

Dokumentacja

Michał Dobranowski

Wiktor Perczak

31 grudnia 2023

Spis treści

| | | |
|----------|------------------------------------|-----------|
| 1 | Część techniczna | 2 |
| 1.1 | Wymagania | 2 |
| 1.2 | Moduł geometry | 2 |
| 1.2.1 | Rectangle | 2 |
| 1.3 | Moduł tree | 2 |
| 1.4 | Moduł quadtree | 2 |
| 1.4.1 | _Node | 2 |
| 1.4.2 | Quadtree | 3 |
| 1.5 | Moduł kd_tree | 3 |
| 1.5.1 | _Node | 3 |
| 1.5.2 | KdTree | 4 |
| 1.6 | Moduł quick_select | 4 |
| 2 | Część użytkownika | 5 |
| 2.1 | Użycie Quadtree | 5 |
| 2.2 | Użycie KdTree | 5 |
| 3 | Wizualizacja | 5 |
| 3.1 | Wizualizacja QuadtreeVis | 6 |
| 3.2 | Wizualizacja KdTreeVis | 7 |
| 4 | Porównanie – testy czasowe | 8 |
| 4.1 | Wyniki | 8 |
| 4.2 | Wnioski | 10 |
| | Literatura | 11 |

1 Część techniczna

1.1 Wymagania

Do używania programu potrzebny jest język Python w wersji co najmniej 3.10. Ponadto wykorzystywane są również moduły:

- `numpy` 1.26.2
- `bitalg` napisany przez koło naukowe BIT do wizualizacji danych: <https://github.com/aghbit/Algorytmy-Geometryczne>

1.2 Moduł geometry

1.2.1 Rectangle

Klasa `Rectangle` implementuje prostokąt oraz wiele operacji, które można na nim wykonać:

- `def __eq__(self, other: Rectangle) -> bool` – metoda sprawdzająca, czy dwa prostokąty są sobie równe (czy współrzędne ich wierzchołków są identyczne)
- `def __and__(self, other: Rectangle) -> Rectangle | None` – metoda zwracająca część wspólną dwóch prostokątów, lub `None`, jeśli ona nie istnieje
- `def __contains__(self, item: Rectangle | Point) -> bool` – metoda sprawdzająca, czy jeden prostokąt w pełni zawiera się w drugim

1.3 Moduł tree

Zawiera klasę abstrakcyjną `Tree`, po której dziedziczą `Quadtree` oraz `KdTree`. Posiada dwie metody:

- `def __init__(self, points: list[Point])` – konstruktor klasy
- `def find(self, rectangle: Rectangle) -> list[Point]` – zwraca listę punktów, które znajdują się w zadanym prostokącie

1.4 Moduł quadtree

1.4.1 _Node

Każdy obiekt `_Node` trzyma informację o:

- `self.rectangle` – prostokącie, za który odpowiada dany wierzchołek drzewa
- `self.min_x`, `self.max_x`, `self.min_y`, `self.max_y` – ekstremach prostokąta
- `self.mid_x`, `self.min_y` – wartości w środku prostokąta

- `self.children` – dzieciach danego wierzchołka
- `self.leaf_node` – współrzędnych punktu, jeśli wierzchołek jest liściem

1.4.2 Quadtree

Klasa `Quadtree` umożliwia budowanie drzewa czwórkowego oraz odpowiadanie na zapytania o punkty wewnątrz danego prostokąta. Posiada następujące metody:

- `def __init__(self, points: list[Point])` – tworzy pierwszy (największy) prostokąt, który zostaje zapisany w korzeniu drzewa (`self.root`). Następnie wywołuje z korzenia metodę `__construct_subtree`, która konstruuje drzewo czwórkowe.
- `__construct_subtree(self, node: _Node, points: list[Point])` – konstruuje drzewo czwórkowe, dzieląc kolejne prostokąty na cztery ćwiartki oraz rozdzielając odpowiednio punkty na cztery zbiory. Jeżeli dojdzie do prostokąta, wewnątrz którego jest tylko jeden punkt, to obecnie rozważany wierzchołek (`_Node`) staje się liściem drzewa.

Złożoność: $\mathcal{O}(hn)$, gdzie h to wysokość drzewa czwórkowego.

