# Wyszukiwanie geometryczne
# Przeszukiwanie obszarów ortogonalnych
*Quadtree* i *kd-drzewa*
Dokumentacja projektu

Stanisław Łenyk
Jerzy Wilczek

Styczeń 2021

# Spis treści

# 1 Część techniczna

## 1.1 Wymagania

Aby uruchomić program należy użyć środowiska `Python` w wersji co najmniej `Python 3.8`. Ponadto należy posiadać następujące niestandardowe moduły w środowisku uruchomieniowym:

- `numpy`
- `matplotlib`

## 1.2 Moduł `geometry`

Moduł ten zawiera różne funkcje i struktury pomocnicze wykorzystywane zarówno przez moduł `kd_tree` jak i `quadtree`, jego głównym elementem jest klasa `Rectangle`, implementująca prostokąt, która umożliwia wykonywanie wielu operacji takich jak porównywanie ze sobą prostokątów, sprawdzanie czy prostokąty zawierają się w sobie, obliczanie części wspólnej dwóch prostokątów czy zamianę prostokąta na inną reprezentację takiej figury. Moduł nie zostanie tu szczegółowo opisany, ponieważ implementuje operacje o bardzo niskim poziomie skomplikowania i pełni głównie funkcje pomocniczą i wydzielenia wspólnej części kodu z pozostałych modułów.

## 1.3 Moduł `quadtree`

Moduł implementuje następujące klasy:

- `_Node` – prywatną klasę reprezentującą wierzchołki drzewa
- `Quadtrant(IntEnum)` – typ wyliczeniowy służący do oznaczania kolejnych ćwiartek (NE, NW, SW, SE)
- `Quadtree` – klasę umożliwiającą tworzenie drzewa ze zbioru punktów i przeszukiwanie go
- `View` – wykorzystywaną przez `Quadtree` klasę służącą do wizualizajcji

### 1.3.1 Klasa `_Node`

Każdy wierzchołek przetrzymuje następujące informacje:

- Górną, dolną, lewą oraz prawą granicę obejmowanego obszaru
- Oznaczenie (numerowane klasą `Quadrant(IntEnum)`) kwadrantu, który dany wierzchołek reprezentuje w relacji ze swoim rodzicem. Korzeń drzewa posiada w tym miejscu wartość `None`
- Listę swoich dzieci lub wartość `None` w przypadku braku jakiegokolwiek dziecka

- pola pomocnicze – środki reprezentowanych granic przedziału w poziomie i pionie

Klasa ta posiada także funkcję umożliwiającą dodawanie dzieci wierzchołkowi.

### 1.3.2 Klasa `Quadtree`

Klasa posiada następujące funkcje:

- `__init__(self, points: List[Point])` – konstruktor klasy tworzący korzeń drzewa
- `__create_quadtree(self, node: _Node, points: List[Point])` – prywatną funkcję wywoływaną w konstruktorze i tworzącą drzewo z podanego zbioru punktów
  **Argumenty:** konieczne jest podanie zbioru punktów w formie `List[Tuple [float, float]]`
  **Złożoność obliczeniowa:** $O(hn)$, gdzie $h$ to wysokość drzewa, a $n$ to ilość przechowywanych punktów.
- `find(self, rect: Rectangle, visualize=False)` – funkcję umożliwiającą wyszukanie punktów znajdujących się w zadanym prostokącie
  **Argumenty:** funkcja przyjmuje jako argument obiekt klasy `Rectangle` opisanej w 1.2 oraz opcjonalny argument `bool` informujący czy użytkownik chce dokonać wizualizacji
  **Złożoność obliczeniowa:** $O(hk)$ gdzie $h$ – wysokość drzewa, $k$ – liczba liści odpowiedzialnych za obszar przecinający się z zadanym
- `__find(self, node: _Node, rect: Rectangle, res: List[Point], view)` – prywatną funkcję pomocniczą wywoływaną przez główną funkcję `find`

## 1.4 Moduł `kd_tree`

Moduł implementuje następujące klasy:

- `_Node` - prywatną klasę przechowującą węzeł *kd-drzewa*
- `KDTree` - klasę przechowującą całe *kd-drzewo* i umożliwiającą operację na nim. W tej klasie umieszczony jest cały interfejs publiczny tego modułu.

