Algorytmy geometryczne - projekt

Wyszukiwanie geometryczne przeszukiwanie obszarów ortogonalnych QuadTree oraz KD-drzewa

Maksymilian Siemek

Hubert Kukla

Spis treści

1. Wprowadzenie 5
2. Część techniczna 5
   2.1. Wymagania techniczne 5
   2.2. Schemat pakietów 6
   2.3. Pakiet *quadtree* 6
      2.3.1. Moduł *quadtree* 6
         2.3.1.1. Klasa *QuadTree* 6
         2.3.1.2. Metoda *___init___* 7
         2.3.1.3. Metoda *is_leaf* 7
         2.3.1.4. Metoda *subdivide* 7
         2.3.1.5. Metoda *_child_for_point* 7
         2.3.1.6. Metoda *insert* 7
         2.3.1.7. Metoda *query* 8
         2.3.1.8. Funkcja *bounding_rect_pairwise* 8
   2.4. Pakiet *kdtree* 9
      2.4.1. Moduł *kdtree* 9
         2.4.1.1. Klasa *KDNode* 9
         2.4.1.2. Klasa *KDTree* 9
         2.4.1.3. Metoda *build* 9
         2.4.1.4. Metoda query_range 9
         2.4.1.5. Metoda search 9
      2.4.2. Moduł kdtreeVis 9
         2.4.2.1. Klasa KDNode 10
         2.4.2.2. Klasa KDTree 10
         2.4.2.3. Funkcja kd_search_plain 10
         2.4.2.4. Funkcje pomocnicze prostokątów 10
         2.4.2.5. Funkcja world_bounds 10
         2.4.2.6. Funkcja draw_kdtree_splits 10
         2.4.2.7. Funkcja kd_query_with_visualization 10
         2.4.2.8. Funkcje renderujące (render_kdtree_image / gif) 11
         2.4.2.9. Funkcje uruchamiające (runners) 11
      2.4.3. Moduł *visualization* 11
         2.4.3.1. Funkcja *draw_kd_tree_structure* 11
         2.4.3.2. Funkcja *kd_query_with_visualization* 11
         2.4.3.3. Funkcja *render_kdtree_gif* 11
         2.4.3.4. Uwagi o danych wejściowych 12
         2.4.3.5. Przykładowe użycie 12
   2.5. Pakiet *data_generators* 12
      2.5.1. Moduł *data_generators* 12
         2.5.1.1. Funkcja *generate_uniform_points* 12
         2.5.1.2. Funkcja *generate_normal_points* 12
         2.5.1.3. Funkcja *generate_collinear_segment_points* 12
         2.5.1.4. Funkcja *generate_rectangle_points* 13
         2.5.1.5. Funkcja *generate_square_points* 13
         2.5.1.6. Funkcja *generate_grid_points* 13
         2.5.1.7. Funkcja *generate_clustered_points* 13
      2.5.2. Moduł *query_picker* 13
         2.5.2.1. Funkcja *load_points_csv_xy* 13

2.5.2.2. Funkcja *save_query_rect* ... 13
2.5.2.3. Funkcja *rect_from_corners* ... 13
2.5.2.4. Funkcja *pick_query_for_existing_csv* ... 13
2.5.2.5. Funkcja *pick_queries_in_folder* ... 14
2.5.3. Moduł *generate_custom* ... 14
2.5.4. Moduł *custom_data_generator* ... 14
2.5.4.1. Funkcja *save_csv* ... 14
2.5.4.2. Funkcja *save_query_rect* ... 14
2.5.4.3. Funkcja *rect_from_corners* ... 14
2.5.4.4. Funkcja *make_custom_points_and_query* ... 14
2.6. Pakiet *points_util* ... 15
2.6.1. Moduł *points_classes* ... 15
2.6.1.1. Klasa *Point* ... 15
2.6.1.2. Klasa *Rect* ... 15
2.6.1.3. Właściwości (Properties) *left, right, bottom, top* ... 15
2.6.1.4. Funkcja *contains_point* ... 15
2.6.1.5. Funkcja *intersects* ... 15
2.6.2. Moduł *points_loaders* ... 15
2.6.2.1. Funkcja *load_points_csv* ... 15
2.6.2.2. Funkcja *load_query_rect* ... 16
2.6.3. Moduł *prepare_queries* ... 16
2.6.3.1. Funkcja *prepare_queries_for_N* ... 16
2.6.3.2. Funkcja *prepare_queries_for_custom* ... 16
2.6.4. Moduł *time_compare* ... 16
2.6.4.1. Funkcja *load_xy_csv* ... 16
2.6.4.2. Funkcja load_query_rect ... 16
2.6.4.3. Funkcja *dataset_name_from_file* ... 16
2.6.4.4. Funkcja bench_both_file ... 16
2.6.4.5. Funkcja bench_both_all ... 17
2.6.4.6. Funkcja save_separate_tables_both ... 17
3. Część użytkownika ... 17
3.1. Pakiet *quadtree* ... 17
3.1.1. Moduł *quadtree* ... 17
3.1.1.1. Inicjalizacja struktury danych ... 17
3.1.1.2. Zapytanie o punkty z zadanego obszaru ... 18
3.1.1.3. Generowanie animacji w formacie *.gif* oraz zdjęć (*.png*) ... 18
3.2. Pakiet *kdtree* ... 19
3.2.1. Moduł *kdtree* ... 19
3.2.1.1. Inicjalizacja struktury danych i zapytanie o punkty z zadanego obszaru ... 19
3.2.1.2. Generowanie animacji w formacie *.gif* oraz zdjęć (*.png*) ... 20
3.2.1.3. Funkcje odpowiedzialne za generowanie *gif*-ów oraz plików *.png* dla każdego zbioru danych o liczności n (*runnery*) ... 21
4. Sprawozdanie ... 22
4.1. Opis problemu ... 22
4.2. Realizacja ... 22
4.3. Wyniki ... 23
4.3.1. Zbiór punktów z rozkładu jednostajnego ... 23
4.3.2. Zbiór punktów z rozkładu normalnego ... 24
4.3.3. Zbiór punktów na siatce ... 25
4.3.4. Zbiór punktów w klastrach ... 26

4.3.5. Zbiór punktów wygenerowanych na prostej 27
4.3.6. Zbiór punktów na obwodzie prostokąta 28
4.3.7. Zbiór punktów na dwóch bokach kwadratu oraz dwóch jego przekątnych 29
4.4. Wnioski 30

1. Wprowadzenie

Dokument ten skupia się na implementacji struktur *KD-drzewa* i *QuadTree* w środowisku *Python*. Obie struktury odgrywają istotną rolę w sprawnym przetwarzaniu zapytań przestrzennych, znajdując szerokie zastosowanie w aplikacjach typu GIS czy nowoczesnych systemach bazodanowych.

Wykorzystanie *KD-drzew* pozwala na szybkie wykonywanie zapytań o obiekty w zadanym zakresie przestrzeni wielowymiarowej – jest to niezbędne przy badaniu zagęszczenia danych czy w pracy systemów GPS. Natomiast *QuadTree* znajduje swoje główne zastosowanie w operacjach dwuwymiarowych; ułatwia ono zarządzanie sektorami w systemach śledzenia ruchu oraz optymalizuje wyświetlanie map poprzez warstwowe ładowanie danych.

2. Część techniczna

2.1. Wymagania techniczne

Program był uruchamiany w interpreterze języka Python 3.12. Potrzebne biblioteki zostały wymienione w pliku *requirements.txt*. Między innymi są to: *pandas*, *matplotlib* oraz *numpy*.

Aby zainstalować wszystkie z nich, należy wykonać polecenie:

```
pip install -r requirements.txt
```

W realizacji projektu wykorzystano narzędzie wizualizacji autorstwa Koła Naukowego BIT dostępne pod adresem: https://github.com/aghbit/Algorytmy-Geometryczne.

2.2. Schemat pakietów

```
QuadTree_and_KDTree/
├── data_generators/
│   ├── custom_data_generator.py
│   ├── data_generator.py
│   ├── generate_custom.py
│   └── query_picker.py
├── KDTree/
│   ├── results/
│   ├── kdtree.py
│   ├── kdtreeVis.py
│   ├── run_KDTree.py
│   └── visualization.py
├── output/
├── points_util/
│   ├── points_classes.py
│   ├── points_loaders.py
│   ├── prepare_queries.py
│   └── time_compare.py
├── QuadTree/
│   ├── results/
│   ├── algorithmsVis.py
│   ├── quadtree.py
│   └── run_QuadTree.py
├── times/
└── visualizer/
```

2.3. Pakiet *quadtree*

2.3.1. Moduł *quadtree*

Moduł implementuje strukturę *QuadTree* w wariancie „punkty tylko w liściach". Oznacza to, że węzły wewnętrzne nie przechowują danych punktowych — służą wyłącznie jako podział przestrzeni na cztery ćwiartki (NW, NE, SW, SE). Punkty są magazynowane wyłącznie w liściach, a podział następuje dopiero po przekroczeniu pojemności liścia.