- `def __find(self, node: _Node, rectangle: Rectangle, res: list[Point])` – metoda, która służy do znajdowania punktów wewnątrz zadanego prostokąta. Rekurencyjnie znajduje prostokąty, które w pełni zawierają się w prostokącie z zapytania i dodaje liście poniżej do odpowiedzi.
- `def find(self, rectangle: Rectangle) -> list[Point]` – wywołuje metodę `__find()` i zwraca listę punktów, które mieszczą się w zadanym prostokącie. Złożoność: $\mathcal{O}(hk)$, gdzie k to liczba znalezionych punktów.

1.5 Moduł `kd_tree`

1.5.1 `_Node`

Każdy obiekt `_Node` trzyma informację o:

- `self.left` – lewym dziecku wierzchołka
- `self.right` – prawym dziecku wierzchołka
- `self.rectangle` – prostokącie, który powstał przez ograniczenia przodków
- `self.leafs` – liście wszystkich liści, które są potomkami danego wierzchołka
- `self.leaf_point` – współrzędnych punktu, jeśli dany wierzchołek jest liściem

1.5.2 KdTree

Klasa `KdTree` umożliwia budowanie drzewa oraz odpowiadanie na zapytania 2D, gdyż taki był temat zadania. Kod można jednak bardzo łatwo przekształcić, tak aby umożliwiał operacje na dowolnej liczbie wymiarów. Klasa `KdTree` posiada następujące metody:

- `def __init__(self, points: list[Point])` – konstruktor klasy, do którego przekazuje się listę punktów, a następnie konstruowane jest kd-drzewo. Tworzony jest też parametr `self.root`, dzięki któremu ma się dostęp do całego drzewa.
- `def build_tree(self, points: list[Point], depth: int, rectangle: Rectangle) -> _Node` – metoda, która rekurencyjnie buduje kd-drzewo. W każdej instancji dzieli dany zbiór punktów na dwa zbiory względem współrzędnej podziału, którą jest mediana (wyliczana algorytmem `quick_select`). Następnie wywołuje się rekurencyjnie z dzieci wierzchołka, który jest obecnie rozważany. W liściach zapisuje punkty z zadanego zbioru. Złożoność: $\mathcal{O}(n \log n)$.
- `def __find(self, node: _Node, rectangle: Rectangle, res: list[Point])` – metoda, która służy do znajdowania punktów wewnątrz zadanego prostokąta. Jeśli prostokąt z danego wierzchołka w pełni zawiera się, w prostokącie z zapytania to dodaje do odpowiedzi wszystkie liście poniżej. Jeśli zawiera się tylko częściowo, wywołuje się rekurencyjnie z jego dzieci.
- `def find(self, rectangle: Rectangle) -> list[Point]` – wywołuje metodę `__find()` i zwraca listę punktów, które mieszczą się w zadanym prostokącie. Złożoność: $\mathcal{O}(\sqrt{n} + k)$, gdzie k to liczba znalezionych punktów.

1.6 Moduł `quick_select`

Służy do zwracania mediany nieposortowanego zbioru punktów w czasie $\mathcal{O}(n)$. Działanie jest identyczne z sortowaniem przez scalanie, dzieli zbiór na dwa zbiory względem zadanej wartości, aż odnajdzie medianę. Posiada trzy funkcje:

- `def quick_select(points: list[Point], l: int, r: int, k: int, depth: int) -> Point` – zwraca medianę zbioru punktów, względem danego wymiaru (`depth: int`)
- `def partition(points: list[Point], l: int, r: int, depth: int) -> int` – porządkuje punkty na dwa zbiory: mniejsze od punktu porządkującego lub większe od niego i zwraca indeks pierwszego punktu ze zbioru punktów większych od punktu porządkującego
- `def rand_partition(points: list[Point], l: int, r: int, depth: int) -> int` – losowo wybiera punkt porządkujący, a następnie wywołuje funkcję `partition()`

2 Część użytkownika

W tej sekcji zaprezentowane są przykładowe użycia klas `Quadtree` i `KdTree`: skonstruowanie drzewa oraz znalezienie punktów, leżących wewnątrz zadanego prostokąta.

2.1 Użycie Quadtree

```
from quadtree import Quadtree
from geometry import Rectangle
```

```
points = [(0, 0), (1.5, 1), (2, 3), (2, 0), (0.5, 1.5)]
rectangle = Rectangle(0, 1.5, 1, 3)
```

```
tree = Quadtree(points)
print(tree.find(rectangle))
```

Odpowiedź dla przykładu: $[(0.5, 1.5), (1.5, 1)]$.

