Wyszukiwanie geometryczne - algorytmy Quadtree oraz Kd_tree

02.01.2025

Jan Martowicz, Wiktor Onik

Plan prezentacji

Przedstawienie problemu

Przedstawienie problemu

Dany jest zbiór punktów P na płaszczyźnie. Chcemy odpowiedzieć na pytanie: „Które punkty znajdują się wewnątrz określonego prostokąta?”,
czyli dla zadanych x_1, x_2, y_1, y_2 znaleźć $x_p, y_p \in P$ takie, że

$$x_1 \leq x_p \leq x_2, y_1 \leq y_p \leq y_2$$

Najprostsze podejście

Rozwiązaniem trywialnym jest sprawdzenie wszystkich punktów Złożoność czasowa: $O(n)$

```
def points_in_rectangle(points, rectangle):

    x1, y1 = rectangle[0]
    x2, y2 = rectangle[1]

    return [
        (x, y) for x, y in points
        if x1 <= x <= x2 and y1 <= y <= y2
    ]
```

Algorytm Quadtree

Opis struktury algorytmu

Quadtree (inaczej drzewo czwórkowe) jest drzewiastą strukturą danych gdzie:

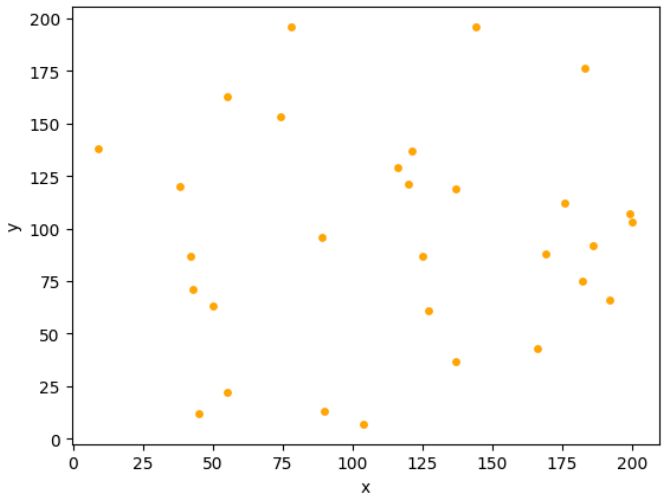
1. Każdy wierzchołek drzewa odpowiada prostokątowi na płaszczyźnie.
2. Każdy wierzchołek posiada dokładnie czworo dzieci, lub nie posiada ich wcale
3. Liść w drzewie odpowiada pojedynczemu punktowi na płaszczyźnie.
4. Można określić ile dzieci będzie miał ostatni wierzchołek który nie jest liściem (capacity)

Quadtree - implementacja

```
class Quadtree:
    def __init__(self, points: list[Point], max_points_per_node: int = 4):
        self.max_points_per_node = max_points_per_node
        self.max_rectangle = RectangleArea(
            min(points, key=lambda p: p.x).x,  # Minimalna wartość x
            min(points, key=lambda p: p.y).y,  # Minimalna wartość y
            max(points, key=lambda p: p.x).x,  # Maksymalna wartość x
            max(points, key=lambda p: p.y).y,  # Maksymalna wartość y
        )
        self.root = self.build_tree(self.max_rectangle, points)

class QuadtreeNode:
    def __init__(self, rectangle: RectangleArea) -> None:
        self.rectangle = rectangle  # Całkowity obszar tego węzła
        self.points = []  # Punkty w tym węźle
        self.upper_left = None  # Lewy górny kwadrant
        self.upper_right = None  # Prawy górny kwadrant
        self.lower_left = None  # Lewy dolny kwadrant
        self.lower_right = None  # Prawy dolny kwadrant
        self.is_leaf = True  # Czy jest liściem (czy ma dzieci)
```

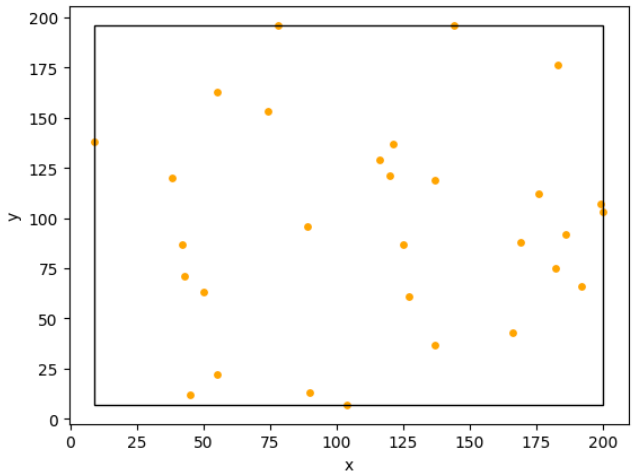
Quadtree - budowa drzewa

Mamy początkowo zbiór punktów:



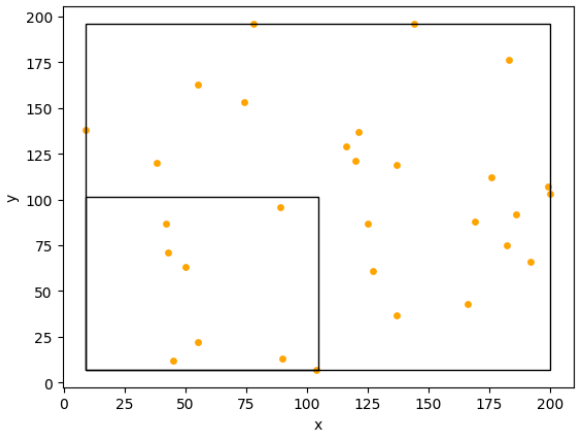
Quadtree - budowa drzewa (ii)

Wyznaczamy najmniejszy prostokąt obejmujący wszystkie punkty - będzie on korzeniem drzewa.

