Łukasz Dubiel
Paweł Kruczkiewicz

# Projekt

Temat nr 7 - Wyszukiwanie geometryczne –przeszukiwanie obszarów ortogonalnych, quadtree oraz kd-drzewa

## 1. Cel ćwiczenia

Celem projektu jest zaimplementowanie struktur kd-tree oraz quad tree oraz porównanie ich efektywności w wyszukiwaniu podzbioru punktów na płaszczyźnie należących do zadanego obszaru prostokątnego.

Projekt znajduje się w repozytorium: https://github.com/pkrucz00/projektGeometryczne.git

## 2. Implementacja Quad Tree

Implementacja quad tree została dokonana w oparciu o rozdział dotyczący tego zagadnienia z książki „Computional Geometry", *Mark de Berg, Otfried Cheng, Marc van Kreveld, Mark Overmars,* opis tej struktury i pseudokod z https://en.wikipedia.org/wiki/Quadtree oraz informację z wykładu. Znajduje się ona w module ***quadTree*** w kodzie programu

### Szczegóły implementacji

Implementacja jest utrzymana w konwencji obiektowej, składają się na nią następujące klasy:

- **Point**, której obiekty przechowują współrzędne xy punktu, zawiera podstawowe metody jak __hash__, czy __str__,

- **Rect**, której obiekty przechowują współrzędne x1, x2, y1, y2, ograniczające prostokąt o bokach równoległych do odpowiednich osi, zawiera ona następujące metody:

  - ***contains(self, p)***, która przyjmuje obiekt klasy Point i zwraca wartość *bool*, określającą, czy punkt ten należy do obszaru,
  - ***include(self, rect),*** która przyjmuje jako argument obiekt klasy Rect i sprawdza czy odpowiadający mu obszar jest podzbiorem obszaru odpowiadającego obiektowi, na którym ta metoda jest wywoływana,
  - ***getSplittingLines(self),*** zwraca listę linii (par wierzchołków, jako tuple) dzielących dany obszar na cztery przystające, (używane w wizualizacji),
  - ***getListOfSides(self)***, jak wyżej, ale do boków danego prostokąta,
  - ***normalizeToSquare(self)***, modyfikuje współrzędne x1, x2, y1, y2, tak by obszar wyznaczany przez nie był minimalnym kwadratem zawierającym poprzedni obszar
  -

- **Quad**, która jest klasą opakowującą dla klasy Rect

- **QTNode**, której obiekty są węzłami drzewa, przechowują one obiekt klasy Quad (.quad), określające obszar, z którym koresponduje dany węzeł, dzieci (.SE, SW, .NW, .NE), jeśli węzeł nie jest liściem oraz listę punktów (.points) jeżeli jest liściem lub nie jest, a drzewo skonstruowane jest z parametrem umożliwiającym przechowywanie referencji na punkty również w węzłach wewnętrznych, w celu np. szybszego wyszukiwania (kosztem pamięci). Metody tej klasy są metodami pomocniczymi dla metod klasy QuadTree:

    - ***__init__(self, quad, isLeaf, points)*** – konstruktor przyjmuję obiekt klasy Quad, wartość bool, określającą, czy węzeł ma być liściem (domyślnie False), listę punktów (obiektów klasy Point) (domyślnie None)
    - ***splitQTNode(self, P, nodeCapacity, nodeContainsPoints)*** – metoda tworzy dzieci danego węzła odpowiednio dzieląc punkty (obiekty klasy Point) z argumentu P, nodeContainsPoints określa, czy węzeł, na którym wykonywana jest metoda ma przechowywać punkty,
    - ***getDepth(self)*** – zwraca głębokość poddrzewa danego węzła,
    - ***_search(self, queryRect, repPoints)*** – metoda pomocnicza dla metody ***serach*** z klasy QuadTree,
    - ***reportAllRec(self, repPoints)*** – metoda przeszukuje poddrzewo dane węzła i dodaje do listy repPoints punkty znalezione w liściach,
    - ***instantReportAll(self)*** – zwraca punkty przechowywane w węźle lub None jeżli drzewo nie przechowuje punktów w węzłach wewnętrznych,
    - ***_searchWithConsiInclusions(self, queryRect, repPoints)*** – metoda pomocnicza dla metody ***seachInRangeWithoutConsiInclusion*** z klasy QuadTree,

- **QuadTree**, której obiekty-drzewa mogą być konstruowane z listy punktów, wtedy automatycznie zostaje wyznaczony minimalny kwadrat zawierający wszystkie punkty, który stanowi obszar root-a lub/i z zadanym obszarem root-a (obiektem **Rect**). W konstruktorze można zadać by drzew mogło przechowywać duplikaty, by w każdym węźle przechowywana była lista punktów zawartych w obszarze węzła oraz można określić ilość punktów jaką może przechowywać węzeł. Klasa zawiera metodę do wyszukiwania punktów w danym obszarze prostokątnym, a także dodawania i usuwania punktów z drzewa, metody:

    - ***__init__(self, P, quad0, nodeCapacity, eachNodeContainsPoints, allowDubplicate)***, P – lista punktów, jako obiektów klasy Point lub tupli, bądź listy współrzędnych, quad0 – obszar root-a (domyślnie None), nodeCapacity – maksymalna ilość punktów w jednym liściu (domyślnie 1), *eachNodeContainsPoints* – wartość bool określająca, czy w każdym węźle mają być przechowywane wierzchołki (domyślnie False), *allowDuplicate* – wartość bool, określa czy w drzewie mogą się pojawiać duplikaty punktów. **Złożoność: O(n(d+1)),** gdzie d to głębokość drzewa

- ***getDepth(self)*** – zwraca głębokość drzewa,
- ***searchInRange(self, queryRect)*** – zwraca listę wykrytych punktów, jako parametr przyjmuje obiekt klasy Rect,
- ***seachInRangeWithoutConsiInclusion(self, queryRec)*** – jak wyżej, ale w przeszukiwaniu zawsze zostaje osiągnięty poziom liścia,
- ***addPoint(self, p)*** – dodaje punkt do drzewa, przyjmuje obiekt klasy Point lub tuplę / listę współrzędnych, zwraca True, jeżeli procedura dodania punktu powiedzie się, w przeciwnym przypadku False,
- ***deletePoints(self, p)*** – usuwa punkt z drzewa, argument jak wyżej, zwraca True, jeżeli procedura usunięcia punktu powiedzie się, w przeciwnym przypadku False

- **QuadTreeVis**, której obiekt jest wizualizatorem dla danego obiektu QuadTree, będącego parametrem konstruktora, zawiera metody zwracające wizualizację (obiekty klasy Plot ( z narzędzia graficznego )) dla procesu konstrukcji drzewa oraz szukania punktów w zadanym zakresie:
  - ***getConstructionVis(self)*** – zwraca obiekt Plot dla wizualizacji konstrukcji drzewa,
  - ***getRangeQueryVis(self)*** – zwraca obiekt Plot dla wizualizacji przeszukiwania drzewa

Moduł **QuadTreeMain** zawiera przykładową wizualizację dla różnych typów danych.