2.2 Użycie KdTree

```
from kd_tree import KdTree
from geometry import Rectangle
```

```
points = [(0, 0), (1.5, 1), (2, 3), (2, 0), (0.5, 1.5)]
rectangle = Rectangle(0, 1.5, 1, 3)
```

```
tree = KdTree(points)
print(tree.find(rectangle))
```

3 Wizualizacja

Do wizualizacji wykorzystano narzędzie koła naukowego BIT (<https://github.com/aghbit/Algorytmy-Geometryczne>). Napisano dwie klasy `QuadtreeVis` oraz `KdTreeVis`, które umożliwiają:

- podział przestrzeni na prostokąty
- zobrazowanie zbioru znalezionych punktów dla danego zapytania
- wygenerowanie pliku GIF, który wizualizuje kolejne kroki algorytmu

3.1 Wizualizacja QuadtreeVis

Klasa `QuadtreeVis` dziedziczy po `Quadtree` dodatkowo dodając elementy wizualizacji.

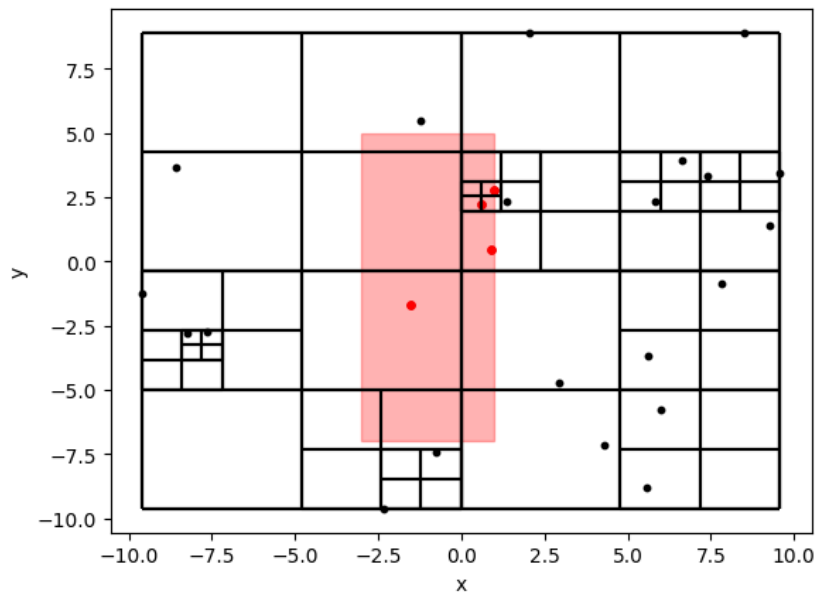
Przykładowy kod na zwizualizowanie procesu tworzenia quadtree (na losowo wygenerowanym zbiorze):

```
from quadtree_visualization import QuadtreeVis
from geometry import Rectangle
from numpy.random import seed, uniform

def generate_uniform_points(left, right, n):
    return list(zip(uniform(left, right, size=n), uniform(left, right,
        ↪ size=n)))

points = generate_uniform_points(-10, 10, 25)
rectangle = Rectangle(-3, 1, -7, 5)
quadtree = QuadtreeVis(points)
quadtree.add_grid()
quadtree.find(rectangle)
quadtree.vis.show()
```

Wykres wygenerowany przez ten program przedstawia poniższy rysunek.



Rysunek 1: Wizualizacja drzewa czwórkowego.

Aby wygenerować plik GIF do wyświetlania kolejnych kroków algorytmu, należy zamiast: `quadtree.vis.show()`, użyć `quadtree.vis.show_gif()`.

3.2 Wizualizacja KdTreeVis

Klasa `KdTreeVis` działa identycznie jak klasa `KdTree`, ale dodatkowo dodaje elementy wizualizacji, zarówno podczas konstruowania drzewa, jak i odpowiadania na zapytanie.

