Przeszukiwanie obszar· w ortogonalnych QuadTree i KD-Tree Dokumentacja techniczna projektu

Szymon Tyburczy Marek SwakoŒ2024/2025

1. Wstęp do sprawozdania oraz dokumentacji

W projekcie opracowaliśmy implementacje dwóch struktur danych: KD-Tree oraz QuadTree. Obie struktury zostały zaprojektowane z myślą o efektywnym przeszukiwaniu zbioru punktów w dwuwymiarowej przestrzeni euklidesowej. Funkcjonalność obejmuje możliwość inicjalizacji przy użyciu statycznego zbioru punktów oraz wyszukiwanie wszystkich elementów znajdujących się w określonym obszarze prostokątnym.

2. Informacje o programie

Struktury zostały zaimplementowane w języku Python. Projekt zawiera moduły odpowiadające za realizację tych struktur, nazwane zgodnie z ich funkcjonalnością: QuadTree.py oraz KDtree.py.

Testy oraz generator danych są odpowiednio w plikach TEST.py oraz GENERATE.py

Projekt znajduje sie na githubie pod linkiem: https://github.com/SzymonTyburczy/Geometric-Algorithms

3. Wymagane pliki

Aby działanie było poprawne należy pobrać wszystkie odpowiednie pliki lub potrzebne pliki zamieszczone są w zipie.
Biblioteki używane w celach tworzenia to:
numpy, random, os, matplotlib, imageio - pip install matplotlib imageio

W algorytmie QuadTree obrano następujące oznacznia przy szacowaniu złożoności obliczeniowej danych operacji:

- M: Oznacza liczbę punktów, które są wstawiane do drzewa QuadTree. To dane wejściowe funkcji.
- N: Oznacza liczbę węzłów w drzewie QuadTree. Liczba węzłów zależy od tego, jak głęboko drzewo zostało podzielone.
- K: Oznacza liczbę punktów zwróconych w wyniku wyszukiwania w zadanym obszarze.

4. QuadTree

Algorytm QuadTree składa się z 3 różnych klas.

- class Point
- class QuadTreeNode
- class QuadTree

4.1 Klasa Point

Klasa Point zawiera w sobie konstruktor punktu na płaszczyźnie który tworzy punkt o współrzędnych (x, y) oraz metode __repr__ zwracającą reprezentację tekstową punktu w postaci Point(x, y).

Złożoność: Konstruktor Point - O(1)

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Point({self.x}, {self.y})"
```

4.2 Klasa QuadTreeNode

Klasa QuadTreeNode to klasa reprezentująca pojedynczy węzeł w drzewie QuadTree.
Każdy taki węzeł odpowiada pewnemu prostokątnemu obszarowi w przestrzeni 2D i przechowuje informacje o punktach należących do tego obszaru. Węzeł może być:

- Liściem – przechowuje punkty bez podziału na podobszary.
- Węzłem wewnętrznym – podzielony na cztery podobszary (ćwiartki) z odnośnikami do dzieci.

Złożoność: Konstruktor QuadTreeNode- O(1)

```
class QuadTreeNode:
    def __init__(self, x_min, y_min, x_max, y_max):
        self.x_min = x_min #lewy dolny rog wezla
        self.y_min = y_min #lewy dolny rog wezla
        self.x_max = x_max #prawy gorny rog wezla
        self.y_max = y_max #prawy gorny rog wezla

        self.points = []
        self.nw = None
        self.ne = None
        self.sw = None
        self.se = None
        self.is_leaf = True
```

W kodzie klasy **QuadTreeNode** uściślamy przedziały danego węzła

- lista points przechowuje punkty w tym węźle (tylko jeśli jest liściem bo jeśli nie jest liściem to ma dzieci i nie ma punktów)
- nw, ne, sw, se określają podział na 4 części zgodnie z zasadą tworzenia QuadTree
- Na samym końcu znajduje się informacja czy węzeł jest liściem

x_min (float): Minimalna wartość współrzędnej x dla obszaru korzenia.
y_min (float): Minimalna wartość współrzędnej y dla obszaru korzenia.
x_max (float): Maksymalna wartość współrzędnej x dla obszaru korzenia.
y_max (float): Maksymalna wartość współrzędnej y dla obszaru korzenia.

4.3 Klasa QuadTree

Złożoność: Konstruktor QuadTree- O(1)

Klasa **QuadTree** to klasa główna algorytmu zawierająca wiele metod

Na samym początku możemy znaleźć konstruktor klasy w którym tworzymy korzeń drzewa

- self.root to **QuadTreeNode** pokrywający cały podany obszar. Początkowo jest liściem.
- x_min, y_min, x_max, y_max – definiują wymiary głównego prostokąta, który będzie obsługiwać Quadtree (czyli obszar, w którym spodziewamy się wszystkich punktów).

