Przeszukiwanie geometryczne KDTree i QuadTree

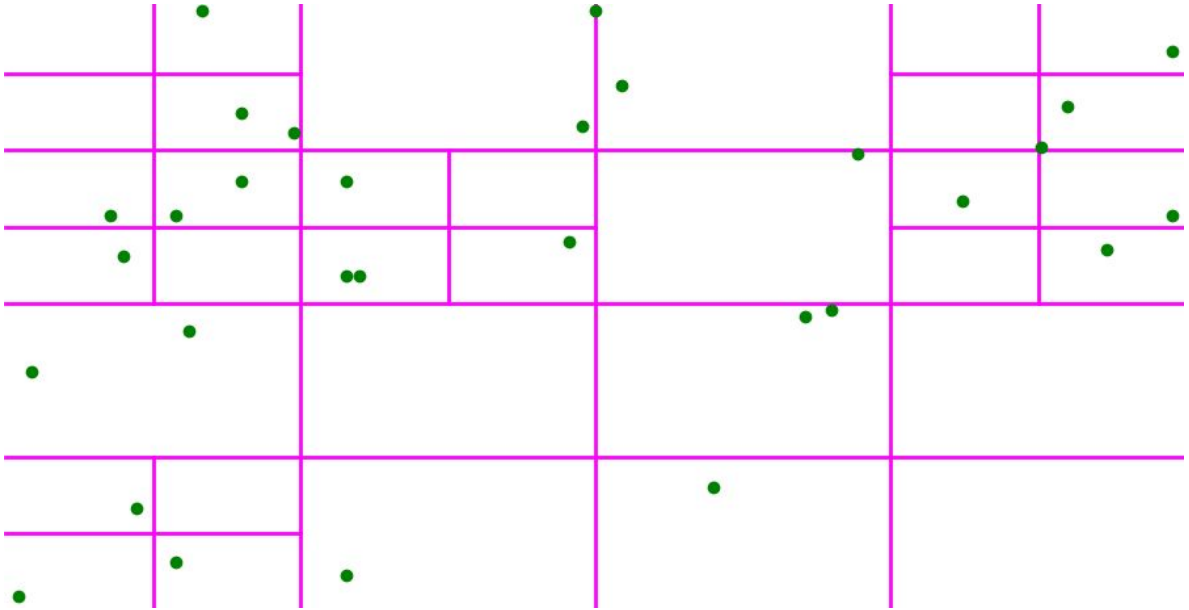
Opis projektu

Dane – zbiór punktów P na płaszczyźnie. Zapytanie: dla zadanych x_1, x_2, y_1, y_2 znaleźć punkty q ze zbioru P takie, że $x_1 \leq q_x \leq x_2$, $y_1 \leq q_y \leq y_2$.

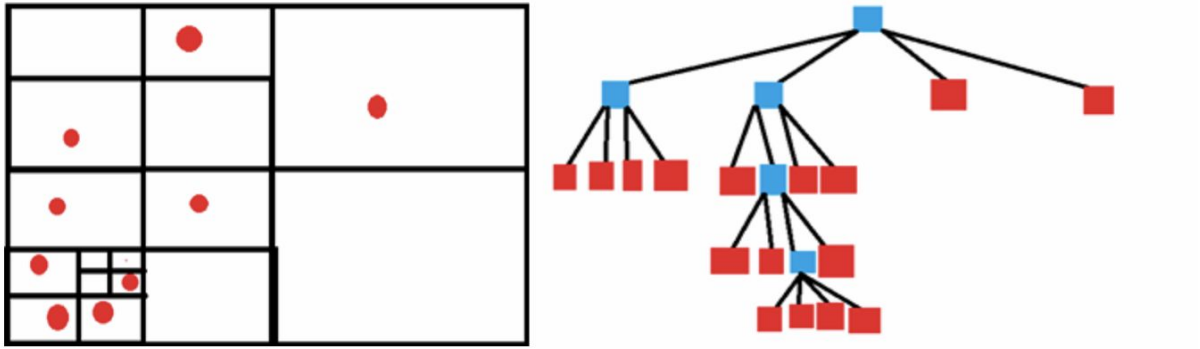
Celem projektu jest zaimplementowanie odpowiednich struktur danych – quadtree oraz kd-drzew, które pozwalają szybko odpowiadać na takie zapytania. Program ma służyć jako narzędzie dydaktyczne do objaśnienia tworzenia struktury i realizacji zapytań.