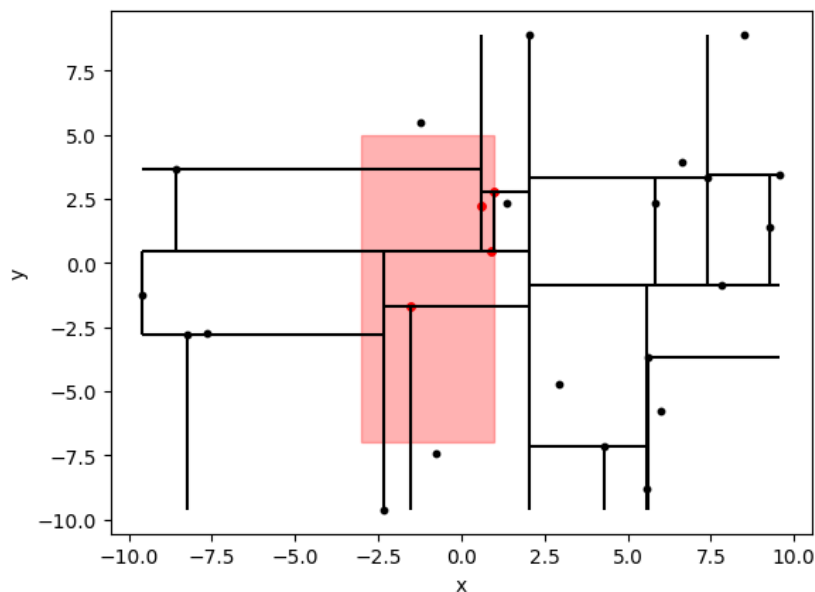
Przykładowe użycie klasy `KdTreeVis` dla losowo wygenerowanego zbioru punktów:

```
from kd_tree_visualization import KdTreeVis
from geometry import Rectangle
from numpy.random import seed, uniform

def generate_uniform_points(left, right, n):
    return list(zip(uniform(left, right, size=n), uniform(left, right,
        ↪ size=n)))

points = generate_uniform_points(-10, 10, 25)
rectangle = Rectangle(-3, 1, -7, 5)
kd_tree = KdTreeVis(points)
kd_tree.find(rectangle)
kd_tree.vis.show()
```

Wykres wygenerowany przez ten program przedstawia poniższy rysunek.



Rysunek 2: Wizualizacja kd-drzewa.

Aby wygenerować plik GIF do wyświetlania kolejnych kroków algorytmu, należy zamiast: `kd_tree.vis.show()`, użyć `kd_tree.vis.show_gif()`.

4 Porównanie – testy czasowe

Porównanie obu struktur danych przeprowadzono przy użyciu dwóch zbiorów testujących:

- $\mathcal{A}(n)$ – zbiór n punktów $P \in \mathcal{S} = [-1000, 1000]^2$ wygenerowanych losowo za pomocą funkcji `numpy.random.uniform`,
- $\mathcal{B}(n)$ – zbiór n punktów, z których połowa to zbiór $\mathcal{A}(n/2)$, a reszta to translacja tego zbioru o wektor $[10^{-8}, 0]$.

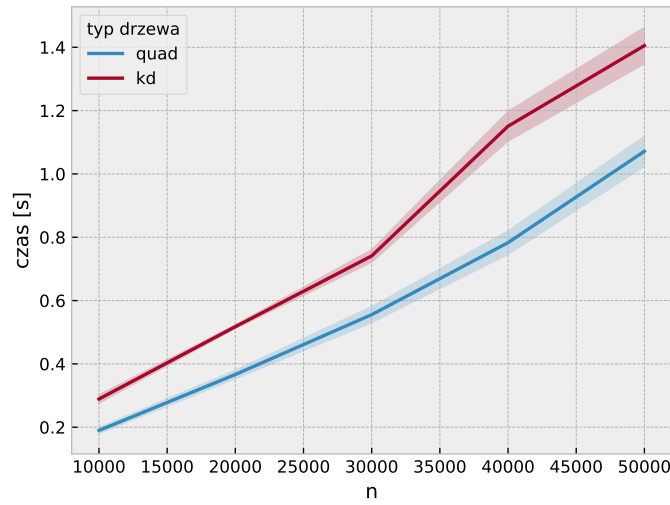
Dla każdego zbioru wybrano po 5 wartości n , dla których przeprowadzono testy. Każdy z testów polegał na wygenerowaniu 15 zbiorów testujących i sprawdzenia czasu wykonywania:

1. konstrukcji drzewa,
2. zapytania o mały prostokąt (losowy, mający powierzchnię równą $\frac{1}{100}$ powierzchni prostokąta \mathcal{S}),
3. zapytania o duży prostokąt (losowy, mający powierzchnię równą $\frac{1}{4}$ powierzchni prostokąta \mathcal{S}),

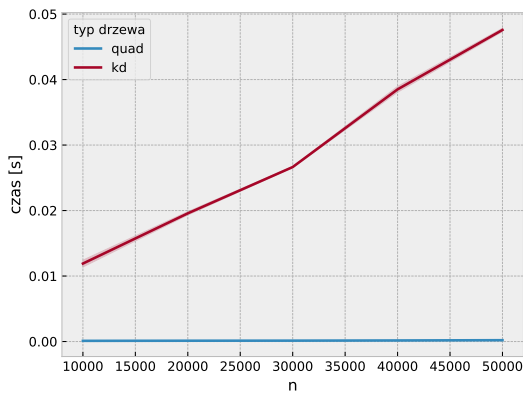
dla każdej ze struktur. Średni czas dla każdej konfiguracji przedstawiono w tabelach [1a](#) i [1b](#) na końcu dokumentu.

