Wyszukiwanie geometryczne – przeszukiwanie obszarów ortogonalnych – Quadtree oraz Kd-drzewa Dokumentacja

Spis treści

1. Wymagania techniczne ..... 1.
2. Wstęp teoretyczny ..... 2.
2. 1. Algorytm Quadtree ..... 2.
2. 2. Algorytm Kd-tree ..... 3.
3. Opracowanie szczegółów implementacji ..... 4.
3. 1. Moduł geo_structures ..... 4.
3. 2. Moduł quadtree ..... 4.
3. 3. Moduł kd_tree ..... 5.
3. 4. Moduł get_median ..... 6.
4. Wizualizacja ..... 6.
5. Instrukcja obsługi programu ..... 7.
5. 1. Kd_Tree ..... 7.
5. 2. Quadtree ..... 7.
5. 3. Wizualizacja Kd_tree ..... 8.
5. 4. Wizualizacja Quadtree ..... 8.
6. Porównanie struktur - testy czasowe ..... 9.
6. 1. Rezultaty dla zbioru A ..... 11.
6. 2. Rezultaty dla zbioru B ..... 13.
6. 3. Rezultaty dla zbioru C ..... 15.
7. Wnioski ..... 17.
8. Podsumowanie ..... 17.
9. Bibliografia ..... 17.

1. Wymagania techniczne

Implementacja znajduje się na platformie Github w formie repozytorium: https://github.com/onikw/quadtree_kdtree_project

Aby uruchomić implementację zalecamy następujące kroki:

- sklonuj powyższe repozytorium na przykład za pomocą `clone https://github.com/onikw/quadtree_kdtree_project.git`
- używając programu `Anaconda` przejdź do lokalizacji pobranego projektu
- stwórz środowisko:
  ```
  conda create --name quad_kd python=3.9
  conda activate quad_kd
  ```
- następnie uruchom:
  ```
  pip install -U pip setuptools wheel
  pip install -e .
  ```

Alternatywnie można zainstalować wymagane dependencje ręcznie, informacje o nich znajdują się w pliku `requirements.txt`.

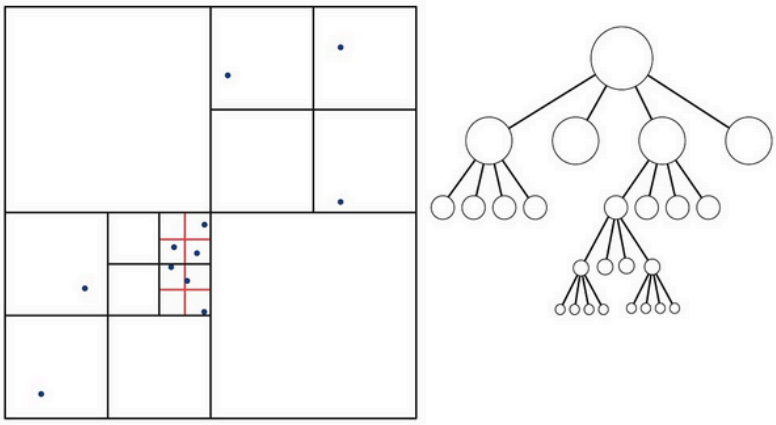
2. Wstęp teoretyczny

2. 1. Algorytm Quadtree

Quadtree to struktura danych w formie drzewa, zaprojektowana do podziału przestrzeni dwuwymiarowej na mniejsze obszary. Proces ten polega na dzieleniu płaszczyzny na równe ćwiartki, a następnie rekurencyjnym podziale każdej z nich, aż spełnione zostaną określone warunki zatrzymania.

W naszym przypadku prostokątna płaszczyzna będzie dzielona na ćwiartki, gdy w danym obszarze znajduje się więcej niż jeden punkt. Podział kończy się, gdy w każdym z obszarów pozostanie maksymalnie jeden punkt. Alternatywnie można przyjąć ograniczenie w postaci maksymalnej głębokości drzewa, co ogranicza liczbę podziałów, pozwalając na obecność większej liczby punktów w jednym obszarze. Podobnie można ustalić inną wartość niż 1 jako akceptowalną liczbę punktów w jednym obszarze.

Cała podzielona płaszczyzna jest reprezentowana przez drzewo czwórkowe. Korzeń drzewa symbolizuje główny prostokąt, a jego gałęzie odpowiadają czterem ćwiartkom powstałym w wyniku podziału. Kolejne poziomy drzewa w analogiczny sposób reprezentują podziały następnych prostokątów. Liście drzewa to obszary, które nie zostały podzielone dalej, spełniając warunek zatrzymania.



Rysunek 1: Schemat drzewa Quadtree

Przeszukiwanie drzewa Quadtree rozpoczynamy od korzenia, wykonując następujące operacje w kolejnych węzłach:

- **Całkowite zawarcie w obszarze poszukiwań:** Jeśli prostokąt reprezentowany przez dziecko węzła mieści się całkowicie w przeszukiwanym obszarze, wszystkie punkty należące do tej gałęzi (rozpoczynającej się od tego węzła) również znajdują się w obszarze. Dodajemy te punkty do zbioru wyników i nie wchodzimy głębiej w gałąź (odcinamy ją od dalszego przeszukiwania).
- **Brak zawarcia w obszarze poszukiwań:** Jeśli prostokąt dziecka węzła w całości znajduje się poza przeszukiwanym obszarem, oznacza to, że żadne punkty z tej gałęzi nie należą do obszaru. W takiej sytuacji kończymy przeszukiwanie tej gałęzi (odcinamy ją).
- **Częściowe zawarcie w obszarze poszukiwań:** Jeśli prostokąt dziecka węzła częściowo zawiera się w obszarze, wchodzimy głębiej w tę gałąź. Jeśli natrafimy na liść, weryfikujemy, czy punkty z tego liścia rzeczywiście znajdują się w przeszukiwanym obszarze. Te, które należą do obszaru, dodajemy do zbioru wyników.