Należy zrobić analizę porównawczą obu podejść

QuadTree



Quadtree jest to strukturą danych będąca drzewem, które służy do podziału przestrzeni dwuwymiarowej na mniejsze części, dzieląc ją na równe ćwiartki, a następnie dzieląc je na kolejne ćwiartki itd.

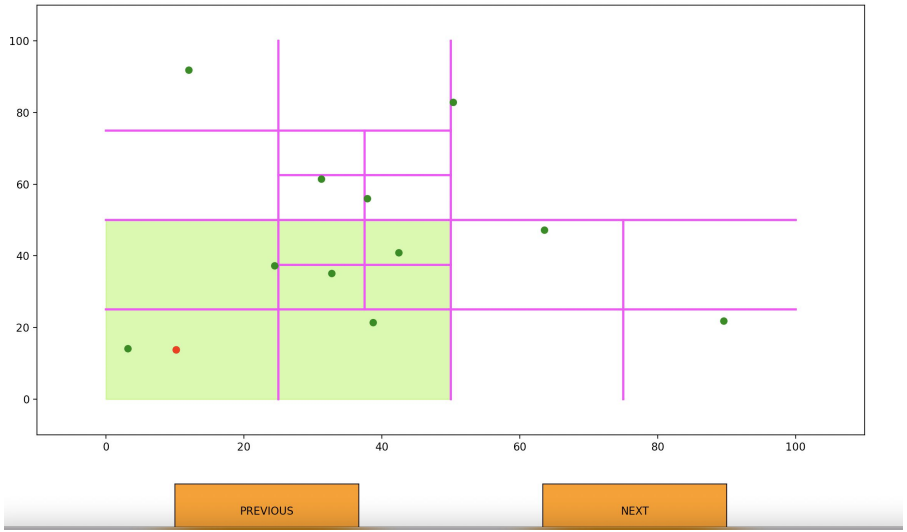


Jest to struktura szczególnie przydatna do:

- przetwarzania obrazu
- generowania siatki
- skutecznego wykrywania kolizji

Quadtree - struktura

```
class QuadTree:
        def __init__(self, …):
                self._points = … # list
                self._capacity = … # int
                self._node = …# QuadTreeNode

class QuadTreeNode:
        def __init__(self, …):
                self.boundary = … #Rect
                self.divided= …# bool
                self.points = … # list
                self.capacity = … # int
                self.right_upper = …# Rect
                self.right_down = … # Rect
                self.right_upper = …# Rect
                self.right_down = … # Rect
```



Quadtree - budowanie

kod:

```
def insert(self,points):
    if not self.boundary.contains_point(point):
        self._node.insert(point):
            return False
    self.points.append(point)
    if.self.capacity < len(self.points):
        if not self.divided:
            self._subdivide()
            self.divided = True
            for p in self.points:
                self.right_upper.insert.(point)
                    or self.right_down.insert.(point)
                    or self.left_upper.insert.(point)
                    or self.left_down.insert.(point)
        else:
            self.right_upper.insert.(point)
                or self.right_down.insert.(point)
                or self.left_upper.insert.(point)
                or self.left_down.insert.(point)
    return True
```