```
def __init__(self, x_min, y_min, x_max, y_max, capacity=4):
    """
    Tworzy korzeń drzewa z zadaną pojemnością liścia (capacity),
    która określa maksymalną liczbę punktów przed podziałem.
    """
    self.root = QuadTreeNode(x_min, y_min, x_max, y_max)
    self.capacity = capacity  # maksymalna liczba punktów w węźle przed podziałem czyli 4
```

Kolejną metodą jest **metoda insert_point oraz insert.**
Metoda insert_point to publiczna metoda wstawiania obiektu klasy Point do całego drzewa,
„właściwe" wstawianie dzieje się w metodzie insert(node, point), która:
Sprawdza, czy node jest liściem (node.is_leaf).
Jeśli jest liściem i ma mniej niż 4 punkty dodaje punkt do node.points.
Jeśli jest liściem, ale przekroczył limit (capacity = 4), wówczas:
▪ dzieli węzeł (subdivide(node)) na4 dzieci,

- przenosi wszystkie dotychczasowe punkty w node.points do odpowiednich dzieci,

▪ wstawia także nowy punkt do

właściwego dziecka.
Jeśli node nie jest liściem, od razu rekurencyjnie kieruje punkt do któregoś z podwęzłów (NW, NE, SW, SE) w zależności od współrzędnych.

Złożoność: insert - O(logN)

```
def insert_point(self, x, y):
    """
    Publiczna metoda wstawiania punktu (x, y) do całego drzewa.
    """
    p = Point(x, y)
    self.insert(self.root, p)


def insert(self, node, point):
    """
    Wstawia punkt do drzewa, zaczynając od podanego węzła `node`.
    """
    # Jeśli węzeł jest liściem
    if node.is_leaf:
        # Jeśli jest jeszcze miejsce, to dodaj punkt
        if len(node.points) < self.capacity:
            node.points.append(point)
            return
        else:
            # Przekroczono pojemność -> podział
            self.subdivide(node)
            # Przenieś istniejące punkty do odpowiednich dzieci
            while node.points:
                p = node.points.pop()
                self.insert_into_child(node, p)
            # Teraz wstawiany punkt również ląduje w odpowiednim dziecku
            self.insert_into_child(node, point)
    else:
        # Jeśli węzeł nie jest liściem -> rekurencyjnie wstaw do dziecka
        self.insert_into_child(node, point)
```

Metoda insert_into_child(self, node, point):
Pomocnicza metoda określająca, do którego dziecka węzła należy wstawić dany punkt.
Parametry:

- node (QuadTreeNode): Węzeł, którego dzieci przeszukujemy.
- point (Point): Punkt do wstawienia. Oblicz współrzędne środka prostokąta reprezentowanego przez węzeł:

Na podstawie położenia punktu:

Jeśli lewy górny róg podziału , wstaw do dziecka NW.

Jeśli prawy górny róg podziału , wstaw do dziecka NE.

Jeśli lewy dolny róg podziału, wstaw do dziecka SW.

Jeśli prawy dolny róg podziału, wstaw do dziecka SE.

Rekurencyjnie wywołaj insert na odpowiednim dziecku.

Metoda subdivide(self, node)

Dzieli węzeł na cztery mniejsze prostokątne obszary, tworząc dzieci NW, NE, SW i SE.

Parametry:

- node (QuadTreeNode): Węzeł do podzielenia.

Oblicza środek prostokąta i tworzy cztery nowe węzły reprezentujące podobszary

- NW ()
- NE ()
- SW ()
- SE ()

Ustaw nowo utworzone węzły jako dzieci node.
Ustaw flagę is_leaf na False bo skoro ma dzieci to to nie liść.

Złożoność:insert_into_child - O(1)

Złożoność:subdivide - O(1)

```
def insert_into_child(self, node, point):
    """
    Pomocnicza funkcja określająca, do którego dziecka węzła wstawić dany punkt. obliczamy srodek prostokata
    """
    mid_x = (node.x_min + node.x_max) / 2.0
    mid_y = (node.y_min + node.y_max) / 2.0

    # W zależności od położenia punktu wybieramy NW, NE, SW lub SE
    if point.x <= mid_x and point.y >= mid_y:
        # NW
        self.insert(node.nw, point)
    elif point.x > mid_x and point.y >= mid_y:
        # NE
        self.insert(node.ne, point)
    elif point.x <= mid_x and point.y < mid_y:
        # SW
        self.insert(node.sw, point)
    else:
        # SE
        self.insert(node.se, point)
```

```
def subdivide(self, node):
    """
    Dzieli węzeł (node) na 4 dzieci: NW, NE, SW, SE.
    """
    mid_x = (node.x_min + node.x_max) / 2.0
    mid_y = (node.y_min + node.y_max) / 2.0