4.1 Wyniki

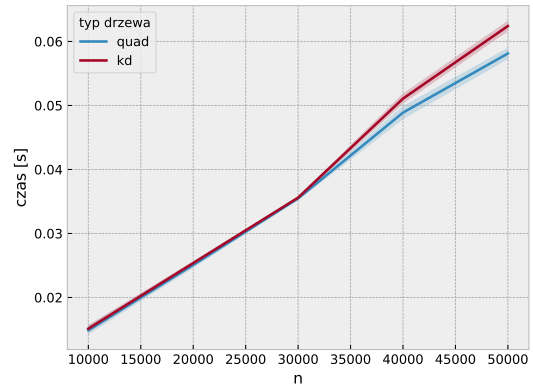
Dla zbiorów typu \mathcal{A} drzewo czwórkowe okazało się być wyraźnie lepsze. Szybciej można je skonstruować (rysunek [3](#)), minimalnie szybciej również odpowiada na zapytania o duży prostokąt (rysunek [4b](#)). W zdecydowanie lepszym czasie odpowiada też na zapytania o mały prostokąt (rysunek [4a](#)).



Rysunek 3: Porównanie czasu konstrukcji drzew dla zbioru \mathcal{A} .



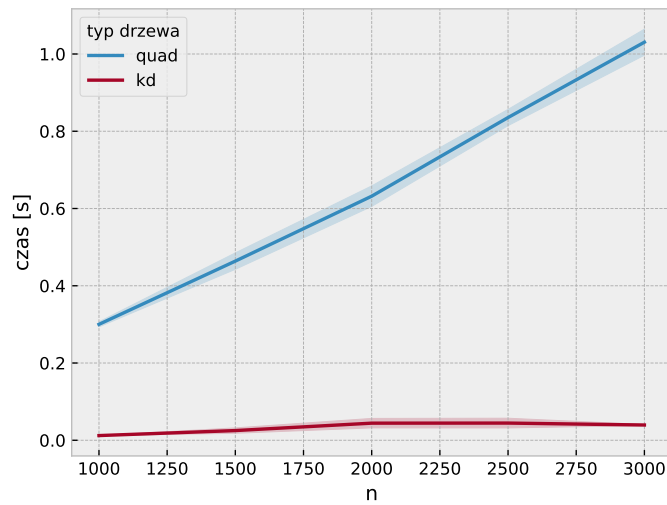
(a) zapytania o mały prostokąt



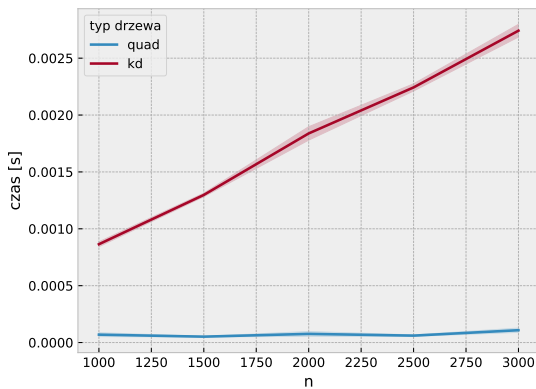
(b) zapytania o duży prostokąt

Rysunek 4: Porównanie czasu zapytania dla zbioru \mathcal{A} .