### 1.4.1 Klasa `_Node`

Klasa ta zawsze posiada następujące pola:

- `points` - lista punktów przechowywanych w prostokącie, za który jest odpowiedzialny dany węzeł
- `is_leaf` - wartość typu prawda-fałsz informująca o tym, czy dany węzeł jest liściem

- `region` - prostokąt, za który dany węzeł jest odpowiedzialny

Dodatkowo jeżeli dany węzeł nie jest liściem posiada pola:

- `division_axis_type` - wartość oznaczająca równolegle do której osi układu współrzędnych przebiega linia dzieląca dany węzeł na potomków
- `__point_comparing_key` - prywatną funkcję wydobywającą z punktów współrzędną, według której punkty są porównywane przy określaniu który punkt leży w którym dziecku węzła
- `dividing_line` - wartość oznaczająca współrzędną, na której położona jest linia podziału na dzieci
- `left` oraz `right` - wskaźniki na dzieci węzła

Klasa posiada również następujące funkcje:

- `__init__(self, points: List[Point], region: Rectangle = None)` - konstruktor klasy, konstruujący również dzieci danego węzła (jeśli jakieś posiada). Parametry funkcji oznaczają:
  - `points` - listę punktów przechowywanych w węźle
  - `region` - prostokąt, za który odpowiedzialny jest dany węzeł. W przypadku konstruowania korzenia drzewa parametr ten jest ustawiany jako pusty i obliczany automatycznie przez konstruktor przy użyciu funkcji `rectangle_from_points(points)` z modułu `geometry`

  Konstruktor samodzielnie decyduje czy podzielić węzeł pionowo czy poziomo wybierając tę oś podziału, która stworzy jak najbardziej optymalną strukturę drzewa.
  **Złożoność obliczeniowa:** $O(nlog(n))$, gdzie $n$ to ilość przechowywanych punktów.
- `__median(self)` - prywatna funkcję pomocniczą obliczająca medianę punktów przechowywanych w danym węźle względem jego osi podziału. Jeśli węzeł posiada więcej niż 1000 punktów, funkcja losowo wybiera 1000 z nich, i to z tych punktów oblicza medianę, co nadal pozwala na uzyskanie optymalnego podziału, a zapewnia że funkcja działa w stałej złożoności obliczeniowej.
  **Złożoność obliczeniowa:** $O(1)$
- `get_divider_line(self)` - zwraca linię podziału węzła (używana przy wizualizacji)
  **Złożoność obliczeniowa:** $O(1)$
- `get_lines_from_node(self)` - zwraca prostokąt za który jest odpowiedzialny dany węzeł oraz linię podziału węzła (używana przy wizualizacji)
  **Złożoność obliczeniowa:** $O(1)$

### 1.4.2 Klasa `KDTree`

Klasa odpowiedzialna za przechowywanie drzewa, enkapsulację wyszukiwania w drzewie oraz udostępnianie zwizualizowanego drzewa.

Posiada ona następujące pola:

- `__root` - wskaźnik na korzeń drzewa
- `__rectangles` oraz `__dividers` - linie wyznaczające prostokąty, na które jest podzielone drzewo (pola używane przy wizualizacji)

Klasa implementuje następujące funkcje:

- `__init__(self, points)` - konstruktor klasy, parametr `points` oznacza listę punktów, które ma przechowywać struktura.
  **Złożoność obliczeniowa:** $O(nlog(n))$, gdzie $n$ oznacza liczbę przechowywanych punktów
- `search(self, x_min, x_max, y_min, y_max, visualize=False)` - funkcja służąca do wyszukiwania w drzewie. Jeśli parametr `visualize` jest nieustawiony lub ustawiony na wartość `False` funkcja zwraca listę punktów ze struktury leżących wewnątrz prostokąta $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$ w postaci `List[Tuple[float, float]]`. Jeśli parametr `visualize` jest ustawiony na `True`, funkcja zwraca taką samą listę punktów oraz listę scen (klasa `Scene`) przedstawiających zwizualizowane kolejne kroki podejmowane przez algorytm wyszukiwania.
  **Złożoność obliczeniowa:** $O(k + \sqrt{n})$, gdzie $k$ oznacza ilość punktów w wyjściu, a $n$ - ilość punktów w strukturze
- `get_visualized(self)` - zwraca scenę (klasa `Scene`) przedstawiającą zwizualizowane drzewo.
  **Złożoność obliczeniowa:** $O(1)$

### 1.4.3 Funkcje nienależące do żadnej klasy

Wszystkie funkcje zawarte w module, które nie należą do żadnej klasy są oznaczone jako prywatne. Pełnią one rolę funkcji pomocniczych. Są to:

- `_kd_search(node, rectangle, frames=None)` - funkcja pomocnicza do wyszukiwania w poddrzewa o korzeniu w węźle `node` punktów leżących wewnątrz prostokąta `rectangle`. Parametr `frames` powinien być tablicą akumulującą kolejne klatki wizualizacji (jeśli program ma przeprowadzić wizualizację)
- `_get_lines_from_subtree(node)` - funkcja zwracająca linie wyznaczające prostokąty, na które jest podzielone poddrzewo o korzeniu w węźle `node` (używana przy wizualizacji)

## 1.5 Moduł `draw_tool`

Moduł ten zawiera narzędzie wizualizacyjne autorstwa mgr. inż. Krzysztofa Podsiadło.

## 1.6 Moduł `gen_data`

Moduł ten umożliwia bardzo proste wygenerowanie danych do przetestowania działania struktur. Zawiera on dwie funkcje:

- `gen_points(scope=(0, 100), n=100)` - funkcja zwracająca tablicę zawierającą `n` losowych punktów należących do obszaru *scope* × *scope*
- `gen_rect(scope=(0, 100))` - funkcja zwracająca prostokąt należący do obszaru *scope* × *scope*

## 1.7 Moduł `time_test`

Moduł ten służy do automatycznego testowania modułów `quadtree` oraz `kd_tree`. Zawiera on jedną klasę - `Tester` i kilka metod nienależących do żadnej klasy. Klasa `Tester` zawiera następujące metody:

- `__init__(self, n_values, rectangle_amount_per_test, scope=(0, 100))` - konstruktor klasy, który generuje automatycznie paczki testów według podanego w parametrach opisu.
  - `n_values` powinien być listą zawierającą liczby całkowite odpowiadające ilościom punktów w kolejnych paczkach testów.
  - `rectangle_amount_per_test` powinien być liczbą całkowitą oznaczającą ile prostokątów będzie wyszukiwanych w każdej paczce testów
  - `scope` oznacza zakres z jakiego wybierane są współrzędne punktów i prostokątów
- `print_tests_csv(self, buildup_tester, search_tester, filename)` - funkcja przyjmująca dwa wskaźniki na funkcje testujące odpowiednio czasy konstrukcji struktury i czasy wyszukania w strukturze na wszystkich paczkach testów i wypisująca wyniki do plików `filename_buildup.csv` oraz `filename_search.csv`
- `print_tests_both_trees_csv(self, bas_filename)` - funkcja testująca obie struktury i wypisująca wyniki do plików `base_filename_quadtree_buildup.csv`, `base_filename_quadtree_search.csv`, `base_filename_kd_tree_buildup.csv` oraz `base_filename_kd_tree_search.csv`

Moduł zawiera również metody `test_kd_buildup`, `test_quadtree_buildup`, `test_kd_search` oraz `test_quadtree_search`, które wykonują pomiary czasowe danej struktury na jednej paczce testów

# 2 Część użytkownika

Aby skorzystać z modułów należy:

1. Zaimportować moduł `quadtree` lub `kd_tree`
2. Skonstruować obiekt odpowiednio klasy `Quadtree` lub `KDTree`. Podawane jako parametr punkty muszą być zapisane w formie `List[Tuple[float, float]]`
3. Wyszukać prostokąty w drzewie

## 2.1 Moduł `quadtree`

Listing 1: Przykładowe uruchomienie modułu `quadtree`

```
from quadtree import Quadtree
from geometry import Rectangle

points = [(1.5, 2), (4, 6), (2, 2), (3.2, 4.1)]
tree = Quadtree(points)  # skonstruowanie obiektu klasy Quadtree

print(tree.find(Rectangle(-2, 6, 2, 4.5)))  # wyszukanie prostokata
# [(3.2, 4.1)]

print(tree.find(Rectangle(-10, 10, -20, 15)))
# [(4, 6), (3.2, 4.1), (1.5, 2), (2, 2)]
```

## 2.2 Moduł *kd-drzewa*

Listing 2: Przykładowe uruchomienie modułu `kd_tree`

```
from kd_tree import KDTree # importowanie modulu

points = [(0, 0), (1, 1), (2, 2)]
tree = KDTree(points) # skonstruowanie obiektu klasy KDTree

print(tree.search(0.5, 1.5, 0.5, 1.5)) # wyszukanie prostokata
# program wypisze: [(1, 1)]

print(tree.search(0.5, 2, 0.5, 2)) # wyszukanie prostokata
# program wypisze: [(1, 1), (2, 2)]
```

## 2.3 Moduł wizualizacji

Aby zwizualizować działanie algorytmów należy użyć programu *Jupyter Notebook*. Wizualizację przeprowadza się w Jupyter'owym *zeszycie* - pliku `.ipynb`. W komórce w takim zeszycie należy:

1. Wstawić linijkę kodu potrzebnego do działania wizualizacji - `%matplotlib notebook`

2. Zaimportować wszystkie elementy narzędzia rysującego (moduł `draw_tool`)

3. Zaimportować moduł, który ma zostać zwizualizowany

4. Użyć modułu w sposób umożliwiający zdobycie obiektów klasy `Scene` dla `KDTree` albo klasy `Plot` dla `Quadtree`

5. Z uzyskanych obiektów klasy `Scene` utworzyć obiekt klasy `Plot` (pominąć w przypadku `Quadtree`)

6. Na obiekcie klasy `Plot` wywołać metodę `.draw()` *Uwaga: jedna komórka z zeszytu może narysować tylko jeden obiekt Plot - jeśli istnieje potrzeba wykonania dwóch rysunków należy użyć dwóch komórek zeszytu*

Poniżej zamieszczamy szczegółowe instrukcje dot. wizualizacji, jednak warto wspomnieć, że do projektu załączamy plik `visualizer.ipynb`, w którym zamieszczone są działające przykłady wizualizacji struktur.

### 2.3.1 Wizualizacja struktury *quadtree*

Listing 3: Przykładowa wizualizacja modułu `quadtree`

```
%matplotlib notebook # wymagana linijka kodu
from draw_tool import * # importowanie narzedzia graficznego
from quadtree import Quadtree # importowanie modulu
```

### 2.3.2 Wizualizacja struktury *kd-tree*

Aby zdobyć scenę przedstawiającą skonstruowane drzewo należy na obiekcie klasy `KDTree` wywołać metodę `.get_visualized()`

Listing 4: Przykładowa wizualizacja struktury `KDTree`

```
%matplotlib notebook # wymagana linijka kodu
from draw_tool import * # importowanie narzedzia graficznego
from kd_tree import KDTree # importowanie modulu

points = [(1.5, 2), (4, 6), (2, 2), (3.2, 4.1)]
tree = Quadtree(points)  # skonstruowanie obiektu klasy Quadtree

# zapisanie zwracanego obkiektu klasy Plot
# (konieczne dodanie argumentu True do tree.find)
plot = tree.find(Rectangle(-2, 6, 2, 4.5), visualize=True)

# narysowanie scen zebranych w trakcie wykonywania algorytmu
plot.draw()
```

Aby zdobyć listę scen przedstawiających kolejne kroki algorytmu wyszukiwania w drzewie należy przy wywoływaniu funkcji `.search(...)` ustawić w niej parametr `visualize=True`. Spowoduje to, że zwróci ona poza standardową listą punktów również listę odpowiednich scen.