    # Tworzymy 4 podwęzły
    # NW
    node.nw = QuadTreeNode(
        x_min=node.x_min,
        y_min=mid_y,
        x_max=mid_x,
        y_max=node.y_max
    )
    # NE
    node.ne = QuadTreeNode(
        x_min=mid_x,
        y_min=mid_y,
        x_max=node.x_max,
        y_max=node.y_max
    )
    # SW
    node.sw = QuadTreeNode(
        x_min=node.x_min,
        y_min=node.y_min,
        x_max=mid_x,
        y_max=mid_y
    )
    # SE
    node.se = QuadTreeNode(
        x_min=mid_x,
        y_min=node.y_min,
        x_max=node.x_max,
        y_max=mid_y
    )

    node.is_leaf = False
```

query_range(self, node, x1, y1, x2, y2, found_points):

Przeszukuje węzeł i jego dzieci w poszukiwaniu punktów znajdujących się w zadanym prostokącie.

Parametry:

node (QuadTreeNode): Węzeł, od którego zaczynamy wyszukiwanie.

x1, y1 (float): Lewy dolny róg prostokąta wyszukiwania.

x2, y2 (float): Prawy górny róg prostokąta wyszukiwania.

found_points (list): Lista, do której dodawane są znalezione punkty.

Działanie:

Jeśli obszar węzła nie nachodzi na zadany prostokąt, zakończ (brak wyników).

Jeśli węzeł jest liściem:

Przejrzyj wszystkie punkty w węźle.

Dodaj punkty znajdujące się w prostokącie do found_points.

Jeśli węzeł posiada dzieci, rekurencyjnie przeszukaj każde dziecko

```
def query_range(self, node, x1, y1, x2, y2, found_points):
#       """
        Wyszukuje punkty w zadanego prostokąta [x1, x2] x [y1,
 y2],
        zaczynając od węzła `node`.

        Wyniki dokłada do listy `found_points`.
        """
        # Jeśli nie ma przecięcia obszarów, to kończymy bo
        if not self.intersects(node, x1, y1, x2, y2):
            return

        # Jeśli węzeł jest liściem, to sprawdź wszystkie punkty
        if node.is_leaf:
            for p in node.points:
                if (x1 <= p.x <= x2) and (y1 <= p.y <= y2):
                    found_points.append(p)
        else:
            # Przeszukaj rekurencyjnie dzieci
            self.query_range(node.nw, x1, y1, x2, y2, found_points)
            self.query_range(node.ne, x1, y1, x2, y2, found_points)
            self.query_range(node.sw, x1, y1, x2, y2, found_points)
            self.query_range(node.se, x1, y1, x2, y2, found_points)
```

Złożoność: query_range - O(k+logN)

intersects(self, node, x1, y1, x2, y2)

Sprawdza, czy obszar prostokąta węzła przecina się z zadanym prostokątem.

Parametry:

node (QuadTreeNode): Węzeł do sprawdzenia.

x1, y1 (float): Lewy dolny róg prostokąta wyszukiwania.

x2, y2 (float): Prawy górny róg prostokąta wyszukiwania.

Zwraca:

True, jeśli obszary się przecinają.

False w przeciwnym razie.

Złożoność: intersects - O(1)

```
def intersects(self, node, x1, y1,
x2, y2):
        """
        Sprawdza, czy obszar węzła
[x_min, x_max] x [y_min, y_max]
        nachodzi na [x1, x2] x [y1, y2].
        """
        return not (node.x_max < x1 or
node.x_min > x2 or
                    node.y_max < y1 or
node.y_min > y2)
```

insert_point(self, x, y): Publiczna metoda do wstawiania punktu do drzewa.
Parametry: x, y (float): Współrzędne punktu do wstawienia.
Działanie:
Tworzy obiekt Point z podanymi współrzędnymi, a następnie wywołuje insert dla korzenia drzewa.

```
def insert_point(self, x, y):
    """
    Publiczna metoda wstawiania punktu (x, y) do całego drzewa.
    """
p = Point(x, y)
self.insert(self.root, p)
```

Złożoność: insert_point - O(logN)

Złożoność: query - O(K+logN)
Wywołuje query_range O(K+logN) i przekształca wynik funkcja change O(K)

query(self, x1, y1, x2, y2):

Publiczna metoda do wyszukiwania punktów w prostokącie.

Parametry:

x1, y1 (float): Lewy dolny róg prostokąta wyszukiwania. x2, y2 (float): Prawy górny róg prostokąta wyszukiwania. Zwraca:

Lista punktów (w formacie) znajdujących się w prostokącie.

Działanie:

Tworzy pustą listę found_points.

Wywołuje query_range dla korzenia drzewa.

Zwraca listę znalezionych punktów w formacie krotek

```
def query(self, x1, y1, x2, y2):
    """
    Publiczna metoda wyszukiwania punktów w prostokącie (x1, y1, x2, y2).
    Zwraca listę znalezionych punktów.

    """
        found_points = []
        self.query_range(self.root, x1, y1, x2, y2, found_points)
    return change(found_points)
```

find_end_points nie dotyczy już bezpośrednio żadnej z klas zadaniem tej funkcji jest wyszukanie granic dla korzenia QuadTree

Złożoność: find_end_points - O(M)
Znajduje minimalne i maksymalne wartości współrzędnych punktów iterując przez M punktów.

