Wyszukiwanie geometryczne
KDTree i QuadTree
Dokumentacja

Bartłomiej Wiśniewski
Cyprian Neugebauer
grupa: wtorek 12:50

Spis treści

Kd-tree i Quadtree – dokumentacja

1. Część techniczna

1.1 Wymagania

Program do działania wymaga Jupyter Notebook’a (Python 3.9), biblioteki numpy w

wersji 1.20.1 oraz biblioteki matplotlib.

Wszystkie zależności zostały zapisane w pliku env.yaml tak więc w praktyce wystarczy wykonać komendę ”conda env create --force -f env.yml” która utworzy nowe środowisko na podstawie pliku, a następnie je aktywować za pomocą komendy „conda activate trees-project”.

1.2 Opis głównych modułów

kd_tree.py – moduł zawierający implementację kd-tree

quadtree.py - moduł zawierający implementację drzewa czwórkowego (quadtree)

visualizers.py – moduł zawierający implementację klas wizualizujących operacje na drzewach

visualization.ipynb – dokument Jupyter Notebook’a stanowiący interfejs użytkownika

1.3 Opis programu

1.3.1 Kd-tree

Implementacja opiera się na informacjach podanych na wykładzie. Wartość podziału podczas budowania drzewa obliczana jest jako $v_{split} = \frac{v_{\lfloor n \div 2 \rfloor - 1} + v_{\lfloor n \div 2 \rfloor}}{2}$ gdzie n to liczba punktów w aktualnie rozpatrywanym obszarze, a v_i to rozpatrywana na aktualnym poziomie rekurencji współrzędna $i - tego$ punktu w posortowanej (po tej współrzędnej) tablicy tych punktów. Nak każdym poziomie rekurencji brana jest pod uwagę $k - ta$ współrzędna $k = depth \bmod dim$, gdzie depth to aktualna głębokość rekurencji, a dim to ilość wymiarów przestrzeni. Wartości $v_{\lfloor n \div 2 \rfloor - 1}$ i $v_{\lfloor n \div 2 \rfloor}$ znajdowane są za pomocą funkcji nupmy.partition, która wykorzystuje algorytm quickselect. Informacja o wartości współrzędnej podziału zapisywana jest z węźle drzewa. Następnie punkty dzielone są na te których współrzędna v jest $v \leq v_{split}$ i resztę. Te grupy punktów są przekazywane na niższy poziom rekurencji.
Wyszukiwanie punktów w kd-drzewie jest wierną implementacją pseudokodu pokazanego na wykładzie.

1.3.2 Quadtree

Implementacja opiera się definicji drzewa czwórkowego podanego na stronie https://en.wikipedia.org/wiki/Quadtree. Po podaniu punktów przez użytkownika drzewo jest budowane w kolejności podania punktów. Każdy węzeł drzewa, odpowiadający pewnej płaszczyźnie może przechowywać 4 punkt. Gdy w danym węźle znajdą się już 4 punkty płaszczyzna odpowiadająca węzłowi zostaje podzielona na 4 równe części, każda z nowych części jest nowym węzłem, który także może pomieścić 4 punkty. Dodatkowo w każdym węźle znajduje się lista z wszystkimi punktami z poddrzewa danego węzła, aby przyśpieszyć funkcjonalność zapytania o wszystkie punkty z zadanego przedziału. Użytkownik ma możliwość wstawiania nowych punktów do drzewa, ale także wyświetlenia wszystkich punktów znajdujących się w zadanych przez użytkownika obszarze zdefiniowanym przez 2 podane punkty.

2. Część użytkownika

2.1 Uruchomienie programu

Aby uruchomić program należy wykonać kroki z sekcji 1.1 Wymagania, a następnie otworzyć za pomocą Jupyter Notebook'a plik visualization.ipynb. Można w nim zadawać punkty na płaszczyźnie i uruchamiać przeszukiwanie tej płaszczyzny za pomocą zaimplementowanych drzew.

2.2 Funkcje dostępne dla użytkownika

Kd-tree

2.2.1 Stworzenie drzewa

Aby stworzyć drzewo należy użyć konstruktora klasy KDTree z podanymi parametrami.

Przykładowe stworzenie drzewa może wyglądać następująco:

kd_tree = KDTree(no_of_dimensions, points, visualize), gdzie

no_of_dimensions - liczba naturalna oznaczająca liczbę współrzędnych każdego punktu

points - lista z punktami w postaci listy np.float64,

visualize - wartość logiczna informująca czy drzewo będzie wizualizowane

2.2.2 Zapytanie o wszystkie punkty z zadanego przedziału

Aby zapytać o wszystkie punkty z zadanego przedziału należy użyć funkcji find_points_in_area z podanym parametrem area.