W wizualizacji quadTree przyjęto następującą konwencję kolorów:

- dla konstrukcji quad tree:
  - linie niebieskie wytyczają obszary jakie już zostały zawarte w węzłach drzewa,
  - linie zielone wytyczają obszary aktualnie analizowane, w przypadku dzielenia obszaru na cztery – alokowaniu kolejnych węzłów-dzieci, są to linie dzielące, a w przypadku gdy analizowany obszar należy do liścia zielone są jego boki,
  - jasnoniebieskie są punkty jeszcze nie umieszczone w pewnym liściu,
  - czerwone są punkty, które zostały już umieszczone w odpowiednim liściu
  - szare linie (opcjonalne) wytyczają obszary w pełni skonstruowanym drzewie
- dla wyszukiwania punktów w zakresie:
  - linie niebieskie wytyczają obszary należące do węzłów już przeanalizowanych,
  - linie zielone wytyczają obszary, aktualnie analizowane, jeżeli zielone są boki pewnego kwadratu, to znaczy, że ten obszar jest rozłączny z danym zakresem lub jest w nim zawarty, ale to pod warunkiem, że w drzewie w każdym węźle przechowywane są punkty i wyszukiwanie może od razu raportować te punkty bez schodzenia w głąb drzewa. Jeżeli zielone są linie podziału to znaczy, że w dalszych krokach program analizuje kolejne dzieci węzła,
  - jasno niebieskie są punkty nie zaklasyfikowane jako należące do zadanego obszaru,
  - czerwony są punkty, które zostały zaklasyfikowane jako należące do zadanego obszaru

- szare linie (opcjonalne) wytyczają obszary korespondujące z węzłami nie odwiedzonymi przy przeszukiwaniu

# 3. Implementacja KD-Tree

Implementacja została opracowana na podstawie książki „Computional Geometry" autorów Mark de Berg, Otfried Cheng, Marc van Kreveld, Mark Overmars oraz wykładu

## Szczegóły implementacji

Moduł zawiera 3 pliki: **kdtreeAuxClasses.py, kdtree.py, kdtreevis.py**

1. **kdtreeAuxClasses.py** implementuje 3 klasy pomocnicze dla KD-Tree:
    1. **Range –** obiekty tej klasy reprezentują prostokąt. Wykorzystuje się go przy przeszukiwaniu drzewa. Obszar, z którego punkty chcemy wydobyć, reprezentowany jest jako obiekt klasy Range. Obszar, za który „odpowiedzialny" jest dany węzeł w drzewie, również reprezentowany jest obiektem tej klasy.
    (**UWAGA nr 1**: oba te użycia różnią się nieznacznie tym, że obszar przeszukiwania jest domknięty, a obszar węzła jest domknięty na dolnej i prawej linii, lecz otwarty na górnej i lewej. Zostało to uwzględnione w metodach **isContainedIn** oraz **intersects** opisanych niżej)
    Klasa ta zawiera atrybuty:
        - **x1, x2, y1, y2 –** współrzędne linii, którymi ograniczony jest prostokąt (**UWAGA nr 2:** dla uproszczenia x1 <= x2 oraz y1 <= y2)

    **Metody:**
    - **isContainedIn(self, other)** – sprawdza, czy dany prostokąt zawiera się w podanym (uwzględnia uwagę nr 1)
        - Zwraca wartość typu **bool**
    - **isPointInRange(self, point) –** sprawdza, czy podany punkt zawiera się w prostokącie
        - Zwraca wartość typu **bool**
    - **intersects(self, other) –** sprawdza, czy obiekt Range przecina się z danym (uwzględnia uwagę nr 1)
        - Zwraca wartość typu **bool**
    - **returnSplit(self, axis, line)** - na podstawie danej linii przecięcia oraz numeru osi zwraca lewy i prawy prostokąt nią przecięty
        - Parametry: **self –** obiekt Range, **axis** – nr osi (0 – oś x, 1 – oś y), **line –** współrzędna linii
        - Zwraca dwa obiekty typu **range –** lewe przecięcie i prawe przecięcie
    - **returnSplit(self, axis, line)** - na podstawie danej linii przecięcia oraz numeru osi zwraca lewy i prawy prostokąt nią przecięty

    **Metody pomocnicze:**
    - **__str__(self)** – zwraca łańcuchową reprezentację obiektu
    - **__copy__(self) –** zwraca kopię obiektu

- **__corrdValidation(self) –** sprawdza, czy warunek z uwagi 2 jest spełniony

**b. LeafNode -** liść KD-tree - przechowuje jeden punkt (klasę tę można pominąć w implementacji KD-tree, jednak pozwala ona lepiej zwizualizować KD-drzewo) Atrybuty:

- **point –** tupla z koordynatami punktu, który przechowuje liść

**c. Node –** węzeł KD-tree. Atrybuty:

- **splitCoord –** współrzędna podziału (**Uwaga 3:** nie zapisujemy, której osi współrzędnych dotyczy współrzędna, ponieważ przechodząc drzewo, można wydedukować to na podstawie ich głębokości)
- **left –** wskaźnik na kolejny węzeł bądź liść zawierający punkty o mniejszej bądź równej współrzędnej na odpowiedniej osi niż wartość wskazana przez linię podziału
- **right –** jak wyżej, lecz współrzędne punktu/punktów przechowywanych przez wskazany liść/węzeł są większe

**2. kdtree.py –** główny plik modułu zawierający jedynie jedną klasę – implementację KD-tree.

**a. KDTree –** dokładny opis znajduje się w prezentacji

**Atrybuty:**

- **maxRange –** obiekt klasy Range, który przechowuje najmniejszy prostokąt zawierający wszystkie dane punkty.
- **kdTreeRoot –** wskaźnik na pierwszy węzeł KD-Tree

**Metody publiczne (interfejs):**

- **Konstruktor –** przyjmuje listę punktów. Tworzy dwie listy na bazie danej listy punktów – jedną posortowaną wg współrzędnej x oraz drugą po współrzędnej y – i na ich podstawie oblicza maxRange oraz buduje drzewo (szczegóły w prezentacji)
  - **ZŁOŻONOŚĆ: O(nlogn)**
- **printTree(self) –** wypisuje drzewo (funkcja służąca do debugowania programu)
- **search(self, searchRange,** node=None, nodeRange=None, depth=0**)** – rekurencyjna funkcja, wyszukująca wszystkie punkty znajdujące się w zadanym przedziale
  - **Parametry:**
    - **searchRange –** obiekt typu Range, reprezentujący zadany przez użytkownika przedział;
    - **node** (domyślna wartość: None) **–** obecnie przetwarzany węzeł/liść drzewa. Jeśli node jest Nonem, to przypisuje się mu wartość self.kdTreeRoot;
    - **nodeRange** (domyślna wartość – None) – przedział, za który „odpowiada” dany węzeł
    - **depth** (domyślna wartość – 0) – obecna głębokość drzewa
  - **Zwraca:** listę punktów zawartych w zadanym przedziale
  - **ZŁOŻONOŚĆ: O(sqrt(n) + k)** gdzie k to liczba znalezionych punktów