2.3.1.1. Klasa *QuadTree*

Reprezentuje pojedynczy węzeł drzewa wraz z jego obszarem (*boundary*). Węzeł może być liściem (przechowuje punkty w *self.points*) albo węzłem wewnętrznym (posiada czterech potomków i nie przechowuje punktów).

Parametry konfiguracyjne:

- *capacity*: maksymalna liczba punktów, jaką może przechować liść przed podziałem.
- *max_depth*: ograniczenie głębokości, które chroni przed nieskończonym dzieleniem (np. gdy wiele punktów wpada w bardzo mały obszar).
- *depth*: aktualna głębokość węzła w drzewie (używana do kontroli *max_depth*).

2.3.1.2. Metoda *___init___*

Inicjalizuje węzeł *QuadTree*, zapisując jego granice (*boundary*) oraz ustawienia (*capacity*, *max_depth*, *depth*). Tworzy pustą listę punktów, oznacza węzeł jako niepodzielony (*divided = False*) i przygotowuje pola na dzieci (*nw*, *ne*, *sw*, *se*).

2.3.1.3. Metoda *is_leaf*

Zwraca informację, czy dany węzeł jest liściem. W praktyce jest to negacja flagi *divided*: jeśli węzeł nie został podzielony, to przechowuje punkty i jest liściem.

2.3.1.4. Metoda *subdivide*

Dzieli obszar węzła (*boundary*) na cztery równe części. Na podstawie środka (*cx*, *cy*) oraz połówek rozmiarów (*hw*, *hh*) oblicza prostokąty potomków o wymiarach *hw/2* i *hh/2*:

- NW: lewa-górna ćwiartka,
- NE: prawa-górna ćwiartka,
- SW: lewa-dolna ćwiartka,
- SE: prawa-dolna ćwiartka.

Następnie tworzy cztery obiekty *QuadTree* jako dzieci, zwiększając *depth* o 1, oraz ustawia *divided = True*, przez co bieżący węzeł przestaje być liściem.

2.3.1.5. Metoda *__child_for_point*

Metoda pomocnicza wybierająca odpowiednie dziecko dla punktu *p*. Zakłada, że węzeł jest już podzielony (*divided == True*) oraz że punkt należy do *boundary*. Sprawdza kolejno, w którym prostokącie potomka leży punkt (używając *contains_point*) i zwraca właściwy węzeł dziecka.

2.3.1.6. Metoda *insert*

Wstawia punkt do struktury.

Logika działania:

- Najpierw sprawdza, czy punkt mieści się w *boundary*. Jeśli nie, zwraca *False* (punkt nie pasuje do tego drzewa).
- Jeśli bieżący węzeł jest liściem:
  - gdy liczba punktów jest mniejsza niż *capacity* lub osiągnięto *max_depth*, punkt trafia do *self.points* i metoda zwraca *True*,
  - w przeciwnym razie węzeł ulega podziałowi (*subdivide*).
- Po podziale punkty zgromadzone wcześniej w liściu są przenoszone do odpowiednich dzieci:
  - *old_points* przechowuje poprzednią listę,
  - *self.points* zostaje wyzerowane, bo węzeł staje się węzłem wewnętrznym,
  - każdy punkt z *old_points* jest kierowany do właściwego dziecka przez *__child_for_point* i ponownie wstawiany rekurencyjnie.
- Na końcu (po przeniesieniu starych punktów) wstawiany jest nowy punkt *p* do odpowiedniego dziecka.

Jeśli węzeł nie jest liściem, metoda od razu wybiera właściwe dziecko (*__child_for_point*) i deleguje wstawienie rekurencyjnie.

2.3.1.7. Metoda *query*

Wykonuje zapytanie zakresowe: zwraca listę punktów leżących wewnątrz prostokąta *range_rect*. Działa rekurencyjnie i wykorzystuje odcinanie gałęzi, aby nie schodzić do obszarów, które na pewno nie mają wyniku.

Zasady:
- Jeśli *found* nie jest podane, tworzona jest nowa lista wyników.
- Jeśli *boundary* węzła nie przecina się z *range_rect* (test *intersects*), metoda natychmiast zwraca *found* bez dalszej rekurencji.
- Jeśli węzeł jest liściem, sprawdza każdy punkt w *self.points* i dodaje do wyników te, które spełniają *contains_point*.
- Jeśli węzeł jest wewnętrzny, rekurencyjnie odpytuje wszystkie cztery dzieci (*nw*, *ne*, *sw*, *se*) i zwraca wspólną listę wyników.

2.3.1.8. Funkcja *bounding__rect__pairwise*

Wyznacza prostokąt graniczny obejmujący wszystkie punkty wejściowe, zwracając obiekt *Rect* w postaci (środek + połówki boków). Funkcja stosuje podejście „pairwise min/max", które redukuje liczbę porównań w porównaniu do prostego skanowania każdej współrzędnej niezależnie.

Przebieg:
- Konwertuje *points* do listy i sprawdza, czy nie jest pusta (dla pustej listy rzucany jest *ValueError*).
- Inicjalizuje wartości *min_x*, *max_x*, *min_y*, *max_y*:
  - dla jednego punktu bezpośrednio z niego,
  - dla parzystej liczby punktów startuje od pierwszej pary, ustawiając min/max po 1 porównaniu na oś,
  - dla nieparzystej liczby punktów startuje od pierwszego punktu.
- Następnie przetwarza punkty parami:
  - w parze (*a*, *b*) najpierw ustala lokalne *low_x/high_x* i *low_y/high_y*,
  - aktualizuje globalne min/max tylko tam, gdzie jest to potrzebne.
- Na końcu przelicza reprezentację min/max na format prostokąta:
  - *cx*, *cy* jako środki przedziałów,
  - *hw*, *hh* jako połowy rozmiarów z dodanym marginesem *padding*, aby prostokąt miał niezerowy rozmiar i był bezpieczny numerycznie.

Zwracany *Rect* może być użyty jako *boundary* świata dla QuadTree (np. do inicjalizacji drzewa obejmującego całą chmurę punktów).

2.4. Pakiet *kdtree*

2.4.1. Moduł *kdtree*

2.4.1.1. Klasa *KDNode*

Reprezentuje pojedynczy węzeł w drzewie k-wymiarowym (tutaj $k = 2$). Przechowuje obiekt typu Point oraz referencje do dwóch dzieci: left (punkty o mniejszych współrzędnych na danej osi) oraz right (punkty o współrzędnych większych lub równych).

2.4.1.2. Klasa *KDTree*

Główna klasa realizująca strukturę drzewa binarnego, który dokonuje rekurencyjnego podziału przestrzeni naprzemiennie wzdłuż osi X oraz osi Y. Dzięki takiemu podziałowi, drzewo pozwala na znaczną optymalizację zapytań zakresowych (range searching) w porównaniu do przeglądu naiwnego.

2.4.1.3. Metoda *build*

Prywatna funkcja rekurencyjna budująca drzewo. W każdym kroku:Wybiera oś podziału na podstawie głębokości drzewa (depth % 2).Sortuje punkty względem wybranej osi i wyznacza medianę.Punkt będący medianą staje się korzeniem obecnego poddrzewa, co gwarantuje, że drzewo będzie zrównoważone.Proces jest powtarzany dla pozostałych punktów, aż do wyczerpania zbioru.

2.4.1.4. Metoda query_range

Publiczny interfejs służący do wyszukiwania punktów. Przyjmuje obiekt klasy Rect jako obszar zapytania i inicjuje rekurencyjne przeszukiwanie struktury od korzenia, zwracając listę wszystkich punktów spełniających kryteria.

2.4.1.5. Metoda search

Sercem algorytmu wyszukiwania jest logika przycinania gałęzi (pruning). Funkcja nie przeszukuje całego drzewa, lecz sprawdza, czy punkt w bieżącym węźle zawiera się w prostokącie range_rect oraz porównuje granice prostokąta (low, high) z wartością podziału w węźle (val).Odwiedza lewe dziecko tylko wtedy, gdy dolna granica zapytania jest mniejsza od mediany.Odwiedza prawe dziecko tylko wtedy, gdy górna granica zapytania jest większa lub równa medianie.Dzięki temu eliminowane są z obliczeń całe obszary przestrzeni, które na pewno nie zawierają wyników, co skraca czas wyszukiwania do skali logarytmicznej w typowych przypadkach.

