Wyszukiwanie geometryczne – przeszukiwanie obszarów ortogonalnych

KDTree i QuadTree

Dokumentacja

Andrzej Karciński i Aleksandra Smela

algorytmy geometryczne

styczeń 2023

Spis treści

1 Część techniczna

1.1 Wymagania

Projekt został zaimplementowany w języku Python (3.9). Wykorzystany został Jupyter Notebook oraz zintegrowane środowisko programistyczne dla Pythona (PyCharm 2021.2.3).

Do działania wymagane są biblioteki matplotlib oraz numpy.

1.2 Opis głównych modułów

KDTree.py
Moduł zawierający implementację struktury KDTree.

QuadTree.py
Moduł zawierający implementację struktury QuadTree.

visualization.ipynb
Interfejs zawierający implementację struktur z możliwością wizualizacji. Plik stanowi interfejs dla użytkownika.

testGenerator.ipynb
Interfejs zawierający funkcje umożliwiające generowanie danych testowych.

tests.ipynb
Interfejs zawierający funkcje umożliwiające testowanie zbiorów danych wczytanych z pliku oraz wybrane przeprowadzone testy.

test_gen.py
Moduł zawierający takie same funkcje jak interfejs testGenerator.ipynb.

1.3 Opis programu

1.3.1 KDTree

Zaimplementowana struktura służy do przechowywania i organizacji punktów przestrzeni dwuwymiarowej. Umożliwia budowanie drzewa oraz wyszukiwanie punktów z zadanego przedziału.

Implementacja opiera się na informacjach z wykładu i prezentacji do wykładu.

1.3.1.1 Budowanie drzewa

Do zbudowania drzewa służy konstruktor obiektu klasy KDTree.

1. Sposób podziału punktów: Na parzystych poziomach punkty zostały podzielone względem współrzędnej y, a na nieparzystych względem współrzędnej x.
2. Punkty podziałów: Dla zbalansowania każdorazowo punkty były sortowane względem danej współrzędnej, a punkt przedziału był wybierany jako średnia k-tego punktu i k+1-ego punktu, gdzie k to [liczba punktów]//2-1.
3. Rekurencja: Następnie algorytm rekurencyjnie tworzył poddrzewa dla zbiorów punktów, otrzymanych w danym podziale.
4. Warunek brzegowy: Jeżeli zbiór punktów składa się z jednego punktu, tworzymy liść, do którego przypisujemy ten punkt.

1.3.1.2 Klasy własnej implementacji wykorzystane do realizacji struktury

KDTree

Zaimplementowana struktura *KDTree* zawiera atrybut *root* (korzeń) który jest obiektem typu *KDTNode*.

KDTNode

KDTNode reprezentuje węzeł drzewa KDTree oraz zawiera atrybuty: *value, no_points, left, right*. Jeżeli *KDTNode* jest:

- liściem drzewa, to atrybuty te przyjmują wartości:
  - *value*: punkt reprezentowany przez krotkę postaci (x,y), gdzie x, y to współrzędne punktu;
  - *no_points*: 1;
  - *left*: None;
  - *right*: None.
- węzłem niebędącym liściem, to atrybuty te przyjmują wartości:
  - *value*: obiekt klasy *OrthagonalRange*, reprezentujący przedział punktów, które są przechowywane w prawym i lewym poddrzewie;
  - *no_points*: liczbę punktów przechowywanych w prawym i lewym poddrzewie;
  - *left*: obiekt klasy *KDTree* – lewe poddrzewo;
  - *right*: obiekt klasy *KDTree* – prawe poddrzewo.

OrthagonalRange

OrthagonalRange służy do reprezentacji przedziałów ortogonalnych. Zawiera atrybuty: x_1, x_2, y_1, y_2, reprezentujące odpowiednio prawą, lewą, górną, dolną granicę przedziału.

Klasa posiada metody:

- *a.intersect(b: OrthagonalRange)* – sprawdzającą czy przedziały a i b się przecinają;

- *a.contain(obj: KDTNode)* – sprawdzającą czy przedział a zawiera przedział lub punkt z pola *value* obiektu obj;
- *a.to_list()* – zwracającą reprezentację zawierającą listę linii reprezentujących granice danego przedziału.

