AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE
AGH UNIVERSITY OF KRAKOW

Przeszukiwanie obszarów ortogonalnych

Autorzy: Dariusz Marecik, Piotr Sękulski

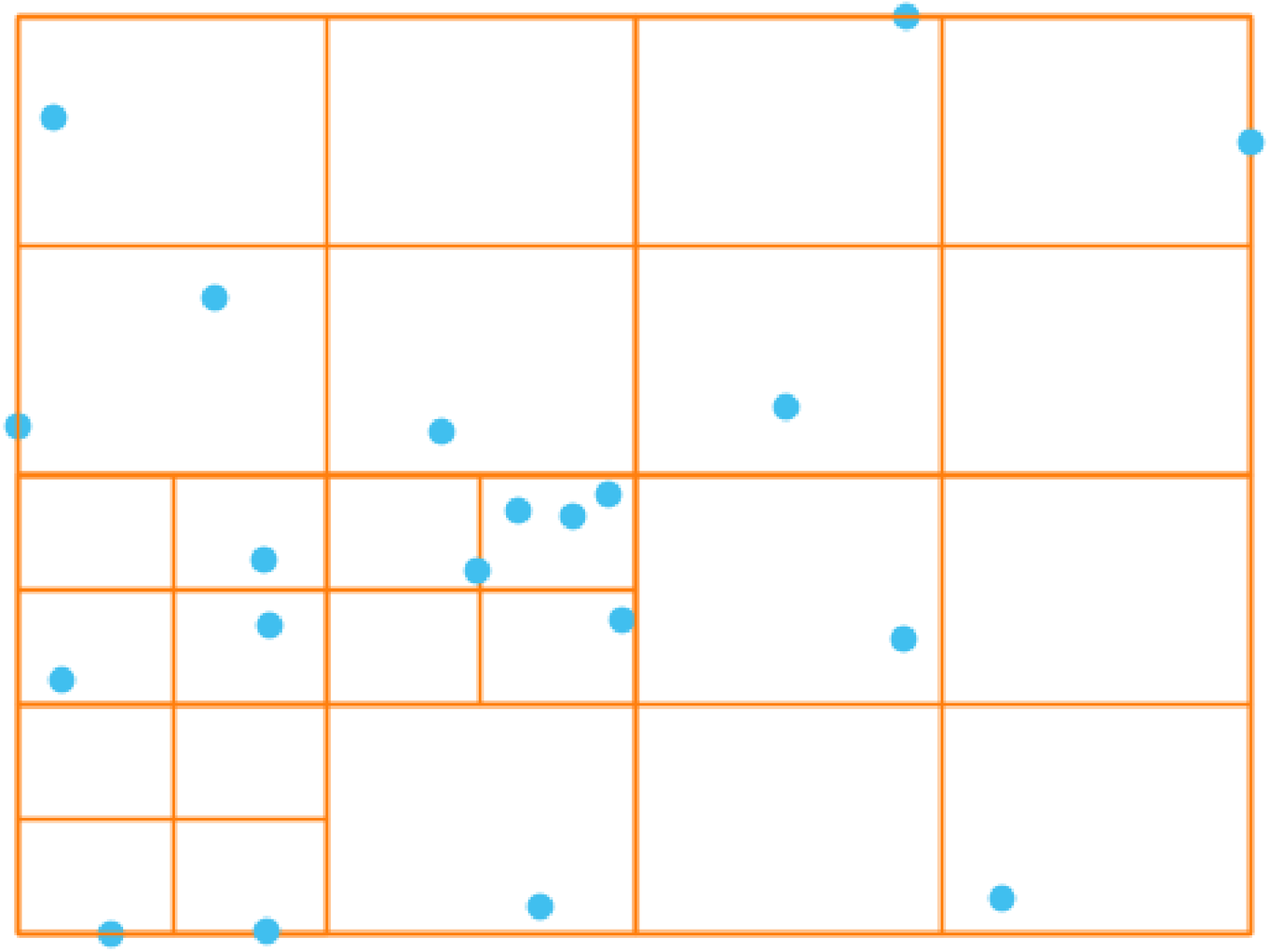
Data

Temat i cel projektu

Dane: Zbiór punktów *P* na płaszczyźnie.

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

QuadTree

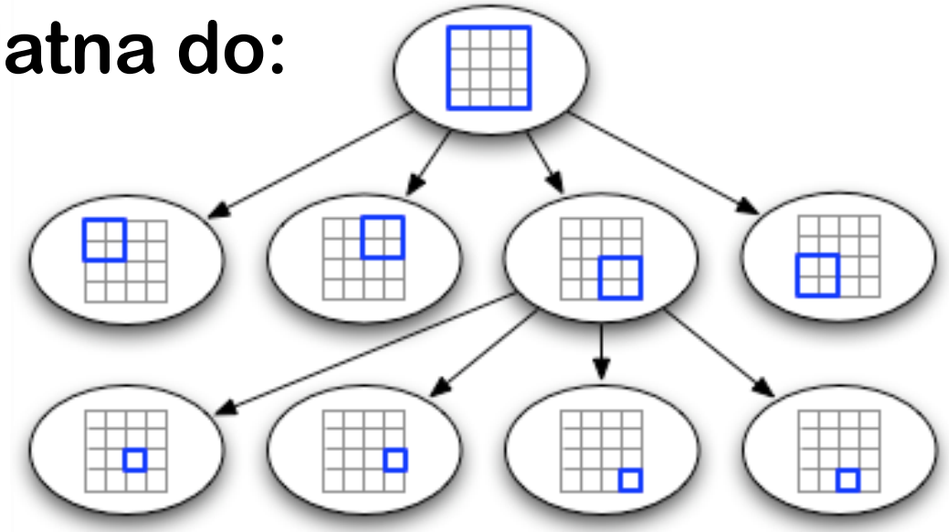


Opis

Quadtree jest to strukturą 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.