**Metody prywatne i chronione (pomocnicze):**

- **__initAux(self, pointsXSorted, pointsYSorted, depth=0)** – rekurencyjna funkcja tworząca KD-tree. Dokładny opis działania znajduje się w prezentacji
  - **Parametry:**
    - **pointsXSorted, pointsYSorted** – punkty posortowane względem współrzędnej x i y (**UWAGA 4** – obie listy przechowują te same punkty, lecz w różnej kolejności)
    - **depth** (domyślna wartość – 0) – obecna głębokość drzewa
  - **Zwraca:**
    - **Obiekt klasy LeafNode** jeżeli dane tablice były jednoelementowe (zwracany obiekt zawiera zawarty w tablicach punkt)
    - **Obiekt klasy Node** w przeciwnym wypadku, zawierający linię przecięcia, lewy oraz prawy węzeł/liść

- **_split(self, x_sorted, y_sorted, axis) –** funkcja obliczająca punkt przecięcia dwóch tablic i dzieląca je wg tej granicy.
  - **Parametry:**
    - **x_sorted, y_sorted** – posortowane punkty odpowiednio wg x i y
    - **axis** – numer osi przecięcia
  - **Zwraca:**
    - Wartości tablicy **result –** 4 listy wynikowe podziału danych tablic wg obliczonej granicy
    - **splitPointCoordinate –** obliczona granica podziału
  - **ZŁOŻONOŚĆ: O(n)**
- **__findMaxRange(self, pointsXSorted, pointsYSorted)–** funkcja zwracająca atrybut maxRange na bazie dwóch posortowanych tablic.
- **_reportSubtree(self, node)** – funkcja zwracająca listę punktów znajdujących się w liściach danego węzła KD-tree

**3. kdtreevis.py –** plik odpowiedzialny za wizualizację inicjalizacji oraz przeszukiwania KD-tree. Zawiera 2 klasy:

**a. Visualizer –** wizualizator drzewa, zapamiętujący jego obecny stan tak, aby mozna było w łatwy sposób stworzyć sceny do narzędzia graficznego oraz na koniec je zwrócić.

**Atrybuty:**

- **__colors –** słownik zawierający kolory używane przy wizualizacji (atrybut prywatny)
- **setOfPoints** – początkowy zbiór punktów
- **maxRange –** przechowuje atrybut maxRange obiektu KDTreeVis (opisanego wyżej w atrybutach KDTree). Domyślnie: None
- **searchRange –** przechowuje obiekt klasy Range reprezentujący obszar przeszukiwania KD- tree. Domyślnie: None
- **lines –** linie podziału, przechowywane w węzłach obiektu klasy KDTreeVis. Domyślnie: pusta lista
- **reportedPoints** – punkty znalezione w trakcie przeszukiwania drzewa. Domyślnie: pusta lista

- **initScenes –** sceny pokazujące inicjalizację KD- tree. Domyślnie: scena pokazująca jedynie punkty z parametru **setOfPoints**
- **searchScenes** – sceny pokazujące przeszukiwanie KD-tree. Domyślnie: pusta lista

**Metody publiczne (interfejs):**

- **konstruktor –** przyjmuje listę punktów, zwraca obiekt klasy Visualizer
- **setMaxRange(self, maxRange)** – ustawia atrybut maxRange na bazie danego parametru
- **setSearchRange(self, searchRange)** – ustawia atrybut searchRange na bazie danego parametru
- **addLine(self, splitCoord, smallerBound, biggerBound, axis)** – dodaje linie przecięcia na bazie współrzędnej przecięcia (**splitCoord**), osi (**axis**), oraz granic przedziału (**smallerBound, biggerBound)**
- **addPoint(self, point)** – dodaje punkt do reportedPoints
- **makeScene(self, currPoints=None, currRange=None)** – tworzy nową scenę na bazie danych parametrów oraz własnych atrybutów oraz dodaje ją do odpowiedniego atrybutu (**initScenes** lub **searchScenes**)
    - **Parametry:**
        - **currPoints –** obecnie przetwarzane punkty (w czasie inicjalizacji)
        - **currRange –** obecnie analizowany obszar (w czasie wyszukiwania)
- **getInitScenes(self)** – zwraca sceny inicjalizacji
- **getSearchScenes(self) –** dodaje ostatnią scenę (jedynie punkty wraz z zaznaczonymi odnalezionymi punktami oraz przeszukiwany obszar) do scen przeszukiwania i je zwraca.
- **clear(self)** – ustawia wartości domyślne dla atrybutów tak, aby można było znów zwizualizować przeszukiwanie KD-tree

**Metody prywatne (pomocnicze):**

- **__getRangeLines(self,rangeObject) –** zwraca listę odcinków danego prostokąta (reprezentowanego jako obiekt klasy Range) lub pustą listę jeśli parametr jest wartością None

**b. KDTreeVis –** klasa dziedzicząca klasy KDTree. Zasadniczo różni się tym, że zawiera dodatkowe pole **vis**, zawierające obiekt klasy Visualizer potrzebny do graficznego przedstawienia działania. Większość klas z macierzystej klasy została nadpisana w taki sposób, aby obiekt klasy Visualizer zapisywał sceny przedstawiające najważniejsze etapy działania algorytmu inicjalizacji i przeszukiwania. Poza tym owe metody przyjmują, działają i zwracają to samo, co metody przez nie nadpisywane.

Do tych metod należą:

- **Konstruktor** – posiada stworzenie nowego obiektu Visualizer i przypisanie go do atrybutu **vis.** Dodaje obiekt MaxRange do **self.vis**
- **__initAux(self, pointsXSorted, pointsYSorted, depth=0)** – tworzy scenę przed podziałem punktów (zaznaczając wszystkie punkty zawarte w drzewie danego węzła) oraz po podziale, z dorysowaniem linii.

- **search(self, searchRange, node=None, nodeRange=None, depth=0)** -metoda ta:
    1. w przypadku pierwszego zejścia rekurencyjnego ustawia w wizualizatorze obszar wyszukiwania (metodą **setSearchRange**) oraz dodaje pierwszą scenę
    2. w przypadku liścia dodaje punkt do wizualizatora i dodaje nową scenę
    3. w przypadku węzła dodaje sceny z zaznaczeniem punktów po odpowiednim podziale, jeżeli przedział węzła zawiera punkt wspólny z przeszukiwanym przedziałem
- **__reportSubtree(self, node)** - metoda ta znajduje znalezione punkty do wizualizatora

Ponadto w owej klasie występują metody rozszerzające nadrzędną klasę. Są one związane przede wszystkim z obsługą wizualizatora:

- **getInitPlot(self)** – zwraca obiekt typu Plot na bazie scen inicjalizacji z wizualizatora
- **getSearchPlot(self)** - zwraca obiekt typu Plot na bazie scen przeszukiwania z wizualizatora
- **clearVis(self)** – czyści sceny przeszukiwania z wizualizatora (**UWAGA 5 –** zaleca się używanie tej metody po każdym wywołaniu getSearchPlot, w przeciwnym wypadku obiekt Plot będzie zawierał kolejne sceny z różnych przeszukiwań)

W wizualizacji KD-Tree przyjęto następującą konwencję kolorów:

- punkty:
    - zielononiebieski – punkty z wejściowego zbioru
    - pomarańczowoczerwony – obecnie przetwarzane punkty. Przy inicjalizacji są to punkty, na których dokona się podziału bądź doda do liścia, a przy przeszukiwaniu – znajdujące się w analizowanym liściu
    - fuksja – znalezione punkty
- linie:
    - niebieski – linie podziału KD-Tree (UWAGA: długość linii podziału jest umowna. Na wizualizacji przedstawiono je jako odcinki od najmniejszej do największej współrzędnej (na osi przeciwnej do osi podziału) punktu w analizowanym zbiorze)
    - wygaszony turkus – odcinki prostokąta reprezentowanego przez obiekt KDTree.MaxRange
    - żółty – prostokąt przeszukiwania
    - czerwony – obszar, za który odpowiedzialny jest obecny węzeł

Do generowania danych użyta została klasa **testDataGenerator**, która zawiera metody służące do generowania danych różnych typów:

- ***inRect(self, n, x1, x2, y1, y2)*** - n punktów leżącyc wewnątrz prostokąta ograniczonego przez *x1, x2, y1, y2,*
- ***sinus(self, n, x1, x2, y1, y2, lapses, fi)*** - ***n*** punktów leżących na wykresie sinus-a mieszczącym się w prostokącie o podanych ograniczeniach, zawierającym lapses pełnych okresów i przesuniętym o ***fi*** *(radiany),*
- ***rectEdges(self, n, x1, x2, y1, y2)*** - ***n*** punktów znajdujących się na bokach danego prostokąta,

- ***rectDiagonals(self, n1, n2, x1, x2, y1, y2)*** - ***n*** punktów znajdujących się na przekątnych danego prostokąta,
- ***quadSplitLines(self, n1, n2, x1, x2, y1, y2)*** - odpowiednio **n1** i **n2** punktów znajdujących się na linii pionowej i poziomej, które dzielę prostokąt na cztery przystające,
- ***cirque(self, n, x, y, R)*** - ***n*** punktów na okręgu o podanym środku **(x, y)** i promieniu **R,**
- ***circle(self, n, x, y, R)*** - ***n*** punktów w kole o podanym środku **(x, y)** i promieniu **R,**
- ***archimedeanSpiral(self, n, x, y, maxR, lapses, direct)*** - ***n*** punktów leżących na spirali Archimedesa o podanym środku **(x, y)**, promieniu maksymalnym **maxR,** liczbie pełnych obiegów **laspes** oraz kierunku zgodnym z ruchem wskazówek zegara jeśli **direct=1** lub przeciwnym jeśli **direct=-1**

Do wizualizacji wykorzystano program mgr. inż Krzysztofa Podsiadło z modyfikacjami. Plik znajduje się w module **AuxFiles** pod nazwą **tool.py**

# 4. Część użytkownika

Główny program znajduję się w ProjectNotebook.ipynb. Dla każdej z badanych struktur zawiera testowanie, wizualizację konstrukcji i wyszukiwania dla prostych danych wpisanych ręcznie oraz oddzielnie wizualizację dla danych generowanych przy pomocy klasy testDataGenerator. W odpowiednich miejscach w programie zawarte są przykładowe wywołania funkcji generujących, z których jedną wybraną należy odkomentować i uruchomić program.

W plikach kdTreeMain.py oraz quadTreeMain.py znajdują się programy do wizualizacji konstrukcji i przeszukiwania dla odpowiednich struktur. Importują one odpowiednio kdTreeVis.py, quadTreeVis.py oraz testDataGenerator.py (zawierający klasę do generowania danych losowych określonego typu). W każdym pliku są napisane przykładowe wywołania funkcji generujących dane, z którymi należy postąpić jak opisano wyżej.

# 5. Sprawozdanie

Testy czasowe i analiza porównawcza obu struktur.

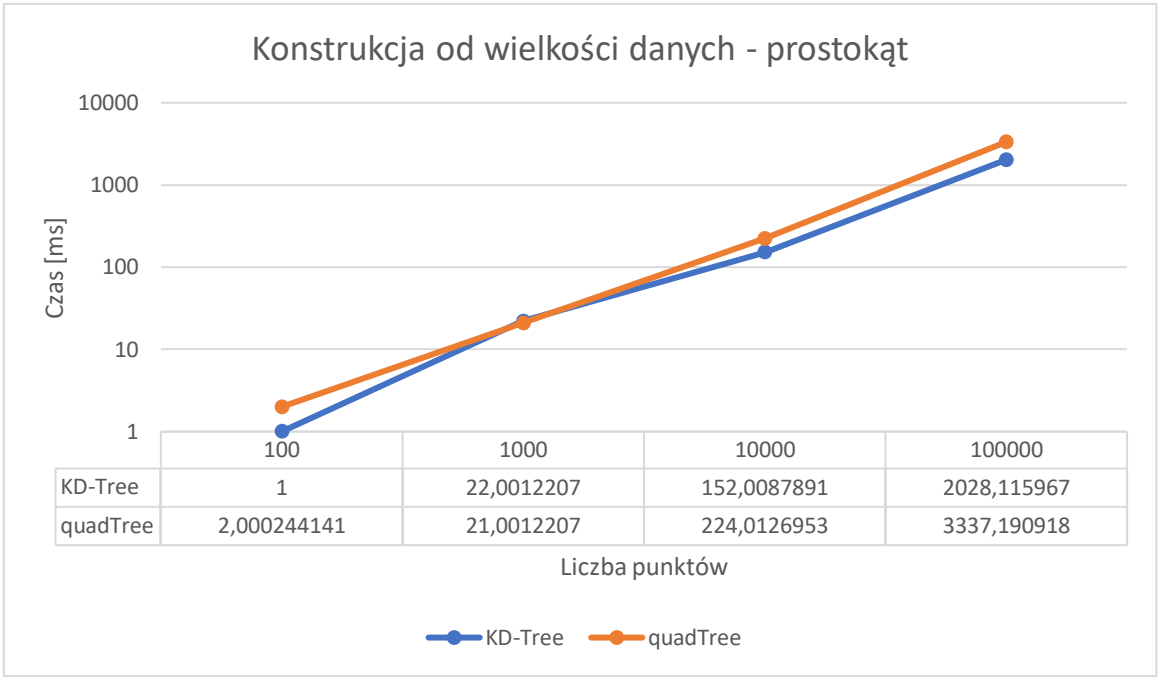
W tej części dokumentacji przedstawiono wyniki testów czasowych obu struktur. Kod testów dostępny jest w plikach **tests.py** oraz **testV2.py**

## Inicjalizacja

W tym doświadczeniu zbadano czas konstrukcji KD-Tree oraz Quadtree.
Obie struktury testowano dla różnych typów danych wejściowych.