Złożoność obliczeniowa przeszukiwania Quadtree wynosi $O(dl)$, gdzie: d to głębokość drzewa, l to liczba liści reprezentujących prostokąty częściowo zawierające się w przeszukiwanym obszarze.

W praktyce przeszukiwanie Quadtree jest znacznie mniej kosztowne niż jego konstrukcja. Struktura ta może być bardzo efektywna w określonych sytuacjach i dla odpowiednich zbiorów punktów. Niemniej jednak w innych przypadkach może okazać się mniej wydajna niż przeszukiwanie metodą liniową, polegającą na sprawdzeniu każdego punktu osobno. Dlatego ważne jest rozważne jej stosowanie i dostosowanie do charakterystyki zbioru punktów.

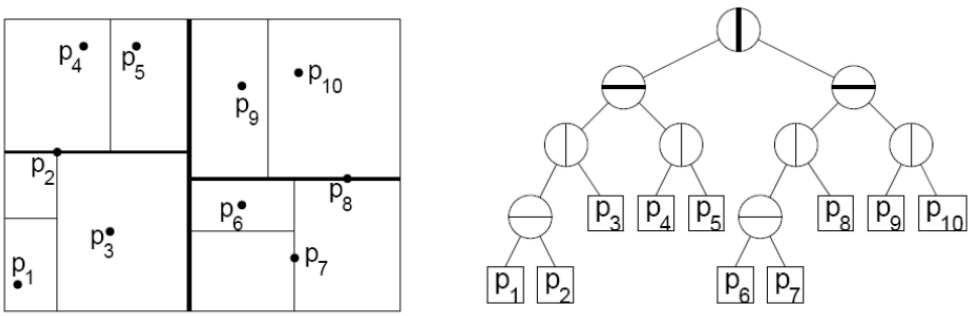
W dalszej części sprawozdania omówimy, jakie zbiory punktów najlepiej nadają się do wykorzystania Quadtree, na podstawie testów przeprowadzonych na zaimplementowanym algorytmie.

2. 2. Algorytm Kd-tree

KD-Tree (drzewo kd) to struktura danych w postaci drzewa binarnego, która służy do efektywnego podziału przestrzeni k-wymiarowej. Proces budowy drzewa rozpoczyna się od wyboru punktu podziału, którym jest mediana punktów w bieżącej przestrzeni, wyznaczana za pomocą algorytmu QuickSelect. Przed wyborem mediany ustalany jest wymiar podziału. Można to zrobić na dwa sposoby: naprzemiennie, wybierając kolejne wymiary (np. x, y, z), lub wybierając ten wymiar, w którym różnica współrzędnych punktów jest największa. W naszej implementacji zastosowaliśmy pierwszą metodę.

Po ustaleniu wymiaru i punktu podziału przestrzeń dzielona jest na dwie części: punkty mniejsze od mediany trafiają do jednej gałęzi drzewa, a punkty większe lub równe medianie do drugiej. Węzły drzewa, które odpowiadają podzielonej przestrzeni, przechowują informacje o wymiarze, na podstawie którego dokonano podziału. Proces podziału trwa tak długo, aż w danej podprzestrzeni pozostanie dokładnie jeden punkt. Wówczas punkt ten staje się liściem drzewa, wychodzącym z węzła reprezentującego przestrzeń nadrzędną.

Opisany proces pozwala na efektywne organizowanie danych w przestrzeni, co czyni KD-Tree skutecznym narzędziem do wyszukiwania punktów, ich sąsiadów lub określonych obszarów.



Rysunek 2: Schemat drzewa Kd-tree

Złożoność obliczeniowa tworzenia KDTree wynosi $O(kn \log n)$, dla pewnych implementacji $O(n \log n)$, gdzie n to ilość punktów, a k - ilość wymiarów (w naszej implementacji $k = 2$), wynika ona bezpośrednio z głębokości drzewa, która wynosi $\log n$, jeżeli zakładamy, że drzewo jest zbalansowane, a takie właśnie tworzymy. Algorytm poszukiwania punktów należących do konkretnego obszaru jest niemal identyczny jak ten przy wykorzystaniu quadtree. Wykorzystujemy natomiast do tego strukturę KDTree co wpływa na poprawę złożoności dla niektórych przypadków. Dla drzewa zrównoważonego złożoność czasowa zliczania wynosi $O(\sqrt{n})$, złożoność czasowa wyszukiwania $O(\sqrt{n} + k)$, gdzie k - jest ilością punktów wynikowych, a złożoność pamięciowa

$O(n)$. Widać, więc że tak jak i w quadtree, przeszukiwanie drzewa jest znacząco mniej kosztowne niż jego stworzenie. Porównywaniem QuadTree oraz KDTree zajmiemy się w dalszej części sprawozdania na podstawie konkretnych przykładów, dzięki którym będziemy mogli dostrzec wady i zalety, a także podobieństwa i różnice tych dwóch podejść rozwiązujących problem zaklasyfikowania punktów z danego zbioru do konkretnego obszaru.

3. Opracowanie szczegółów implementacji

3. 1. Moduł geo_structures

Jest to plik zawierający struktury pomocnicze to właściwej części algorytmu

- ```
  class Point:
      def __init__(self, x, y):
          self.x = x
          self.y = y
  ```

`Point` reprezentuje punkt w dwuwymiarowej przestrzeni. Posiada następujące metody:

1. `__str__(self)` - reprezentacja przy wywołaniu `print` w postaci `Point(x,y)`
2. `get(self,dim)` - przydatna dla Kd-tree, zwraca współrzędną x lub y punktu, zależnie od potrzeby.
3. `__eq__` oraz `__hash__` - niezbędne do odpowiedniego operowania obiektami klasy Point w wielu strukturach danych

- ```
  class RectangleArea:
      def __init__(self, min_x: float, min_y: float, max_x: float, max_y: float):
          self.min_x = min_x
          self.max_x = max_x
          self.min_y = min_y
          self.max_y = max_y
  ```

Struktura przechowuje wymiary prostokątnego obszaru (lewy dolny i prawy górny róg). Posiada następujące metody:

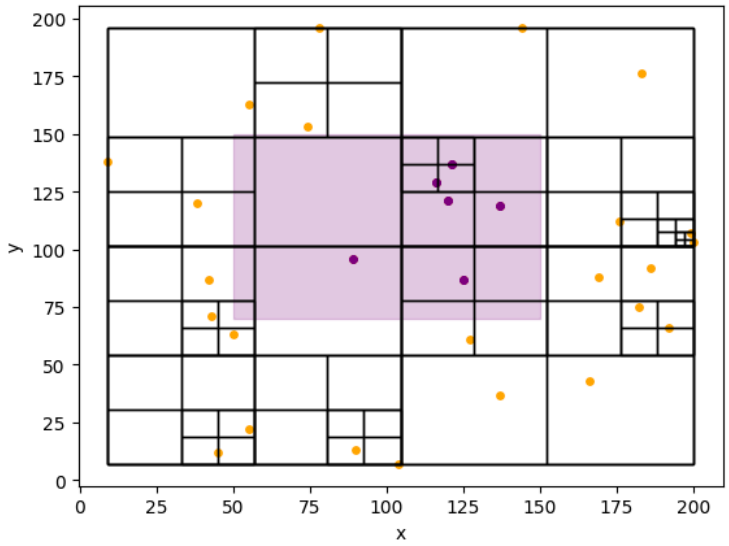
1. `__and__(self, other: RectangleArea) -> RectangleArea | None` - metoda przymuje drugi obiekt klasy `RectangularArea` i wyznacza ich część wspólną. Jeśli nie mają części wspólnej - zwraca `None`.
2. `__contains__(self, item: RectangleArea | Point) -> bool:` - przymuje `Point` albo `Rectangular` i zwraca `Bool` informujące o tym czy punkt lub prostokąt należy (w całości) do innego prostokąta.
3. `get_extrema(self) -> tuple[float, float, float, float]` - Zwraca granice prostokąta jako krotkę (min_x, min_y, max_x, max_y).
4. `__eq__(self, other: RectangleArea) -> bool` - Metoda sprawdza, czy dwa prostokąty są identyczne (mają te same granice).

3. 2. Moduł quadtree

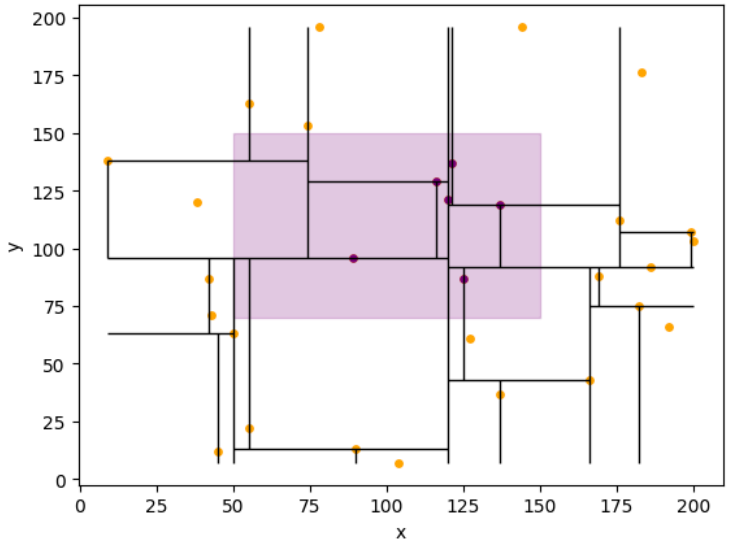
Plik zawiera implementację algorytmu przeszukiwania ortogonalnego Quadtree.

- ```
  class Quadtree:
      def __init__(self, points, capacity: int = 1):
          self.capacity = capacity
          self.max_rectangle = RectangleArea(
              min(points, key=lambda p: p.x).x,  # Minimalna wartość x
              min(points, key=lambda p: p.y).y,  # Minimalna wartość y
              max(points, key=lambda p: p.x).x,  # Maksymalna wartość x
              max(points, key=lambda p: p.y).y,  # Maksymalna wartość y
  ```
  ```

```
        )
        self.root = self.build_tree(self.max_rectangle, points)
```

`self.capacity` - oznacza maksymalną ilość punktów, które mogą znaleźć się wdanym prostokącie.
`self.max_rectangle` - to najmniejszy prostokąt obejmujący wszystkie zadane punkty.

`Quadtree` posiada następujące metody:

1. `def build_tree(self, rectangle: RectangleArea, points) -> QuadtreeNode:` - metoda buduje drzewo. Obiket klasy `Quadtree` jest korzeniem, a elementami niżej w drzewie są `QuadTreeNode`, dlatego ta metoda zwraca obiekt właśnie tej klasy. Funkcja ta wywołuje się rekurencyjnie, dopóki ilość punktów w każdym prostokącie będzie mniejsza lub równa `capacity`. Opis działania: Będąc w danym prostokącie, sprawdzamy czy ilość punktów które trzeba tam umieścić jest $\leq$ `capacity`. Jeśli tak jest, to zwracamy `QuadtreeNode` i kończymy budowanie drzewa w tym miejscu. W przeciwnym przypadku dzielimy prostokąt na 4 takie same części i każdej przydzielamy punkty które się w niej znajdują. Wywołujemy `build_tree` dla każdej takiej części
2. `def find_recursion(self, node: QuadtreeNode, rectangle: RectangleArea) -> list[Point]:` - metoda rekurencyjnie przeszukuje drzewo `Quadtree`, zaczynając od korzenia.