Jest to struktura szczególnie przydatna do:

- przetwarzania obrazu
- generowania siatki
- skutecznego wykrywania kolizji



Struktura QuadTree

Żeby wybudować QuadTree każdy węzeł drzewa musi posiadać pewne parametry:

- Połączenie z węzłami potencjalnych dzieci
- Maksymalną liczbę punktów, która może znajdywać się w drzewie
- Prostokąt obszaru który reprezentuje
- Punkty które znajdują się w węźle
- Aktualna głębokość drzewa
- Czy drzewo jest podzielone, to znaczy czy posiada dzieci

Struktura QuadTree - implementacja

```python
def __init__(self, rectangle, max_points = 1, depth = 0):
    self.nw = None # lewy górny czworokąt
    self.ne = None # prawy górny czworokąt
    self.sw = None # lewy donly czworokąt
    self.se = None # prawy dolny czworokąt
    self.rectangle = rectangle # czworokąt ograniczający dany węzeł
    self.max_points = max_points # maksymalna liczba punktów w węźle
    self.points = [] # punkty w węźle
    self.depth = depth # głęgokość węzła
    self.divided = False # flaga mówiąca o tym czy węzeł ma dzieci
```

Wstawianie punktu

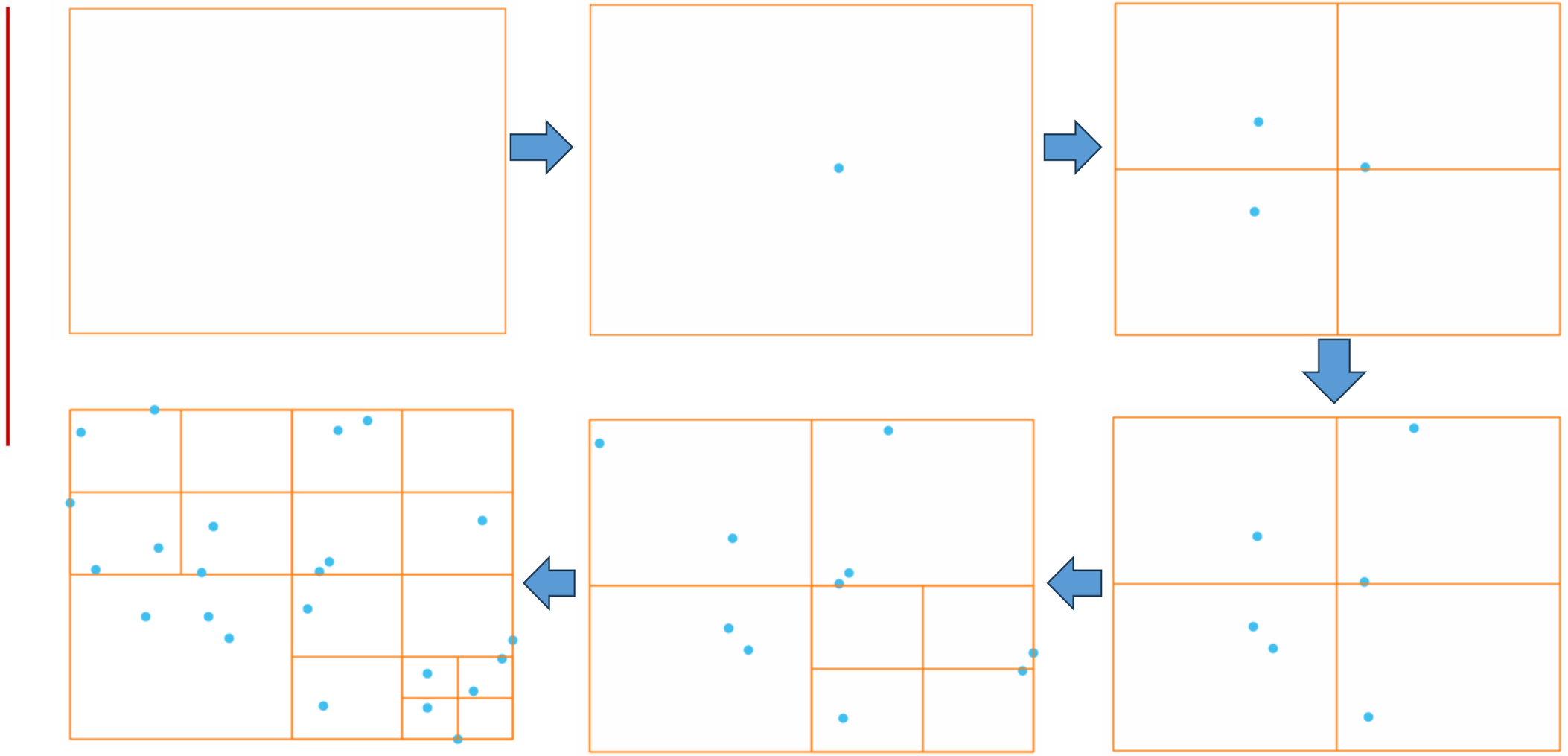
1. Sprawdzenie czy punkt zawiera się w prostokącie węzła jeżeli nie zaprzestanie sprawdzania tej części drzewa.
2. Sprawdzenie czy można wstawić punkt do aktualnego węzła. Jeśli tak to dodanie go do niego.
3. Sprawdzenie czy węzeł ma dzieci. Jeśli nie to podzielenie prostokątu drzewa na cztery mniejsze i ustawienie ich jako dzieci węzłu. Rekurencyjne wejście do dzieci węzłu i powtórzenie procedury wstawiania.
4. Dodanie punktu do struktury.