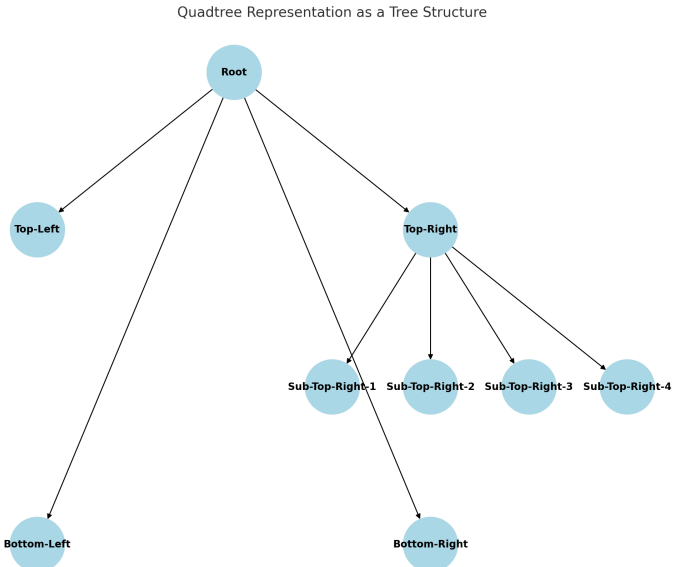


Quadtree - budowa drzewa (iii)

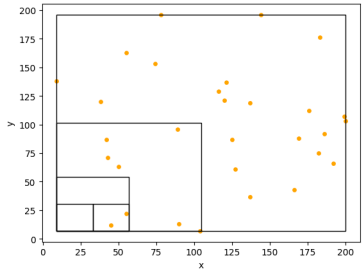
Teraz prostokąt dzielimy na 4 takie same części - będą to dzieci aktualnego prostokąta. Tworzymy 4 poddrzewa do momentu kiedy punkty w każdym prostokącie $\leq$ capacity.

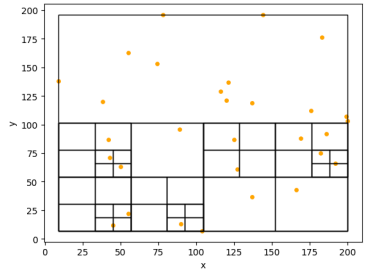


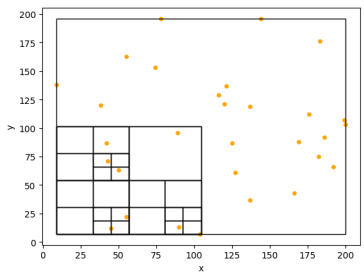
Quadtree - budowa drzewa (iv)

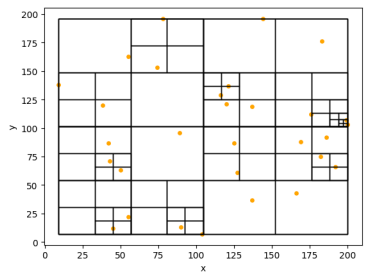


Quadtree - budowa drzewa (v)



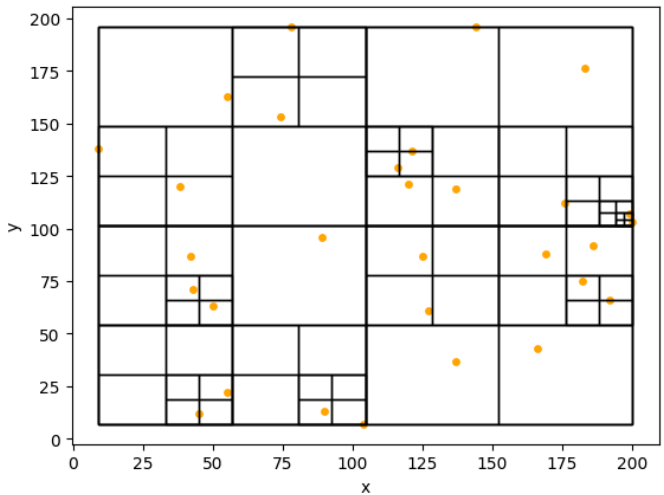






Quadtree - budowa drzewa (vi)

Ostatecznie tak wygląda zbudowane drzewo:



Quadtree - znajdowanie punktów

Znajdowanie punktów działa w następujący sposób:

1. przeszukujemy drzewo, dla każdego niebędącego liściem wierzchołka sprawdzając, czy odpowiadający mu prostokąt ma część wspólną z prostokątem z zapytania.
2. Jeśli tak, to schodzimy wgłąb tego poddrzewa
3. Gdy dojdziemy do liścia, sprawdzamy czy odpowiadający mu punkt należy do szukanego prostokąta.

Implementacja znajdowania punktów

```
def find_recursion(
    self, node: QuadtreeNode, rectangle: RectangleArea
) -> list[Point]:
    res = []
    if (node.rectangle & rectangle is None):  # Jeśli prostokąty nie mają wspólnego obszaru
        return res

    # Jeśli to liść, sprawdzamy punkty
    if node.is_leaf:
        res.extend([p for p in node.points if p in rectangle])
    else:
        # Rekurencyjnie sprawdzamy dzieci (cztery ćwiartki)
        res.extend(self.find_recursion(node.lower_left, rectangle))
        res.extend(self.find_recursion(node.lower_right, rectangle))
        res.extend(self.find_recursion(node.upper_left, rectangle))
        res.extend(self.find_recursion(node.upper_right, rectangle))

    return res

def find(self, rectangle: RectangleArea) -> list[Point]:
    return self.find_recursion(self.root, rectangle)
```

Quadtree - złożoności

n - liczba punktów w zbiorze

h - wysokość drzewa

Złożoność konstrukcji $\longrightarrow O(h \cdot n)$ W najgorszym wypadku $h = n$, wtedy gdy punkty są bardzo skoncentrowane i dla każdego punktu trzeba budować nowy poziom.