- ```
  class QuadtreeNode:
      def __init__(self, rectangle: RectangleArea) -> None:
          self.rectangle = rectangle  # Całkowity obszar tego węzła
          self.points = []  # Punkty w tym węźle
          self.upper_left = None  # Lewy górny kwadrant
          self.upper_right = None  # Prawy górny kwadrant
          self.lower_left = None  # Lewy dolny kwadrant
          self.lower_right = None  # Prawy dolny kwadrant
          self.is_leaf = True  # Czy jest liściem (czy ma dzieci)
  ```

Każdy obiekt `QuadtreeNode` przechowuje informacje na temat punktów, które zawiera, obszaru który obejmuje i swoich dzieci.

3. 3. Moduł kd_tree

Plik zawiera implementację przeszukiwania `Kd_tree`

- ```
  class KdTree:
      def __init__(self, points: list[Point]):

          self.points = points
          self.max_rectangle = RectangleArea(
              min(points, key=lambda p: p.x).x,  # Minimalna wartość x
              min(points, key=lambda p: p.y).y,  # Minimalna wartość y
              max(points, key=lambda p: p.x).x,  # Maksymalna wartość x
              max(points, key=lambda p: p.y).y,  # Maksymalna wartość y
          )
          self.root = self.build_tree(self.points, 0, self.max_rectangle)
  ```

Klasa zawiera metody:

1. `def build_tree(self, points: list[Point], depth: int, rectangle: RectangleArea) -> KdTreeNode:` - rekurencyjna metoda budująca drzewo, rozpoczyna swoja działanie od korzenia i kończy działanie gdy `KdTreeNode` zawiera 1 element
2. `def find(self, rectangle: RectangleArea) -> list[Point]:` - metoda znajduje punkty w danym obszarze przeszukując drzewo rekurencyjnie.

- ```
  class KdTreeNode:
      def __init__(self, axis: int | None, rectangle: RectangleArea) -> None:
          self.axis = axis  # określa którą oś rozpatrujemy tj. dla K=2 czy według
  osi X czy Y/
          self.rectangle = rectangle  # obszar który jesr reprezentowany przez
  poddtrzewo tego wierzchołka
          self.left_node = None  # lewe dziecko
          self.right_node = None  # prawe dziecko
          self.leafs_list = []  # list liści w poddrzewie
          self.leaf_point = None  # jeśli node jest liście to znajduje się tu punkt
  ```

Obiekt `KdtreeNode` zawiera informacje na temat zajmowanego przez niego obszaru, głębokości oraz dzieci.

3. 4. Moduł get_median

Jest to zestaw funkcji pomocniczych wykorzystywanych przy wyszukiwaniu `KdTree`. Funkcje zawarte w `get_media` mają na celu znalezienie mediany.

- ```
  def get_median(
      points: list[Point], l: int, r: int, k: int, depth: int, K: int
  ) -> Point:
  ```

funkcja ta rekurencyjnie wyszukuje mediany opierając swoje działanie na szukaniu pivota co jest znaną i wydajną metodą szukania mediany. Zwracanym wynikiem jest punkt, który pełni rolę mediany w danym wymiarze, wyznaczonym przez głębokość w drzewie KD.

- ```
  def rand_partition(points: list[Point], l: int, r: int, depth: int, K: int) -> int:
  ```

Metoda rand_partition służy do losowego podziału listy punktów w celu ułatwienia wyszukiwania mediany. Jej działanie polega na wybraniu losowego elementu (pivotu) z listy, zamianie go z ostatnim elementem, a następnie reorganizacji listy tak, aby wszystkie elementy mniejsze od pivotu znalazły się po jego lewej stronie, a większe po prawej. Na końcu metoda zwraca indeks, pod którym znajduje się pivot po dokonanym podziale.

4. Wizualizacja

Wizualizacje powyższych algorytmów zostały napisane w oparciu o Visualizer stworzony przez Koło Naukowe BIT. Dokumentacja odnośnie tego wizualizera dostępna jest pod adresem: https://github.com/aghbit/Algorytmy-Geometryczne.
Zdecydowaliśmy się stworzyć osobne klasy służące do wizualizacji, ponieważ gdybyśmy te wizualizacje wykonywali na naszych oryginalnych klasach, czas wykonania programu nie byłby tylko czasem działania naszego algorytmu, ale również wykonania wizualizacji. Dlatego w trosce o odpowiednie pomiary czasu stworzyliśmy osobne klasy do wizualizacji, które są kopiami oryginalnych klas z dodaniem Visualizera.
  ```



Rysunek 3: Przykładowa wizualizacja QuadTree



Rysunek 4: Przykładowa wizualizacja Kd_Tree

5. Instrukcja obsługi programu

5. 1. Kd_Tree

Przykładowy prosty program wykorzystujący nasz algorytm KdTree:

```
points=[Point(1,5),Point(3,4), Point(1,2)] #lista punktów
kd=KdTree(points) #zbudowanie drzewa i dodanie punktów
area=RectangleArea(2,2,5,5) #określenie obszaru w którym szukane będą punkty
found_points=kd.find(area) #znalezienie punktów
print(found_points) #wyświetlenie wyniku
```

W powyższym przykładzie wynik to: `[Point(3, 4)]`

5. 2. Quadtree

Przykładowy prosty program wykorzystujący nasz algorytm Quadtree:

```
points=[Point(1,5),Point(3,4), Point(1,2)] #lista punktów
qt=Quadtree(points,1) #zbudowanie drzewa i dodanie punktów, określenie ile punktów
```

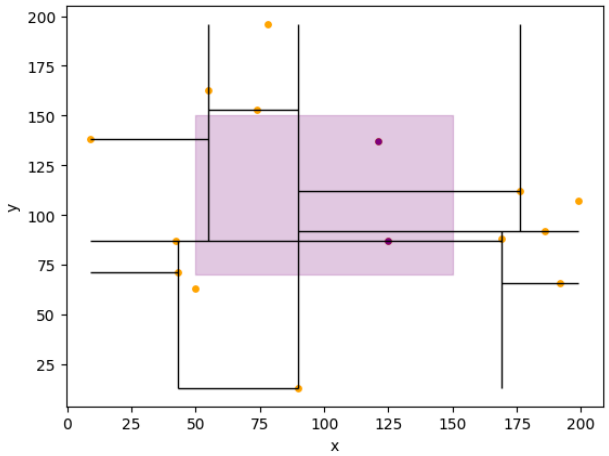
```
może być w jednym prostokącie
area=RectangleArea(2,2,5,5) #określenie obszaru w którym szukane będą punkty
found_points=qt.find(area) #znalezienie punktów
print(found_points) #wyświetlenie wyniku
```