1.3.1.3 Wyszukiwanie punktów z zadanego przedziału

Do wyszukiwania punktów z zadanego przedziału służy metoda *search.(D: OrthagonalRange):*

- Jeżeli przedział reprezentowany przez lewe lub prawe poddrzewo danego węzła:
    - zawiera się w D, to zwracam wszystkie liście z tego poddrzewa;
    - przecina się z D, to rekurencyjnie przeszukuję to poddrzewo.
- Warunek brzegowy to natrafienie w rekurencji na liść – w takim przypadku sprawdzam czy punkt z liścia należy do D i zwracam w zależności od wyniku.

1.3.2 QuadTree

1.3.2.1 Budowanie drzewa

Do zbudowania QuadTree należy wykorzystać konstruktor do tego stworzony.

- Punkty umieszczamy w węzłach drzewa, które to są dzielone na 4 jednakowej wielkości obszary tak zwane dzieci: prawy-górny, lewy-górny, prawy-dolny, lewy-dolny.
- Następnie następuje przejście rekurencyjne dzielące aktualny węzeł na 4 poddrzewa

1.3.2.2 Klasy własnej implementacji wykorzystane do realizacji struktury

QuadTree

Struktura QuadTree przechowująca korzeń root typu QTNode, lewy dolny wierzchołek minimalnego prostokąta zawierającego wszystkie punkty oraz długość oraz wysokość tego prostokąta a także wszystkie punkty podanego zbioru.

Klasa posiada metodę:

- *a.find_points(lower_left: Point, upper_right: Point)* - metoda pozwalająca na sprawdzenie, które z punktów leżą wewnątrz zadanego prostokąta za pomocą 2 wierzchołków, a następnie zwracająca listę tych punktów.

QTNode

Zaimplementowana struktura węzła w drzewie posiadająca listę dzieci typu QTNode, punkty zawarte w danym prostokącie opisanym przez pola lower_left przechowującym współrzędne lewego dolnego wierzchołka oraz długość i wysokość. Jeżeli węzeł jest liściem to jego tablica dzieci jest pusta, w innym przypadku zawiera 4 dzieci.

Klasa posiada metody:

- *a.seek_points(lower_left: Point, width: float, height: float)* - metoda zwraca punkty zawarte wewnątrz zadanego prostokąta poszukując punktów jedynie wśród punktów węzła, w którym aktualnie się znajdujemy.
- *a.new_point(xadd: float, yadd: float)* - metoda tworząca nowy obiekt klasy Point przesunięty o xadd i yadd względem aktualnego lewego dolnego wierzchołka badanego węzła.
- *a.divide_node()* - metoda, która rekurencyjnie dzieli węzły w taki sposób aby w każdym z nich nie znajdowało się więcej niż zadaną liczbę ustalaną podczas inicjalizowania QuadTree punktów.
- *a.find(rec_lower_left: Point, rec_upper_right: Point, result: [Point])* - metoda wywoływana podczas wywołania metody *find_points* na obiekcie typu QuadTree. Metoda zwraca listę punktów zawartych w danym poddrzewie na którego korzeniu wykonaliśmy tą metodę.

1.3.2.2.1 Point

Zaimplementowana klasa reprezuntująca punkt w przestrzeni przyjmująca w konstruktorze 2 parametry x i y danego punktu.

1.3.2.3 Wyszukiwanie punktów z zadanego przedziału

W celu wyszukania punktów należy wywołać metodę *find_points(lower_left: Point, upper_right: Point).*

- Następuje pierwsze wywołanie funkcji *find* z klasy QTNode.
- Rozważamy 4 przypadki dla węzła:
- Węzeł jest dzieckiem i wtedy sprawdzamy każdy z punktów należących do tego węzła czy leży wewnątrz zadanego prostokąta.
- Węzeł całkowicie zawiera się wewnątrz prostokąta, wtedy wszystkie punkty z węzła dodajemy do wyniku końcowego.
- Węzeł nie przecina badanego obszaru a więc nie ma potrzeby szukać punktów w jego poddrzewach.
- Jeżeli nie nastąpił żaden z powyższych przypadków wywołujemy rekurencyjnie metodę find na dzieciach węzła.

2 Część użytkownika

2.1 Uruchomienie programu

visualization.ipynb
Moduł ten należy otworzyć w Jupyter Notebook. Zawiera on wizualizację wybranych zbiorów danych i umożliwia wczytanie danych z pliku i zwizualizowanie wybranych algorytmów.

testGenerator.ipynb
Moduł ten należy otworzyć w Jupyter Notebook. Umożliwia on generowanie danych i zapisywanie ich do plików. Generowanie danych może zajść poprzez wprowadzenie ich myszką lub wylosowanie zbiorów o konkretnych właściwościach.

