzić, że jest bardzo niestabilne, chociaż przedstawione wykresy są narysowane na podstawie uśrednionych wyników z trzech pomiarów. Problemem jest oczywiście losowość punktów co może powodować sytuacje gdy KDTree nie zachowuje się najlepszym sposobem (takie jak proste linie i tp). Da się stwierdzić, że istnieje tendencja do zwiększenia czasu konstruowania. Jest ona w większości spowodowana szukaniem najlepszego wymiaru dla podziału co jest $O(kn)$. To staje się dominującym czynnikiem przy konstruowaniu. Jest ono więc $O(kn \log n)$. Dałoby się stworzyć $O(n \log n)$ zmieniając wymiary po kolei, ale powodowałoby to gorszą strukturę drzewa, co wpływa wprost na czas przeszukiwania. Więc uznaliśmy, że warto posłużyć się dłuższym konstruowaniem, które jest jednorazowe, żeby ulepszyć wyniki częstych operacji (podobnie jak działa język Julia, długa komplikacja i dużo optymalizacji powoduje, że sam skompilowany kod jest bardzo wydajny).

5 Wnioski

Na podstawie testów przeprowadzonych dla wyżej przeanalizowanych zbiorów, możemy stwierdzić, że programy przez nas zaimplementowane działają prawidłowo i poprawnie wyznaczają podzbiór punktów należących do zadanej płaszczyzny. Dla każdego z testowanych przypadków, czas konstruowania struktur był podobny (odpowiednio dla danej liczby punktów), jest to kolejnym argumentem poświadczającym o poprawności działania zaimplementowanych struktur. Analiza badanych zbiorów pozwala nam stwierdzić, że dla punktów rozmiieszczonych równomiernie na płaszczyźnie (zbiór 1 i 3) najrozsądniej wykorzystać strukturę KDTree. Dla punktów pogrupowanych w kilka skupisk, oraz tych tworzących rozkład normalny (zbiór 2 i 4) optymalniej jest użyć struktury QuadTree. Dla zbiorów złożonych z gęstego skupiska punktów i paru punktów nie należących do tego skupiska, o tym, która struktura jest bardziej wydajna, decyduje ilość punktów należących do tego zbioru (zbiory 5.1 i 5.2). Natomiast dla zbiorów złożonych z punktów należących do prostych równoległych do osi OX lub OY (zbiór 6 i 7) należy koniecznie skorzystać ze struktury QuadTree, gdyż znacznie przyspiesza to działanie programu w porównaniu do KDTree. Przetestowanie programu dla różnych wartości capacity udowadnia, że zaimplementowane przez nas struktury osiągają złożoność lepszą od liniowej, co pozwala nam przypuszczać, że nasze programy osiągnęły złożoność oczekiwana.