W powyższym przykładzie wynik to: `[Point(3, 4)]`

5. 3. Wizualizacja Kd_tree

Przykładowy prosty program umożliwiający wizualizację.

```
points=[
    Point(192, 66), Point(55, 163), Point(50, 63), Point(186, 92), Point(74, 153),
    Point(125, 87), Point(78, 196), Point(43, 71), Point(90, 13), Point(199, 107),
    Point(176, 112), Point(9, 138), Point(169, 88), Point(42, 87), Point(121, 137)
]
kdtree = Kd_tree_vis(points)
found = kdtree.find(RectangleArea(50, 70, 150, 150))
viskd = kdtree.get_vis()
viskd.show()
```

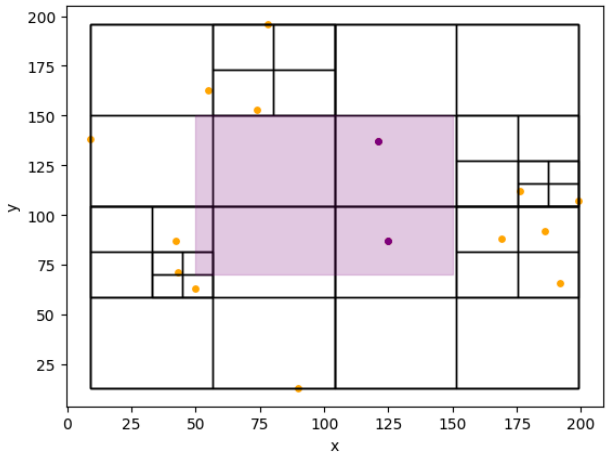


Rysunek 5: Rezultat powyższego programu

5. 4. Wizualizacja Quadtree

Przykładowy prosty program umożliwiający wizualizację.

```
points=[
    Point(192, 66), Point(55, 163), Point(50, 63), Point(186, 92), Point(74, 153),
    Point(125, 87), Point(78, 196), Point(43, 71), Point(90, 13), Point(199, 107),
    Point(176, 112), Point(9, 138), Point(169, 88), Point(42, 87), Point(121, 137)
]
quad_tree=Quadtree_vis(points,1)
found = quad_tree.find(RectangleArea(50, 70, 150, 150))
vis=quad_tree.get_vis()
vis.show()
```

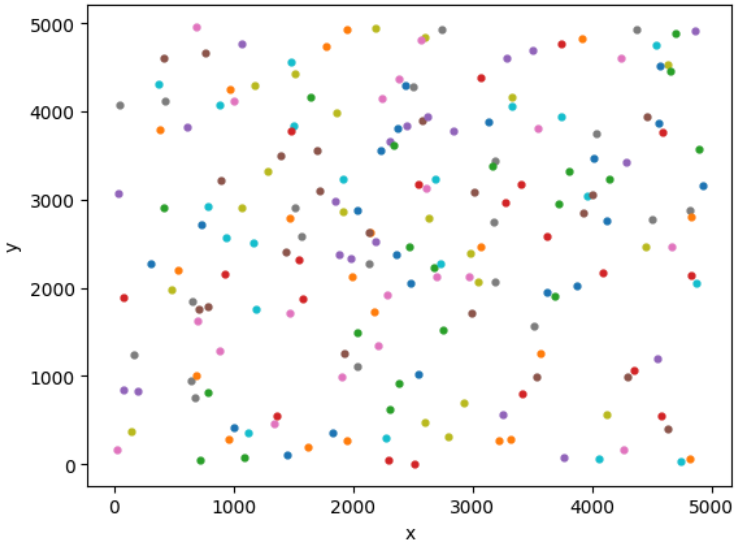


Rysunek 6: Rezultat powyższego programu

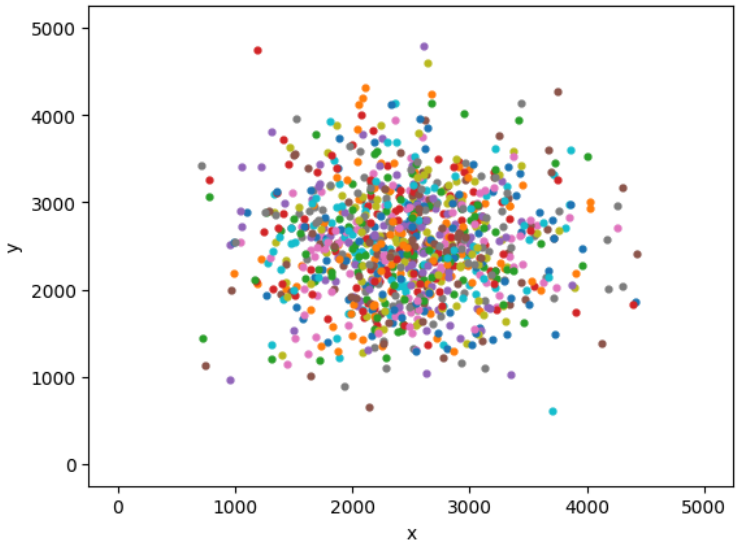
6. Porównanie struktur - testy czasowe

Obydwie struktury przetestowane zostały na trzech rodzajach zbiorów punktów:

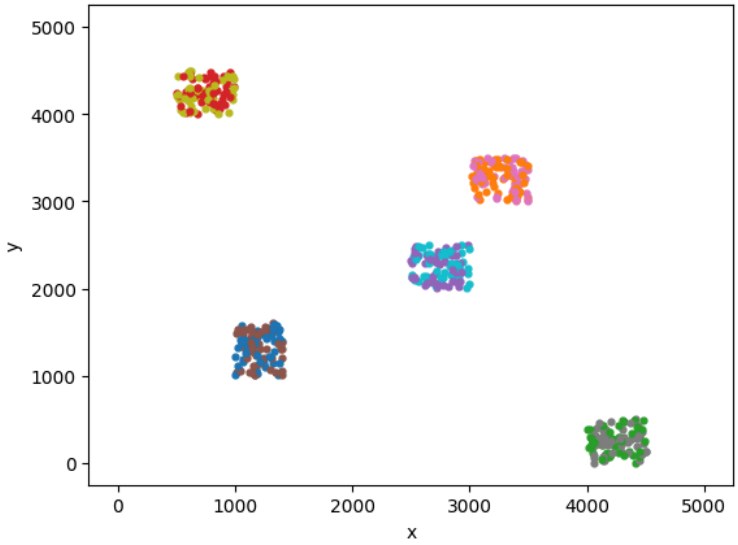
- $A(n)$ - zbiór n jednostajnie rozłożonych punktów będących w zakresie $[0, 5000]^2$ wygenerowany przy pomocy funkcji `numpy.random.uniform`
- $B(n)$ - zbiór n punktów o rozkładzie normalnym będących w zakresie $[0, 5000]^2$ wygenerowany przy pomocy funkcji `numpy.random.normal`
- $C(n)$ - zawiera 5 klastrów, każdy o rozkładzie jednostajnym z n/5 punktów będących w zakresie $[0, 5000]^2$ wygenerowany przy pomocy funkcji `numpy.random.uniform`



Rysunek 7: Wizualizacja zbioru A



Rysunek 8: Wizualizacja zbioru B



Rysunek 9: Wizualizacja zbioru C

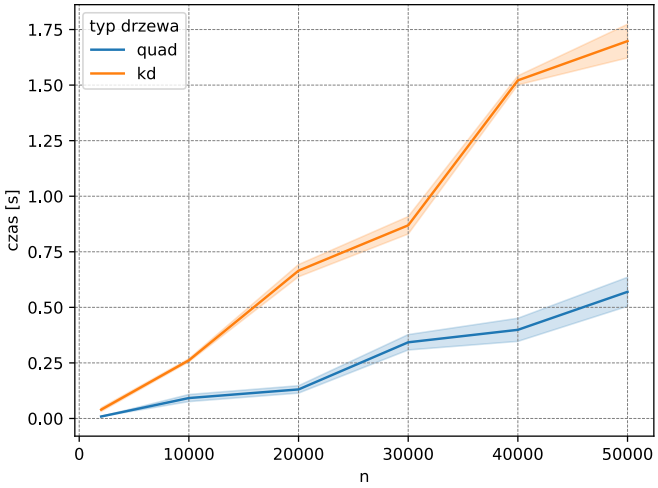
Dla każdego zbioru wybrano 6 wartości $n \in [2000, 10000, 20000, 30000, 40000, 50000]$ na których to przeprowadzono test, każdy test został powtórzony 10 razy aby rezultaty były bardziej miarodajne. Testy przeprowadzono na obydwu strukturach.

Sprawdzono czasy:
- konstrukcji drzewa
- zapytania o mały obszar
- zapytania o duży obszar

6. 1. Rezultaty dla zbioru A

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0088	0.0397
10000	0.0919	0.2618
20000	0.1307	0.6650
30000	0.3428	0.8691
40000	0.3988	1.5213
50000	0.5698	1.6980

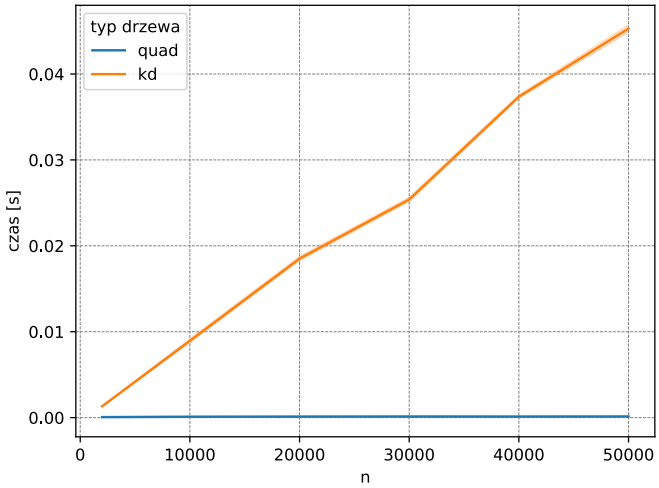
Tabela 1: Czasy konstrukcji drzew dla zbioru A



Rysunek 10: Wizualizacja czasu konstrukcji drzew dla zbioru A

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0001	0.0013
10000	0.0001	0.0089
20000	0.0001	0.0185
30000	0.0001	0.0254
40000	0.0001	0.0374
50000	0.0001	0.0453

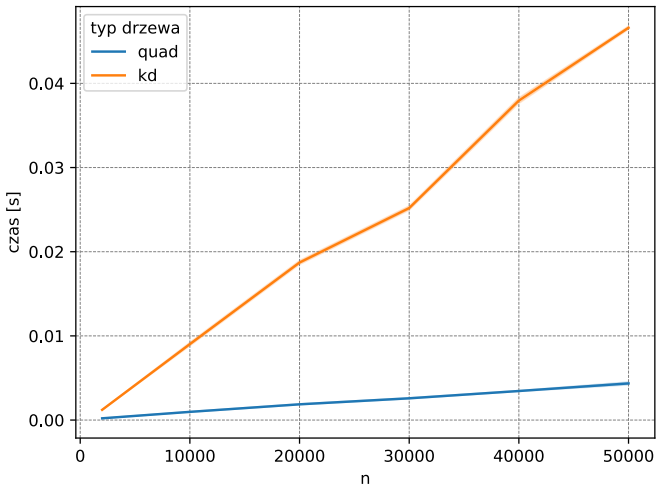
Tabela 2: Czasy odpowiedzi na zapytania o mały obszar dla zbioru A



Rysunek 11: Wizualizacja czasu odpowiedzi na zapytania o mały obszar dla zbioru A