2.1 Funkcje dostępne dla użytkownika

2.1.1 Wczytanie zbioru punktów z pliku (visualization.ipynb)

```
#aby wczytać zbiór punktów z pliku json należy zastosować funkcję file_to points

#funkcja ta została zaimportowana z modułu test_gen.py

linear = file_to_points('test_collections\\linear.json')

#zbiory danych można wygenerować stosując moduł testGenerator (patrz2.1.4)
```

2.1.2 KDTree (visualization.ipynb)

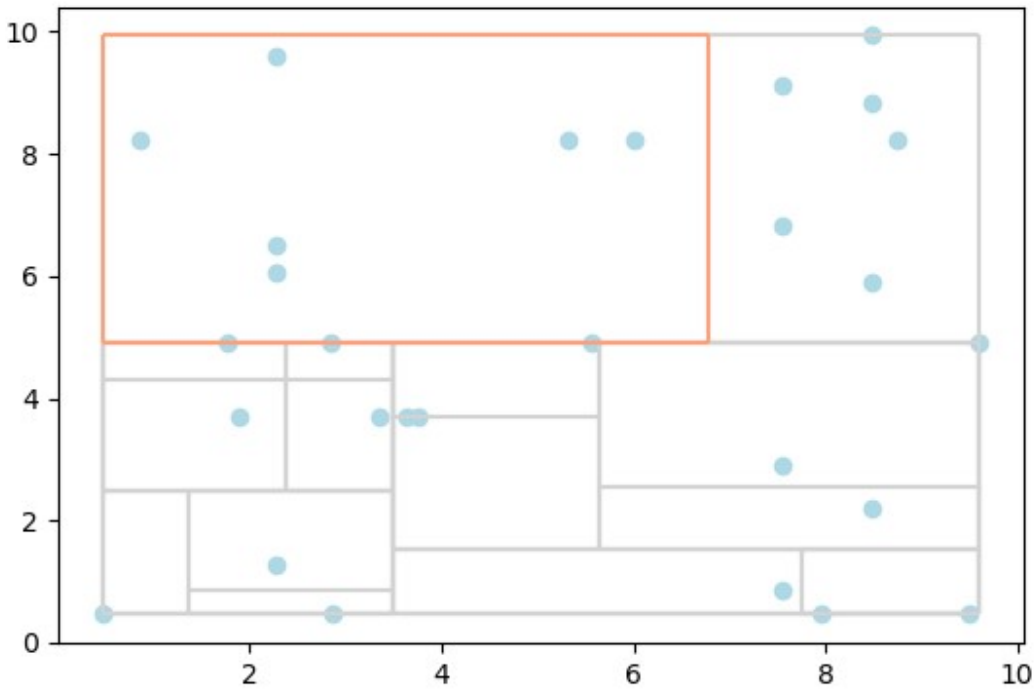
2.1.2.1 Stworzenie drzewa

```
#aby zbudować i zwizualizować budowę KDTree należy

#stworzyć nowy wizualizator typu KDTreeVisualizer

visualizer = KDTreeVisualizer(linear)

#do konstruktora KDTree_v przekazać zbiór punktów oraz stworzony wizualizator

kd_tree = KDTree_v(linear, visualizer)

#dostęp do scen wizualizacji można otrzymać odwołując się do visualizer.scenes

plot = Plot(scenes = visualizer.scenes)

plot.draw()
```

```
#do wizualizacji można użyć również funkcji pomocniczej create_KDTree_visualization(P)

#funkcja przyjmuje zbiór punktów P i zwraca krotkę: (zbudowane drzewo, sceny możliwe
do zwizualizowania)

kd_tree, scenes = create_KDTree_visualization(linear)

plot = Plot(scenes = scenes)

plot.draw()
```

Wizualizacja

Przykładowa scena z wizualizacji:



Objaśnienia kolorów wizualizacji:

- **jasnoniebieski** – wszystkie punkty,
- **łososiowy** – granice obszaru aktualnego poddrzewa,
- **jasnoszary** – granice obszarów w wcześniej rozpatrzonych poddrzewach.