Jeśli punkty są rozłożone równomiernie $h = \log(n)$

Złożoność zapytania $\longrightarrow O(h \cdot k)$, gdzie k to liczba punktów zwróconych w zapytaniu

Złożoność pamięciowa $\longrightarrow O(h \cdot n)$

Algorytm Kd-Tree

Opis struktury algorytmu

KD-drzewo (ang. kd-tree) to specjalny rodzaj drzewa binarnego używanego do organizowania punktów w przestrzeni wielowymiarowej. Jego struktura opiera się na następujących zasadach:

1. Podział według wymiaru: Każdy poziom drzewa odpowiada za inny wymiar współrzędnych (np. naprzemiennie oś X i oś Y). Podział przestrzeni następuje względem tego wymiaru.
2. Wartość w wierzchołku: Wierzchołek drzewa przechowuje współrzędną, względem której podzielono zbiór punktów na danym etapie.
3. Podział na podstawie mediany: Punkt podziału wybierany jest na podstawie mediany wartości współrzędnej w danym wymiarze, co minimalizuje wysokość drzewa.

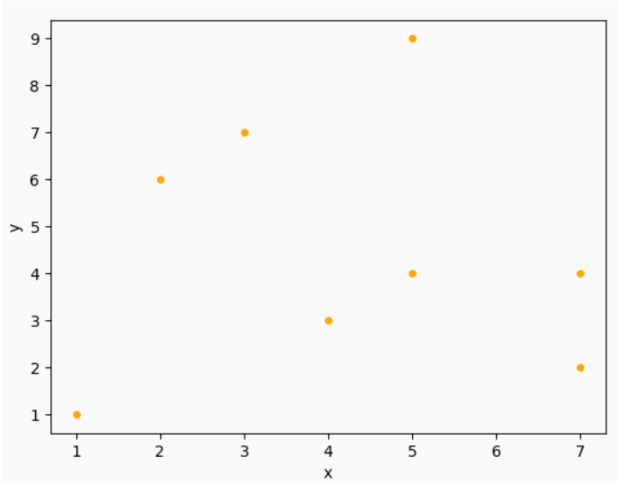
Opis struktury algorytmu (ii)

4. Przydzielanie punktów do potomków:
   - Punkty o współrzędnej mniejszej od wartości w wierzchołku trafiają do lewego poddrzewa.
   - Punkty o współrzędnej większej trafiają do prawego poddrzewa.
   - Punkty równe medianie są przypisywane naprzemiennie do jednego z poddrzew (w celu zrównoważenia drzewa).
5. Liście drzewa: Każdy liść drzewa KD odpowiada dokładnie jednemu punktowi z przestrzeni, bez dalszego podziału.

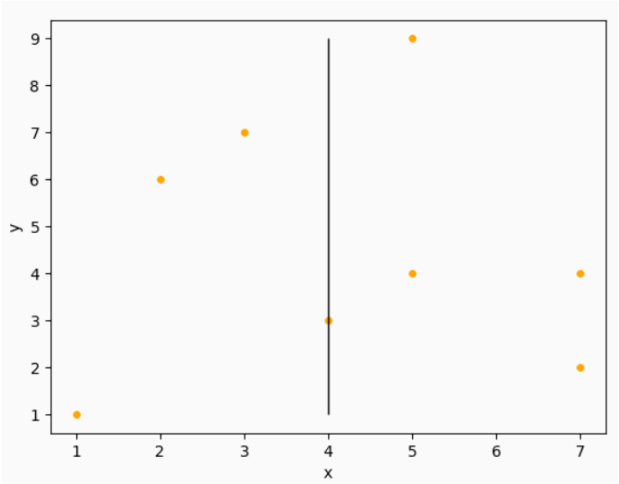
Kd-Tree - implementacja

```
class KdTreeNode:
    def __init__(self, axis: int | None, rectangle: RectangleArea) -> None:

        self.axis = axis  # określa którą oś rozpatrujemy tj. dla K=2 czy według osi X czy Y/
        self.rectangle = rectangle  # obszar który jesr reprezentowany przez poddtrzewo tego wierzchołka
        self.left_node = None  # lewe dziecko
        self.right_node = None  # prawe dziecko
        self.leafs_list = []  # list liści w poddrzewie
        self.leaf_point = None  # jeśli node jest liście to znajduje się tu punkt


class KdTree:
    def __init__(self, points: list[Point]):

        self.points = points
        self.max_rectangle = RectangleArea(
            min(points, key=lambda p: p.x).x,  # Minimalna wartość x
            min(points, key=lambda p: p.y).y,  # Minimalna wartość y
            max(points, key=lambda p: p.x).x,  # Maksymalna wartość x
            max(points, key=lambda p: p.y).y,  # Maksymalna wartość y
        )
        self.root = self.build_tree(self.points, 0, self.max_rectangle)
```