```
def find_end_points(points):
    x_min = float('inf')
    x_max = float('-inf')
    y_min = float('inf')
    y_max = float('-inf')

    for x, y in points:
        x_min = min(x_min, x)
        x_max = max(x_max, x)
        y_min = min(y_min, y)
        y_max = max(y_max, y)
return x_min, y_min, x_max, y_max
```

Funkcja CreateQuad buduje QuadTree używajac odpowiednich przedziałów oraz wstawia wszystkie punkty w stworzone drzewo.

Złożoność: CreateQuad - O(MlogN)

1. **Szukamy granic obszaru O(M)**
2. **Tworzy QuadTree O(1)**
3. **Wstawia każdy punkt O(MlogN)**
   Zatem :O(M+MlogN)=O(MlogN)

```
def CreateQuad(points_to_insert):
    qt =QuadTree(find_end_points(points_to_insert)[0],
find_end_points(points_to_insert)[1],
                          find_end_points(points_to_insert)[2],
              find_end_points(points_to_insert)[3], capacity=4)

    for (x, y) in points_to_insert:
        qt.insert_point(x, y)
    return qt
```

Funkcja change zamienia punkt klasy Point na zwykłego floata w celu porównania wyników z KDTree

Złożoność: change - O(K)
Iteracja przez listę punktów: O(K), gdzie K to liczba punktów

```
def change(result):
    new_result = []
for i in range(len(result)):
        new_result.append((result[i].x,
result[i].y))

    return new_result
```

Ogólny opis działania algorytmu oraz jego zamysł:

Algorytm QuadTree tworzy obiekt klasy QuadTree z odpowiednio znalezionymi wartościami końcowymi. Wykonuje się to w funkcji CreateQuad. Po odpowiednim wywołaniu tworzony jest korzeń drzewa. Następnym krokiem jest wstawienie wszystkich punktów do drzewa, co również robi funkcja CreateQuad. W CreateQuad wywoływana jest metoda insert_point() na wcześniej stworzonym obiekcie QuadTree. Publiczna metoda insert_point() tworzy punkt klasy Point i wywołuje metodę insert() na korzeniu.

Metoda insert sprawdza, czy węzeł jest liściem: jeśli nie jest, to rekurencyjnie wstawiamy punkt do dziecka za pomocą metody insert_into_child(). Natomiast jeśli węzeł jest liściem, sprawdzamy, czy jest w nim jeszcze miejsce. W sytuacji, gdy miejsce jest, dodajemy punkt. Natomiast gdy miejsca nie ma, wywoływana jest metoda subdivide(), a punkty przenoszone są do odpowiednich dzieci.

Wyszukiwanie odbywa się za pomocą metody query(), w której wywoływana jest metoda query_range(). Wyszukuje ona punkty zadanego prostokąta [x1, x2] x [y1, y2], zaczynając od węzła node. Wykorzystywana jest w niej metoda intersects(), której zadaniem jest sprawdzenie, czy obszar węzła [x_min, x_max] x [y_min, y_max] nachodzi na wprowadzone punkty graniczne.

5. KD-Drzewa

Algorytm KD-Tree składa się z klasy głównej KDTree, dwóch podklas definiujących węzły oraz metod budującej drzewo i realizującej algorytm przeszukiwania zakresowego z metodami pomocniczymi.

5.1 Konstruktory klas

5.1.1 Klasa KDTree

Główna klasa KDTree implementuje strukturę drzewa. Umożliwia inicjalizację drzewa przy użyciu zbioru punktów oraz wyszukiwanie punktów w zadanym obszarze.

Przyjmuje tablicę krotek, zawierających współrzędne punktów.

Konstruktor klasy KDTree:

```
class KDtree:
    def __init__(self,points):
        xpoints = self.sort_by_dim(points,0)
ypoints = self.sort_by_dim(points,1)
                    self.root =
self.build_tree([xpoints,ypoints],0)
```

xpoints, ypoints: punkty posortowane względem odpowiednio współrzędnej x oraz y

self.root: korzeń utowrzony za pomocą metody build_tree

5.1.2 Klasa Vertex

Klasa Vertex reprezentuje wewnętrzny węzeł drzewa KD.

Przyjmuje krotkę zawierającą współrzędne punktu oraz prawe i lewe dziecko, czyli kolejne wierzchołki drzewa.

Konstruktor klasy Vertex:

```
class Vertex:
    def __init__(self, point, left=None, right=None):
        self.point
= point
self.left = None
self.right = None
```

point: punkt węzła, według niego tworzony jest podział płaszczyzny na dwie podpłaszczyzny w trakcie tworzenia oraz przeszukiwania drzewa

left, right: odniesienia do dzieci węzła

5.1.3 Klasa Leaf

Klasa Leaf reprezentuje liść drzewa KD, przechowuje pojedynczy punkt, ale w przeciwieństwie do klasy Vertex, nie posiada dzieci.

Przyjmuje krotkę zawierającą współrzędne punktu

Konstruktor klasy Leaf:

```
class Leaf:
 def __init__(self, point):
       self.point = point
```

point: przechowuje punkt

5.2 Metody klasy KDTree

5.2.1 Metoda get_root:

Getter podający korzeń drzewa.

5.2.2 Metoda sort_by_dim

Metoda statyczna, tworząca kopię zbioru punktów i sortująca ją według zadanego wymiaru.

Przyjmuje tablicę punktów do posortowania, oraz indeks współrzędnej według którego ma zostać wykonane sortowanie.