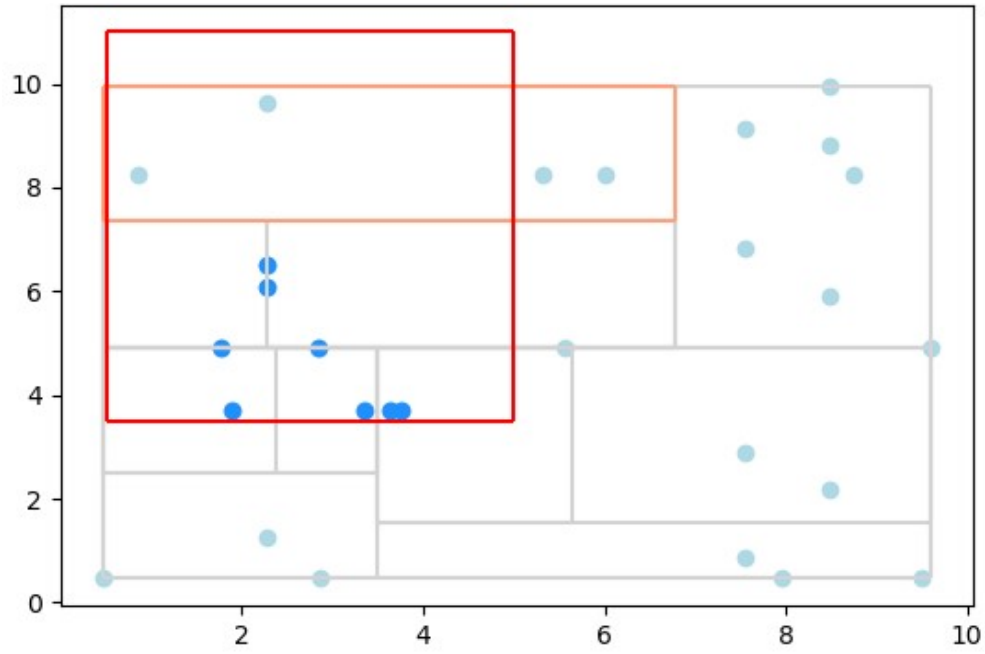
2.1.2.2 Zapytanie o punkty z zadanego przedziału

```
#aby zbudować i zwizualizować budowę KDTree należy
#stworzyć nowy wizualizator typu KDTreeVisualizer
visualizer = KDTreeVisualizer(linear)
#do konstruktora KDTree_v przekazać zbiór punktów oraz stworzony wizualizator
kd_tree = KDTree_v(linear, visualizer)
#dostęp do scen wizualizacji można otrzymać odwołując się do visualizer.scenes
plot = Plot(scenes = visualizer.scenes)
plot.draw()
```

```
#do wizualizacji można użyć również funkcji pomocniczej find_points_visualization(P, x1,x2,y1,y2)
#funkcja przyjmuje zbiór punktów P i granice szukanego przedziału
found_points, scenes = find_points_visualization(linear,0.5,5,3.5,11)
#funkcja zwraca znalezione punkty oraz sceny możliwe do zwizualizowania
plot = Plot(scenes=scenes)
plot.draw()
#funkcja dodaje do wizualizacji kilka końcowych scen:
#koloruje na granatowo wszystkie znalezione punkty
#usuwa szare krawędzie KD drzewa reprezentującego zbiór punktów
```

Wizualizacja

Przykładowa scena z wizualizacji:



Objaśnienia kolorów wizualizacji:

- **jasnoniebieski** – wszystkie punkty,
- **niebieski** – znalezione punkty z zadanego obszaru,
- **czerwony** – zadany do przeszukania obszar,
- **łososiowy** – granice obszaru aktualnie przeszukiwanego poddrzewa,
- **jasnoszary** – granice obszarów w wcześniej rozpatrzonych poddrzewach.