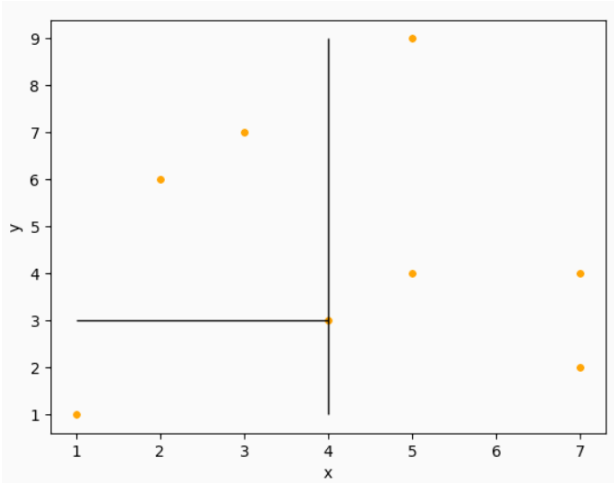
Kd-Tree - budowa drzewa



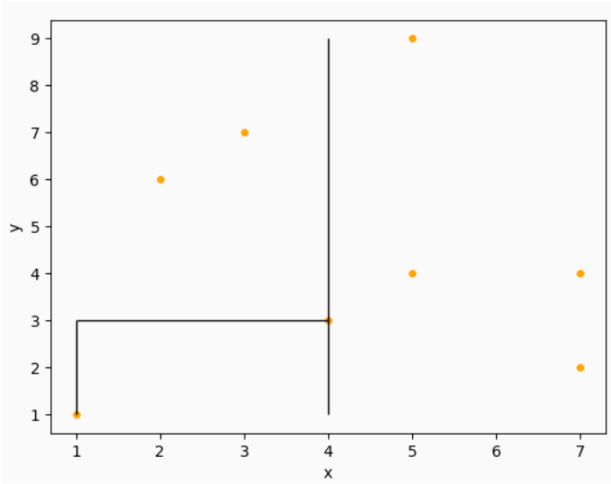
Kd-Tree - budowa drzewa (ii)



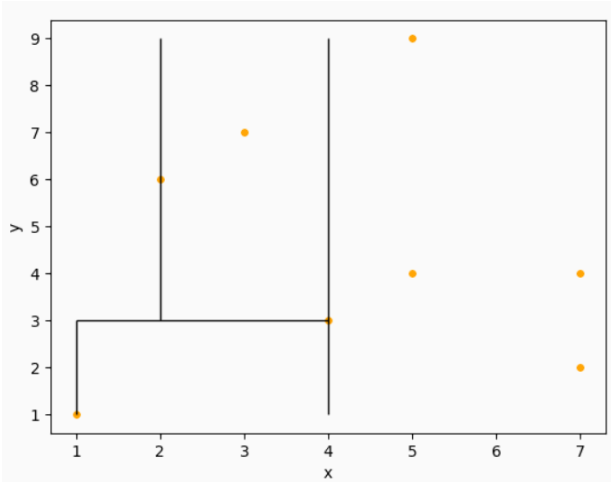
Kd-Tree - budowa drzewa (iii)



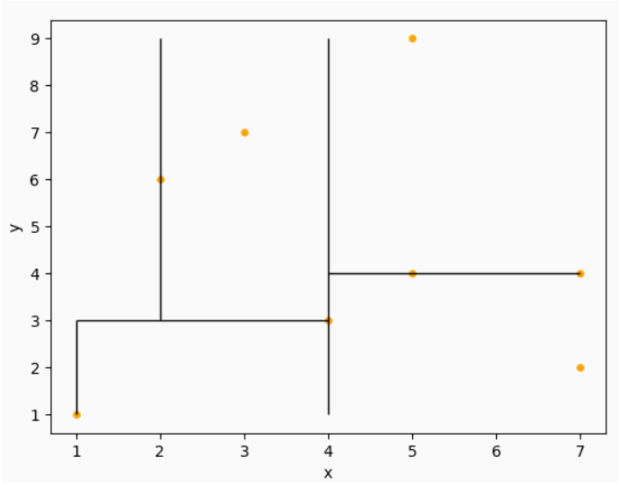
Kd-Tree - budowa drzewa (iv)



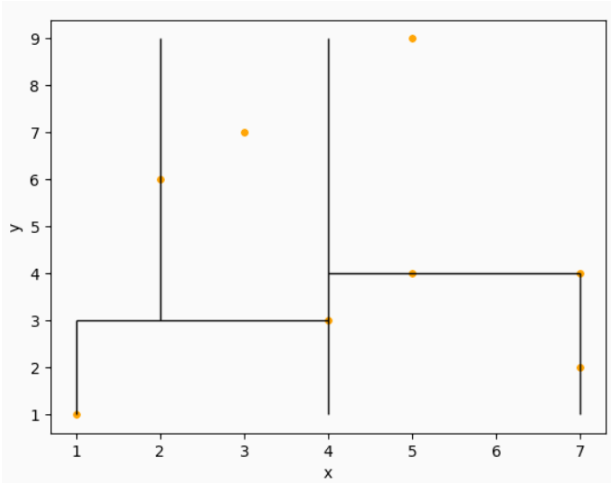
Kd-Tree - budowa drzewa (v)



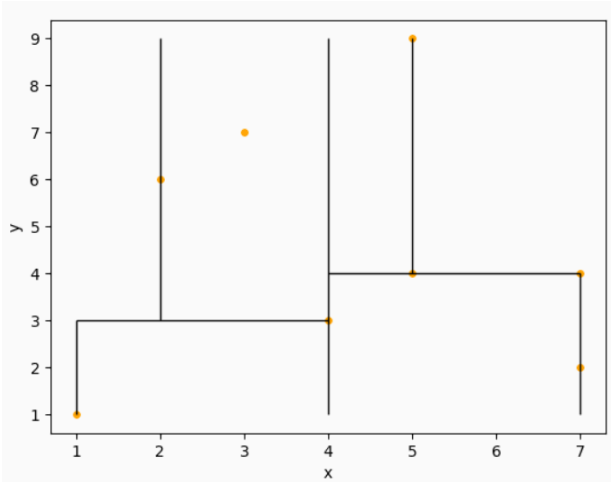
Kd-Tree - budowa drzewa (vi)