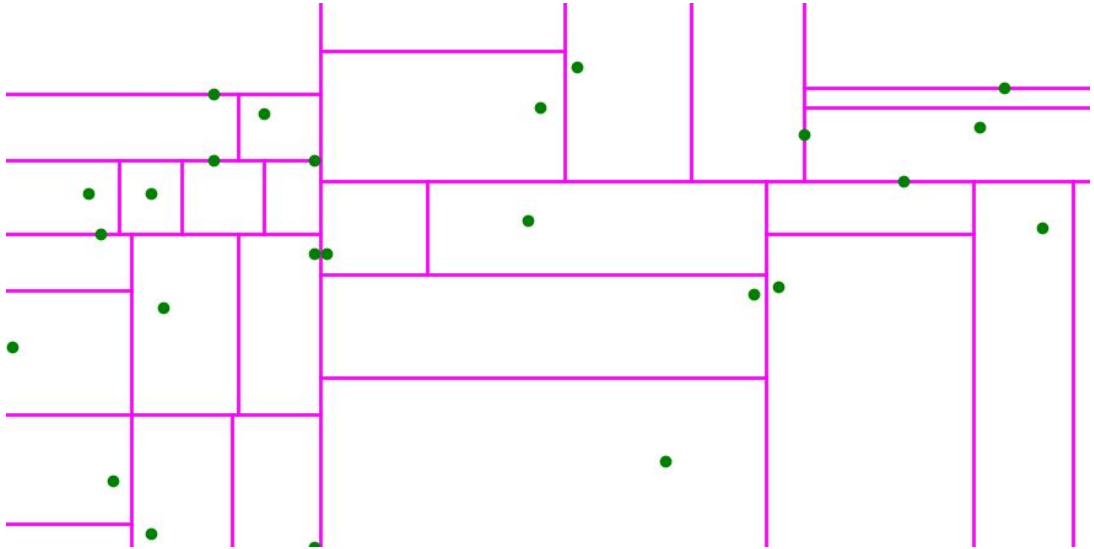
- Konstruowanie Quadtree polega na umieszczeniu każdego z punktów ze zbioru w drzewie, po kolei wywołując metodę insert dla danych części drzewa
- jeżeli punkt nie zawiera się w tej części drzewa zwracamy False
- w przeciwnym wypadku sprawdzamy czy mamy wystarczająco miejsca w danej ćwiartce by umieścić tam punkt
- jeżeli tak to go tam umieszczamy i zwracamy True
- jeżeli nie, sprawdzamy czy prostokąt jest podzielony
- jeżeli nie, to dzielimy go na ćwiartki
- dla każdej z ćwiartek danej części wywołujemy metodę insert, w której wykonujemy wyżej opisane czynności

Quadtree - przeszukiwanie

```
def points_in_rec(self,points,rect):
        if not rect.overlaps(self.boundary):
                return [ ]
        if.rect.contains_rect(self.boundary):
                return self.points
        result = []
        if self.devided:
                result.extend(self.left_down.points_in_rec(rect)
                result.extend(self.left_upper.points_in_rec(rect)
                result.extend(self.right_upper.points_in_rec(rect)
                result.extend(self.right_down.points_in_rec(rect)
        else:
                found_points = [ point for point in self.points
                        if rect.contains_point(point)]
                result.extend(found_points)
        return result
```

- Przeszukiwanie QuadTree polega na wchodzeniu w głąb obszaru otaczającego obszar przeszukiwany
- Polega ona na przeszukaniu w następujący sposób każdej z ćwiartek obecnie przeszukiwanego prostokąta
- Jeżeli dana ćwiartka nie przecina się z obszarem przeszukiwanym to zwracamy [ ] oznaczającą brak punktów w tej ćwiartce, należących do obszaru przeszukiwanego
- Jeżeli ćwiartka zawiera się całkowicie w tym obszarze to zwracamy wszystkie punkty należące do niej
- Jeżeli obszary te się przecinają to sprawdzamy czy jest podzielony
- Jeżeli tak to wchodzimy w głąb powtarzając powyższe czynności dla tej ćwiartki
- Jeżeli nie, sprawdzamy wśród wszystkich punktów zawierających się w tej ćwiartce, które należą do przeszukiwanego obszaru

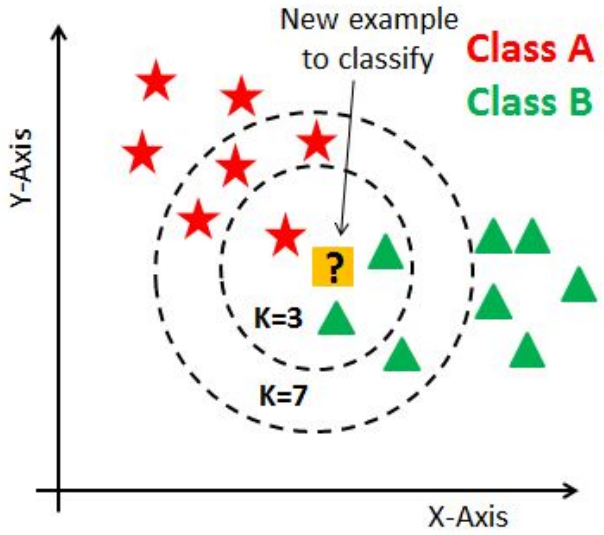
KDTree



Pełna implementacja i dokumentacja jest tu

KD-drzewo - drzewo binarne używane do podziału k-wymiarowej przestrzeni i przechowywania punktów do niej należących. Zapewnia podstawowe operacje:

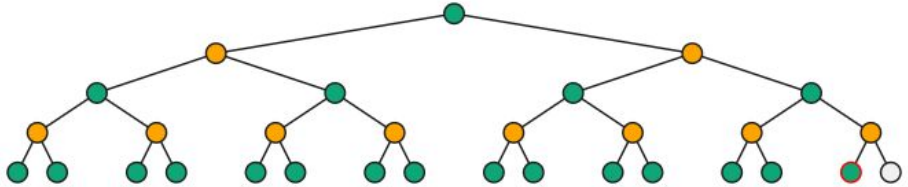
- sprawdzenie czy istnieje punkt w danym drzewie -- O(logn)
- znalezienie wszystkich punktów pomiędzy punktami q_{min} i q_{max} -- $O(n^{1/2} + k)$
- znalezienie k najbliższych punktów do podanego -- O(klogn)



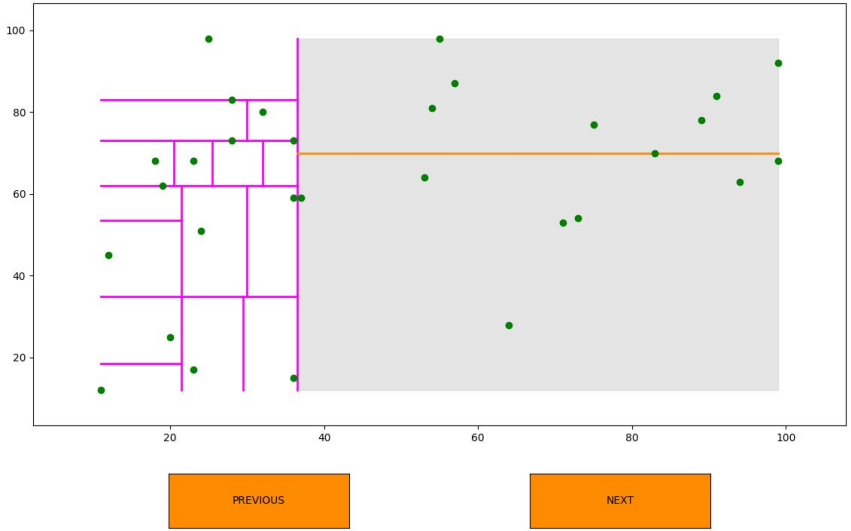
Najbardziej popularnym zastosowaniem drzewa jest algorytm kNN (k nearest neighbours) - podstawowy algorytm klasyfikacji uczenia maszynowego.

Także bardzo przydatnym jest wydajne filtrowanie danych. Jeśli mamy zbiór obiektów z pewnymi cechami i chcemy wybrać takie, wartości cech których mieszczą się w pewnych przedziałach, to możemy potraktować ich jak wielowymiarowe punkty. Tym zastosowaniem i zajmiemy się.

KDTree - struktura



```
class KDTreeNode:
        def __init__(self, …):
                self._left = … # KDTreeNode
                self._right = … # KDTreeNode
                self._points = … # list
                self._condition = … # function
                self._rect = … # Rect
                self._leaf = … # bool
```



Powyższa struktura reprezentuje węzeł kd-drzewa. Węzeł trzyma swoje dzieci, jeżeli nie jest liściem. Także wszystkie punkty w swoim poddrzewie. Dodatkowo funkcję, która zwraca True lub False, która odpowiednio wskazuje podział. Węzeł trzyma także obszar, który jest przez niego zajęty.

* przedstawiony kod jest tylko pokazowym

KDTree - budowanie

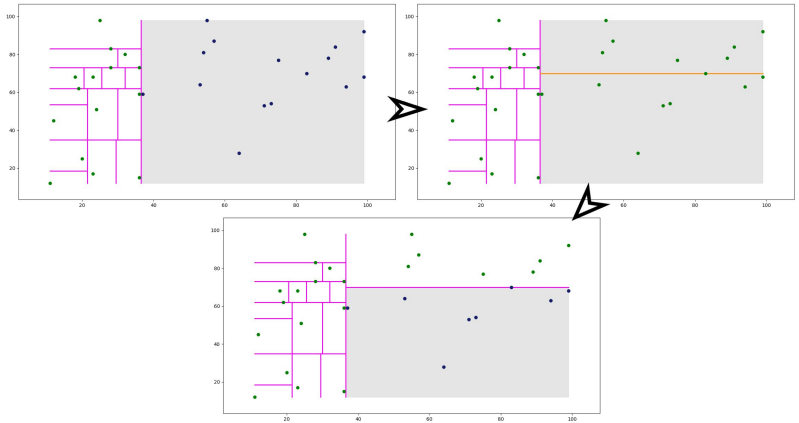
```
def _build_tree(self, points, rect, ...):
        node = KDTreeNode(...)

        if len(points) == 1:
                node.leaf = True
                return node

        axis = self._get_condition_axis(...)
        threshold = self._get_condition_threshold(...)

        node.left = self._build_tree(
                [p for p in points if node.condition(p)], ...)
        node.right = self._build_tree(
                [p for p in points if not node.condition(p)], …)

        return node
```