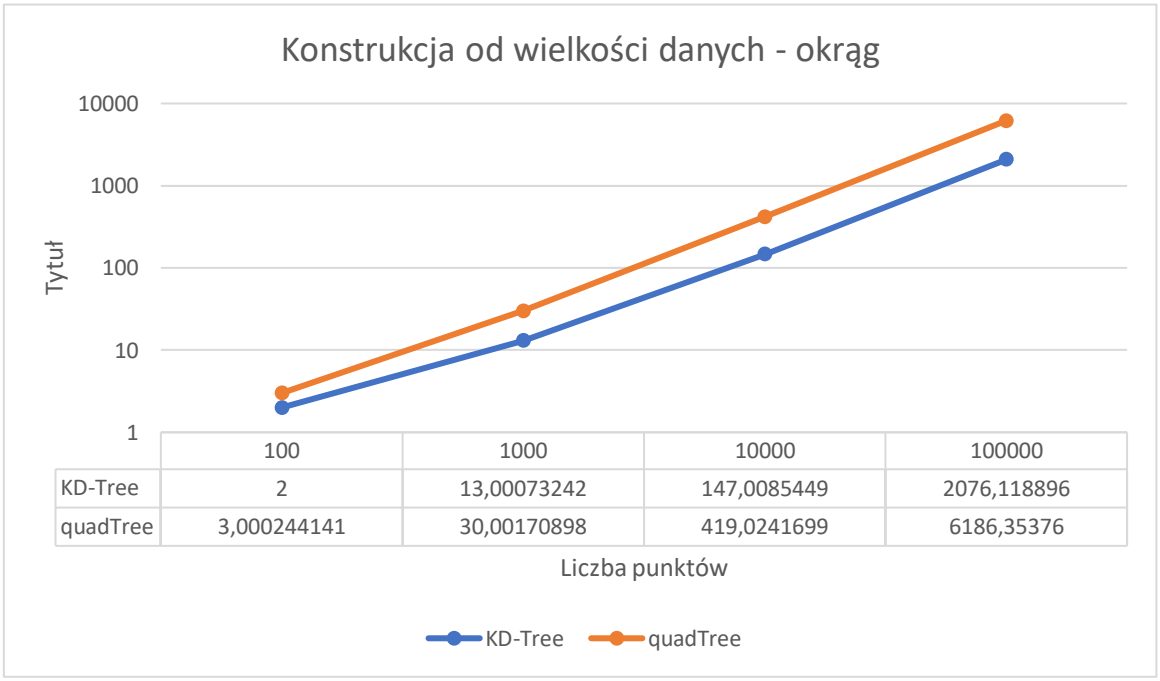
a) Punkty wylosowane na powierzchni prostokąta [-100, 100]x[-100, 100]
b) Punkty wylosowane na okręgu ( środek (0,0), promień 100 )
c) Przekątne kwadratu jak w punkcie a)

Wyniki przedstawiono na wykresach 1.-3. (**UWAGA** – zastosowano skalę logarytmiczną)



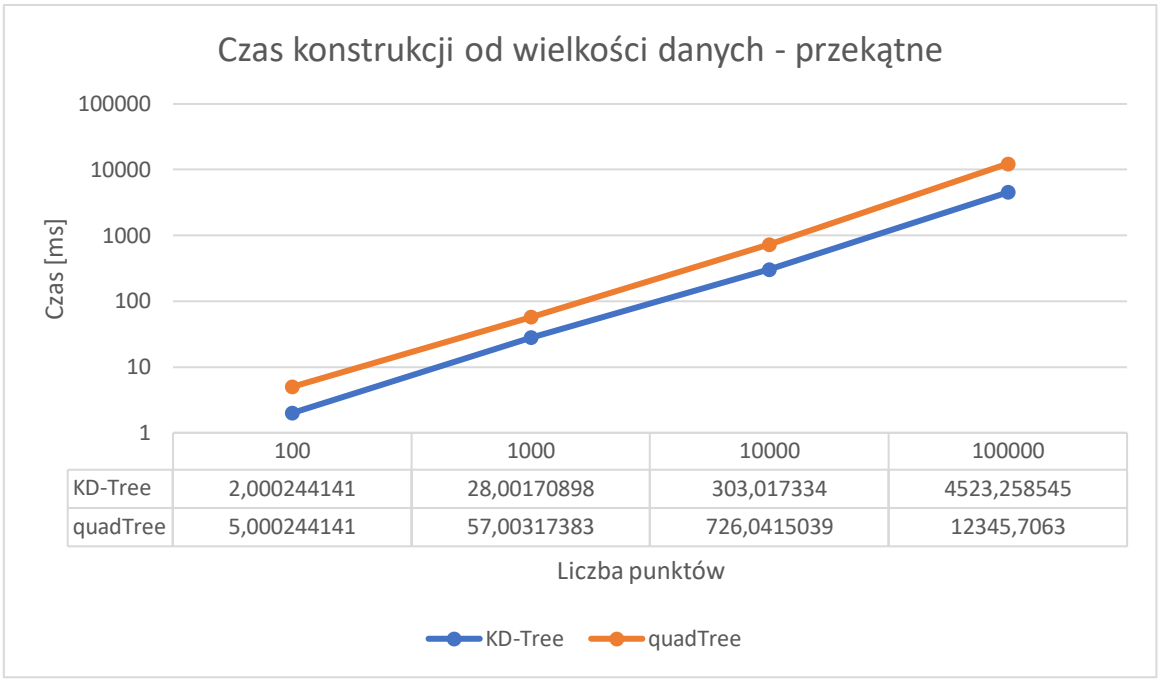

| | 100 | 1000 | 10000 | 100000 |
|---|---|---|---|---|
| KD-Tree | 1 | 22,0012207 | 152,0087891 | 2028,115967 |
| quadTree | 2,000244141 | 21,0012207 | 224,0126953 | 3337,190918 |

*Wykres 1. – zależność czasu budowy od wielkości danych dla punktów wylosowanych na prostokącie*




| | 100 | 1000 | 10000 | 100000 |
|---|---|---|---|---|
| KD-Tree | 2 | 13,00073242 | 147,0085449 | 2076,118896 |
| quadTree | 3,000244141 | 30,00170898 | 419,0241699 | 6186,35376 |

*Wykres 2. – zależność czasu budowy od wielkości danych dla punktów wylosowanych na okręgu*