2.4.2. Moduł kdtreeVis

Moduł ten stanowi kluczowy element projektu w zakresie wizualizacji. Łączy on logiczną strukturę KD-drzewa z silnikiem graficznym, umożliwiając generowanie statycznych obrazów oraz animacji procesów wyszukiwania.

2.4.2.1. Klasa KDNode

Podstawowa jednostka strukturalna drzewa. Przechowuje pojedynczy obiekt punktu oraz referencje do dwóch potencjalnych dzieci (lewego i prawego), reprezentujących podziały przestrzeni.

2.4.2.2. Klasa KDTree

Główna klasa zarządzająca strukturą. W konstruktorze przyjmuje listę punktów i uruchamia proces budowy drzewa.

build: Funkcja rekurencyjna budująca drzewo. W każdym kroku wybiera oś podziału (naprzemiennie X i Y), sortuje punkty według tej osi i wybiera medianę jako korzeń poddrzewa, co zapewnia optymalne wyważenie struktury.query_range: Publiczna metoda pozwalająca na znalezienie wszystkich punktów w zadanym prostokącie.

2.4.2.3. Funkcja kd_search_plain

Realizuje standardowe, efektywne przeszukiwanie drzewa bez narzutu wizualizacyjnego. Sprawdza, czy punkt w bieżącym węźle mieści się w zapytaniu i decyduje, do których gałęzi (dzieci) należy zajrzeć na podstawie współrzędnych obszaru.

2.4.2.4. Funkcje pomocnicze prostokątów

Zbiór funkcji zapewniających kompatybilność między różnymi reprezentacjami obszarów:rect_lrbt: Konwertuje parametry prostokąta na format (lewo, prawo, dół, góra).rect_contains_point: Uniwersalna metoda sprawdzająca przynależność punktu do obszaru.add_rect_lrbt: Ułatwia rysowanie krawędzi prostokąta na warstwie wizualizatora.

2.4.2.5. Funkcja world_bounds

Analizuje cały zbiór punktów, aby wyznaczyć minimalne i maksymalne współrzędne. Dodaje margines bezpieczeństwa (pad_ratio), aby chmura punktów była estetycznie wycentrowana na generowanych obrazach.

2.4.2.6. Funkcja draw_kdtree_splits

Funkcja rekurencyjna, która nanosi na wykres linie podziału przestrzeni. Wizualizuje, jak drzewo „kroi” płaszczyznę na coraz mniejsze obszary w zależności od głębokości węzła.

2.4.2.7. Funkcja kd_query_with_visualization

Najbardziej rozbudowana funkcja wyszukiwania. Oprócz logiki filtrowania punktów, w każdym kroku komunikuje się z obiektem Visualizer. Podświetla obecnie sprawdzaną linię podziału (na pomarańczowo), zaznacza znalezione punkty (na czerwono) i opcjonalnie pokazuje obszary, które zostały odrzucone (pruning). Każda taka operacja staje się pojedynczą klatką animacji.

2.4.2.8. Funkcje renderujące (render_kdtree_image / gif)

render_kdtree_image_from_csv: Tworzy statyczny plik PNG przedstawiający finalny stan drzewa: chmurę punktów, wszystkie linie podziału oraz zaznaczony obszar zapytania wraz z wynikami.render_kdtree_gif_from_csv: Generuje animację GIF pokazującą proces przeszukiwania „krok po kroku". Pozwala to zrozumieć, dlaczego algorytm pomija niektóre gałęzie drzewa.

2.4.2.9. Funkcje uruchamiające (runners)

run_images_for_N_kdtree oraz run_gifs_for_N_kdtree: Automatyzują przetwarzanie całych folderów z danymi dla konkretnej liczebności punktów N.run_for_custom_kdtree: Specjalny skrypt do obsługi plików stworzonych ręcznie przez użytkownika w folderze custom.

2.4.3. Moduł *visualization*

Moduł odpowiada za wizualizację działania KD-Tree podczas wykonywania zapytania zakresowego. Łączy strukturę danych (KDTree) z silnikiem graficznym (Visualizer), aby wygenerować animację GIF pokazującą kolejne odwiedzane obszary oraz punkty spełniające warunek zapytania.

2.4.3.1. Funkcja *draw_kd_tree_structure*

Rysuje statyczną „mapę" podziałów KD-Tree w zadanym obszarze świata. Dla każdego węzła wyznacza oś podziału na podstawie głębokości (naprzemiennie X oraz Y), a następnie dodaje do wizualizatora pojedynczy odcinek reprezentujący linię cięcia przestrzeni.

W kolejnym kroku wylicza dwa nowe prostokąty odpowiadające przestrzeni dzieci węzła (lewego i prawego) i rekurencyjnie kontynuuje rysowanie, aż do osiągnięcia limitu głębokości (*max_levels*) lub końca gałęzi.

2.4.3.2. Funkcja *kd_query_with_visualization*

Realizuje rekurencyjne przeszukiwanie KD-Tree z jednoczesną animacją. Funkcja działa w obrębie aktualnego obszaru (*current_rect*), który jest dynamicznie wyznaczany dla kolejnych węzłów na podstawie ich punktu podziału.

W każdej iteracji:
- odrzuca gałąź, jeśli *current_rect* nie przecina się z *query_rect*,
- podświetla aktualnie odwiedzany prostokąt (wizualny krok algorytmu),
- sprawdza, czy punkt w węźle znajduje się w *query_rect* i w razie sukcesu dodaje go do listy wyników oraz zaznacza na wykresie,
- dzieli *current_rect* na prostokąty dzieci i rekurencyjnie schodzi do lewego oraz prawego poddrzewa,
- po zakończeniu obsługi węzła usuwa podświetlenie, aby animacja pokazywała aktualny „stan przeszukiwania".

2.4.3.3. Funkcja *render_kdtree_gif*

Główna funkcja generująca animację GIF z przebiegu zapytania w KD-Tree. Inicjalizuje obiekt *Visualizer*, przygotowuje scenę oraz uruchamia proces budowy drzewa i wizualizacji przeszukiwania.

Kroki działania: 1) Tworzy wizualizator, dodaje siatkę, ustawia równe skale osi i tytuł wykresu. 2) Rysuje wszystkie punkty wejściowe jako tło (dla kontekstu przestrzennego). 3) Buduje strukturę KD--Tree na podstawie listy punktów (*tree = KDTree(points)*). 4) Rysuje linie podziału drzewa w granicach *world_rect* za pomocą *draw_kd_tree_structure*, ograniczając głębokość, aby obraz pozostał czytelny. 5) Dodaje na wykres prostokąt zapytania (*query_rect*), aby użytkownik widział obszar wyszukiwania. 6) Uruchamia animowane przeszukiwanie poprzez *kd_query_with_visualization*, które krok po kroku podświetla odwiedzane obszary i zaznacza znalezione punkty. 7) Zapisuje wynik jako plik GIF (*vis.save_gif*) z zadanym interwałem między klatkami i wypisuje ścieżkę do pliku wyjściowego.

2.4.3.4. Uwagi o danych wejściowych

- *points*: lista obiektów typu *Point*, które stanowią dane wejściowe do budowy KD-Tree.
- *query_rect*: prostokąt zapytania typu *Rect*, definiujący obszar wyszukiwania.
- *world_rect*: prostokąt świata typu *Rect*, określający zakres rysowania i obszar startowy rekurencji.
- *out_gif*: nazwa/ścieżka pliku wynikowego GIF.

2.4.3.5. Przykładowe użycie

Na końcu pliku tworzony jest przykładowy zbiór punktów losowych, definiowany jest *world_rect* (zakres rysowania) oraz *query_rect* (obszar zapytania), po czym wywoływana jest funkcja *render_kdtree_gif* w celu wygenerowania animacji.

2.5. Pakiet *data_generators*

Zawiera funkcje odpowiedzialne za generowanie zbiorów danych o różnych charakterystykach. Wykorzystano biblioteki *numpy*, *matplotlib* oraz *math*.

2.5.1. Moduł *data_generators*

Zawiera funkcje generujące zbiory punktów o różnych właściwościach.

2.5.1.1. Funkcja *generate_uniform_points*

Generuje punkty rozrzucone równomiernie wewnątrz kwadratu/prostokąta zdefiniowanego przez granice *left* i *right*.

2.5.1.2. Funkcja *generate_normal_points*

Generuje punkty o rozkładzie normalnym (Gaussa). Najwięcej punktów znajduje się blisko środka (*mean*), a im dalej, tym jest ich mniej. Tworzy to charakterystyczną „rozmytą plamę".

2.5.1.3. Funkcja *generate_collinear_segment_points*

Generuje punkty leżące na jednej linii prostej (współliniowe) między punktami *a* i *b*.

2.5.1.4. Funkcja *generate_rectangle_points*

Generuje punkty leżące tylko na obwodzie prostokąta. Funkcja najpierw oblicza długości boków, a następnie rozdziela punkty proporcjonalnie do ich długości.