```
def sort_by_dim(points, dim):

   # stworzenie kopii zbioru punktów
   sorted_points = points[:]
   # sortowanie punktów względem wybranej
współrzędnej     sorted(sorted_points, key=lambda x:
x[dim])
   # zwrócenie posortowanych punktów
   return sorted_points
```

5.2.3 Metoda divide_points

Metoda dzieląca podany zbiór punktów na lewy i prawy względem wybranej współrzędnej. Używana w budowaniu drzewa.

Przyjmuje tablicę tablic, które należy podzielić według wskazanej współrzędnej oraz tą współrzędną.

```
def divide_points(self,points,dim):
    # sprawdzenie czy zbiór punktów nie jest pusty
    if not points[dim]:
        return None
    # sprawdzenie czy zbiór punktów zawiera tylko jeden punkt
    if len(points[dim]) == 1:
        return points[dim][0], [], []
    # wyznaczenie mediany punktów względem wybranej
współrzędnej    median = points[dim][(len(points[dim]) - 1) // 2]
[dim]
    median_point = points[dim][(len(points[dim]) - 1) // 2]
    # stworzenie dwóch zbiorów punktów: lewego i prawego
    left = []
    right = []
    # podział punktów na zbiory lewy i prawy
    for i in range(2):left_points = [point for point in points[i] if (point[dim] <= median and point !=
median_point)]
        right_points = [point for point in points[i] if point[dim] >
median]        left.append(left_points)
    right.append(right_points)
    # zwrócenie mediany punktów oraz zbiorów
lewego    return median_point, left, right
```

5.2.4 Metoda build_tree

Rekurencyjna metoda budująca drzewo KD.

Przyjmuje tablicę tablic punktów, posortowanych według kolejnych współrzędnych, oraz głębokość rekurencji, która służy do określenia współrzędnej, według której ma zostać wykonany podział punktów

```
def build_tree(self,points,depth):
    # wyznaczenie współrzędnej, względem której dokonujemy
podziału    dim = depth % 2
    # sprawdzenie czy zbiór punktów zawiera tylko jeden punkt
    # jeśli tak, zwracamy liść z tym punktem
    if len(points[dim]) == 1:
```

```
            return self.Leaf(points[dim][0])
        # podział zbioru punktów na zbiory lewy i prawy
        division = self.divide_points(points, dim)
        # jeśli podział istnieje, to wyznaczamy medianę punktów, lewy i prawy
    zbiór     if division:
            median, left, right = division
        else:
            return None
        # stworzenie wierzchołka drzewa KD
        root = self.Vertex(median)
        # rekurencyjne wywołanie metody build_tree dla zbiorów lewego i prawego
    root.left = self.build_tree(left, depth + 1)
        root.right = self.build_tree(right, depth + 1)
        # zwrócenie korzenia drzewa KD
        return root
```

5.2.4 Metoda search_area

Metoda rekurencyjna realizująca algorytm przeszukiwania zakresowego na drzewie KD.

Przyjmuje krotkę zawierającą współrzędne lewego dolnego i prawego górnego punktu ograniczającego zadany obszar, aktualnie rozpatrywany wierzchołek oraz głębokość rekurencji.

```
    def search_area(self,area,root,depth):
        # wyznaczenie współrzędnej, względem której dokonujemy
    podziału     dim = depth % 2
        # sprawdzenie czy korzeń drzewa nie jest pusty
        if root is None:
            return []
        # sprawdzenie czy korzeń drzewa jest klasy Vertex
        if root.getclass() == self.Vertex:
            # stworzenie dwóch zbiorów punktów: lewego i prawego
       left = []
       right = []
       # sprawdzenie czy obszar przecina się z lewym i prawym
poddrzewem         if root.point[dim]>= area[0][dim]:
    left = self.search_area(area,root.left,depth+1)
        if root.point[dim]<= area[1][dim]:
    right = self.search_area(area,root.right,depth+1)
        # sprawdzenie czy korzeń drzewa należy do zadanego obszaru
            if area[0][0] <= root.point[0] <= area[1][0] and area[0][1] <= root.point[1] <=
        area[1][1]:
            return left + [root.point] + right
```

```
    else:
            return left + right
        else:
         # sprawdzenie czy liść należy do zadanego obszaru
         if (area[0][0] <= root.point[0] <= area[1][0]) and (area[0][1] <= root.point[1]
    <= area[1][1]):
             return
[root.point]   return
[]
```

5.3 Metody klas Vertex i Leaf

Klasa Vertex oraz Leaf zawierają analogiczne metody zwracające klasę obiektu def