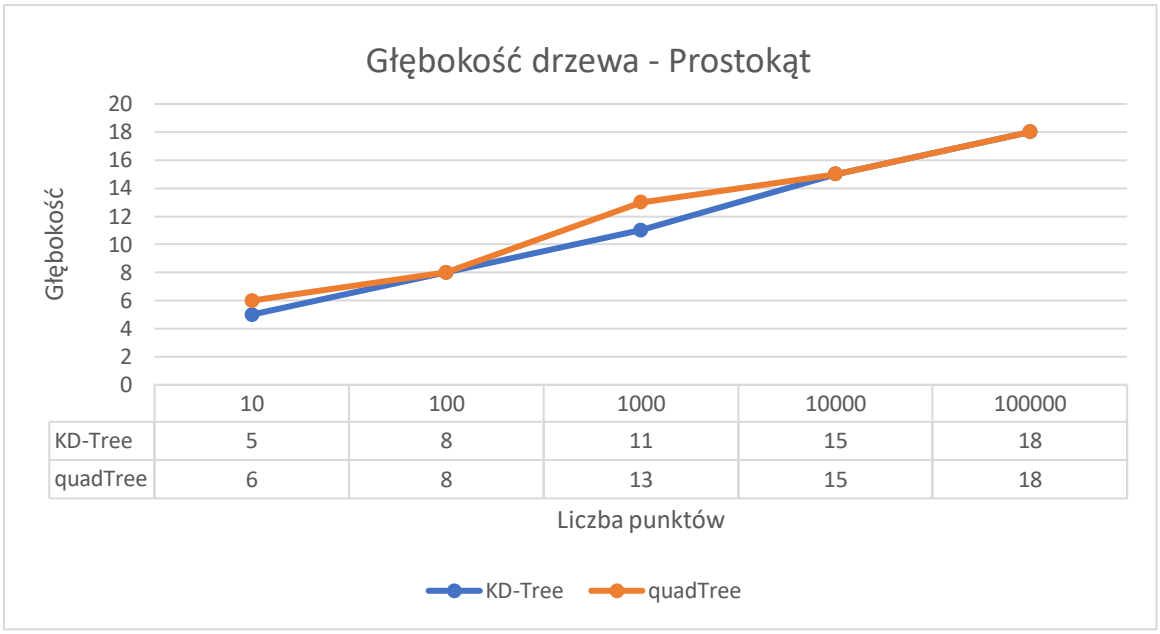

| | 100 | 1000 | 10000 | 100000 |
|---|---|---|---|---|
| KD-Tree | 2,000244141 | 28,00170898 | 303,017334 | 4523,258545 |
| quadTree | 5,000244141 | 57,00317383 | 726,0415039 | 12345,7063 |

*Wykres 3 – zależność czasu budowy od wielkości danych dla punktów wylosowanych na przekątnej prostokąta*

Jak pokazuje powyższy przykład, w większości przypadków konstrukcja KD-tree będzie szybsza. Ponieważ jest ona niemal całkowicie zależna jedynie od wielkości danych można się spodziewać podobnego czasu konstrukcji dla każdego zestawu danych. Tymczasem dla quadtree rozkład punktów jest znaczący – o ile przy losowym rozkładzie punktów na prostokącie quadtree było gorsze od KD-tree maksymalnie 1,7 raza, tak przy okręgu i przekątnych różnicę można zaobserwowac już dla zestawu zaledwie 100 punktów.

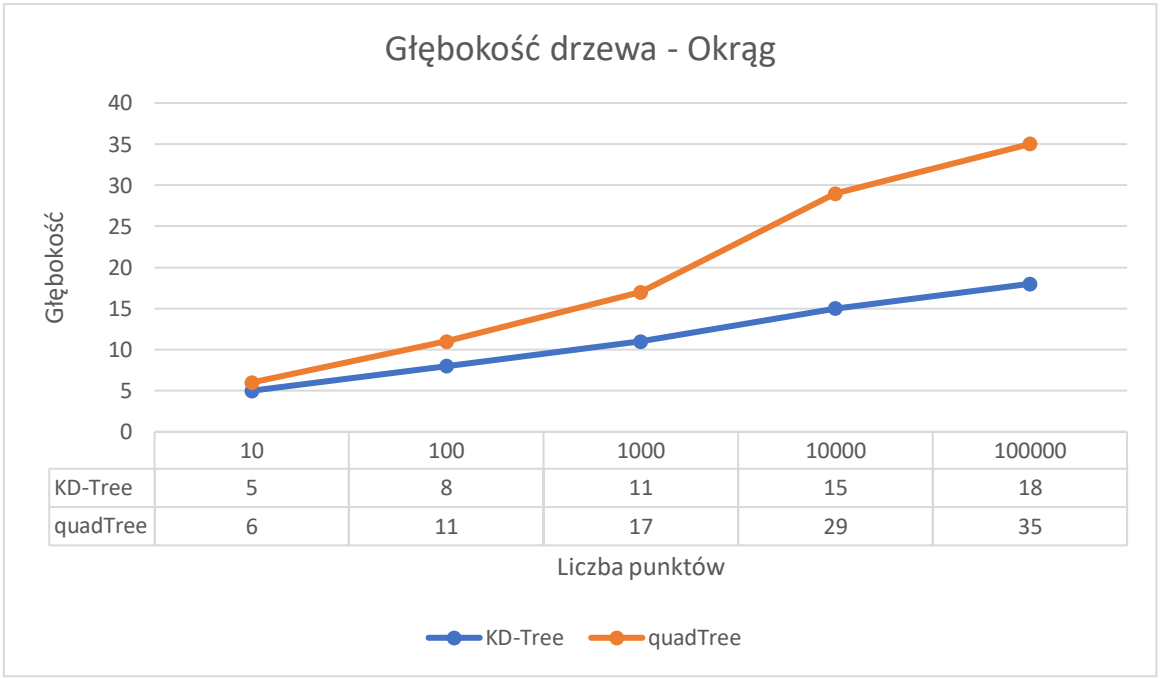
## Głębokość

Ponieważ złożoność asymptotyczna quadTree jest zależna w dużej mierze od głębokości, sprawdzono ten parametr drzewa dla zbiorów identycznych jak w punkcie wyżej.
Wyniki zaprezentowano w formie wykresów (nr 4-6)



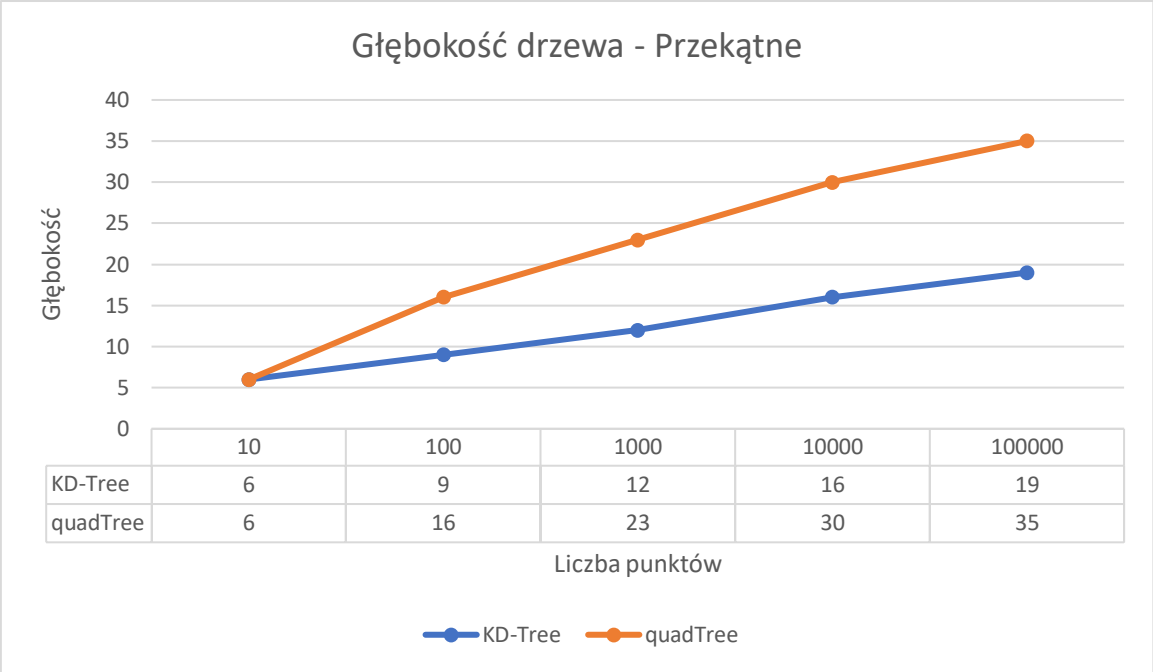

| | 10 | 100 | 1000 | 10000 | 100000 |
|---|---|---|---|---|---|
| KD-Tree | 5 | 8 | 11 | 15 | 18 |
| quadTree | 6 | 8 | 13 | 15 | 18 |