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0002	0.0012
10000	0.0010	0.0090
20000	0.0019	0.0187
30000	0.0026	0.0252
40000	0.0035	0.0380
50000	0.0044	0.0466

Tabela 3: Czasy odpowiedzi na zapytania o duży obszar dla zbioru A

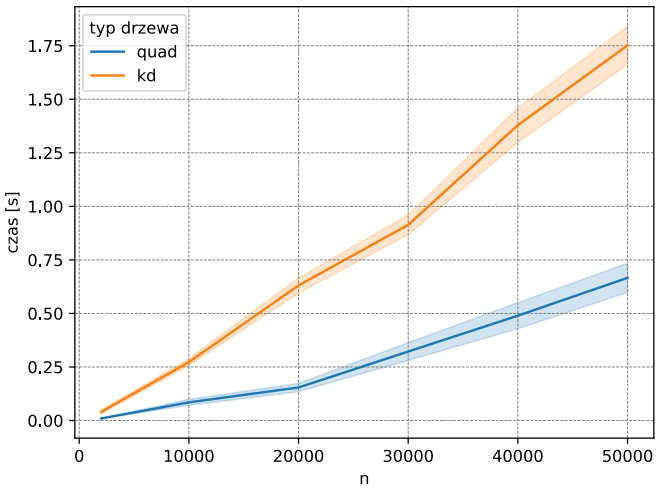


Rysunek 12: Wizualizacja czasu odpowiedzi na zapytania o duży obszar dla zbioru A

6. 2. Rezultaty dla zbioru B

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0094	0.0400
10000	0.0849	0.2727
20000	0.1542	0.6307
30000	0.3220	0.9136
40000	0.4896	1.3791
50000	0.6662	1.7517

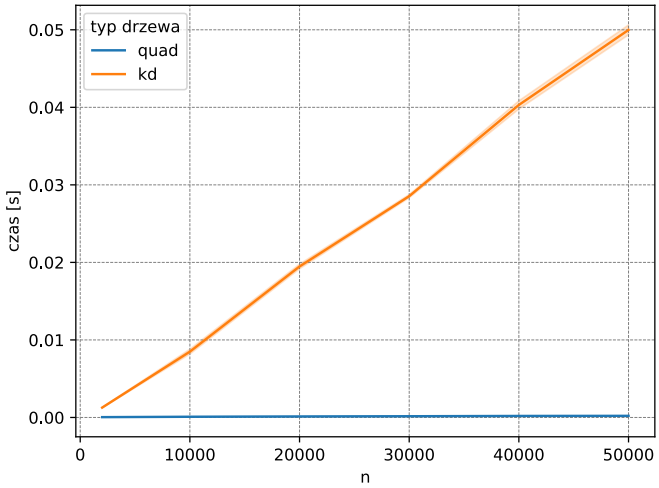
Tabela 4: Czasy konstrukcji drzew dla zbioru B



Rysunek 13: Wizualizacja czasu konstrukcji drzew dla zbioru B

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0000	0.0013
10000	0.0001	0.0085
20000	0.0001	0.0195
30000	0.0002	0.0285
40000	0.0002	0.0403
50000	0.0002	0.0500

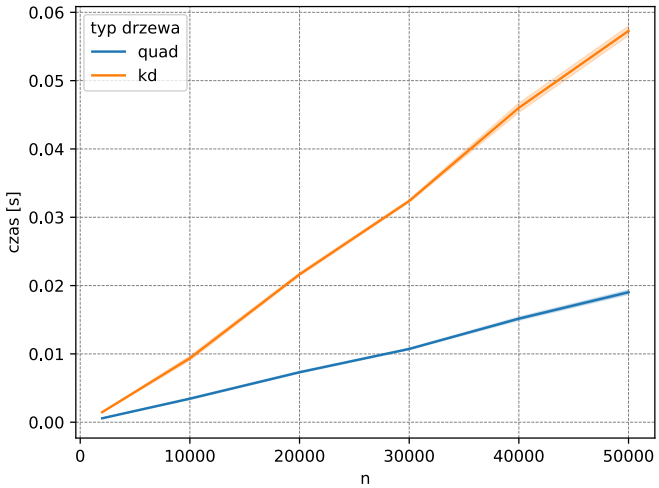
Tabela 5: Czas odpowiedzi na zapytania o mały obszar dla zbioru B



Rysunek 14: Wizualizacja czasu odpowiedzi na zapytania o mały obszar dla zbioru B

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0006	0.0015
10000	0.0034	0.0094
20000	0.0073	0.0216
30000	0.0107	0.0324
40000	0.0152	0.0460
50000	0.0190	0.0573

Tabela 6: Czas odpowiedzi na zapytania o duży obszar dla zbioru B

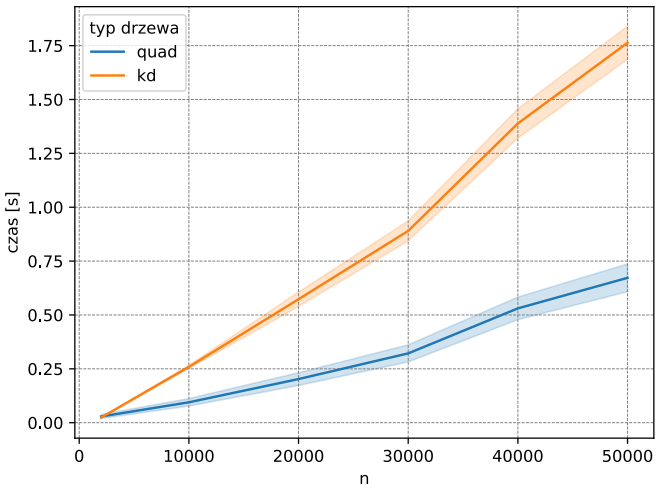


Rysunek 15: Wizualizacja czasu odpowiedzi na zapytania o duży obszar dla zbioru B