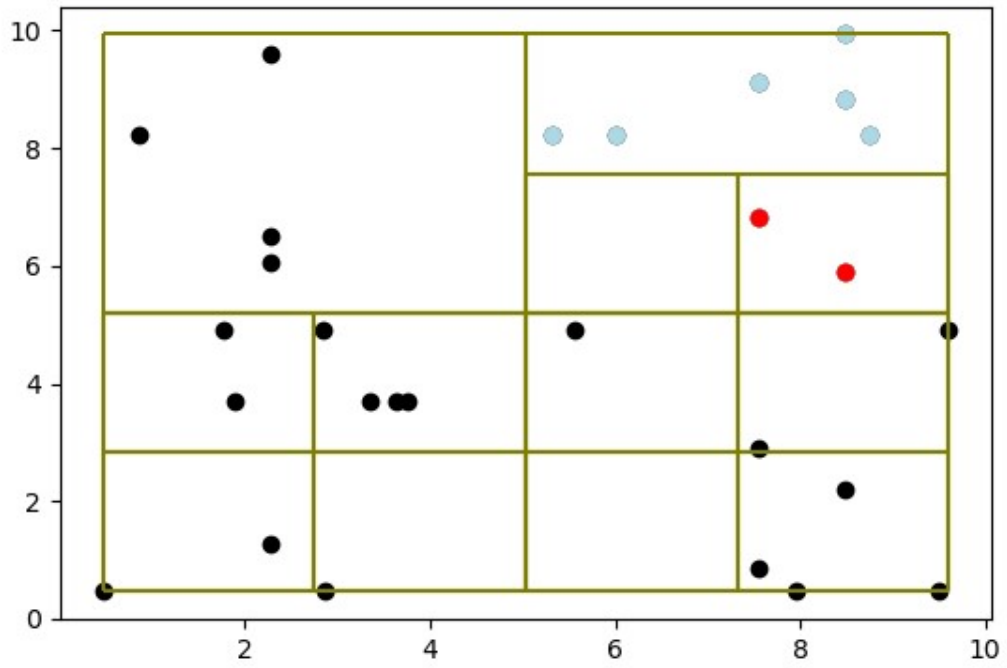
2.1.3 QuadTree (visualization.ipynb)

2.1.3.1 Stworzenie drzewa

```
#aby zbudować i zwizualizować budowę QuadTree należy

#stworzyć tablicę obiektów typu Point

linear_p = []

for i in linear:

    linear_p.append(Point(i[0],i[1]))

#stworzyć nowy obiekt typu QuadTree

#do konstruktora KDTree przekazać tablicę obiektów typu Point oraz  maksymalną liczbę punktów w liściu

QT = QuadTree_v(linear_p, 4)

#aby odwołać się do scen do zwizualizowania należy odwołać się do atrybutu sc obiektu QuadTree

plot = Plot(scenes = QT.sc)

plot.draw()
```

Wizualizacja

Przykładowa scena z wizualizacji:



Objaśnienia kolorów wizualizacji:

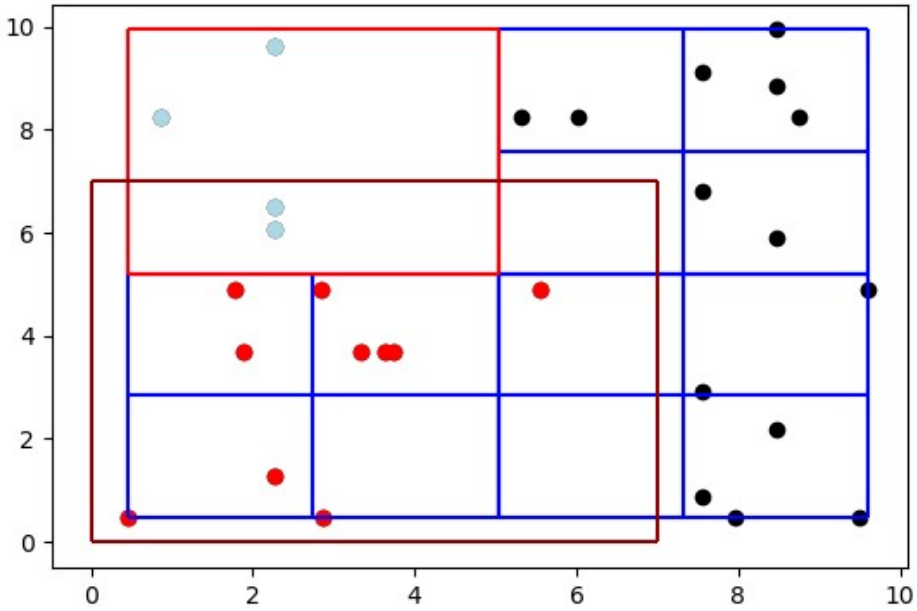
- **czerwony** – punkty aktualnie przetwarzane
- **czarny** – punkty w przedziale rodzica danego poddrzewa
- **szary** – pozostałe punkty
- **pomarańczowy** – linie podziałów w drzewie, jeżeli aktualnie jesteśmy w lewym dolnym poddrzewie
- **oliwkowy** – linie podziałów w drzewie, jeżeli aktualnie jesteśmy w prawym dolnym poddrzewie
- **różowy** – linie podziałów w drzewie, jeżeli aktualnie jesteśmy w lewym górnym poddrzewie
- **cyan** – linie podziałów w drzewie, jeżeli aktualnie jesteśmy w prawym górnym poddrzewie
- **niebieski** – linie podziałów w drzewie, jeżeli aktualnie wracamy z rekurencją