```
getclass(self):
 return self.__class__
```

5.4 Zamysł oraz opis działania KDTree

Drzewo KD to struktura danych umożliwiająca efektywne przeszukiwanie przestrzeni wielowymiarowej.

5.4.1 Inicjalizacja drzewa

Algorytm przyjmuje zbiór punktów przestrzeni dwuwymiarowej, następnie sortuje je według kolejnych współrzędnych, co umożliwia późniejsze łatwe znajdowanie mediany, potrzebne do konstrukcji zrównoważonego drzewa.

5.4.2 Budowa drzewa

Metoda budująca drzewo, rekurencyjnie dzieli przestrzeń na mniejsze podzbiory, wzdłuż mediany bieżącego wymiaru. Mediana staje się węzłem drzewa, a pozostałe punkty tworzą lewe i prawe poddrzewo. Gdy lista punktów ma tylko jeden element, tworzony jest liść, który nie posiada dalszych poddrzew.

5.4.3 Otrzymana struktura

Otrzymane drzewo, dzięki podziałowi wzdłuż mediany, ma zrównoważoną strukturę, co umożliwia efektywne wyszukiwanie punktów.

5.4.4 Przeszukiwanie zakresowe drzewa

Algorytm rozpoczyna przeszukiwanie od korzenia i sprawdza, czy obszar węzła przecina się z zadanym prostokątem. Algorytm rekurencyjnie odwiedza wierzchołki o odpowiednich zakresach i dodaje je do tablicy wynikowej, jeśli zawierają się całkowicie w zadanym obszarze. Algorytm kończy się, gdy zostaną przetworzone wszystkie wierzchołki których zakres przecina się z podanym prostokątem, oraz gdy sprawdzone

zostaną ich liście. Po zakończeniu algorytmu, tablica wynikowa zawiera wszystkie punkty należące do podanego obszaru.

5.5 Złożoność obliczeniowa i pamięciowa KDTree

5.5.1 Klasy w KDTree

Inicjalizacja i budowa KDTree

Złożoność Obliczeniowa

Sortowania punktów według wymiarów korzysta z algorytmu quicksort o złożoności oczekiwanej O(nlogn).

Każde wywołanie metody build_tree wywołuje metodę pomocniczą divide_points, która iteruje po tablicy points, daje to złożoność O(n).

Metoda build_tree jest wywoływana rekurencyjnie, za każdym razem dzieląc punkty na, w przybliżeniu równe, dwa podzbiory, więc jest wywoływana logn razy.

Łącznie budowa drzewa ma złożoność O(nlogn).

Złożoność Pamięciowa

Punkty w drzewie zajmują O(n) pamięci.

Węzłów drzewa jest tyle samo co punktów, jeden punkt to jeden węzeł, więc zajmują razem O(n) pamięci.

Łącznie daje to złożoność pamięciową O(n)

Klasa Vertex

Złożoność Obliczeniowa

Stworzenie obiektu tej klasy kosztuje O(1) obliczeń

Złożoność Pamięciowa

Każdy wierzchołek przechowuje punkt O(1) pamięci, oraz odnośniki do lewego i prawego poddrzewa O(1) pamięci

Łącznie daje to złożoność pamięciową O(1)

Klasa Leaf

Złożoność Obliczeniowa

Stworzenie obiektu tej klasy kosztuje O(1) obliczeń

Złożoność Pamięciowa

Każdy wierzchołek przechowuje punkt O(1) pamięci

5.5.2 Metody w KDTree

Metoda build_tree

Jej złożoność została opisana wyżej we fragmencie opisującym złożoność konstrukcji drzewa