Przykładowe zapytanie o wszystkie punkty z zadanego przedziału:

kd_tree.find_points_in_area(area), gdzie

area - krotka z 2 punktami - lewy dolny punkt i prawy górny punkt przedziału, o który chcemy zapytać (w przypadku przestrzeni dwuwymiarowej). Dla przestrzeni o większej liczbie wymiarów analogiczne punky o większej liczbie współrzędnych.

2.2.3 Wizualizacja wykonanych na drzewie operacji

Aby zwizualizować operacje wykonane na drzewie należy wywołać funkcję kd_tree.visualizer.get_tree_building_plot() lub kd_tree.visualizer.get_searching_plot(i) i przypisać jej wynik do zmiennej. Następnie na tej zmiennej wywołać metodę draw().

Funkcja kd_tree.visualizer.get_tree_building_plot() zwróci wizualizację procesu budowania drzewa, a kd_tree.visualizer.get_searching_plot(i) wizualizację i-tego z kolei (licząc od 0) procesu wyszukiwania punktów. Jeżeli parametr i nie zostanie podany to funkcja zwróci wizualizację ostatniego wyszukiwania.

Przykładowe wizualizacje operacji na drzewie:

plot1 = kd_tree.visualizer.get_tree_building_plot()

plot1.draw()

plot2 = kd_tree.visualizer.get_searching_plot(0)

plot2.draw()

Quadtree

2.2.1 Stworzenie drzewa

Aby stworzyć drzewo należy przypisać do zmiennej konstruktor klasy Quadtree z podanymi parametrami.

Przykładowe stworzenie drzewa może wyglądać następująco:

quadtree = Quadtree(points, boundary, capacity, visualize), gdzie

points - lista z punktami w postaci krotek,

boundary - krotka z 2 punktami - lewy dolny punkt i prawy górny punkt prostokąta zawierającego wszystkie punkty z points

capacity - pojemność punktów w jednym węźle

visualize - wartość logiczna informująca czy drzewo będzie wizualizowane

2.2.2 Wstawianie punktów do drzewa

Do wstawienia punktu do istniejącego już drzewa użytkownik musi użyć funkcji insert z podanymi parametrami.

Przykładowe wstawienie punktu do drzewa:

quadtree.insert(QTNode, point), gdzie

QTNode - zawsze przy wstawianiu wartość ta musi być ustawiona na quadtree.root

point - punkt, który chcemy wstawić do drzewa

2.2.3 Zapytanie o wszystkie punkty z zadanego przedziału

Aby zapytać o wszystkie punkty z zadanego przedziału należy użyć funkcji query_range z podanym parametrem range.

Przykładowe zapytanie o wszystkie punkty z zadanego przedziału:

quadtree.query_range(range), gdzie

range - krotka z 2 punktami - lewy dolny punkt i prawy górny punkt przedziału, o który chcemy zapytać

2.2.4 Wizualizacja wykonanych na drzewie operacji

Aby zwizualizować budowanie drzewa należy wywołać funkcję quadtree.visualizer.create_build_plot() i przypisać jej wynik do zmiennej. Następnie na tej zmiennej wywołać metodę draw().

Przykładowa wizualizacja budowania drzewa:

build_plot = quadtree.visualizer.create_build_plot()

build_plot.draw()

Aby zwizualizować zapytanie o wszystkie punkty z zadanego przedziały należy wywołać funkcję quadtree.visualizer.creat_query_plot() i przypisać jej wynik do zmiennej. Następnie na tej zmiennej wywołać metodę draw().

Przykładowa wizualizacja zapytania:

query_plot = quadtree.visualizer.create_query_plot()

query_plot.draw()

3. Sprawozdanie

3.1 Opis problemu

Celem projektu było zaimplementowanie odpowiednich struktur danych - quadtree oraz kd-drzew, które pozwalają na szybkie przeszukiwanie obszarów ortogonalnych. Na wejściu otrzymujemy zbiór punktów P na płaszczyźnie i należy dla zadanych x_1, x_2, y_1, y_2 znaleźć punkty q ze zbioru P takie, że $x_1 \leq q_x \leq x_2$, $y_1 \leq q_y \leq y_2$.