Listing 5: Przykładowa wizualizacja wyszukiwania w `KDTree`

```
%matplotlib notebook # wymagana linijka kodu
from draw_tool import * # importowanie narzedzia graficznego
from kd_tree import KDTree # importowanie modulu

points = [(0, 0), (1, 1), (2, 2)]
tree = KDTree(points) # skonstruowanie obiektu klasy KDTree

result_points, scenes = tree.search(
    0.5,
    1.5,
    0.5,
    1.5,
    visualize=True
)
# zdobycie listy scen sceny przedstawiajacych
# kolejne kroki wykonywane przez algorytm

plot = Plot(scenes)  # skonstruowanie obiektu Plot
plot.draw()  # narysowanie zebranych scen
```

# 3 Sprawozdanie – testy czasowe

Testy czasowe przeprowadzone zostały na komputerze z systemem Windows 10 x64 z procesorem Intel Core i7-8565U.

Testy wykonane zostały przy użyciu klasy `Tester` z modułu `time_test` omówionego w 1.7.

Struktury były testowane na losowych, ale takich samych zbiorach punktów, a przedstawione wyniki czasowe przeszukiwań są średnią ze 100 losowo wybranych prostokątów należących do odpowiadających kolejnym zbiorom punktów przedziałów.

W ostatnim wierszu każdej z tabel przedstawione są ilorazy czasów działania obu struktur.

| n | quadtree | kd-tree | quad/kd |
|---|---|---|---|
| 10 | 0.000 | 0.000 | 0.333 |
| $10^2$ | 0.001 | 0.002 | 0.311 |
| $10^3$ | 0.009 | 0.026 | 0.363 |
| $10^4$ | 0.164 | 0.360 | 0.456 |
| $10^5$ | 2.473 | 4.185 | 0.591 |

Tablica 1: Czas tworzenia struktur 1.

| n | quadtree | kd-tree | quad/kd |
|---|---|---|---|
| $10^5$ | 2.825 | 4.637 | 0.609 |
| $2 \cdot 10^5$ | 5.417 | 11.502 | 0.471 |
| $3 \cdot 10^5$ | 8.804 | 16.680 | 0.528 |
| $4 \cdot 10^5$ | 11.791 | 21.262 | 0.555 |

Tablica 2: Czas tworzenia struktur 2.

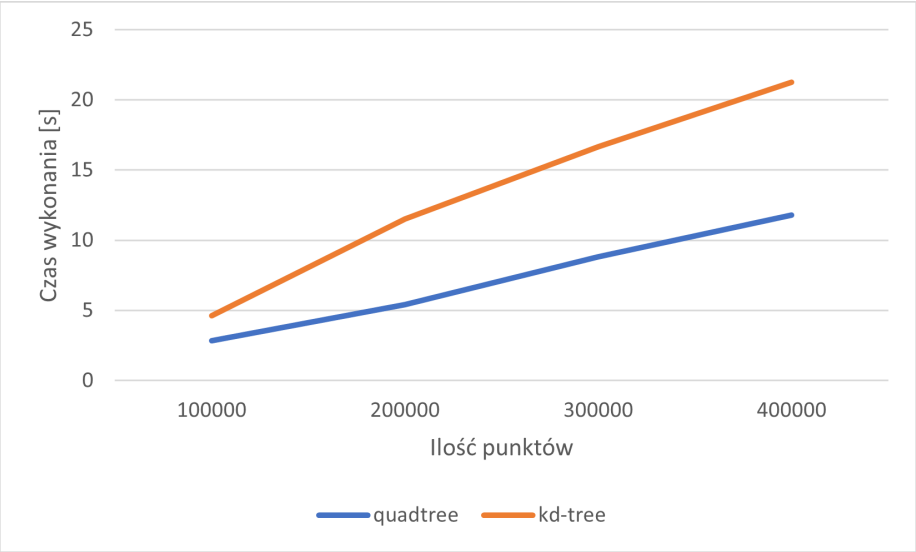
| n | quadtree | kd-tree | quad/kd |
|---|---|---|---|
| 10 | 0.0000062 | 0.0000417 | 0.148 |
| $10^2$ | 0.0000457 | 0.0001891 | 0.241 |
| $10^3$ | 0.0002617 | 0.0006866 | 0.381 |
| $10^4$ | 0.0017161 | 0.0020760 | 0.827 |
| $10^5$ | 0.0198555 | 0.0079135 | 2.509 |