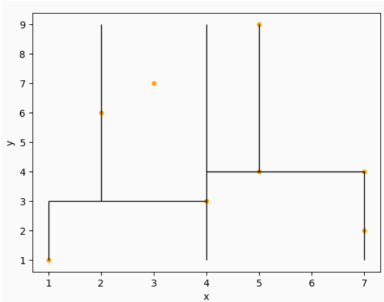
Kd-Tree - budowa drzewa (vii)

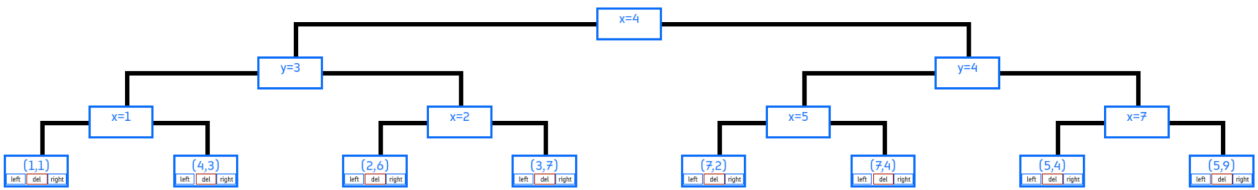


Kd-Tree - budowa drzewa (viii)



Kd-Tree - budowa drzewa (ix)





Kd-Tree – odpowiadanie na zapytania

Każdy wierzchołek (poza liśćmi) reprezentuje określony obszar w przestrzeni, ograniczony przez podziały wynikające z wyższych poziomów drzewa.

Znajdowanie punktów, które należą do zadanego obszaru działa następująco:

1. Algorytm rozpoczyna przeszukiwanie drzewa od korzenia. Jeśli obszar przypisany do wierzchołka w pełni zawiera się w zadanym obszarze, wszystkie punkty (liście) tego wierzchołka są dodawane do wyniku bez dalszego przeszukiwania poddrzew.
2. Jeżeli obszar wierzchołka tylko częściowo pokrywa się z zadanym obszarem, konieczne jest dalsze przeszukiwanie obu jego dzieci (lewego i prawego poddrzewa).
3. Jeśli obszar wierzchołka nie pokrywa się w ogóle z zadanym obszarem, dalsze przeszukiwanie tego poddrzewa zostaje zakończone, a węzeł zostaje pominięty.

Implementacja znajdowania punktów

```
def find_recursive(self, node: KdTreeNode, rectangle: RectangleArea, res: list[Point]):

      if (rectangle & node.rectangle is None):  # szukany obaszar jest poza obecnym obszarem
          return

      if node.leaf_point is not None and node.leaf_point in rectangle:
          res.append(node.leaf_point)  # node jest liściem i jest w obszarze więc dodajemy
          return
      for leaf_node in node.leafs_list:  # wchodzimy głębiej
          self.find_recursive(leaf_node, rectangle, res)

def find(self, rectangle: RectangleArea) -> list[Point]:
      res = []
      self.find_recursive(self.root, rectangle, res)
      return res
```

Kd-Tree – złożoności

n – liczba punktów w zbiorze

konstrukcji drzewa: $O(n \log n)$

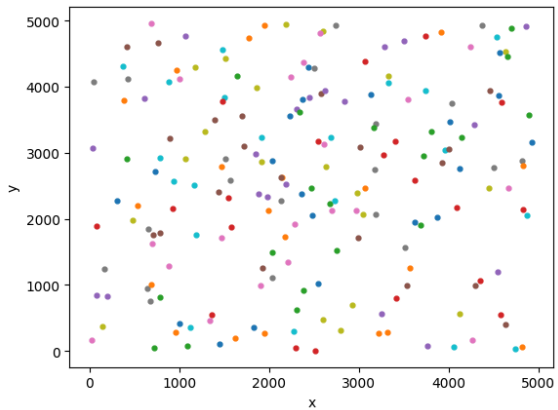
Dla zrównoważonego drzewa:

- złożoność czasowa zliczania: $O(\sqrt{n})$
- złożoność czasowa wyszukiwania: $O(\sqrt{n} + k)$
- złożoność pamięciowa $O(n)$

Porównanie algorytmów

Zbiór A

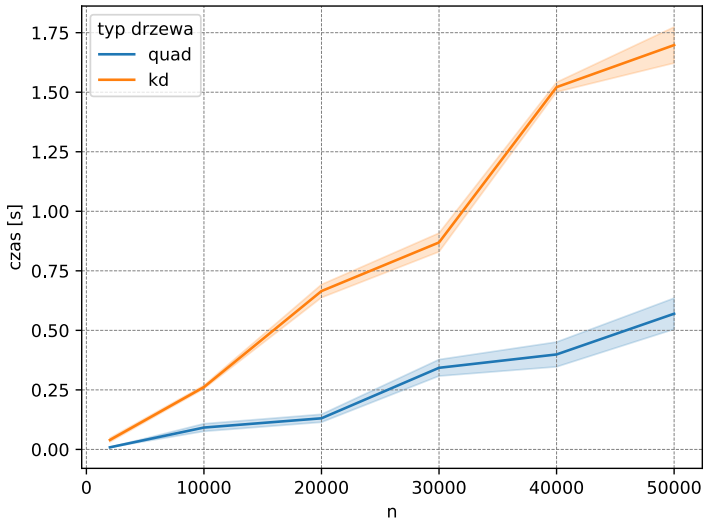
$A(n)$ - zbiór n jednostajnie rozłożonych punktów będących w zakresie $[0, 5000]^2$ wygenerowany przy pomocy funkcji `numpy.random.uniform`