*Wykres 4. – zależność głębokości od wielkość zbioru punktów wylosowanych na prostokącie*




| | 10 | 100 | 1000 | 10000 | 100000 |
|---|---|---|---|---|---|
| KD-Tree | 5 | 8 | 11 | 15 | 18 |
| quadTree | 6 | 11 | 17 | 29 | 35 |

*Wykres 5. – zależność głębokości od wielkość zbioru punktów wylosowanych na okręgu*



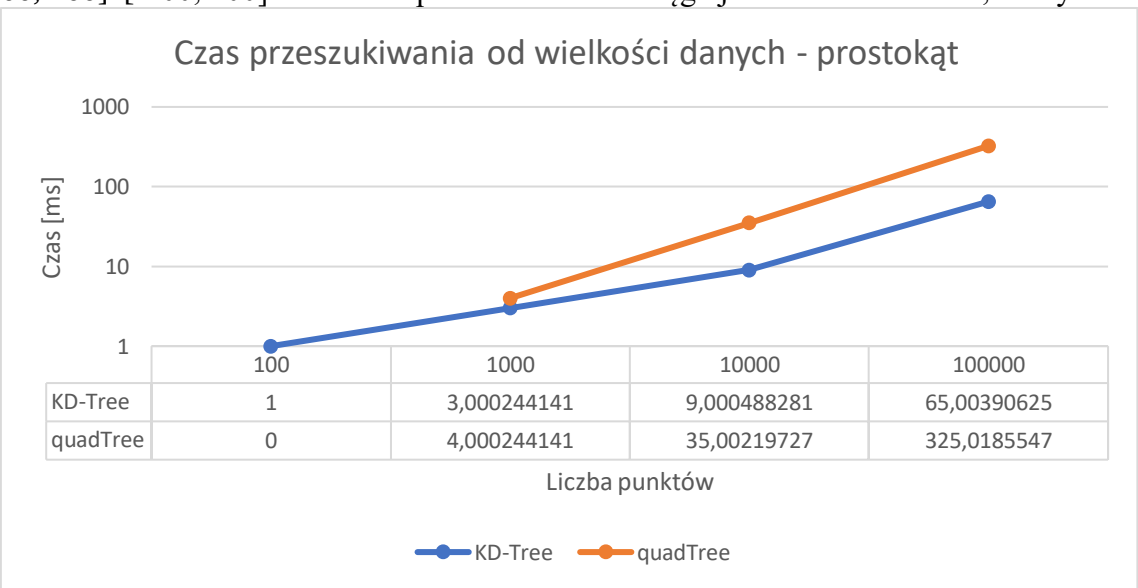

*Wykres 6. – zależność głębokości od wielkości zbioru punktów wylosowanych na przekątnych*

Wykresy jasno pokazują, że rodzaj danych wejściowych determinuje głębokość drzewa quadtree, co jest jedną z przyczyn wolnego czasu konstrukcji dla dwóch ostatnich omawianych zbiorów, tj. okręgu oraz przekątnych. Warto jednak zauważyć, że dla losowych punktów na prostokącie obie struktury mają tę samą głębokość.
Co istotne – dla struktur KD-Tree różnica głębokości jest niemal niezauważalna

## Przeszukiwanie

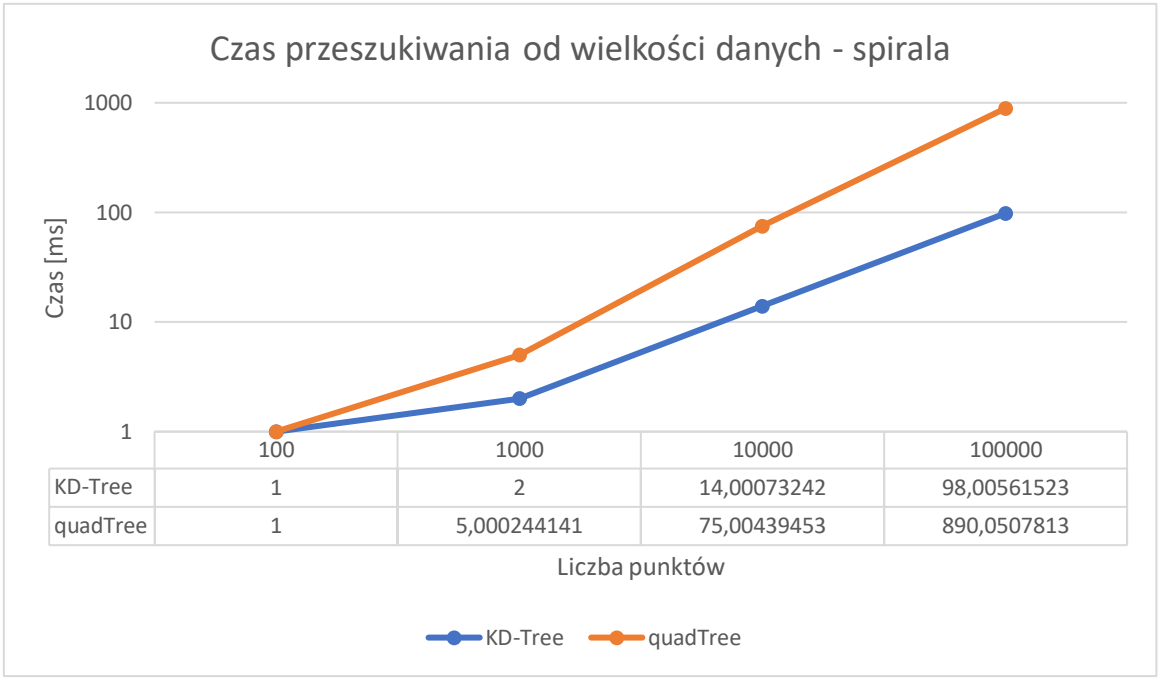
### Zależność od wielkości danych

W tym doświadczeniu zbadano czas przeszukiwania obszaru zajmującego 50% pola prostokąta [-100, 100]x[-100, 100]. Ponieważ przeszukiwanie okręgu jest nierównomierne, zdecydowano