3.2 Wykonane testy

Testy zostały wykonane na 5 losowo wygenerowanych zestawach danych, takich że

$p_1 = (x_1, y_1) = (50, 50)$, $p_2 = (x_2, y_2) = (150, 150)$ oraz punkty p ze zbioru P spełniały zależność $0 \leq p_x \leq 200$, $0 \leq p_y \leq 200$.

Dla każdego zbioru punktów zmierzono czas budowania drzewa oraz czas wyszukiwania punktów spełniających powyższe warunki.

3.3 Uzyskane wyniki

Tabela 1:

	Czas budowania drzewa	
Liczba punktów	QuadTree	KD-Tree
1000	0,01089525223	0,01509690285
10000	0,1561832428	0,186798811
50000	0,8909876347	1,121948004
100000	1,965014696	2,445206642
500000	11,91793108	14,42099214

Wykres 1:

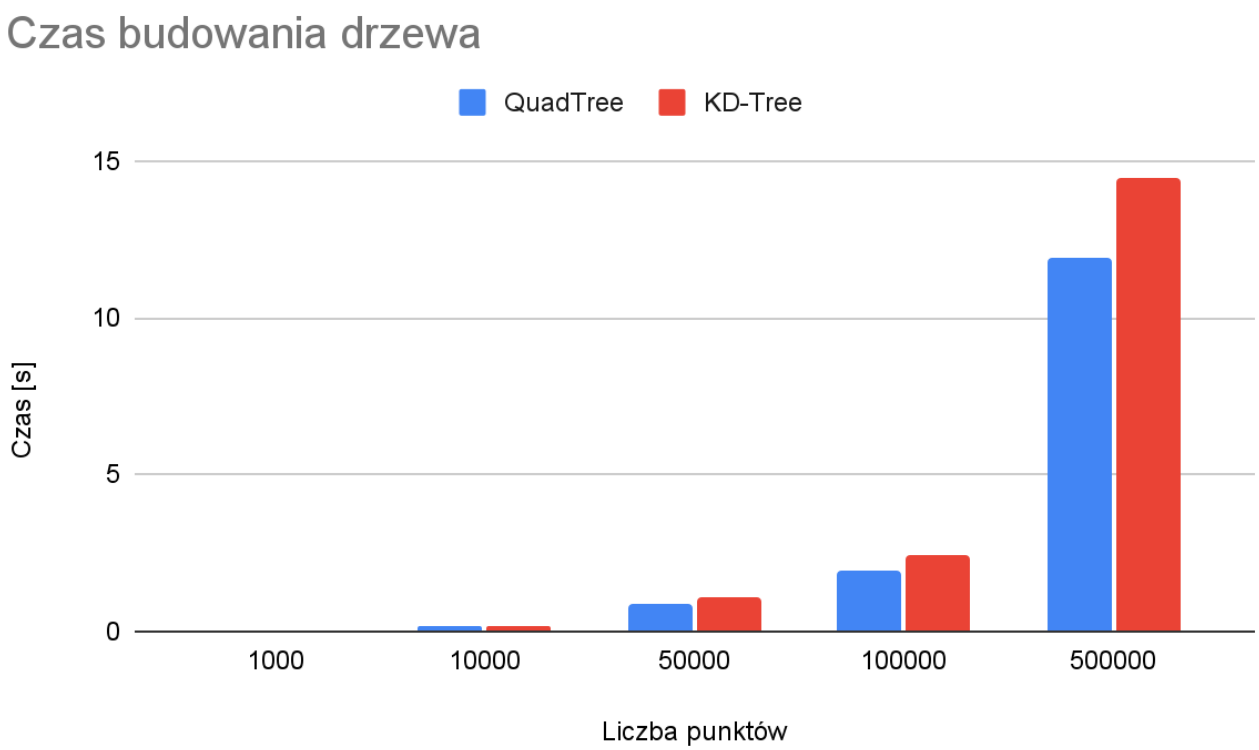


Tabela 2 (punkty z przedziału (0 - 200, 0 - 200), zapytanie z przedziału (50, 50) - (150, 150)):

	Czas przeszukiwania drzewa [s]		
Liczba punktów	QuadTree	KD-Tree	Tablica (Python'owa lista)
1000	0,001585960388	0,001424789429	0,0003609657288
10000	0,005750417709	0,005335569382	0,003895998001
50000	0,002445697784	0,01429772377	0,0180580616
100000	0,003633975983	0,02073216438	0,03485131264
500000	0,01019597054	0,05152344704	0,1690728664