Zbiór A - wyniki czasu konstrukcji

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0088	0.0397
10000	0.0919	0.2618
20000	0.1307	0.6650
30000	0.3428	0.8691
40000	0.3988	1.5213
50000	0.5698	1.6980

Zbiór A - wykres czasu konstrukcji

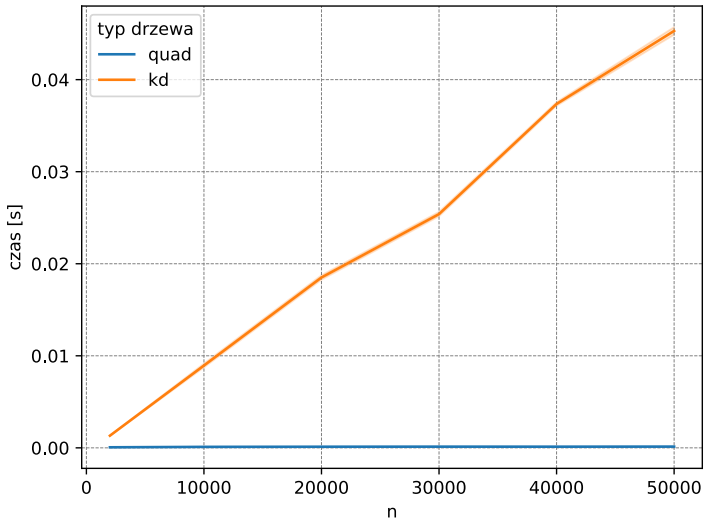


Zbiór A - znajdowanie punktów

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0001	0.0013
10000	0.0001	0.0089
20000	0.0001	0.0185
30000	0.0001	0.0254
40000	0.0001	0.0374
50000	0.0001	0.0453

Tabela 2: Czasy odpowiedzi na zapytania o mały obszar dla zbioru A

Zbiór A - znajdowanie punktów (ii)



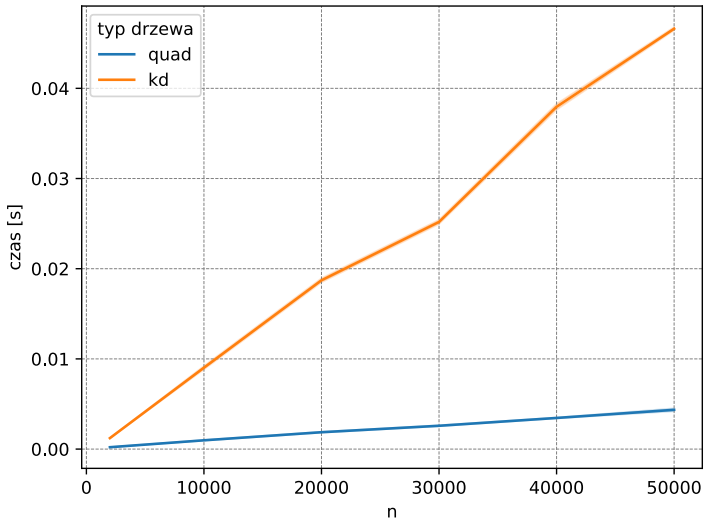
Rysunek 1: Wizualizacja czasu odpowiedzi na zapytania o mały obszar dla zbioru A

Zbiór A - znajdowanie punktów (iii)

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0002	0.0012
10000	0.0010	0.0090
20000	0.0019	0.0187
30000	0.0026	0.0252
40000	0.0035	0.0380
50000	0.0044	0.0466

Tabela 3: Czasy odpowiedzi na zapytania o duży obszar dla zbioru A

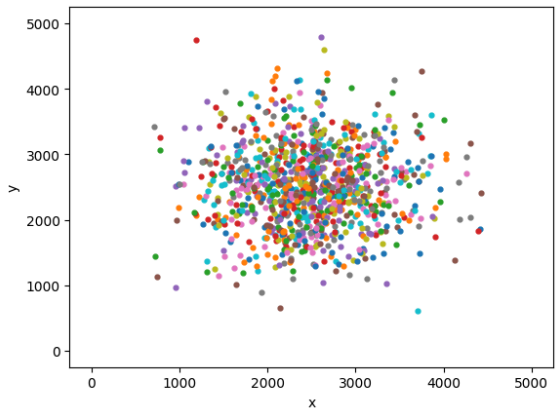
Zbiór A - znajdowanie punktów (iv)



Rysunek 2: Wizualizacja czasu odpowiedzi na zapytania o duży obszar dla zbioru A

Zbiór B

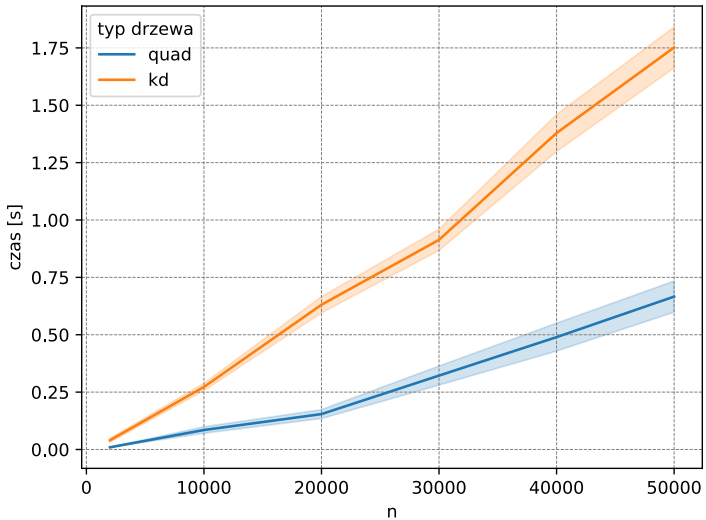
- $B(n)$ - zbiór n punktów o rozkładzie normalnym będących w zakresie $[0, 5000]^2$ wygenerowany przy pomocy funkcji `numpy.random.normal`