Wstawianie punktu - implementacja

Kod:

```
def insert(self, point):
    if point.amount_of_dimensions != 2:
        raise ValueError("Niepoprawny wymiar puntktów! \nQuadtree obsługuje tylko punkty dwueymiarowe!")
    if not self.rectangle.is_point_in_rectangle(point):
        return False
    if len(self.points) < self.max_points:
        self.points.append(point)
        return True

    if not self.divided:
        self.divide()

    return self.se.insert(point) or self.ne.insert(point) or self.sw.insert(point) or self.nw.insert(point)

def divide(self):
    x_1, y_1 = self.rectangle.lower_left.cords
    x_2, y_2 = self.rectangle.upper_right.cords
    c_x = (x_1 + x_2) / 2
    c_y = (y_1 + y_2) / 2
    center = Point((c_x, c_y))
    bounds = (Point((x_1, y_1)), Point((x_2, y_2)), Point((x_2, y_1)), Point((x_1, y_2)))

    self.sw = QuadTree(Rectangle(bounds[0].lower_left(center), bounds[0].upper_right(center)), elf.max_points, self.depth + 1)
    self.ne = QuadTree(Rectangle(bounds[1].lower_left(center), bounds[1].upper_right(center)), self.max_points, self.depth + 1)
    self.se = QuadTree(Rectangle(bounds[2].lower_left(center), bounds[2].upper_right(center)), self.max_points, self.depth + 1)
    self.nw = QuadTree(Rectangle(bounds[3].lower_left(center), bounds[3].upper_right(center)), self.max_points, self.depth + 1)
    self.divided = True
```

Budowa drzewa - wizualizacja



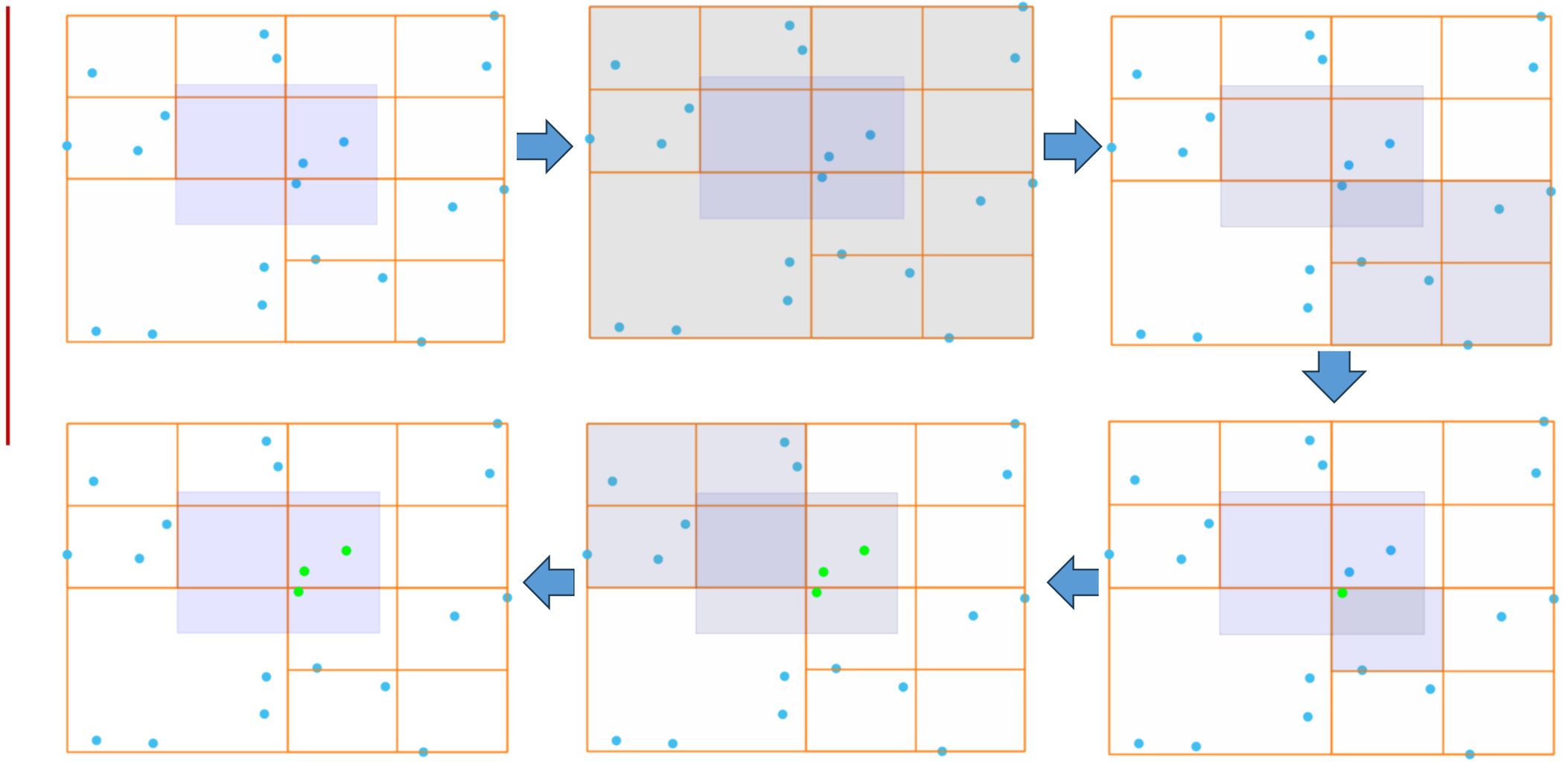
Przeszukiwanie struktury

1. Sprawdzenie czy prostokąt węzełu ma jakąś część wspólną z przeszukiwanym obszarem jeśli nie to kończymy przeszukiwanie tej części drzewa.
2. Sprawdzenie czy punkty zapisane w węźle zawierają się w szukanym prostokącie, jeśli tak to dodanie jego współrzędnych do listy znalezionych wcześniej punktów.
3. Rekurencyjne przeszukanie węzłów dzieci.
4. Zwrócenie znaleźnych do tej pory punktów.