Tablica 3: Czasy wykonywania zapytań 1.

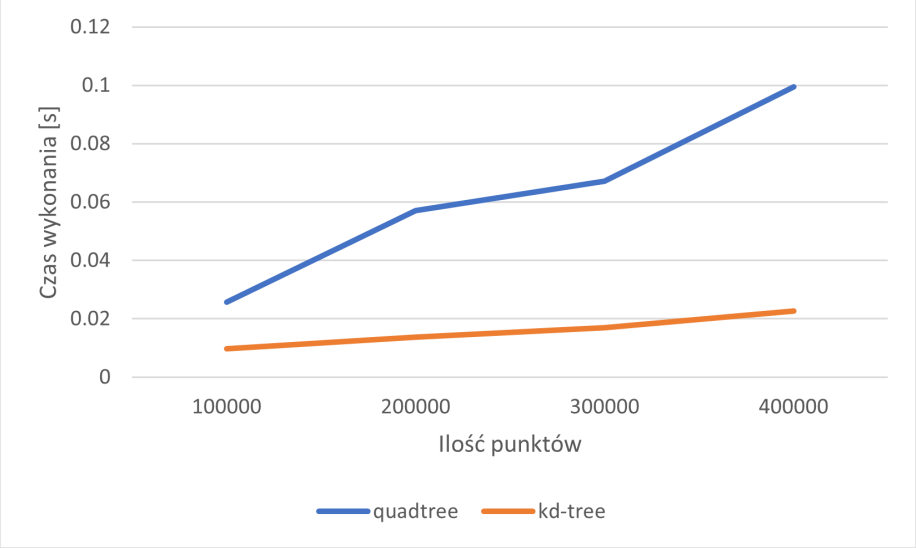
| n | quadtree | kd-tree | quad/kd |
|---|---|---|---|
| $10^5$ | 0.026 | 0.010 | 2.622 |
| $2 \cdot 10^5$ | 0.057 | 0.014 | 4.147 |
| $3 \cdot 10^5$ | 0.067 | 0.017 | 3.973 |
| $4 \cdot 10^5$ | 0.099 | 0.023 | 4.371 |

Tablica 4: Czas wykonywania zapytań 2.

Tworzenie drzewa ćwiartkowego na każdym z testowanych zakresów danych jest ok. 2-3 razy szybsze w stosunku do kd-drzewa. Czasy wykonywania zapytań dla mniejszych $n$ są krótsze w przypadku quadtree, jednakże zwiększanie wejściowej liczby punktów odwraca tę zależność.




Rysunek 1: Porównanie czasów budowania drzewa przez obie struktury




Rysunek 2: Porównanie czasów wykonywania zapytań przez obie struktury

# 4 Bibliografia


1. Robert Bembenik *Metody eksploracji danych z systemów informacji przestrzennej*, październik 2006

2. dr inż. Barbara Głut *Wykład – wyszukiwanie geometryczne*

3. mgr inż. Krzysztof Podsiadło *narzędzie wizualizacyjne*