2.5.1.5. Funkcja *generate_square_points*

Tworzy specyficzny kształt: 4 wierzchołki, punkty na dwóch bokach (osiach) oraz punkty na obu przekątnych kwadratu. Często używane do testowania przypadków szczególnych w algorytmach.

2.5.1.6. Funkcja *generate_grid_points*

Tworzy regularną siatkę punktów (jak na kartce w kratkę) o wymiarach $n \times n$.

2.5.1.7. Funkcja *generate_clustered_points*

Tworzy kilka oddzielnych grup (klastrów). Każda grupa ma swój środek i określone rozproszenie (*std*).

2.5.2. Moduł *query_picker*

Zawiera funkcje odpowiedzialne za tworzenie prostokątu (*query*) dla każdego zbioru danych.

2.5.2.1. Funkcja *load_points_csv_xy*

Wczytuje współrzędne punktów z pliku CSV. Pomija nagłówek i parsuje każdą linię do postaci krotki (x, y), tworząc listę gotową do wizualizacji lub obliczeń.

2.5.2.2. Funkcja *save_query_rect*

Zapisuje parametry prostokąta zapytania (Rect) do pliku CSV. Plik wynikowy zawiera nagłówek oraz wartości: środek prostokąta (cx, cy) oraz jego połowiczne wymiary (hw, hh).

2.5.2.3. Funkcja *rect_from_corners*

Konwertuje współrzędne dwóch dowolnych narożników (x_0, y_0 oraz x_1, y_1) na obiekt klasy Rect. Funkcja automatycznie wyznacza środek i promienie (połowy szerokości/wysokości), dbając o poprawność obliczeń niezależnie od kolejności zaznaczenia punktów.

2.5.2.4. Funkcja *pick_query_for_existing_csv*

Główna funkcja interaktywna. Wyświetla wykres z punktami wczytanymi z pliku CSV. Umożliwia użytkownikowi ręczne zaznaczenie obszaru zapytania (prostokąta) za pomocą prawego przycisku myszy (PPM + przeciąganie). Po naciśnięciu klawisza Enter parametry obszaru są zapisywane w nowym pliku z rozszerzeniem .query.csv.

2.5.2.5. Funkcja *pick_queries_in_folder*

Automatyzuje proces tworzenia zapytań dla całego katalogu. Przeszukuje wskazany folder w poszukiwaniu plików *CSV* (pomijając te, które już są plikami zapytań) i po kolei uruchamia dla nich interaktywne narzędzie wyboru obszaru.

2.5.3. Moduł *generate_custom*

Odpowiada za interfejs do generowania danych przez użytkownika.

2.5.4. Moduł *custom_data_generator*

Odpowiada za zapisanie i zinterpretowanie danych wprowadzonych przez użytkownika.

2.5.4.1. Funkcja *save_csv*

Funkcja pomocnicza, która tworzy folder docelowy (jeśli nie istnieje) i zapisuje listę punktów do pliku CSV z nagłówkiem x, y.

2.5.4.2. Funkcja *save_query_rect*

Zapisuje parametry zdefiniowanego prostokąta zapytania do osobnego pliku CSV. Przechowuje dane o środku (cx, cy) oraz połowicznych wymiarach (hw, hh) obszaru.

2.5.4.3. Funkcja *rect_from_corners*

Przelicza współrzędne dwóch punktów (kliknięcie i zwolnienie myszy) na sformatowany obiekt Rect. Oblicza geometryczny środek i odległości do krawędzi, co pozwala na stworzenie obiektu niezależnie od tego, w którym kierunku przeciągnięto mysz.

2.5.4.4. Funkcja *make_custom_points_and_query*

Uruchamia interaktywne okno edycji oparte na bibliotece matplotlib. Posiada wbudowaną logikę obsługi zdarzeń:

LPM (Lewy Przycisk Myszy): Dodaje nowy punkt w miejscu kliknięcia.

PPM (Prawy Przycisk Myszy): Usuwa punkt znajdujący się najbliżej kursora.

RectangleSelector: Pozwala na rysowanie prostokąta zapytania poprzez przeciąganie myszą.

Klawisz 'Enter': Kończy edycję i zapisuje dane.

Klawisz 'C': Czyści wszystkie postawione punkty.

Klawisz 'R': Resetuje (usuwa) narysowany prostokąt zapytania.

Jeśli użytkownik nie zdefiniuje własnego prostokąta zapytania, funkcja automatycznie tworzy go tak, aby obejmował wszystkie postawione punkty z 5% marginesem bezpieczeństwa.

2.6. Pakiet *points_util*

2.6.1. Moduł *points_classes*

2.6.1.1. Klasa *Point*

Prosta struktura danych reprezentująca punkt w przestrzeni dwuwymiarowej. Użycie parametru frozen=True gwarantuje niezmienność obiektu (niemutowalność), co zapobiega przypadkowym modyfikacjom współrzędnych podczas działania algorytmów.

2.6.1.2. Klasa *Rect*

Reprezentuje prostokąt osiowo wyrównany (AABB – Axis-Aligned Bounding Box). W przeciwieństwie do standardowego zapisu (lewy górny róg + wymiary), klasa ta definiuje prostokąt poprzez jego środek (cx, cy) oraz połowiczne wymiary (hw, hh). Taki zapis znacznie ułatwia proces podziału węzła QuadTree na cztery mniejsze dzieci.

2.6.1.3. Właściwości (Properties) *left, right, bottom, top*

Zestaw dynamicznych właściwości, które przeliczają parametry środka i promieni na klasyczne krawędzie prostokąta. Upraszczają one czytelność kodu w testach logicznych, eliminując konieczność ręcznego dodawania i odejmowania wartości hw i hh w całym programie.

2.6.1.4. Funkcja *contains_point*

Sprawdza, czy dany punkt znajduje się wewnątrz prostokąta. Zastosowano tu logikę półotwartą: punkt należy do obszaru, jeśli jego współrzędne mieszczą się w przedziale [`left`, `right`) oraz [`bottom`, `top`). Jest to kluczowe przy podziale QuadTree – dzięki temu punkt leżący dokładnie na granicy dwóch obszarów zostanie przypisany tylko do jednego z nich, co zapobiega dublowaniu danych.

2.6.1.5. Funkcja *intersects*

Rozstrzyga, czy dwa prostokąty mają ze sobą część wspólną (choćby krawędź). Wykorzystuje metodę zaprzeczenia – zamiast sprawdzać nakładanie się, sprawdza czy prostokąty są od siebie całkowicie odseparowane w poziomie lub pionie. Jeśli nie zachodzi żadna z tych separacji, prostokąty muszą na siebie nachodzić.

2.6.2. Moduł *points_loaders*

2.6.2.1. Funkcja *load_points_csv*

Odpowiada za deserializację danych punktowych. Wczytuje plik *.csv*, pomija nagłówek, a każdą parę współrzędnych zamienia w instancję klasy Point. Dzięki temu w dalszej części algorytmu (np. wewnątrz *QuadTree*) możemy korzystać z czytelnych odwołań typu p.x i p.y zamiast indeksów tablicy *row[0].*

2.6.2.2. Funkcja *load_query_rect*

Wczytuje parametry obszaru zapytania z dedykowanego pliku .query.csv. Pobiera środek oraz połowiczne wymiary prostokąta, zwracając gotowy obiekt klasy *Rect*. Jest to funkcja komplementarna do generatorów, pozwalająca na szybkie odtworzenie zapisanego wcześniej „okna wyszukiwania" bez konieczności jego ponownego rysowania.

2.6.3. Moduł *prepare_queries*

2.6.3.1. Funkcja *prepare_queries_for_N*

Automatyzuje proces tworzenia zapytań dla zbiorów danych o określonej liczebności N. Przeszukuje folder *output* w poszukiwaniu plików *.csv* i dla każdego z nich uruchamia interaktywne narzędzie wyboru obszaru. Funkcja posiada mechanizm overwrite, który pozwala zdecydować, czy istniejące już pliki zapytań mają być pominięte, czy stworzone na nowo.

2.6.3.2. Funkcja *prepare_queries_for_custom*

Działa analogicznie do poprzedniej funkcji, ale koncentruje się na katalogu z niestandardowymi zestawami danych (*output/custom*). Jest to szczególnie przydatne, gdy użytkownik ręcznie przygotował wiele różnych chmur punktów i chce dla każdej z nich szybko zdefiniować obszar testowy bez ręcznego wpisywania ścieżek do skryptu.

2.6.4. Moduł *time_compare*

2.6.4.1. Funkcja *load_xy_csv*

Odpowiada za niskopoziomowe wczytywanie współrzędnych z plików *.csv*. W przeciwieństwie do innych loaderów, zwraca surową listę krotek (x, y), co pozwala na szybkie wstępne przetworzenie danych przed ich konwersją na obiekty klasowe.