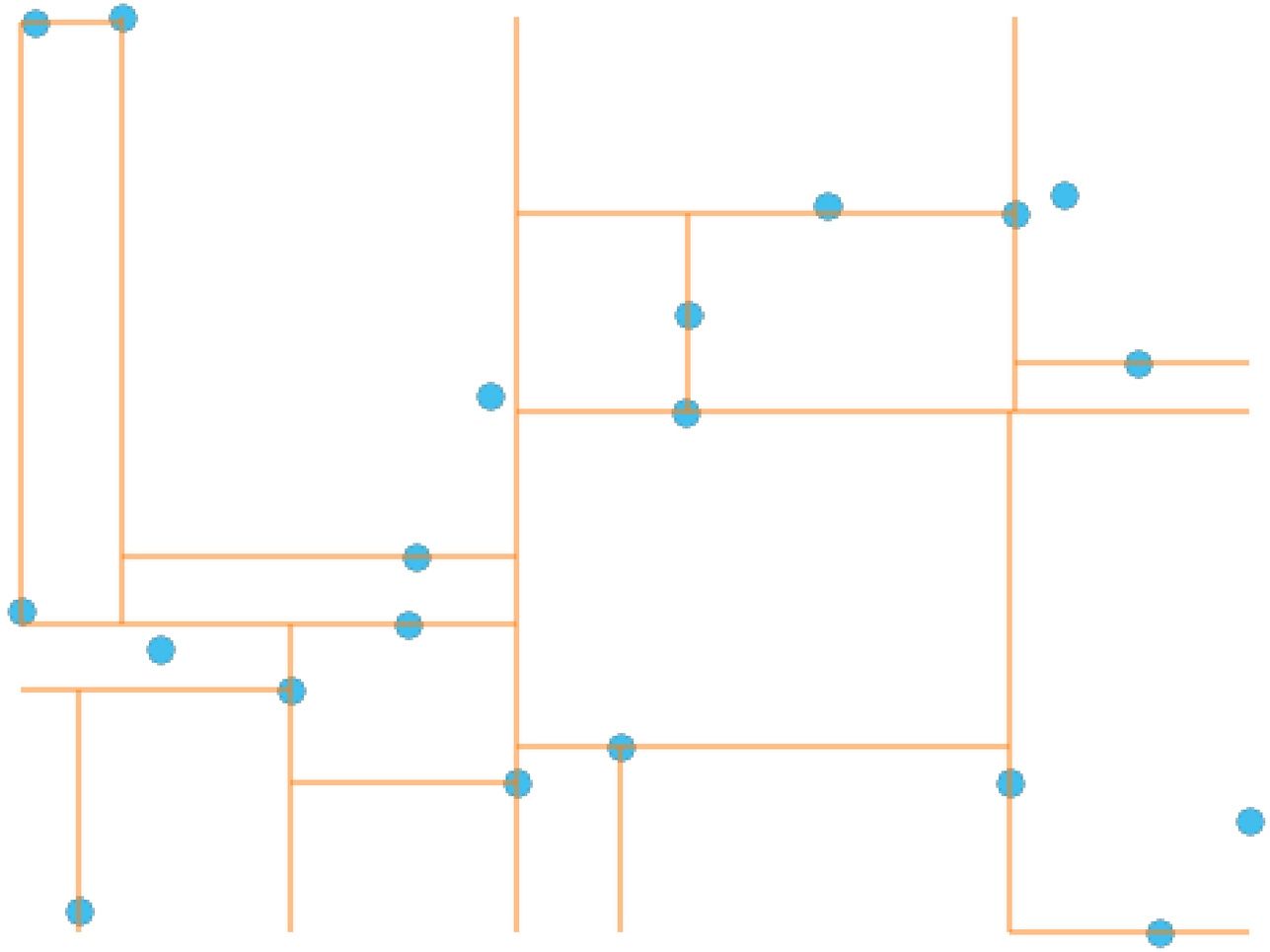
Przeszukiwanie struktury - implementacja

```
def search(self, boundary, found_points):
    if not self.rectangle.is_intersect(boundary):
        return []

    for point in self.points:
        if boundary.is_point_in_rectangle(point):
            found_points.append(point.cords)

    if self.divided:
        self.se.search(boundary, found_points)
        self.ne.search(boundary, found_points)
        self.sw.search(boundary, found_points)
        self.nw.search(boundary, found_points)
    return found_points
```

Przeszukiwanie drzewa - wizualizacja



KDTree

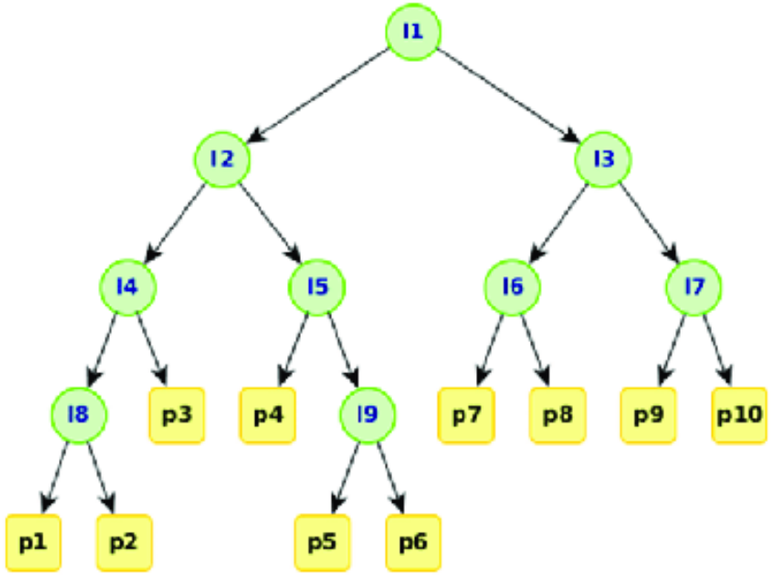


Opis

Struktura ta organizuje punkty w formie binarnego drzewa, gdzie każdy węzeł reprezentuje podział przestrzeni wzdłuż jednej z osi. Przy każdej iteracji podział odbywa się na przemian względem kolejnych wymiarów.

Najbardziej popularnym zastosowaniem drzewa jest algorytm kNN (k nearest neighbours) - podstawowy algorytm klasyfikacji uczenia maszynowego.

Także bardzo przydatnym jest wydajne filtrowanie danych. Jeśli mamy zbiór obiektów z pewnymi cechami i chcemy wybrać takie, wartości cech których mieszczą się w pewnych przedziałach, to możemy potraktować ich jak wielowymiarowe punkty.