Metoda search_area

Złożoność Obliczeniowa:

każdy węzeł na danej głębokości jest odwiedzany, jeśli jego obszar przecina się z zadanym obszarem prostokątnym, w najgorszym przypadku odwiedzone zostaną wszystkie n wierzchołki

Przeglądane są jedynie dzieci, których zakres przecina się z zadanym obszarem, co obniża liczbę odwiedzanych wierzchołków dla typowego przypadku do $O(\sqrt{n})$

Dodawanie punktów k należących do wyniku działa w złożoności O(k)

Łącznie dla średniego przypadku złożoność wynosi $O(\sqrt{n} + k)$, dla przypadku pesymistycznego O(n)

Złożoność Pamięciowa:

Pamięć potrzebna dla stosu rekurencji, głębokość drzewa czyli O(logn)

Znalezione wyniki wpisywane do tablicy rezultatów O(k)

Łącznie złożoność pamięciowa wynosi O(logn + k)

6. Porównanie czasów wykonywania algorytmów

6.1 Dane testowe:

Dane Użyte do porównania czasów budowania struktur

Zbiory testowe 1-5 zawierają kolejno:

Numer Testu	Liczba punktów
1	20,000
2	40,000
3	60,000
4	80,000
5	100,000

Tabela 6.1 liczby punktów dla testów 1-5

Punkty zostały wygenerowane losowo z kwadratu ograniczonego punktami
(-10000,-10000), (10000,10000)

Dane Użyte do porównania czasów przeszukiwania struktur

Zbiory testowe 1-5 są analogiczne jak w tabeli 6.1.

Numer Testu	Liczba punktów	Zakres (współrzędne x, y)	Poszukiwany obszar
6	20	(0.0, 0.0) do (8.0, 8.0)	(0, 0) do (100, 150)
7	10	(0.0, 0.0) do (7.0, 7.0)	(0, 0) do (70, 60)
8	10	(0.0, 0.0) do (10.0, 10.0)	(-10, 0) do (50, 70)
9	10	(-9.5, -9.5) do (9.5, 9.5)	(0, 0) do (10, 15)
10	15	(-4.5, -4.5) do (4.5, 4.5)	(-5, 0) do (10, 2)
11	10	(4.0, 4.0) do (10.0, 10.0)	(0, 0) do (8, 8)
12	100	(-10.0, 0.0) do (10.0, 15.0)	(0, 0) do (7, 7)
13	10	(4.0, 4.0) do (10.0, 10.0)	(-200, -200) do (500, 500)
14	400	(-2.0, 2.0) do (2.0, 25.0)	(-4, -4) do (4, 4)
15	100	(4.0, 4.0) do (10.0, 10.0)	(-2, -2) do (2, 2)
16	25	(4.0, 4.0) do (10.0, 10.0)	(4, 4) do (10, 10)
17	420	(-20.0, -3.0) do (40.0, 10.0)	(-10, 0) do (10, 15)

Tabela 6.2 liczby punktów i zakresy dla testów 6-17

Testy 6 - 10 to losowo rozmieszczone punkty, testy 11-16 to punkty równomiernie rozmieszczone na kwadracie o zadanych wierzchołkach. Test 17 to punkty równomiernie rozłożone na prostokącie o zadanych wierzchołkach.

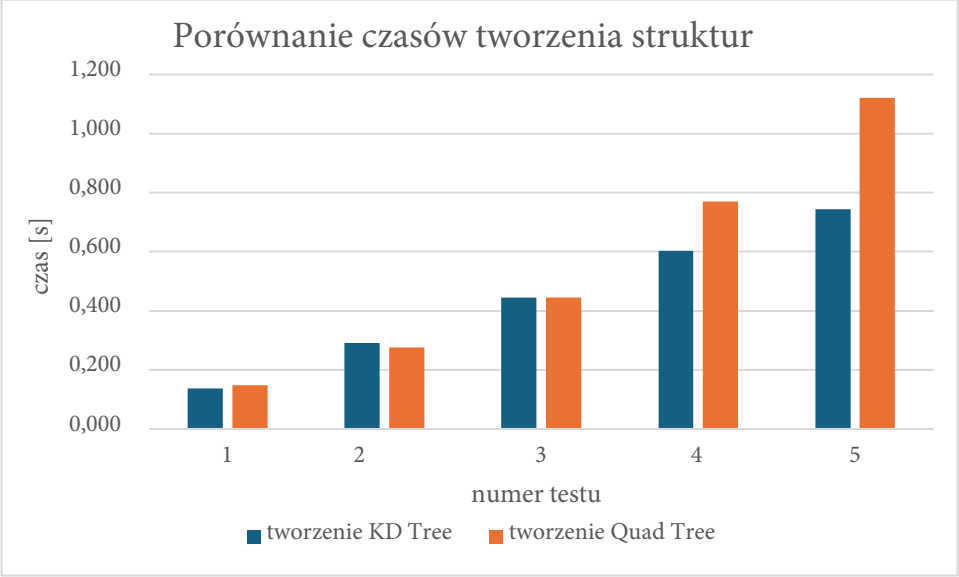
6.2 Porównanie czasów budowania struktur:

Poniżej znajduje się tabela 6.3, która zawiera porównanie czasów budowania struktur KDTree oraz QuadTree.

Numer Testu	Czas tworzenia KD Tree [s]	Czas tworzenia QuadTree [s]
1	0,1380010	0,1484120
2	0,2920639	0,2760429
3	0,4454105	0,4444131
4	0,6021929	0,7693601
5	0,7436359	1,1216311

Tabela 6.3 Porównanie czasów budowania zaimplementowanych struktur

Poniżej znajduje się wykres porównujący czasy tworzenia struktur (Rys 6.1) zrobiony na podstawie tabeli 6.3



Rys 6.1 Porównanie czasu tworzenia struktur