2.6.4.2. Funkcja load_query_rect

Wczytuje parametry obszaru zapytania z pliku o rozszerzeniu .query.csv. Wydobywa dane o środku i promieniach prostokąta, a następnie inicjalizuje obiekt klasy *Rect*, który jest niezbędny do przeprowadzenia testów przeszukiwania obszarów w obu strukturach.

2.6.4.3. Funkcja *dataset_name_from_file*

Pomocnicza funkcja mapująca techniczne nazwy plików na czytelne, polskie opisy (np. „normal" na „rozkład normalny"). Pozwala to na automatyczne generowanie estetycznych raportów i tabel bez konieczności ręcznej edycji nazw zbiorów danych.

2.6.4.4. Funkcja bench_both_file

Kluczowa funkcja przeprowadzająca test porównawczy dla pojedynczego zestawu danych. Dla tego samego zbioru punktów wykonuje następujące operacje:Buduje strukturę QuadTree oraz KD-drzewo,

mierząc czas ich konstrukcji za pomocą perf_counter.Wykonuje zapytanie o punkty wewnątrz zdefiniowanego prostokąta dla obu struktur.Zlicza trafienia i mierzy czas odpowiedzi.Wszystkie te parametry (czasy, liczebność, głębokość) zwraca w formie słownika gotowego do dalszej analizy.

2.6.4.5. Funkcja bench_both_all

Automatyzuje proces testowania dla całej struktury katalogów. Przeszukuje rekurencyjnie wskazany folder, odnajduje pary plików (punkty + zapytanie) i dla każdej pary wywołuje funkcję *bench_both_file*. Wyniki są wyświetlane na bieżąco w konsoli oraz zbierane w zbiorczą listę.

2.6.4.6. Funkcja save_separate_tables_both

Odpowiada za finalne zestawienie wyników i ich eksport. Przekształca zebrane dane w ramkę danych (*DataFrame*), a następnie grupuje je według typu zbioru danych. Wyniki są zapisywane do osobnych plików (Excel lub *.csv*) w folderze *times*. Funkcja automatycznie dba o czytelne nazewnictwo kolumn (np. zamiana *qt_build_s* na *QuadTreeBuildTime*), co ułatwia bezpośrednie wykorzystanie tych danych w pracach naukowych lub dokumentacji technicznej.

3. Część użytkownika

Część opisująca przykłady uruchamiania programu oraz korzystania z jego poszczególnych modułów i pakietów.

3.1. Pakiet *quadtree*

3.1.1. Moduł *quadtree*

Implementacja QuadTree.

3.1.1.1. Inicjalizacja struktury danych

```
xy = load_xy_csv(csv_path)
n = len(xy)
pts = [Point(x, y) for (x, y) in xy]
world = bounding_rect_pairwise(pts)
qt = QuadTree(world, capacity=capacity, max_depth=max_depth)
for p in pts:
    qt.insert(p)
```

Powyższy fragment kodu inicjalizuje drzewo czwórkowe na podstawie punktów wczytanych z pliku CSV. Najpierw dane są zamieniane na listę obiektów Point, a następnie wyznaczany jest minimalny prostokąt obejmujący cały zbiór (bounding_rect_pairwise), który staje się granicą świata drzewa. Na tej podstawie tworzony jest obiekt QuadTree z ustaloną pojemnością liścia (capacity) i maksymalną głębokością (max_depth). Na końcu każdy punkt jest wstawiany do struktury metodą insert, co powoduje rekurencyjny podział przestrzeni na ćwiartki tam, gdzie gęstość danych tego wymaga.

3.1.1.2. Zapytanie o punkty z zadanego obszaru

```
query_rect = load_query_rect(query_path)
qt.query(query_rect)
```

Ten fragment kodu realizuje zapytanie zakresowe w drzewie czwórkowym. Najpierw z pliku wczytywany jest prostokąt zapytania (query_rect), który definiuje interesujący obszar. Następnie wywołanie qt.query(query_rect) przeszukuje tylko te węzły QuadTree, których obszary przecinają się z prostokątem, i zwraca listę punktów leżących wewnątrz zadanego zakresu.

3.1.1.3. Generowanie animacji w formacie *.gif* oraz zdjęć (*.png*)

```
def render_quadtree_gif_from_csv(
    csv_path: str,
    query_square: Rect,
    capacity=8,
    max_depth=16,
    out_gif="quadtree.gif",
    out_png=None,
    sample_points=2000,
    max_levels=10,
    show_pruned=False,
    interval=120
):
```

Powyżej znajduje się nagłówek funkcji render_quadtree_gif_from_csv(…), która ma wczytać dane punktowe z pliku CSV (csv_path), zbudować dla nich strukturę quadtree w zadanym obszarze (query_square) i wyrenderować jej kolejne etapy.

Parametry pozwalają kontrolować m.in.:

- capacity – ile punktów może trafić do liścia zanim nastąpi podział,
- max_depth – maksymalną głębokość drzewa,
- out_gif / out_png – nazwę pliku wynikowego GIF oraz opcjonalny zapis pojedynczej klatki jako obraz,
- sample_points, max_levels, show_pruned, interval – ile punktów/poziomów pokazać, czy rysować „ucięte” gałęzie oraz odstęp czasu między klatkami (ms).

```
def render_quadtree_image_from_csv(
    csv_path: str,
    query_square: Rect,
    capacity=8,
    max_depth=16,
    out_png="quadtree.png",
    sample_points=5000,
    max_levels=10,
):
```

Powyżej znajduje się nagłówek funkcji render_quadtree_image_from_csv(…), która wczytuje punkty z pliku CSV (csv_path), buduje dla nich quadtree w zadanym obszarze (query_square), a następnie zapisuje wynikową wizualizację jako pojedynczy obraz PNG (out_png, domyślnie „quadtree.png”).

Najważniejsze parametry:

- capacity – maks. liczba punktów w węźle/liściu zanim nastąpi podział,
- max_depth – maksymalna głębokość drzewa,
- sample_points – ile punktów wziąć do wizualizacji (tu domyślnie 5000),
- max_levels – ile poziomów drzewa narysować (tu domyślnie 10).

3.2. Pakiet *kdtree*

3.2.1. Moduł *kdtree*

Odpowiedzialny za implementację KD-Tree.

3.2.1.1. Inicjalizacja struktury danych i zapytanie o punkty z zadanego obszaru

```
xy = load_xy_csv(csv_path)
n = len(xy)
pts = [Point(x, y) for (x, y) in xy]
kd = KDTree(list(pts))
query_rect = load_query_rect(query_path)
kd.query_range(query_rect)
```

Ten fragment pokazuje inicjalizację danych wejściowych oraz wykonanie zapytania zakresowego (range query) w strukturze KDTree:

- xy = load_xy_csv(csv_path) – wczytuje z CSV współrzędne punktów (np. listę par (x, y)).
- n = len(xy) – liczy liczbę wczytanych punktów.
- pts = [Point(x, y) for (x, y) in xy] – zamienia surowe pary liczb na obiekty Point.
- kd = KDTree(list(pts)) – buduje drzewo KD (KDTree) na podstawie tych punktów.
- query_rect = load_query_rect(query_path) – wczytuje definicję prostokąta zapytania (obszar, który nas interesuje).
- kd.query_range(query_rect) – wyszukuje w KDTree wszystkie punkty znajdujące się wewnątrz zadanego prostokąta.

3.2.1.2. Generowanie animacji w formacie *.gif* oraz zdjęć (*.png*)

```
def render_kdtree_gif_from_csv(
    csv_path: str,
    query_rect,
    PointClass,
    out_gif="kdtree.gif",
    out_png=None,
    sample_points=2000,
    max_levels=10,
    show_pruned=False,
    interval=120,
):
```

Ten fragment definiuje funkcję odpowiedzialną za generowanie wizualizacji działania KD-drzewa na podstawie danych z pliku CSV. Przyjmuje ścieżkę do pliku z punktami (csv_path) oraz prostokąt zapytania (query_rect), a następnie zapisuje wynik w formie animacji .gif (domyślnie kdtree.gif) oraz opcjonalnie jako obraz .png (out_png). Parametr sample_points ogranicza liczbę rysowanych punktów, aby wizualizacja pozostała czytelna i szybka, natomiast max_levels kontroluje głębokość rysowanych podziałów drzewa. Flaga show_pruned pozwala zdecydować, czy w animacji mają być zaznaczane odrzucane gałęzie, a interval określa odstęp czasu między klatkami GIF.

```
def render_kdtree_image_from_csv(
    csv_path: str,
    query_rect,
    PointClass,
    out_png="kdtree.png",
    sample_points=5000,
    max_levels=10,
):
```

Powyżej znajduje się nagłówek funkcji render_kdtree_image_from_csv(…), która wczytuje punkty z pliku CSV (csv_path), buduje z nich KDTree i zapisuje wizualizację drzewa (podział przestrzeni) jako pojedynczy obraz PNG (out_png, domyślnie „kdtree.png”).