2.1.3.2 Zapytanie o punkty z zadanego przedziału

```
#aby zbudować i zwizualizować wyszukiwanie punktów z zadanego przedziału w KDTree należy

#stworzyć punkt rec_lower_left, czyli lewy dolny wierzchołek prostokąta w którym szukamy punktów,

#oraz analogicznie rec_upper_right, czyli prawy górny wierzchołek prostokąta w którym szukamy punktów,

rec_lower_left = Point(0.0,0.0)

rec_upper_right = Point(7.0,7.0)

#funkcja find_points odpowiada za znajdowanie punktów w zadanym prostokącie

#przyjmuje punkty rec_lower_left i rec_upper_right

found_points, sc = QT.find_points(rec_lower_left, rec_upper_right)

#zwraca tablicę z znalezionymi punktami i sceny, które można zwizualizować

plot = Plot(scenes = sc)

plot.draw()
```

Wizualizacja

Przykładowa scena z wizualizacji:



Objaśnienia kolorów wizualizacji:

- **czerwony** – punkty dodane do wyniku
- **szary** – punkty w danym węźle
- **czarny** – pozostałe punkty
- **czerwony** – linie obszaru aktualnie przetwarzanego węzła
- **bordowy** – linie obszaru zadanego do przeszukania
- **zielony** – linie obszaru aktualnie przetwarzanego węzła, jeżeli jest liściem
- **żółty** – linie obszaru aktualnie przetwarzanego węzła, jeżeli cały zawiera się w przeszukiwanym obszarze
- **niebieski** – linie pozostałych obszarów

2.1.4 Generowanie zbiorów danych (testGenerator.ipynb)

Dostępne funkcje:

- plot_to_points(plot)

Zwraca punkty wprowadzone myszką. Punkty są zwrócone w postaci tablicy dwuelementowych krotek zawierających współrzędne x, y.

plot – płótno, na które zostały wprowadzone punkty to zapisania

- file_to_points(file_points)

Zwraca listę punktów odczytanych do pliku. Punkty są zwrócone w postaci tablicy dwuelementowych krotek zawierających współrzędne x, y.

file_points – ścieżka do pliku

- points_to_file(points, name)

Zapisuje punkty do pliku.

points – tablica dwuelementowych krotek zawierających współrzędne x, y

name – ścieżka do pliku, w którym należy zapisać punkty.

- generate_random(n, ll, ru)

Generuje losowo n punktów z zadanego obszaru. Punkty są zwrócone w postaci tablicy dwuelementowych krotek zawierających współrzędne x, y.

n – liczba punktów do wygenerowania

ll – lewy dolny punkt obszaru, z którego generowane są punkty

ru – prawy górny punkt obszaru, z którego generowane są punkty

- generate_square_diagonal(nside, ndiagonal, ll, ru)

Generuje losowo n punktów na bokach i przekątnych zadanego kwadratu. Włącza w to wierzchołki. Punkty są zwrócone w postaci tablicy dwuelementowych krotek zawierających współrzędne x, y.

nside – liczba punktów do wygenerowania na bokach

ndiagonal - liczba punktów do wygenerowania na przekątnych

ll – lewy dolny punkt obszaru, z którego generowane są punkty

ru – prawy górny punkt obszaru, z którego generowane są punkty

- generate_square(size, ll, ru)

Generuje losowo n punktów na bokach zadanego kwadratu. Punkty są zwrócone w postaci tablicy dwuelementowych krotek zawierających współrzędne x, y.

nsize – liczba punktów do wygenerowania

ll – lewy dolny punkt obszaru, z którego generowane są punkty

ru – prawy górny punkt obszaru, z którego generowane są punkty

- generate_linear(size, nlines, ll, ru)

Generuje n punktów na k pionowych bądź poziomych liniach. Punkty są zwrócone w postaci tablicy dwuelementowych krotek zawierających współrzędne x, y.

size – liczba punktów do wygenerowania

nlines – liczba linii, na których generowane będą punkty

ll – lewy dolny punkt obszaru, z którego generowane są punkty

ru – prawy górny punkt obszaru, z którego generowane są punkty

Interfejs zawiera przykłady generowania zbiorów danych, zapisywania do pliku, odczytywania z pliku oraz wprowadzania własnego zbioru za pomocą myszki.