Wykres 2 (punkty z przedziału (0 - 200, 0 - 200), zapytanie z przedziału (50, 50) - (150, 150)):

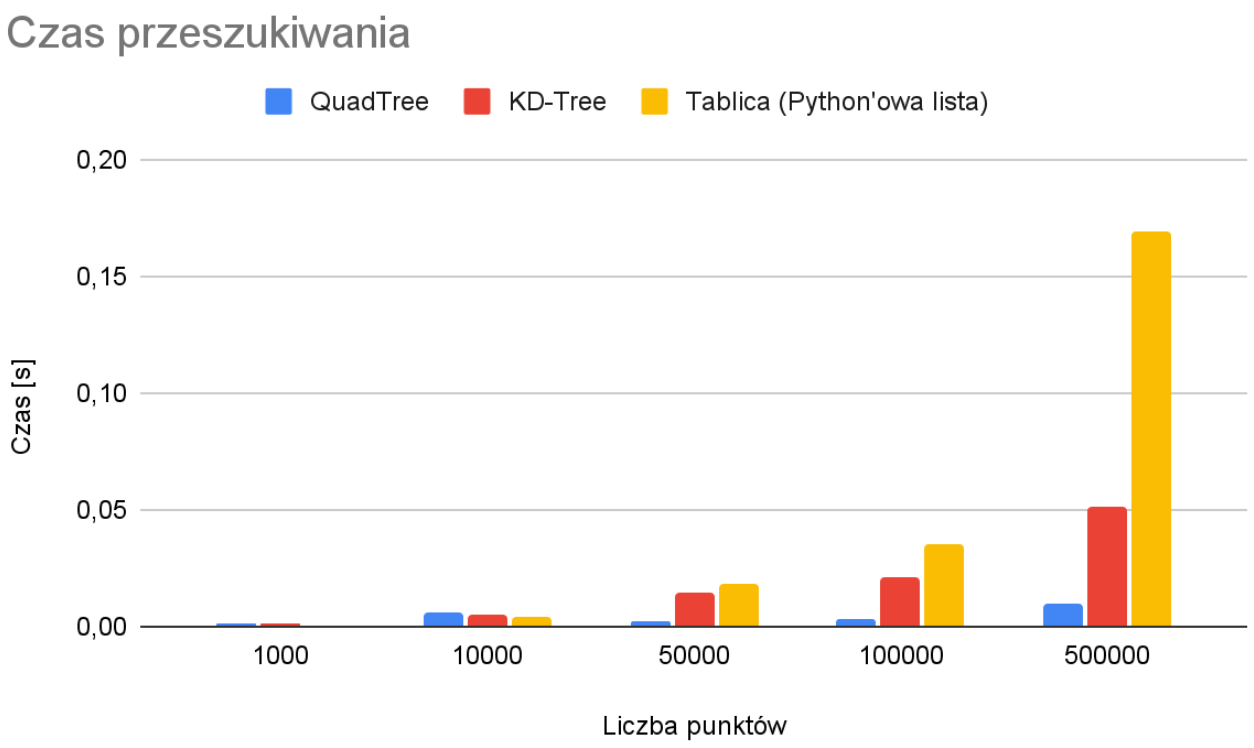
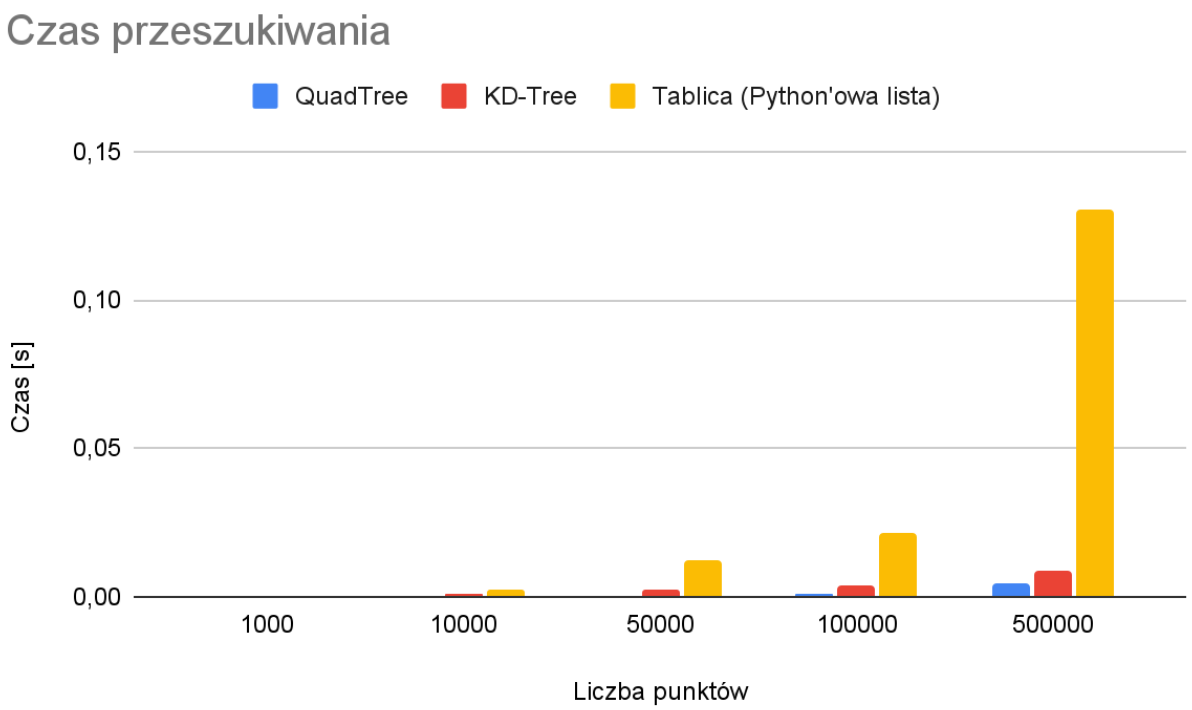