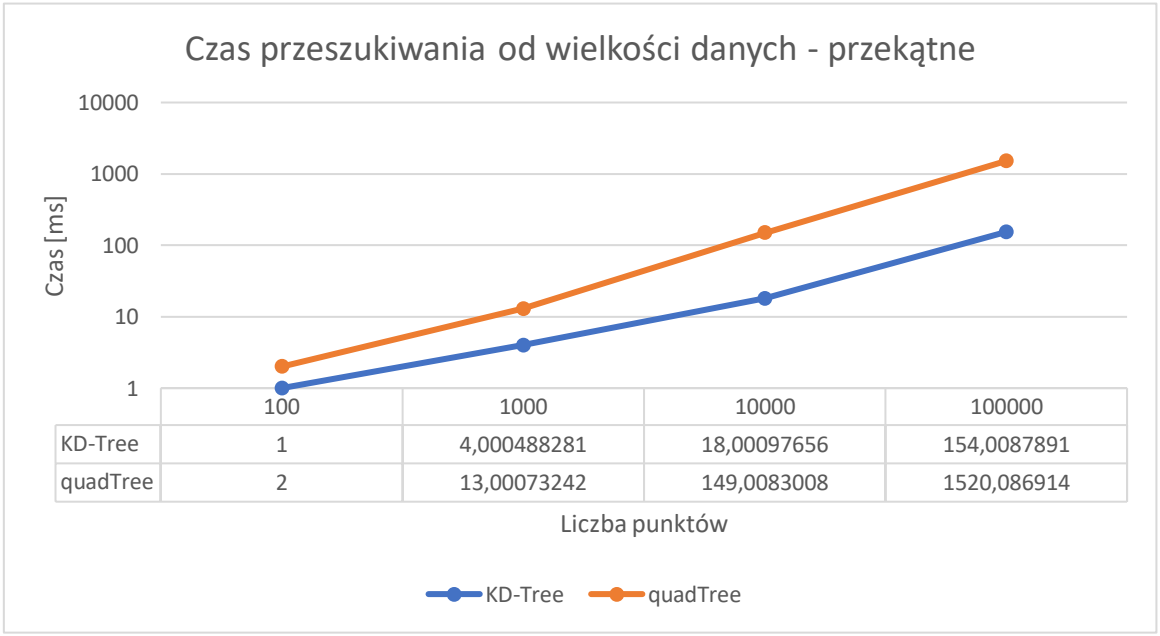

*Wykres 7. – zależność czasu przeszukiwania od wielkości zbioru punktów wylosowanych na prostokącie*

się na użycie zamiast niego punktów na spirali Archimedesa. Wyniki przedstawiono na wykresach 7.-9. (Zastosowano skalę logarytmiczną)




| | 100 | 1000 | 10000 | 100000 |
|---|---|---|---|---|
| KD-Tree | 1 | 2 | 14,00073242 | 98,00561523 |
| quadTree | 1 | 5,000244141 | 75,00439453 | 890,0507813 |

*Wykres 8. – zależność czasu przeszukiwania od wielkości zbioru punktów wylosowanych na spirali*



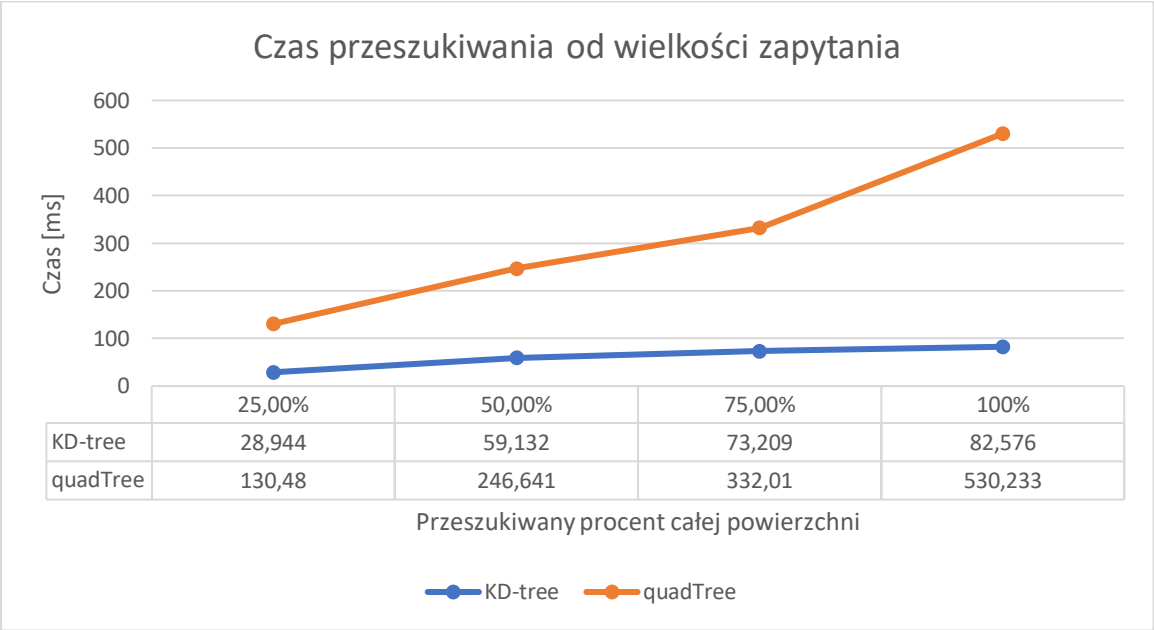

| | 100 | 1000 | 10000 | 100000 |
|---|---|---|---|---|
| KD-Tree | 1 | 4,000488281 | 18,00097656 | 154,0087891 |
| quadTree | 2 | 13,00073242 | 149,0083008 | 1520,086914 |

*Wykres 9. – zależność czasu przeszukiwania od wielkości zbioru punktów wylosowanych na przekątnych*

Powyższe wykresy pokazują, że KD-Tree jest zdecydowanie lepszą strukturą do omawianych zapytań. Zapewnia lepszą szybkość wyszukiwania w każdym przypadku.

**Zależność czasu wyszukiwania od wielkości przeszukiwanej powierzchni**

Na koniec sprawdzono zależność czasu wyszukiwania od wielkości zapytania. W tym celu zmierzono, ile czasu zajmuje znalezienie punktów wylosowanych na kwadracie [-100, 100]x[- 100, 100] i wyszukiwanych na kwadratach będących ułamkiem tej powierzchni (czyli przykładowo dla 25% przeszukiwano kwadrat [-50, 50]x[- 50, 50]). Liczność zbioru wynosi 100 000. Wyniki przedstawiono na wykresie 10.




| | 25,00% | 50,00% | 75,00% | 100% |
|---|---|---|---|---|
| KD-tree | 28,944 | 59,132 | 73,209 | 82,576 |
| quadTree | 130,48 | 246,641 | 332,01 | 530,233 |

*Wykres 10. – zależność czasu przeszukiwania od wielkości zapytania. Zbiór wejściowy to 100 000 punktów wylosowanych na kwadracie.*

Wyniki pokazują, że dla przedstawionych zapytań wraz z liniowym wzrostem wielkości zapytania otrzymujemy w przybliżeniu liniowy wzrost czasu potrzebnego na jego zrealizowanie. Takiego wyniku można się spodziewać, ponieważ dolnym ograniczeniem czasowym na zwrócenie wszystkich jest ich liczba.
Warto jednak zauważyć, że dla zapytań bliskich sprawdzaniu wszystkich punktów, przy drzewach quadTree można zauważyć znaczne pogorszenie szybkości, natomiast dla KD-tree – przeciwnie. Wskazują na to zmiany wartości dla powierzchni z 75% na 100%.

**Podsumowanie**

KD-tree pod względem konstrukcji oraz przeszukiwania obszarów ortogonalnych jest bardziej wydajne niż quadTree – jego konstrukcja trwa krócej, głębokość drzewa jest mniejsza oraz wyszukiwanie jest niemal zawsze szybsze. Jest również odporna na różne nietypowe typy danych, takie jak punkty na okręgu czy prostej.