Struktura drzewa

Inicjalizacja drzewa:

- Ustawienie początkowej osi podziału
- Ustawienie ilości wymiarów drzewa
- Dworzenie kożenia drzewa jako węzła

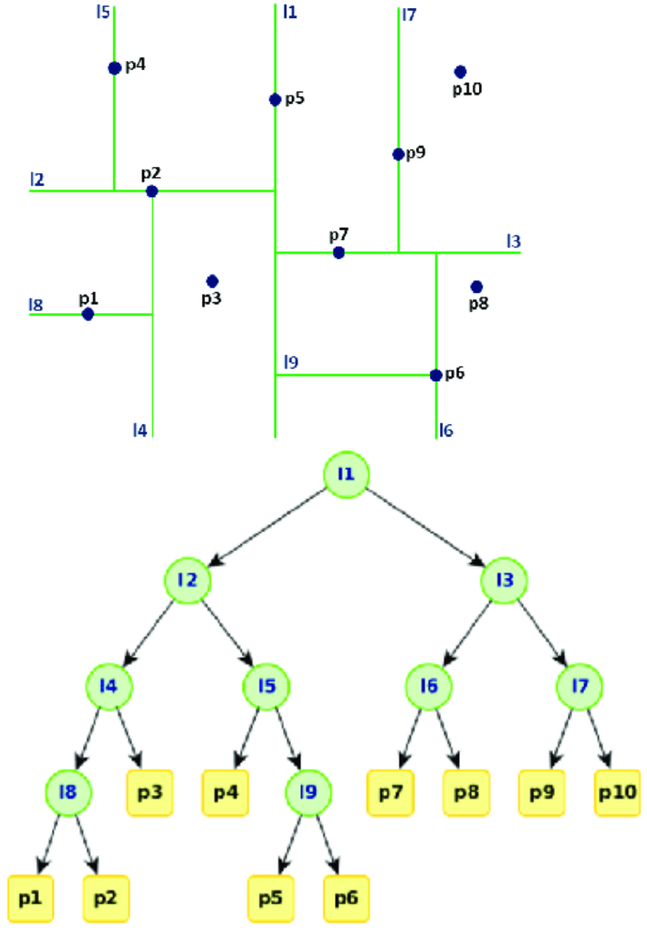
Tworzenie węzła:

- Połączenie dzieci
- Ustawienie osi podziału
- Ustawienie prostokątu obszaru który reprezentuje
- Ustawienie punktu w węźle
- Ustawienie flagi czy punkt zawiera się w węźle

Struktura drzewa - implementacja

```
class KdTree:
    def __init__(self, points, dimensions_amount, begining_axis=0, is_points_in_vertix = True):
        for point in points:
            if len(point)!= dimensions_amount:
                raise ValueError("zbiór punktów nie zgadza się z deklarowaną ilością wymiarów")
        points = [Point(point) for point in points]
        self.begining_axis = begining_axis
        self.root = KdTreeNode(points, dimensions_amount, begining_axis,
                               Rectangle(list_of_Point=points), is_points_in_vertix)
        self.dimensions_amount = dimensions_amount
class KdTreeNode:
    def __init__(self,points, dimensions_amount,depth,rectangle, is_points_in_vertix=True):
        if is_points_in_vertix or len(points)==1:
            self.points = points
        else:
            self.points = []
        self.dimensions_amount = dimensions_amount
        self.depth = depth
        self.dimension_number = self.depth%self.dimensions_amount
        self.left = None
        self.right = None
        self.is_points_in_vertix = is_points_in_vertix
        self.rectangle = rectangle
        if len(points)>1:
            self.build(points)
```

Budowa drzewa



- Ustawienie prostokątu opartego na zbiorze punktów jako korzenia
- Wykonywanie funkcji rekurencyjnej do momentu podziału wszystkich punktów na pojedyncze liście e punktów jako korzeni
- Dla każdego węzła nie będącego liściem dokonywane jest sortowanie punktów w zależności od współrzędnej występującej na poziomie węzła w drzewie. Ustawiany jest środek podziału (median) jako (liczba punktów -1)/2.
- Stawiana jest linia podziału przechodząca przez punkt z indeksem median. kierunek linii zależy od współrzędnej poziomu węzła w drzewie. Tworzone są dzieci węzła posiadające powstałe zbiory punktów

Budowa drzewa

```python
class KdTreeNode:
    def build(self,points):
        for _ in range(self.dimensions_amount):
            points.sort(key = lambda x: x.cords[self.dimension_number])
            median = math.ceil(len(points)/2)
            median-=1
            median = self.bsearch_right(points,self.dimension_number,points[median].cords[self.dimension_number])
            left_median = self.bsearch_left(points,self.dimension_number,points[median].cords[self.dimension_number])
            median+=1

            if median - left_median > 3*len(points)//4 or median == len(points):
                self.depth +=1
                self.dimension_number = (self.dimension_number+1)%self.dimensions_amount
            else:
                break

        self.axis = points[median-1].cords[self.dimension_number]
        left_rec, right_rec = self._split_region(self.rectangle, self.dimension_number, self.axis)
        self.left = KdTreeNode(points[0:median], self.dimensions_amount, self.depth+1, left_rec, self.is_points_in_vertix )
        self.right = KdTreeNode(points[median:], self.dimensions_amount, self.depth+1, right_rec, self.is_points_in_vertix )
```