3 Sprawozdanie

3.1 Opis problemu

Dane: zbiór punktów P na płaszczyźnie; x_1, x_2, y_1, y_2

Zapytanie: dla zadanych x_1, x_2, y_1, y_2 znaleźć punkty q ze zbioru P takie,

że $x_1 \leq q_x \leq x_2,\ \ y_1 \leq q_y \leq y_2$.

Celem projektu jest zaimplementowanie odpowiednich struktur danych – QuadTree oraz KDTree, które pozwalają szybko odpowiadać na takie zapytania.

Istotne są analiza i porównanie algorytmów!

3.2 Wykonane testy

Wszystkie zbiory zostały wygenerowane wewnątrz kwadratu 1000x1000 za pomocą funkcji random.uniform() oraz random.choice().

Zostały przygotowane następujące zbiory do testowania:

3.2.1 random_1eX

testy losowe , X oznacza ilość punktów. Łącznie stworzyliśmy 5 takich testów.

3.2.2 linear_X_Y

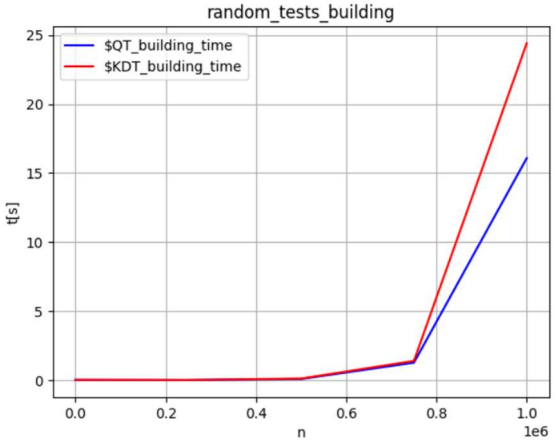
testy typu liniowego w których punkty leżą na pionowych i poziomych liniach, których jest łącznie Y, punktów jest X. Łącznie stworzyliśmy 4 takie testy.

3.2.3 square_diagonal_X

testy w których punkty leżą na dwóch bokach prostokąta a także na obu przekątnych, X oznacza osobno ilość punktów na bokach oraz osobno na przekątnych. Łącznie stworzyliśmy 4 takie testy.

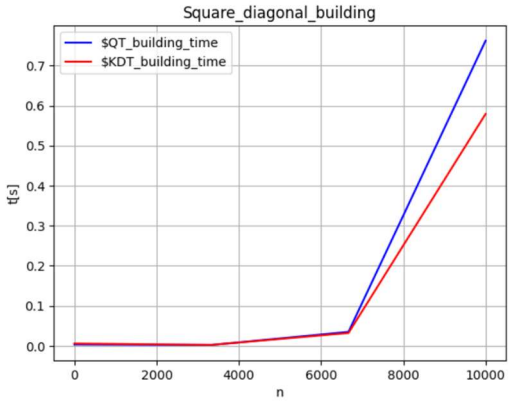
3.3 Wyniki pomiaru czasu budowy struktur

3.3.1 Wyniki random_1eX



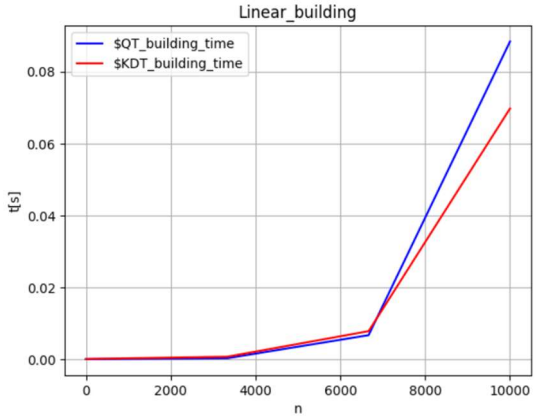
	collection	QT building time [s]	KDT building time [s]
0	random_1e2	0.015739	0.018770
1	random_1e3	0.005139	0.010189
2	random_1e4	0.083910	0.112884
3	random_1e5	1.264895	1.391615
4	random_1e6	16.080887	24.406131

3.3.2 Wyniki diagonal_X



	collection	QT building time [s]	KDT building time [s]
0	square_diagonal_30	0.003463	0.005811
1	square_diagonal_100	0.002216	0.002374
2	square_diagonal_1000	0.034962	0.032159
3	square_diagonal_10000	0.761832	0.579034