Możemy zauważyć, że dla pierwszych 3 testów, czasy są porównywalne, natomiast dla testów zawierających znacząco więcej punktów, testy 4-5, budowanie QuadTree zajmuje coraz więcej czasu.

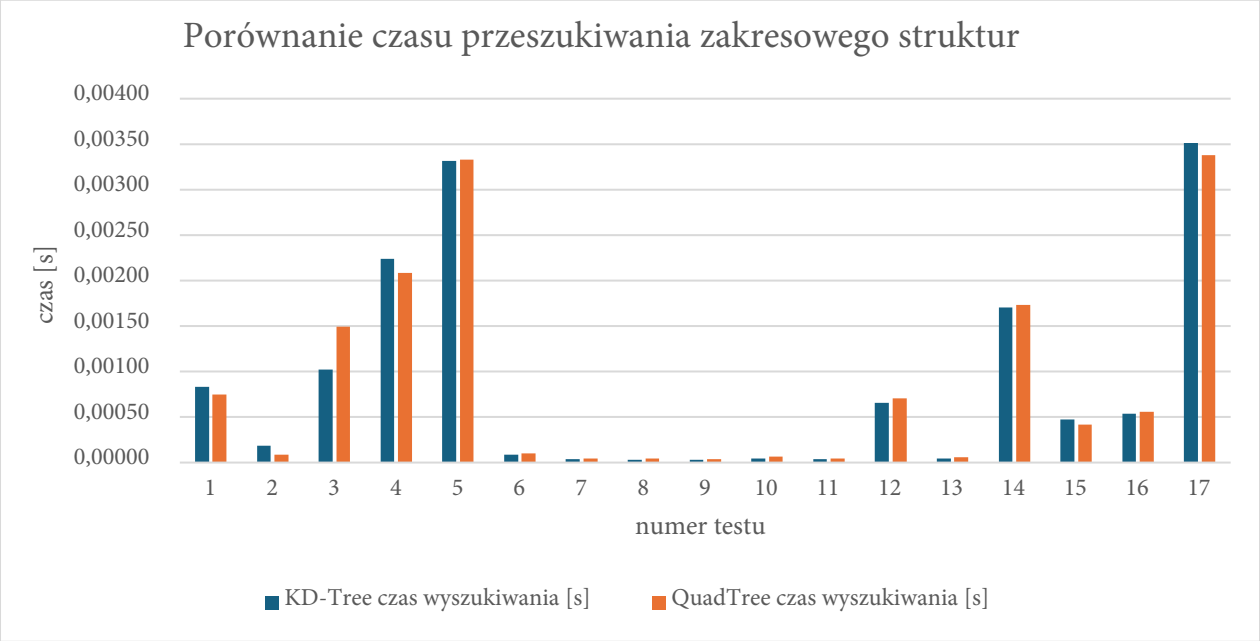
6.3 Porównanie czasów przeszukiwania struktur:

W tabeli 6.4 znajdują się porównania czasów przeszukiwania struktur KDTree i QuadTree

Numer Testu	Czas przeszukiwania KD-Tree [s]	Czas przeszukiwania QuadTree [s]
1	0,0008356	0,0007500
2	0,0001878	0,0000833
3	0,0010255	0,0014964
4	0,0022417	0,0020824
5	0,0033144	0,0033313
6	0,0000855	0,0000981
7	0,0000401	0,0000435
8	0,0000335	0,0000417
9	0,0000324	0,0000397
10	0,0000450	0,0000647
11	0,0000360	0,0000412
12	0,0006589	0,0007048
13	0,0000474	0,0000596
14	0,0017059	0,0017301
15	0,0004753	0,0004140
16	0,0005391	0,0005548
17	0,0035111	0,0033808

Tabela 6.4 Porównanie czasów przeszukiwania obszarowego zaimplementowanych struktur

Poniżej znajduje się wykres porównujący czasy przeszukiwania struktur (Rys 6.2)
zrobiony na podstawie tabeli 6.4



Rys 6.2 Porównanie czasu przeszukiwania struktur

Na podstawie wyników z tabeli 6.4 oraz rysunku 6.3 możemy zauważyć, że czasy przeszukiwania najczęściej są zbliżone dla obu struktur. Dobrze to widać w testach 6-17, które mają niewielką liczbę punktów. Dla dużej liczby punktów w testach 1-5, quadtree często radzi sobie lepiej. Możemy także zauważyć, że przeszukiwanie dużych obszarów z dużą liczbą losowo rozłożonych punktów (np. test 5), zajmuje porównywalną ilość czasu co nieduża liczba punktów rozłożona na bokach prostokąta, czyli zbiór o silnie nierównomiernie rozłożonych punktach.

6.3 Podsumowanie i wnioski:

Na podstawie wyników testów, możemy stwierdzić, że struktura KDTree szybciej buduje się dla dużych zbiorów niż struktura QuadTree. Jeśli chodzi o czas przeszukiwania, to dla większości testów, zaimplementowane algorytmy radzą sobie porównywalnie, z lekką przewagą na korzyść QuadTree. Możemy także zauważyć, że obie struktury słabo radzą sobie przy przeszukiwaniu zbiorów o nierównomiernie rozłożonych punktach takich jak w teście 17 czy 14.

BIBLIOGRAFIA

https://www.researchgate.net/figure/mplementation-of-quadtrees-within-Google-Maps-Note-the-yellow-quadrant-on-each-image_fig3_252774444
https://en.wikipedia.org/wiki/Quadtree
https://mkramarczyk.zut.edu.pl/?cat=M&l=QUAD
https://www.geeksforgeeks.org/quad-tree/
https://www.youtube.com/watch?v=UQ-1sBMV0v4
https://www.youtube.com/watch?v=OJxEcs0w_kE
Mark de Berg "Geometria Obliczeniowa - Algorytmy i Zastosowania"
https://en.wikipedia.org/wiki/K-d_tree
https://www.youtube.com/watch?v=Glp7THUpGow&pp=ygUGa2R0cmVl
https://www.baeldung.com/cs/k-d-trees
https://medium.com/@isurangawarnasooriya/exploring-kd-trees-a-comprehensive-guide-to-implementation-and-applications-in-python-3385fd56a246