Co oznaczają parametry:

- csv_path: str – ścieżka do pliku z punktami,
- query_rect – prostokąt/obszar, który może być zaznaczony na rysunku albo użyty do pokazania zapytania zakresowego,
- PointClass – klasa używana do tworzenia obiektów punktów (np. Point), dzięki czemu funkcja może działać z różnymi implementacjami punktu,
- out_png – nazwa pliku wynikowego,
- sample_points – ile punktów maksymalnie wziąć do wizualizacji (żeby rysunek był czytelny i szybki),
- max_levels – ile poziomów drzewa narysować (ogranicza „gęstość” podziałów na rysunku).

3.2.1.3. Funkcje odpowiedzialne za generowanie *gif*-ów oraz plików *.png* dla każdego zbioru danych o liczności n (*runnery*)

```
def run_gifs_for_N_kdtree(
    N: int,
    sample_points=1500,
    max_levels=10,
    show_pruned=False,
    interval=120,
    also_save_png=True
):
```

```
def run_images_for_N_kdtree(
    N: int,
    sample_points=5000,
    max_levels=10
):
```

```
def run_images_for_N(
    N: int,
    capacity=8,
    max_depth=16,
    sample_points=5000,
    max_levels=10
):
```

```
def run_for_N(
    N: int,
    capacity=8,
    max_depth=16,
    sample_points=1500,
    max_levels=10,
    show_pruned=False,
    interval=120,
    also_save_png=True
):
```

Powyższe fragmenty kodu służą do uzyskania animacji oraz zdjęć dla kolejno KD-drzewa oraz QuadTree. Pojedyncze wywołanie każdej z funkcji tworzy zasoby dla każdego rodzaju zbioru danych o danej liczności n.

4. Sprawozdanie

- System operacyjny: Microsoft Windows (x64)
- Procesor: AMD Ryzen 7 7735HS (3.2 GHz)
- Pamięć RAM: 32 GB (4800 MHz)
- Środowisko: PyCharm
- Język: Python 3.13.5

Precyzja przechowywania zmiennych i obliczeń to *float64*

4.1. Opis problemu

W wielu obszarach informatyki — m.in. w systemach GIS oraz grafice komputerowej — kluczowe znaczenie ma sprawne przeszukiwanie danych przestrzennych. W tym sprawozdaniu porównujemy wydajność dwóch struktur przeznaczonych do partycjonowania przestrzeni: KD-drzewa oraz drzewa czwórkowego (QuadTree). Rozważania ograniczono do przypadku dwuwymiarowego.

Obie struktury przyspieszają wyszukiwanie punktów w zadanym obszarze dzięki rekurencyjnemu dzieleniu przestrzeni. KD-drzewo wykonuje cięcia naprzemiennie względem kolejnych wymiarów, natomiast QuadTree dzieli obszar na cztery równe części. Dla zrównoważonego KD-drzewa czas budowy wynosi $O(n \log n)$, a koszt zapytania zakresowego to $O(\sqrt{n} + d)$, gdzie d oznacza liczbę znalezionych punktów. W przypadku QuadTree budowa ma zwykle złożoność $O(n \log n)$, choć w sytuacji niekorzystnego rozkładu danych może dojść do degeneracji nawet do $O(n^2)$. Dla danych rozłożonych równomiernie wyszukiwanie w QuadTree osiąga złożoność $O(\log n + d)$.

Celem analizy porównawczej jest zbadanie czasu tworzenia struktur oraz efektywności obsługi zapytań przestrzennych przy różnych rozkładach punktów wejściowych.

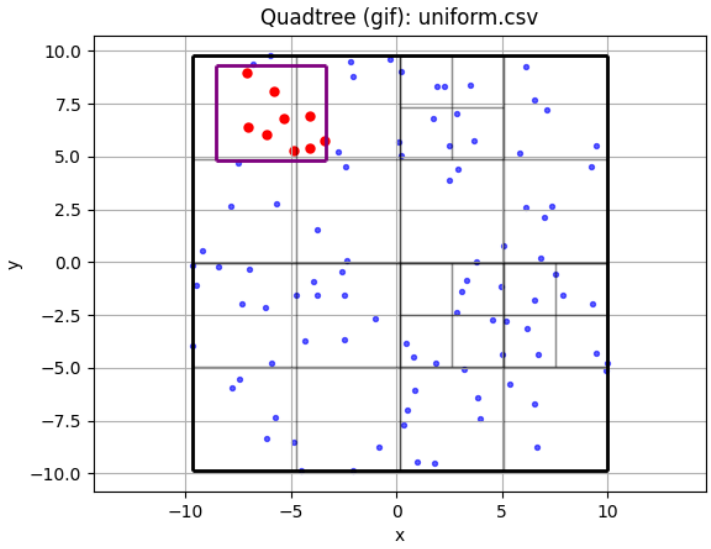
4.2. Realizacja

Na potrzeby porównania wygenerowano zestawy punktów w przestrzeni 2D o zróżnicowanych właściwościach. Dla każdego zbioru przedstawiono wizualizację (dla niewielkiej liczby punktów), graficzne zobrazowanie rezultatu zapytania, a także zestawienie czasów działania. Animacje ilustrujące krok po kroku pracę algorytmów umieszczono w notatniku Jupyter.

4.3. Wyniki

Dla wszystkich zestawów testowych obie struktury zwracały identyczne wyniki zapytań, co potwierdza poprawność przygotowanej implementacji.

4.3.1. Zbiór punktów z rozkładu jednostajnego



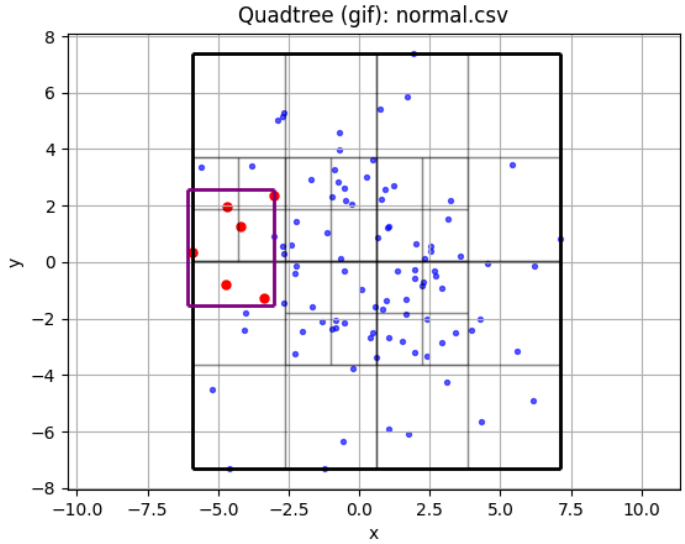
Rysunek 1: Wynik przykładowego zapytania dla zbioru jednostajnego

Liczba punktów		Czas [s]			
Zbioru	Znalezionych	Budowy KD--drzewa	Budowy QuadTree	Zapytania do KD-drzewa	Zapytania do QuadTree
100	9	0,000	0,001	0,000	0,000
1000	73	0,002	0,008	0,000	0,000
10000	464	0,046	0,110	0,001	0,001
100000	5992	0,586	1,476	0,008	0,008
1000000	39212	14,884	21,192	0,062	0,065

Tabela 1: Porównanie czasów budowy i zapytań dla KD-drzewa oraz QuadTree.

Wszystkie wygenerowane zbiory testowe są wizualnie zbliżone do tego pokazanego na Rysunku 1 — różnią się przede wszystkim liczbą punktów. Zgodnie z wynikami przedstawionymi w Tabeli 1, w większości przypadków czas budowy KD-drzewa jest krótszy niż w przypadku QuadTree. Dla małej liczności zbioru zapytania były obsługiwane szybciej przez QuadTree, natomiast przy większej liczbie punktów lepsze wyniki uzyskiwało KD-drzewo.