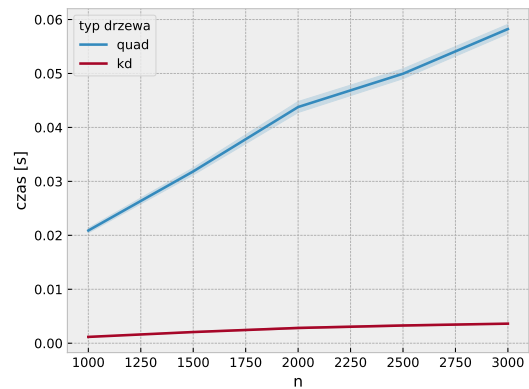
Zupełnie inaczej wygląda sytuacja w przypadku zbiorów typu \mathcal{B} . Zarówno konstrukcja drzewa czwórkowego, jak i użycie go dla zapytań o duży prostokąt, jest zdecydowanie wolniejsze niż w przypadku kd-drzewa (rysunki 5 i 6b). Różnica jest na tyle duża, że testy przeprowadzono na danych o rząd wielkości mniejszych. W zapytaniach o mały prostokąt ponownie lepiej sprawdza się drzewo czwórkowe (rysunek 6a).



Rysunek 5: Porównanie czasu konstrukcji drzew dla zbioru \mathcal{B} .



(a) zapytania o mały prostokąt



(b) zapytania o duży prostokąt

Rysunek 6: Porównanie czasu zapytania dla zbioru \mathcal{B} .

4.2 Wnioski

Wyraźnie widać, że względne odległości między punktami mają ogromne znaczenie przy wyborze odpowiedniej struktury danych. W większości przypadków lepiej sprawdzi się drzewo czwórkowe, ale łatwo również znaleźć takie zbiory punktów, na których jego wydajność bardzo szybko spada.

Lepiej sprawdzi się drzewo czwórkowe w sytuacjach, w których zwykle będziemy pytać o małe podzbiory zbioru punktów (na przykład w aplikacjach typu Google Maps), albo gdy możemy pominąć duże obszary tego zbioru (wykrywanie kolizji i kompresja obrazów).

| typ operacji | n | czas [s] | |
|-----------------|-------|----------|---------|
| | | quadtree | kd-tree |
| 1 | 10000 | 0.1898 | 0.2890 |
| | 20000 | 0.3665 | 0.5176 |
| | 30000 | 0.5555 | 0.7410 |
| | 40000 | 0.7829 | 1.1507 |
| | 50000 | 1.0716 | 1.4053 |
| 2 | 10000 | 0.0001 | 0.0119 |
| | 20000 | 0.0001 | 0.0196 |
| | 30000 | 0.0001 | 0.0266 |
| | 40000 | 0.0002 | 0.0385 |
| | 50000 | 0.0002 | 0.0476 |
| 3 | 10000 | 0.0149 | 0.0151 |
| | 20000 | 0.0251 | 0.0254 |
| | 30000 | 0.0355 | 0.0356 |
| | 40000 | 0.0489 | 0.0511 |
| | 50000 | 0.0581 | 0.0624 |

(a) zbiory $\mathcal{A}(n)$

| typ operacji | n | czas [s] | |
|-----------------|------|----------|---------|
| | | quadtree | kd-tree |
| 1 | 1000 | 0.3001 | 0.0122 |
| | 1500 | 0.4638 | 0.0251 |
| | 2000 | 0.6319 | 0.0443 |
| | 2500 | 0.8355 | 0.0445 |
| | 3000 | 1.0306 | 0.0395 |
| 2 | 1000 | 0.0001 | 0.0009 |
| | 1500 | 0.0001 | 0.0013 |
| | 2000 | 0.0001 | 0.0018 |
| | 2500 | 0.0001 | 0.0022 |
| | 3000 | 0.0001 | 0.0027 |
| 3 | 1000 | 0.0209 | 0.0012 |
| | 1500 | 0.0318 | 0.0021 |
| | 2000 | 0.0438 | 0.0028 |
| | 2500 | 0.0500 | 0.0033 |
| | 3000 | 0.0582 | 0.0036 |

(b) zbiory $\mathcal{B}(n)$

Tabela 1: Porównanie średnich czasów dla poszczególnych operacji, zbiorów i struktur.

Kd-drzewo również ma swoje zalety; warto wymienić między innymi bardzo proste uogólnienie na większą liczbę wymiarów czy łatwiejsze wprowadzenie dodatkowych operacji (na przykład szukania najbliższego sąsiada dla danego punktu).

Literatura

- [1] **Algorytmy geometryczne – wykład**, dr inż Barbara Głut.
- [2] **Quadtrees, Geometric Approximation Algorithms**, prof. Kevin Buchin, TU Dortmund. https://ls11-www.cs.tu-dortmund.de/_media/buchin/teaching/akda_ws21/quadtrees.pdf
- [3] **Wprowadzenie do algorytmów**, Thomas H. Cormen et al.