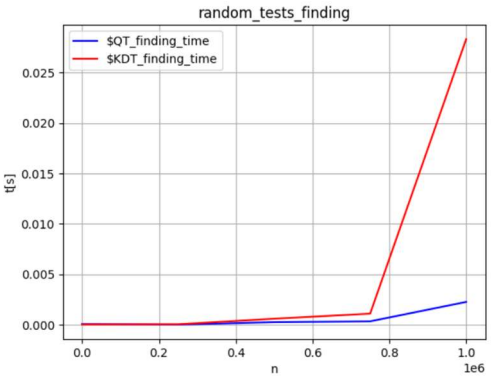
3.3.3 Wyniki linear_X_Y



	collection	QT building time [s]	KDT building time [s]
0	linear_30_10	0.000117	0.000117
1	linear_100_10	0.000382	0.000382
2	linear_1000_10	0.006792	0.006792
3	linear_10000_10	0.088338	0.088338

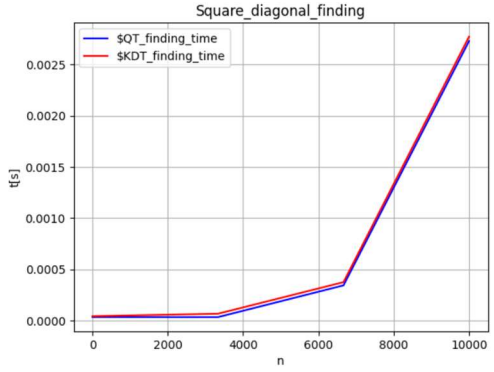
3.4 Wyniki pomiaru czasu przeszukiwania struktur

3.4.1 Wyniki random_1eX



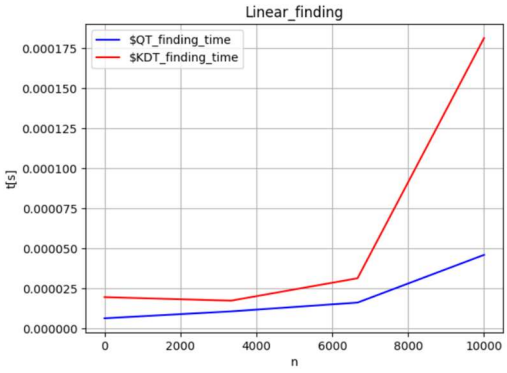
	collection	QT finding time [s]	KDT finding time [s]
0	random_1e2	0.000067	0.000033
1	random_1e3	0.000019	0.000052
2	random_1e4	0.000263	0.000602
3	random_1e5	0.000343	0.001104
4	random_1e6	0.002266	0.028310

3.4.2 Wyniki diagonal_X



	collection	QT finding time [s]	KDT finding time [s]
0	square_diagonal_30	0.000034	0.000043
1	square_diagonal_100	0.000034	0.000067
2	square_diagonal_1000	0.000344	0.000375
3	square_diagonal_10000	0.002729	0.002772

3.4.3 Wyniki linear_X_Y



	collection	QT finding time [s]	KDT finding time [s]
0	linear_30_10	0.000006	0.000006
1	linear_100_10	0.000011	0.000011
2	linear_1000_10	0.000016	0.000016
3	linear_10000_10	0.000046	0.000046

3.5 Wnioski

Na podstawie przeprowadzonych testów i wizualizacji możemy stwierdzić, że zarówno budowanie struktur QuadTree i KDTree i wyszukiwanie punktów z zadanych obszarów działa poprawnie. Zbiory znalezionych punktów były identyczne w przypadku obu struktur.

Podobieństwo QuadTree i KDTree opiera się na idei podziału obszaru na mniejsze części. Jednakże struktury te znacznie się różną. Idea KDTree jest łatwo rozszerzalna to wyższych wymiarów, natomiast QuadTree wyróżnia łatwość dodawania nowych punktów do struktury.

W porównaniu czasów budowania struktury dla zbioru losowo wybranych punktów z zadanego obszaru lepszy wynik uzyskało QuadTree. Jednak KDTree uzyskało lepszy czas dla specyficznych zbiorów postaci punktów wybranych na bokach i przekątnych kwadratu oraz punktów wybranych z poziomych i pionowych prostych.

W porównaniu czasów znajdowania punktów z zadanego obszaru lepsze wyniki uzyskało QuadTree, jednak w przypadku punktów na bokach i przekątnych kwadratu czasy te były bardzo zbliżone.

Podsumowując, QuadTree okazało się szybsze w większości przetestowanych przypadków, jednak dla pewnych specyficznych przypadków to KDTree osiągnęło mniejszy czas.