4.3.2. Zbiór punktów z rozkładu normalnego



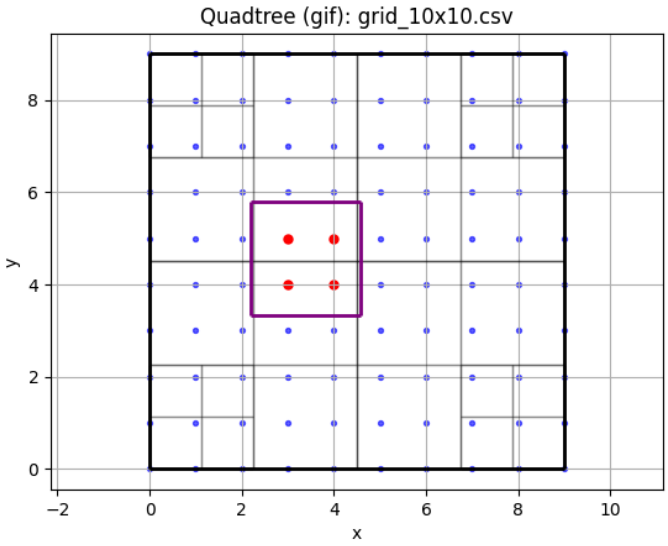
Rysunek 2: Wynik przykładowego zapytania dla zbioru punktów na siatce

Liczba punktów		Czas [s]			
Zbioru	Znalezionych	Budowy KD--drzewa	Budowy QuadTree	Zapytania do KD-drzewa	Zapytania do QuadTree
100	6	0,000	0,001	0,000	0,000
1000	137	0,003	0,008	0,000	0,000
10000	1127	0,058	0,150	0,002	0,002
100000	17069	0,858	1,806	0,025	0,024
1000000	586556	13,786	21,978	0,907	0,885

Tabela 2: Tabela: Porównanie czasów budowy i zapytań dla KD-drzewa oraz QuadTree (rozkład normalny).

W Tabeli 2 przedstawiono pomiary czasowe funkcji dla punktów rozmieszczonych za pomocą rozkładu normalnego. Przy 100 punktach QuadTree buduje się szybciej niż KD-drzewo, natomiast KD-drzewo realizuje zapytania co najmniej dwukrotnie szybciej. Gdy liczba punktów mieści się w przedziale od 100 do 1000, czasy budowy obu struktur są zbliżone, jednak w tym zakresie szybciej odpowiada QuadTree. Z kolei dla większych zbiorów KD-drzewo okazuje się efektywniejsze. W praktyce, dla 100–1000 punktów przewagę w czasie zapytań ma QuadTree, a dla 50 punktów oraz powyżej 100000 punktów to KD-drzewo staje się odpowiednio około dwukrotnie i trzykrotnie szybsze.

4.3.3. Zbiór punktów na siatce



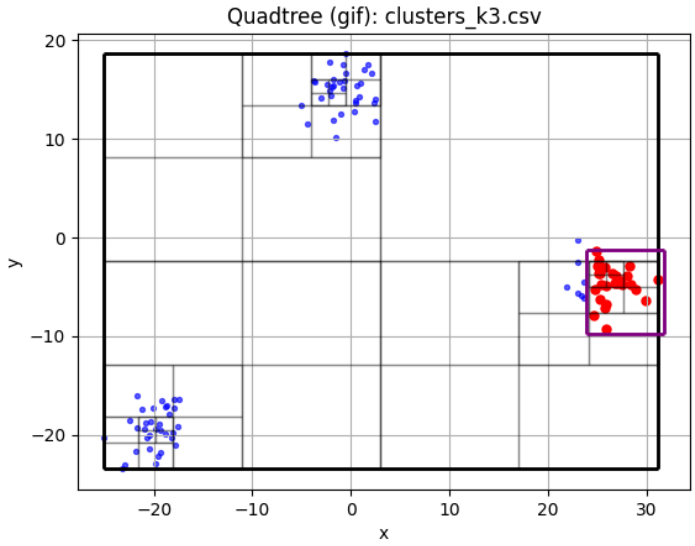
Rysunek 3: Wynik przykładowego zapytania dla zbioru punktów na siatce

Liczba punktów		Czas [s]			
Zbioru	Znalezionych	Budowy KD--drzewa	Budowy QuadTree	Zapytania do KD-drzewa	Zapytania do QuadTree
100	4	0,000	0,000	0,000	0,000
961	48	0,033	0,007	0,000	0,000
10000	378	0,101	0,159	0,001	0,001
99856	3481	0,533	1,417	0,004	0,004
1000000	22748	5,323	15,536	0,022	0,027

Tabela 3: Porównanie czasów budowy i zapytań dla KD-drzewa oraz QuadTree (siatka).

Dla zbioru punktów ułożonych na siatce oba algorytmy zwracają porównywalny wynik zapytania, co widać na wizualizacji (te same punkty trafiają do obszaru). Z Tabeli 3 wynika, że czas budowy KD--drzewa jest konsekwentnie mniejszy niż QuadTree, a różnica rośnie wraz z licznością zbioru (dla 1 000 000 punktów 5,323 s vs 15,536 s). Czasy zapytań dla małych i średnich zbiorów są praktycznie identyczne, natomiast dla największego zbioru KD-drzewo jest nieco szybsze (0,022 s vs 0,027 s). Oznacza to, że przy danych siatkowych KD-drzewo daje lepszą efektywność budowy bez pogorszenia jakości i czasu zapytań.

4.3.4. Zbiór punktów w klastrach



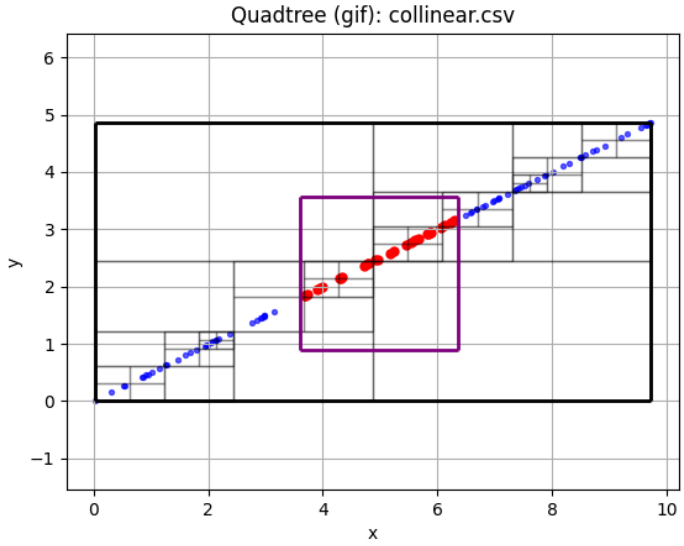
Rysunek 4: Wynik przykładowego zapytania dla zbioru punktów w klastrach

Liczba punktów		Czas [s]			
Zbioru	Znalezionych	Budowy KD--drzewa	Budowy QuadTree	Zapytania do KD-drzewa	Zapytania do QuadTree
99	26	0,000	0,001	0,000	0,000
999	289	0,004	0,011	0,000	0,000
9999	1788	0,086	0,133	0,002	0,002
99999	18764	0,724	1,807	0,025	0,025
999999	330474	13,164	22,010	0,472	0,457

Tabela 4: Tabela: Porównanie czasów budowy i zapytań dla KD-drzewa oraz QuadTree (klastry).

Dla danych skupionych w klastrach QuadTree tworzy znacznie więcej podziałów w rejonach dużej gęstości punktów, co widać na wizualizacji. Z Tabela 4 wynika, że KD-drzewo buduje się szybciej dla wszystkich rozmiarów zbioru, a przewaga staje się wyraźna przy dużej liczbie punktów (np. 13,164 s vs 22,010 s). Czasy zapytań dla małych i średnich zbiorów są praktycznie takie same, natomiast dla największego zbioru QuadTree jest minimalnie szybsze (0,457 s vs 0,472 s). Sugeruje to, że przy klastrach QuadTree może nieznacznie zyskać na czasie zapytań kosztem wyraźnie dłuższej budowy struktury.

4.3.5. Zbiór punktów wygenerowanych na prostej



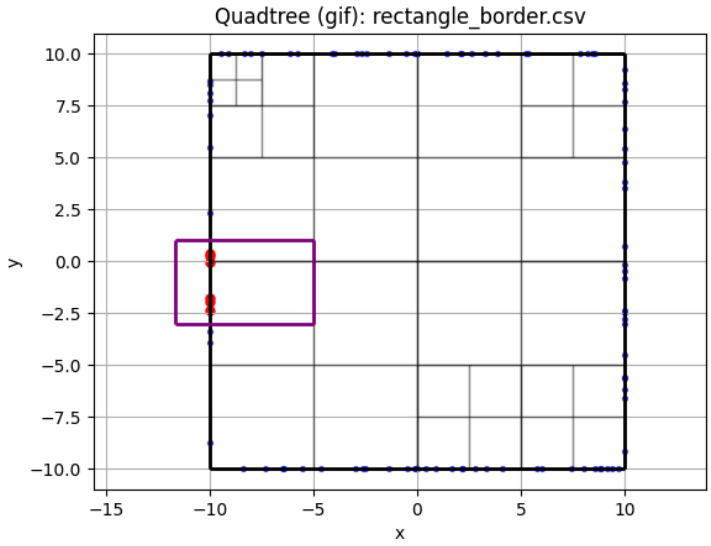
Rysunek 5: Wynik przykładowego zapytania dla zbioru punktów współliniowych

Liczba punktów		Czas [s]			
Zbioru	Znalezionych	Budowy KD--drzewa	Budowy QuadTree	Zapytania do KD-drzewa	Zapytania do QuadTree
100	28	0,000	0,001	0,000	0,000
1000	87	0,003	0,016	0,000	0,000
10000	1054	0,040	0,266	0,002	0,002
100000	18906	0,655	3,366	0,026	0,040
1000000	491113	9,472	33,054	0,756	0,620

Tabela 5: Porównanie czasów budowy i zapytań dla KD-drzewa oraz QuadTree (współliniowe).