Zbiór B - wyniki czasu konstrukcji

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0094	0.0400
10000	0.0849	0.2727
20000	0.1542	0.6307
30000	0.3220	0.9136
40000	0.4896	1.3791
50000	0.6662	1.7517

Zbiór B - wykres czasu konstrukcji

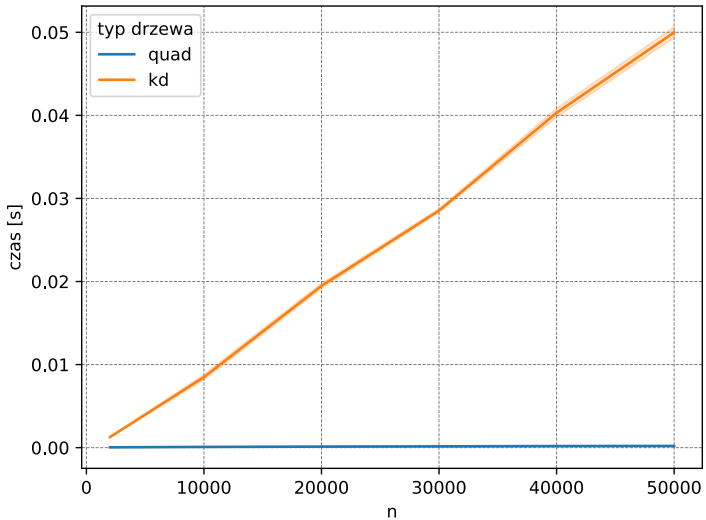


Zbiór B - znajdowanie punktów

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0000	0.0013
10000	0.0001	0.0085
20000	0.0001	0.0195
30000	0.0002	0.0285
40000	0.0002	0.0403
50000	0.0002	0.0500

Tabela 5: Czas odpowiedzi na zapytania o mały obszar dla zbioru B

Zbiór B - znajdowanie punktów (ii)



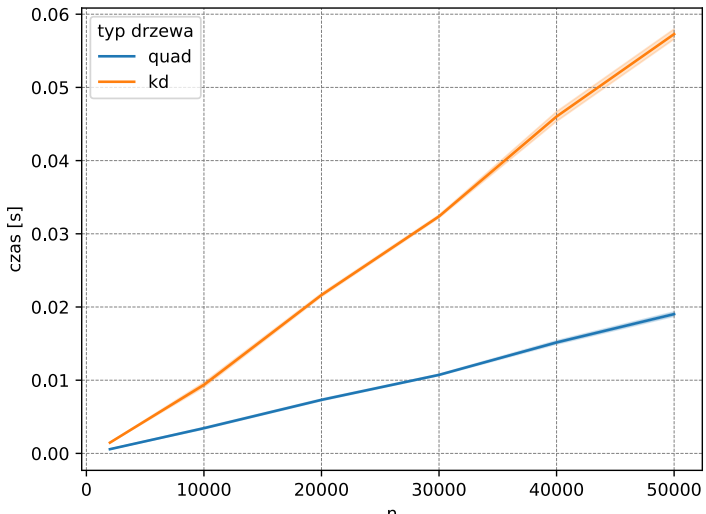
Rysunek 3: Wizualizacja czasu odpowiedzi na zapytania o mały obszar dla zbioru B

Zbiór B - znajdowanie punktów (iii)

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0006	0.0015
10000	0.0034	0.0094
20000	0.0073	0.0216
30000	0.0107	0.0324
40000	0.0152	0.0460
50000	0.0190	0.0573

Tabela 6: Czas odpowiedzi na zapytania o duży obszar dla zbioru B

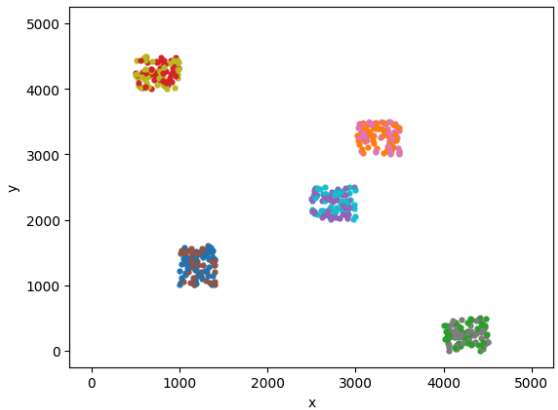
Zbiór B - znajdowanie punktów (iv)