Przeszukiawnie struktury

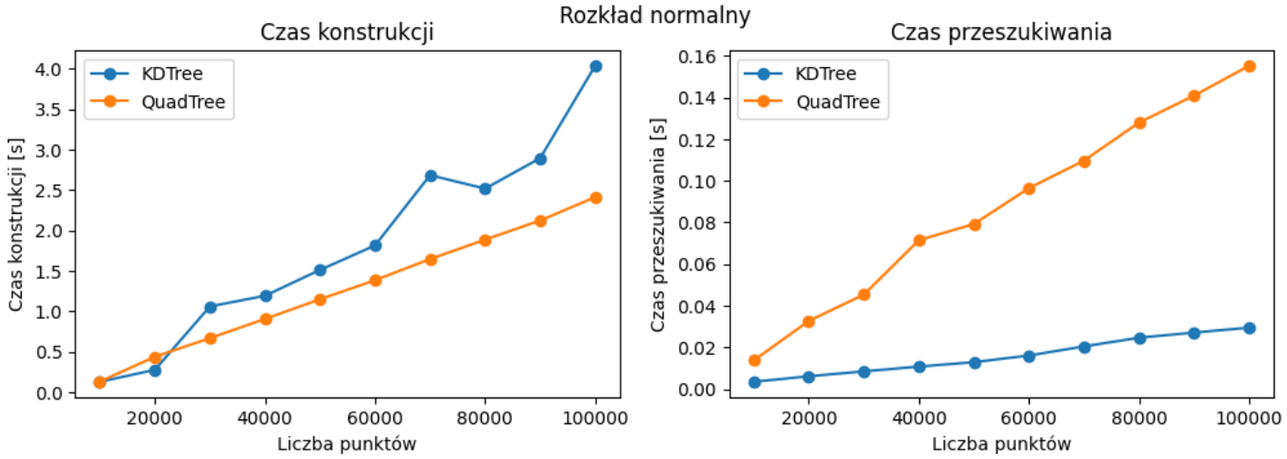
- Przeszukiwanie drzewa zostało zaimplementowane rekurencyjnie.
- Algorytm eksploruje wierzchołki wzdłuż zadanego obszaru poszukiwań.
- Jeśli prostokąt wierzchołka w całości mieści się w tym obszarze, zwracane są wszystkie punkty przechowywane w tym wierzchołku.
- W przypadku liści sprawdzana jest przynależność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

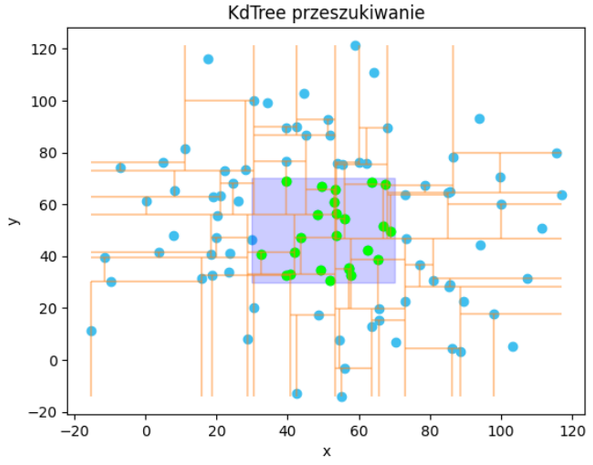
Przeszukiwanie struktury - implementacja

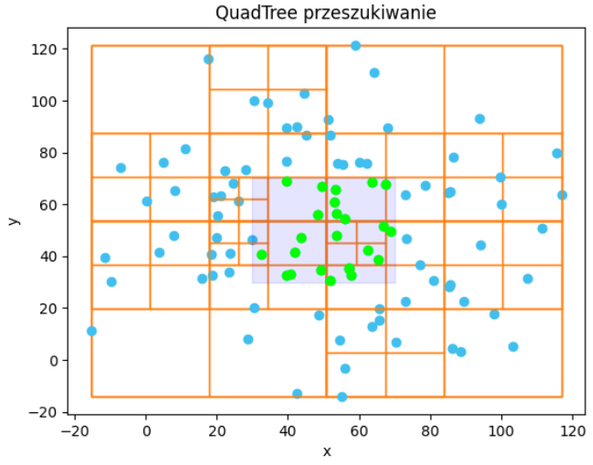
```
class KdTree:
    def check_contains(self,point):
        if not isinstance(point, Point):
            if len(point) != self.dimensions_amount:
                raise ValueError("Podano nieprawidłowy punkt do znalezienia")
            point = Point(point)
        return self.root.check_contains(point)
class KdTreeNode:
        def find_points_in_region(self,region):
        if self.is_leaf():
            is_in = region.is_point_in_rectangle(self.points[0])
            return self.points if is_in else []

        if region.is_contained(self.rectangle):
            return self.get_points()

        if region.is_intersect(self.rectangle):
            return self.left.find_points_in_region(region) + self.right.find_points_in_region(region)
        return []
```

Rozkład normalny

Jest to rozkład statystycznie najczęściej występujący w naturze, jeżeli chcemy, więc przyjrzeć się naszemu problemowi dla najbardziej powszechnych przypadków, powinniśmy wziąć pod lupę właśnie tak wygenerowany zbiór punktów. Jak wychodzi z wykresu KdTree poradziło sobie lepiej niż Quadtree.