Tabela 3 (Punkty z przedziału (0 - 200, 0 - 200), zapytanie z przedziału (40, 40), (60, 60)):

	Czas przeszukiwania drzewa [s]		
Liczba punktów	QuadTree	KD-Tree	Tablica (Python'owa lista)
1000	9,01E-05	0,0004713535309	0,0002958774567
10000	0,0002846717834	0,001134634018	0,002254247665
50000	0,0005795955658	0,002576112747	0,01224994659
100000	0,001064538956	0,003776550293	0,02185440063
500000	0,004234790802	0,008467912674	0,1302320957

Wykres 3 (Punkty z przedziału (0 - 200, 0 - 200), zapytanie z przedziału (40, 40), (60, 60)):



4. Wnioski

Na podstawie przeprowadzonych i przedstawionych powyżej testów możemy stwierdzić, że zaimplementowane struktury i ich funkcjonalności działają poprawnie. Obydwa drzewa realizują ideę podziału przestrzeni na mniejsze części. Rezultaty w postaci znalezionych punktów były identyczne w przypadku obu drzew.
Procesy budowy tych drzew znacznie się od siebie różnią i mają swoje konsekwencje. Do QuadTree w dowolnej chwili możemy w prosty sposób dodać nowy punkt. Nie można tego samego powiedzieć o KD-Tree. Jednak to KD-Tree jest znacznie prostsze w realizacji dla dowolnej liczby wymiarów. Dla wszystkich zestawów danych czas tworzenia drzew był bardzo zbliżony, choć budowanie Quadtree za każdym razem wykonywało się nieco szybciej. Natomiast patrząc na czas wyszukiwania punktów dla zadanego przedziału Quadtree okazał się szybszy w zdecydowanej większości testów. Drzewo czwórkowe w ogólności jest dużo płytszym drzewem niż KD-Tree, w większości przypadków mogło to mieć znaczący wpływ na szybkość algorytmu. Sprawa ma się nieco inaczej w przypadkach, gdy większość punktów jest skupiona w jednym obszarze. Wtedy QuadTree może okazać się głębsze od KD-Tree. Złożoność czasowa pesymistyczna zapytania o wszystkie punkty z zadanego przedziału dla Quadtree to O(n), w przypadku KD-Tree O(sqrt(n) + k), gdzie n to liczba wszystkich punktów, a k to liczba punktów w zadanym przedziale. Dla niewygodnych danych drzewa te mogą osiągać takie złożoności, co można zauważyć w teście z Tabeli 2 dla

n = 10000. Liniowe przejście przez tablice w tym przypadku było nieco szybsze od pozostałych metod, lecz reszta testów nie pozostawia złudzeń co do korzyści z implementacji QuadTree oraz KD-Tree dla problemu zapytania o wszystkie punkty z zadanego przedziału. Dodatkowo im mniejsze są obszary poszukiwań w porównaniu do wszystkich punktów tym korzystniej wypadają drzewa w porównaniu do standardowej iteracji po tablicy.