6. 3. Rezultaty dla zbioru C

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0288	0.0238
10000	0.0946	0.2590
20000	0.2025	0.5731
30000	0.3217	0.8903
40000	0.5305	1.3889
50000	0.6723	1.7632

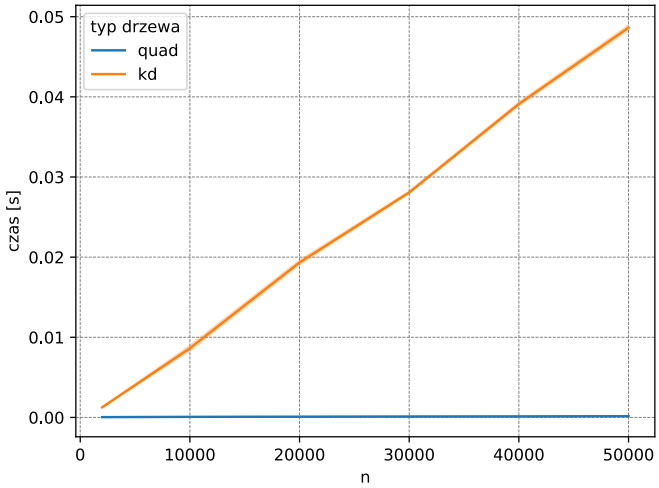
Tabela 7: Czas konstrukcji drzew dla zbioru C



Rysunek 16: Wizualizacja czasu konstrukcji drzew dla zbioru C

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0000	0.0013
10000	0.0001	0.0086
20000	0.0001	0.0193
30000	0.0001	0.0281
40000	0.0001	0.0391
50000	0.0002	0.0486

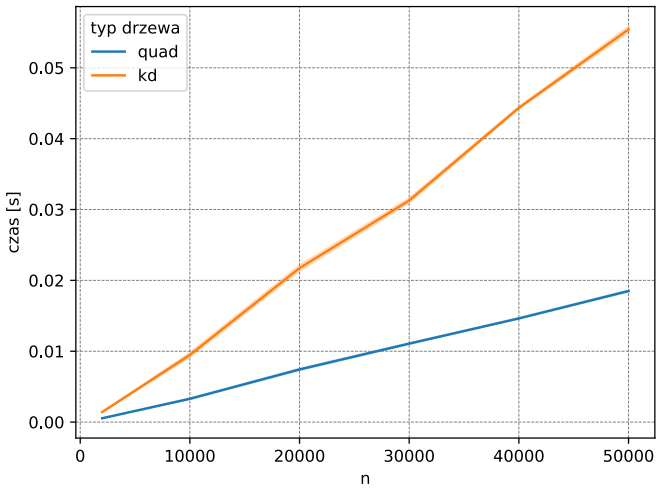
Tabela 8: Czas odpowiedzi na zapytania o mały obszar dla zbioru C



Rysunek 17: Wizualizacja czasu odpowiedzi na zapytania o mały obszar dla zbioru C

Liczba punktów	Czas wykonania struktury Quadtree [s]	Czas wykonania struktury KD-Tree [s]
2000	0.0005	0.0014
10000	0.0033	0.0095
20000	0.0074	0.0217
30000	0.0111	0.0313
40000	0.0146	0.0444
50000	0.0185	0.0554

Tabela 9: Czas odpowiedzi na zapytania o duży obszar dla zbioru C



Rysunek 18: Wizualizacja czasu odpowiedzi na zapytania o duży obszar dla zbioru C

7. Wnioski

Jak wynika z analizy powyższych wykresów, dla zbioru punktów o rozkładzie jednostajnym, normalnym oraz klastrów, czas konstrukcji drzewa jest dłuższy dla algorytmu Kd-Tree. Dzieje się tak dlatego, że algorytm ten wymaga znalezienia mediany przy każdym podziale, co zajmuje czas. Natomiast algorytm Quadtree po prostu zawsze dzieli na 4 równe części. Na postawie danych przedstawionych na wykresach numer 11,12,14,15,17,18 widzimy że odpowiedź na pytanie o tak mały jak i duży obszar jest realizowane istotnie szybciej przez algorytm Quadtree. Jest to spodziewane ponieważ algorytm ten zawsze dzieli obszar na 4 części co sprawia że powstałe drzewo podziału jest płytsze, a co za tym idzie czas odpowiedzi na zapytanie krótszy. Nie oznacza to że drzewo Kd-Tree jest gorszą strukturą, gdybyśmy mieli za zadanie znaleźć najbliższego sąsiada zadanego punktu to struktura ta poradziłaby sobie najpewniej lepiej. Warto również pamiętać że Quadtree nadaje się jedynie do użytku dla przestrzeni $2d$, natomiast Kd-Tree łatwo można używać dla dowolniej przestrzeni K. Znajduje więc ono zastosowanie choćby przy relacyjnych bazach danych gdzie każda kolumna to osobny wymiar w drzewie.

8. Podsumowanie

- Quadtree jest prostsze w budowie i bardziej intuicyjne w przestrzeniach 2D, zwłaszcza przy równomiernym rozkładzie punktów.
- KD-Tree jest bardziej elastyczne i skalowalne, szczególnie dla danych wielowymiarowych lub o nierównomiernym rozkładzie.
- W ogólności Quadtree jest szybsze jeśli chodzi o znajdowanie punktów w określonym obszarze, natomiast KD-tree ma duży potencjał jeśli chodzi o wykonywanie innych operacji - np. wyszukiwanie najbliższego sąsiada danego punktu.
- Wybór odpowiedniego algorytmu zależy od charakterystyki danych (rozkład, liczba wymiarów) i rodzaju operacji, jakie mają być wykonywane.

9. Bibliografia

1. **Algorytmy geometryczne – wykład, dr inż. Barbara Głut.**
2. **Wprowadzenie do algorytmów, Thomas H. Cormen et al**
3. **K-d Trees - Computerphile**: https://www.youtube.com/watch?v=BK5x7IUTIyU
4. **Quadtree - Wikipedia**: https://en.wikipedia.org/wiki/Quadtree