Rysunek 4: Wizualizacja czasu odpowiedzi na zapytania o duży obszar dla zbioru B

Zbiór C

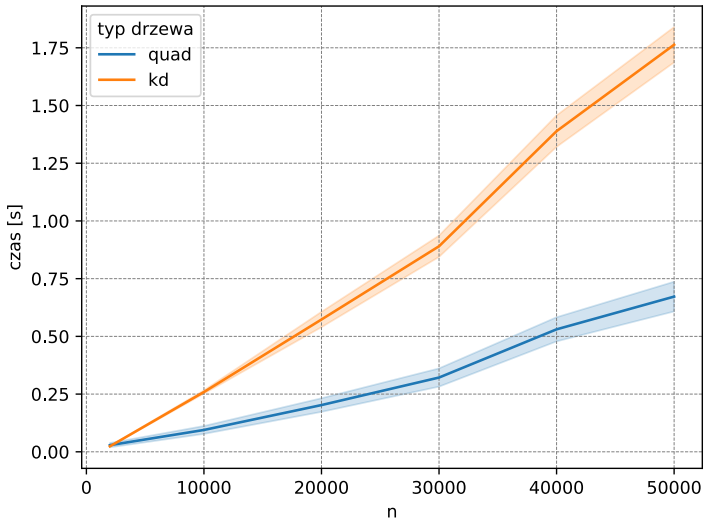
- $C(n)$ - zawiera 5 klastrów, każdy o rozkładzie jednostajnym z n/5 punktów będących w zakresie $[0, 5000]^2$ wygenerowany przy pomocy funkcji `numpy.random.uniform`



Zbiór C - wyniki czasu konstrukcji

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0288	0.0238
10000	0.0946	0.2590
20000	0.2025	0.5731
30000	0.3217	0.8903
40000	0.5305	1.3889
50000	0.6723	1.7632

Zbiór C - wykres czasu konstrukcji

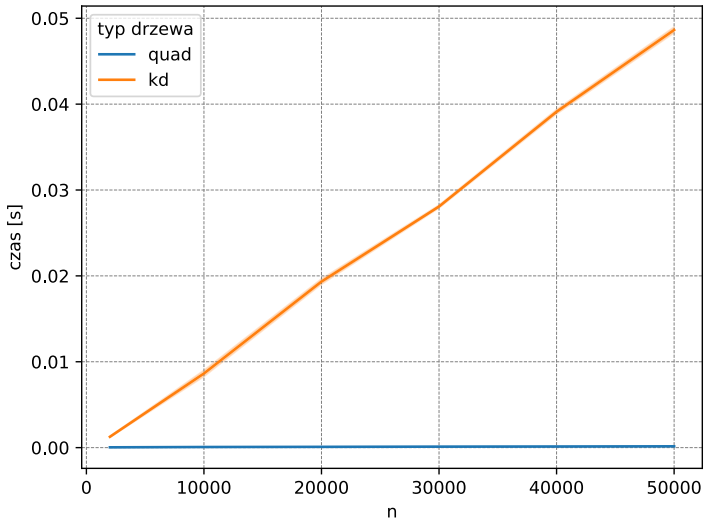


Zbiór C - znajdowanie punktów

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0000	0.0013
10000	0.0001	0.0086
20000	0.0001	0.0193
30000	0.0001	0.0281
40000	0.0001	0.0391
50000	0.0002	0.0486

Tabela 8: Czas odpowiedzi na zapytania o mały obszar dla zbioru C

Zbiór C - znajdowanie punktów (ii)



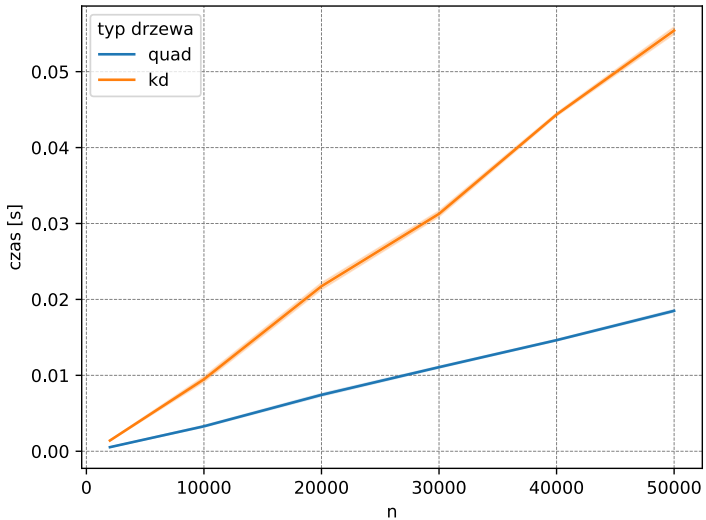
Rysunek 5: Wizualizacja czasu odpowiedzi na zapytania o mały obszar dla zbioru C

Zbiór C - znajdowanie punktów (iii)

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0005	0.0014
10000	0.0033	0.0095
20000	0.0074	0.0217
30000	0.0111	0.0313
40000	0.0146	0.0444
50000	0.0185	0.0554

Tabela 9: Czas odpowiedzi na zapytania o duży obszar dla zbioru C

Zbiór C - znajdowanie punktów (iv)



Rysunek 6: Wizualizacja czasu odpowiedzi na zapytania o duży obszar dla zbioru C

Wnioski

Wnioski

- Quadtree jest prostsze w budowie i bardziej intuicyjne w przestrzeniach 2D, zwłaszcza przy równomiernym rozkładzie punktów.
- KD-Tree jest bardziej elastyczne i skalowalne, szczególnie dla danych wielowymiarowych lub o nierównomiernym rozkładzie.
- W ogólności Quadtree jest szybsze jeśli chodzi o znajdowanie punktów w określonym obszarze, natomiast KD-tree ma duży potencjał jeśli chodzi o wykonywanie innych operacji - np. wyszukiwanie najbliższego sąsiada danego punktu.
- Wybór odpowiedniego algorytmu zależy od charakterystyki danych (rozkład, liczba wymiarów) i rodzaju operacji, jakie mają być wykonywane.