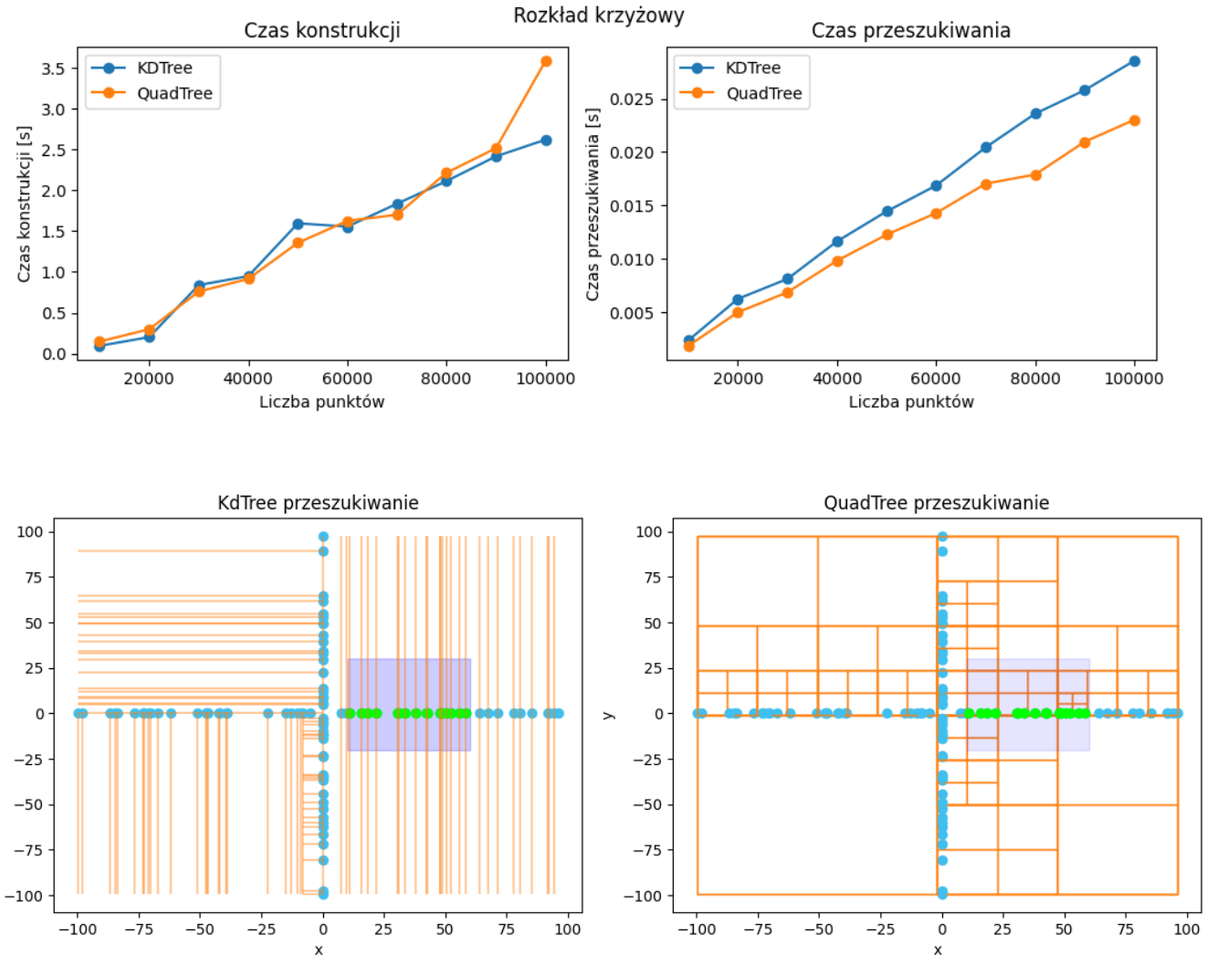






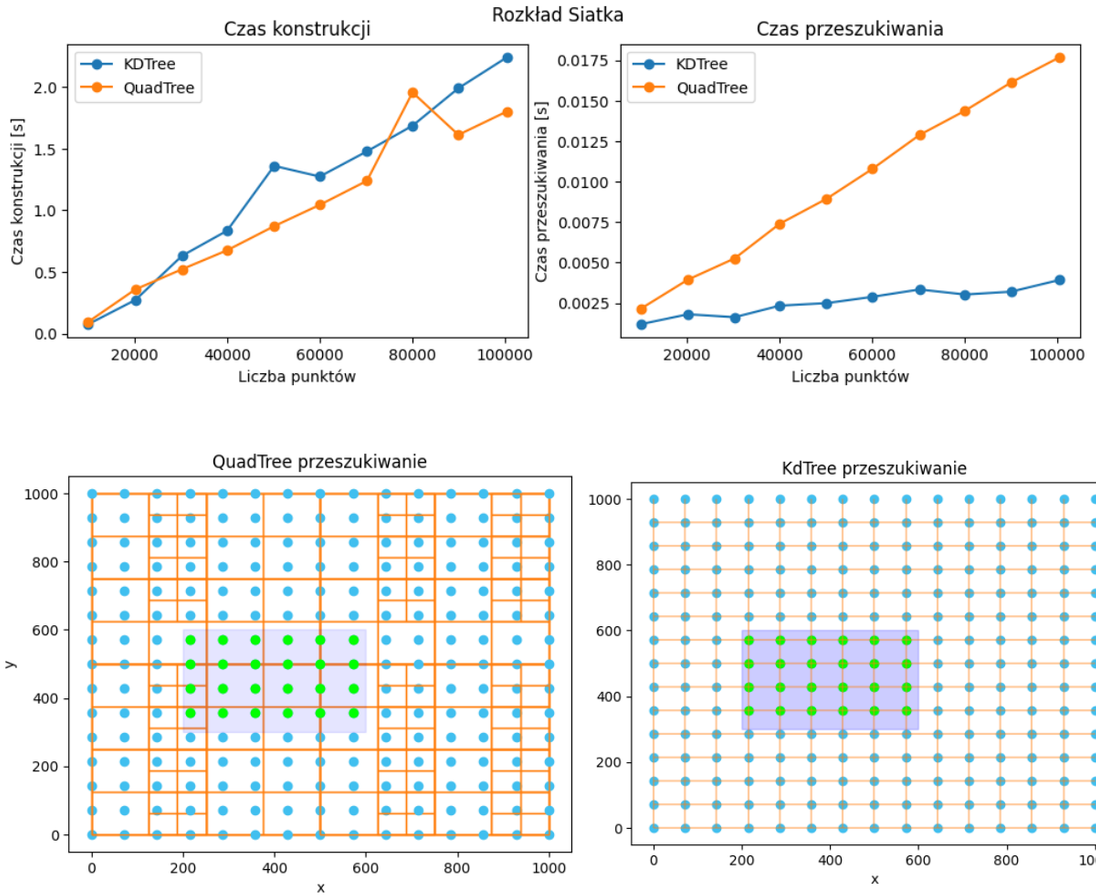
Rozkład krzyżowy

Punkty w tym zbiorze są rozmieszczone w sposób zbliżony do równomiernego na osiach układu współrzędnych, co wpływa na charakter podziału przestrzeni przez struktury danych. W przypadku takiego rozmieszczenia, duża liczba punktów współliniowych może prowadzić do nieoptymalnego dzielenia obszarów, w wyniku czego struktura danych może nieefektywnie zarządzać przestrzenią. Jak widać na wykresie QuadTree poradziło sobie trochę lepiej.