Dla punktów leżących na prostej QuadTree wykonuje wiele podziałów wzdłuż wąskiego obszaru zajętego przez dane, co widać na rysunku i przekłada się na rosnący koszt budowy. Z tabeli wynika, że KD-drzewo buduje się zdecydowanie szybciej, a przewaga ta gwałtownie rośnie wraz z liczbą punktów (dla 1 000 000: 9,472 s vs 33,054 s). Dla małych zbiorów czasy zapytań są porównywalne, natomiast przy większych rozmiarach QuadTree zaczyna być wolniejsze (np. dla 100 000: 0,026 s vs 0,040 s). Przy największym zbiorze QuadTree jest nieznacznie szybsze w zapytaniu (0,620 s vs 0,756 s), ale różnica ta nie rekompensuje znacznie dłuższego czasu konstrukcji. Ogólnie, dla danych współliniowych KD-drzewo wypada korzystniej, szczególnie jeśli struktura ma być budowana wielokrotnie lub dla dużych zbiorów.

4.3.6. Zbiór punktów na obwodzie prostokąta



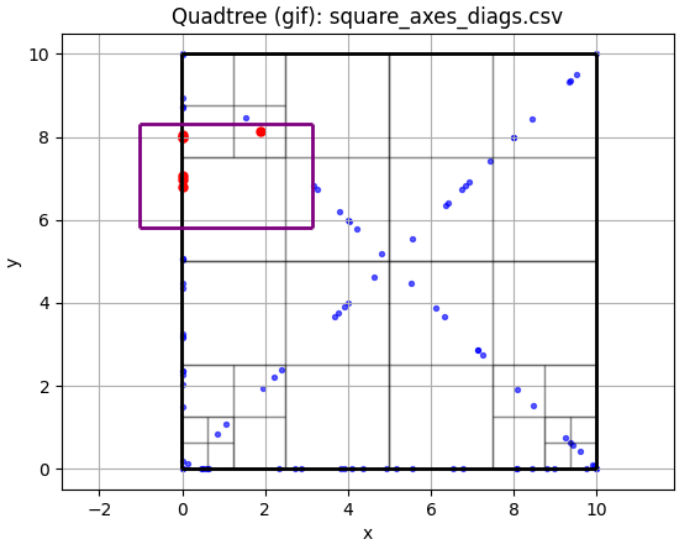
Rysunek 6: Wynik przykładowego zapytania dla zbioru punktów na obwodzie prostokąta

Liczba punktów		Czas [s]			
Zbioru	Znalezionych	Budowy KD--drzewa	Budowy QuadTree	Zapytania do KD-drzewa	Zapytania do QuadTree
100	6	0,000	0,001	0,000	0,000
1000	49	0,002	0,011	0,000	0,000
10000	537	0,033	0,166	0,001	0,001
100000	4917	0,558	2,612	0,007	0,010
1000000	22363	8,282	32,849	0,031	0,041

Tabela 6: Porównanie czasów budowy i zapytań dla KD-drzewa oraz QuadTree (obwód prostokąta).

Dla punktów leżących na obwodzie prostokąta QuadTree wykonuje dodatkowe podziały głównie wzdłuż krawędzi, co zwiększa koszt budowy i jest widoczne na rysunku. Z tabeli wynika, że KD--drzewo buduje się wyraźnie szybciej dla wszystkich rozmiarów zbioru, a różnica rośnie wraz z liczbą punktów (dla 1 000 000: 8,282 s vs 32,849 s). Czasy zapytań są zbliżone dla małych zbiorów, natomiast dla większych danych KD-drzewo utrzymuje przewagę (np. 0,031 s vs 0,041 s dla 1 000 000). W praktyce oznacza to, że przy takim „brzegowym” rozkładzie danych KD-drzewo jest bardziej opłacalne zarówno pod względem konstrukcji, jak i obsługi zapytań.

4.3.7. Zbiór punktów na dwóch bokach kwadratu oraz dwóch jego przekątnych



Rysunek 7: Wynik przykładowego zapytania dla zbioru punktów na dwóch bokach kwadratu oraz dwóch jego przekątnych

Liczba punktów		Czas [s]			
Zbioru	Znalezionych	Budowy KD--drzewa	Budowy QuadTree	Zapytania do KD-drzewa	Zapytania do QuadTree
100	7	0,000	0,001	0,000	0,000
1000	79	0,002	0,011	0,000	0,000
10000	931	0,038	0,203	0,001	0,002
100000	9398	0,665	2,673	0,013	0,020
1000000	35611	10,431	33,780	0,071	0,065

Tabela 7: Porównanie czasów budowy i zapytań dla KD-drzewa oraz QuadTree (kwadrat).

Dla punktów leżących na dwóch bokach kwadratu i jego przekątnych QuadTree wykonuje nierównomierne, gęste podziały w obszarach, gdzie dane są skupione w wąskich pasach, co widać na rysunku. Z tabeli wynika, że KD-drzewo buduje się zdecydowanie szybciej dla każdego rozmiaru zbioru, a przewaga rośnie wraz z licznością (dla 1 000 000: 10,431 s vs 33,780 s). W zapytaniach KD--drzewo jest szybsze dla małych i średnich zbiorów (np. dla 100 000: 0,013 s vs 0,020 s). Dla największego zbioru QuadTree osiąga minimalnie lepszy czas zapytania (0,065 s vs 0,071 s), ale różnica jest niewielka. Sumarycznie KD-drzewo wypada korzystniej, szczególnie gdy istotny jest czas budowy struktury.

4.4. Wnioski

1. Zarówno KD-drzewo, jak i QuadTree poprawnie wykonują wyszukiwanie punktów w określonym obszarze, niezależnie od rodzaju rozkładu danych wejściowych.
2. Dla dużych zbiorów punktów KD-drzewo zwykle osiąga lepsze czasy budowy oraz zapytań, szczególnie gdy dane są równomiernie rozproszone albo mają charakter współliniowy. Wynika to z bardziej zrównoważonej struktury, która ogranicza głębokość drzewa.
3. QuadTree może być korzystniejsze przy małych zbiorach oraz w sytuacjach, gdy punkty są równomiernie rozmieszczone w przestrzeni. Dodatkowo, dla mniej zagęszczonych danych rekurencyjny podział obszaru potrafi skutecznie zmniejszać liczbę wykonywanych operacji względem bardziej złożonych układów.
4. Charakter rozkładu danych w znacznym stopniu determinuje efektywność obu struktur. KD--drzewo lepiej radzi sobie dla danych o rozkładzie jednostajnym, normalnym i współliniowym, natomiast QuadTree ma większy potencjał przy danych o mniejszej gęstości lub bardziej jednorodnym rozmieszczeniu.
5. KD-drzewo jest rozwiązaniem bardziej uniwersalnym i dobrze sprawdza się w zastosowaniach wymagających obsługi dużych zbiorów lub specyficznych rozkładów, podczas gdy QuadTree bywa preferowane dla mniejszych zbiorów, danych mniej zagęszczonych albo wtedy, gdy ważna jest prostota implementacji.
6. Choć w ujęciu teoretycznym złożoności budowy i wyszukiwania obu struktur są zbliżone, w praktyce KD-drzewo zachowuje bardziej stabilną wydajność wraz ze wzrostem liczby punktów. QuadTree natomiast cechuje się większą zmiennością czasów działania w zależności od rozmieszczenia oraz zagęszczenia danych.

Podsumowując, oba podejścia mają swoje zalety i ograniczenia, dlatego wybór odpowiedniej struktury powinien zależeć od charakterystyki danych oraz wymagań aplikacji. KD-drzewo zapewnia większą stabilność i przewagę w zadaniach z dużymi, gęsto rozmieszczonymi zbiorami punktów, natomiast QuadTree może okazać się skuteczniejsze dla mniejszych, mniej zagęszczonych danych oraz w przypadkach, gdy liczy się prostota implementacji.