Zbiór siatka

Zbiór składa się z punktów ułożonych w równych odległościach od siebie. Takie rozmieszczenie przypomina rozkład jednostajny, lecz dokłada problem współliniowych punktów względem osi układu współrzędnych. Jak widać po wykresie lepiej sprawdza do tego zbioru drzewo kd

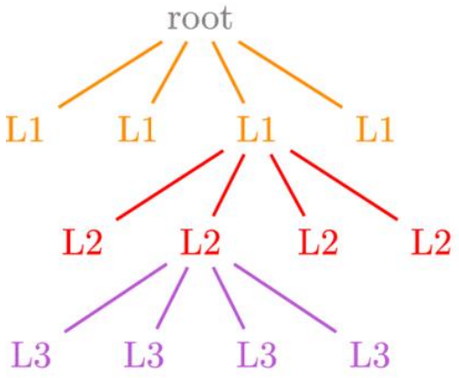


Zastosowania QuadTree

QuadTree jest strukturą używaną głównie w informatyce graficznej, przetwarzaniu obrazu, analizie przestrzennej oraz w problemach związanych z organizacją danych przestrzennych.

Przykłady zastosowania struktury:

- Kompresja obrazu
- Segmentacja obrazu
- Kliping (proces eliminacji obiektów które nie mieszczą się w określonym obszarze widoczności)
- Wyszukiwanie punktów
- Przyspieszanie zapytań
- przestrzennych
- Zarządzanie kolizjami
- Klastrowanie danych
- Renderowanie terenu

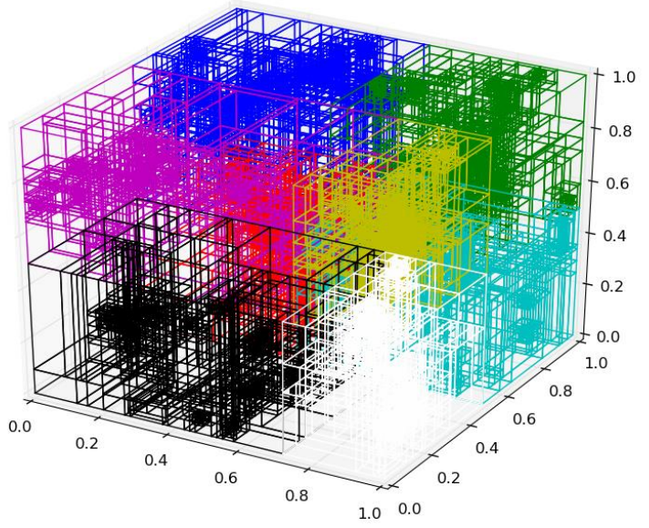


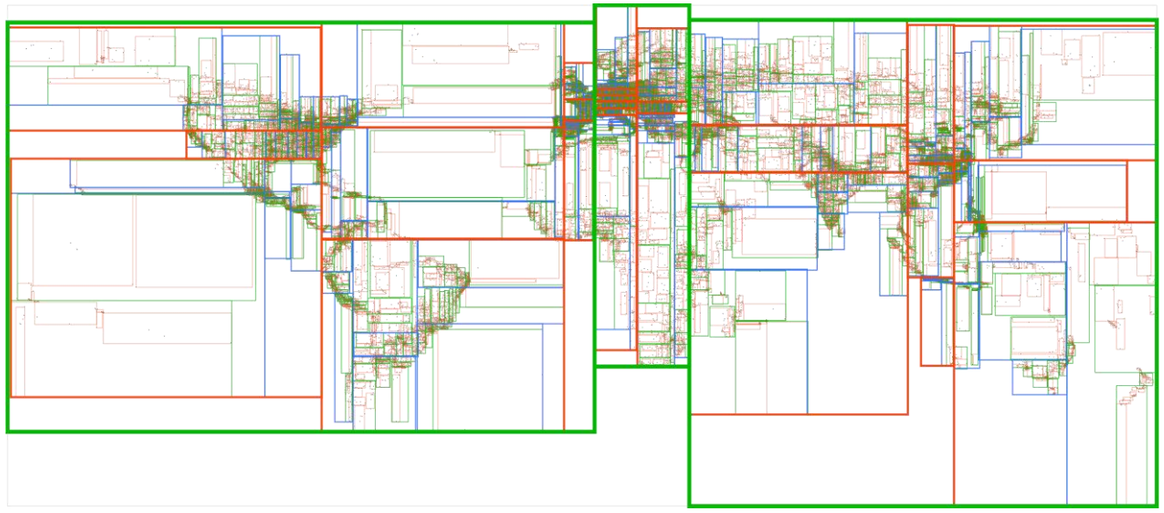
Zastosowania QuadTree

Kd-drzewa są stosowane w różnych dziedzinach, gdzie efektywne przeszukiwanie przestrzeni wielowymiarowej jest kluczowe.

Przykłady zastosowania struktury:

- Bazy danych - wyszukiwanie
- przedziałów (zakresów)
- Grafika komputerowa - renderowanie scen 3D
- Analiza obrazów – wyszukiwanie podobieństw
- Analiza danych przestrzennych - wyszukiwanie najbliższych sąsiadów
- Robotyka i nawigacja – planowanie tras (zapobieganie kolizjom)





Wady i zalety struktur

Quadtree

- Skuteczne w przypadku klastrów danych
- Prosta implementacja w dwóch wymiarach
- Proste zwiększanie liczby punktów w liściu
- Efektywne w statycznych lub mało dynamicznych strukturach
- Mało wydajne w przypadku równomiernie rozłożonych danych
- Mało wydajne w przypadku dynamicznych danych

KDTree

- Efektywne dla równomiernie rozłożonych danych
- Łatwo skalowalne na dane o większej wymiarowości
- Wydajne dla dynamicznych struktur
- Mniejsza czytelność (trudniejsze w implementacji)

Obie struktury zapewniają szybkie przeszukiwanie przestrzeni, jednak są wrażliwe na strukturę danych

Dziękujemy za uwagę

Dariusz Marecik, Piotr Sękulski

Bibliografia

- https://github.com/aghbit/Algorytmy-Geometryczne
- https://en.wikipedia.org/wiki/Quadtree
- https://en.wikipedia.org/wiki/K-d_tree
- https://github.com/FloudMe77/QuadTree_and_KdTree
- https://www.agh.edu.pl/o-agh/multimedia/szablony-prezentacji
- Grafiki DuckDuckGo
- Grafiki Google