Budowanie kd-drzewa jest bardzo proste.

Dostając zbiór punktów szukamy najlepszego podziału na dwie części. Robimy to szukając wymiaru z największym rozstępem i dzielimy po wartości, która jest medianą dla tych punktów w tym wymiarze.

Wszystkie punkty mniejsze od wartości podziału idą do lewego poddrzewa, a reszta do prawego.

Jeżeli jest tylko jeden punkt to tworzymy liść i wracamy.

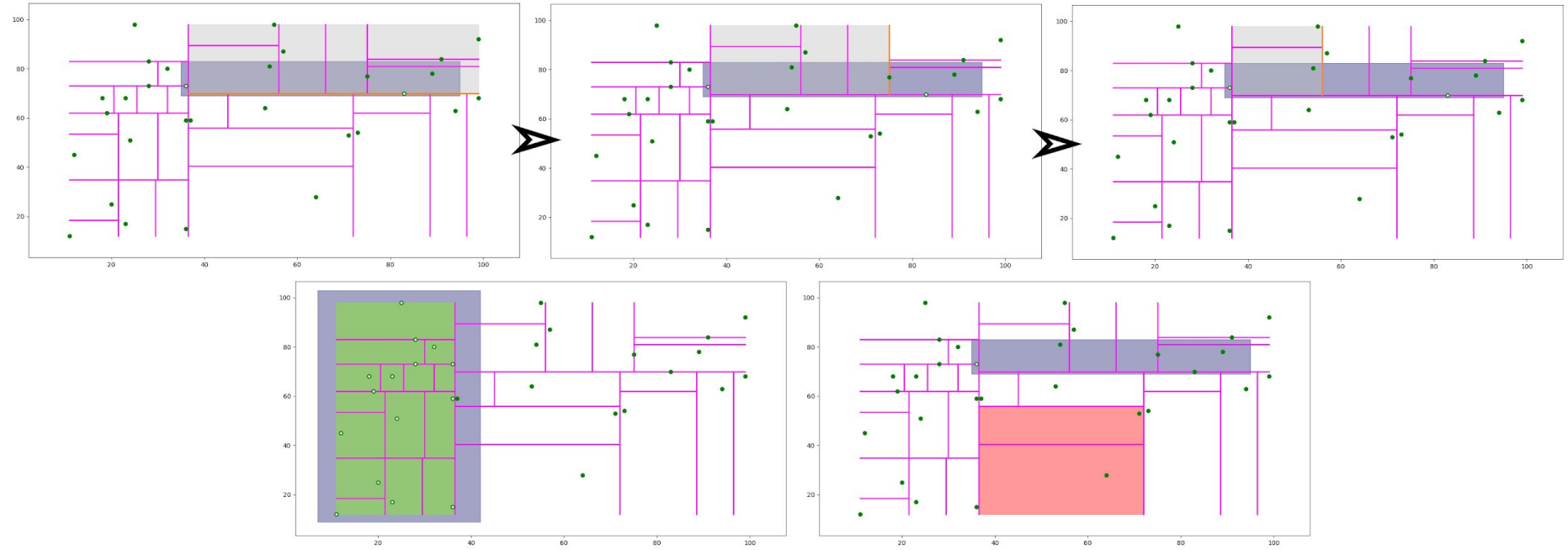


KDTree - przeszukiwanie

```
def find_points_in(self, rect):
        if not self._rect.overlaps(rect):
                return []
        if rect.contains_rect(self._rect):
                return self._points
        if self.is_leaf():
                if rect.contains_point(self._point):
                        return [self._point]
                else:
                        return []

        res = []
        res.extend(self.left.find_points_in(rect))
        res.extend(self.right.find_points_in(rect))
        return res
```

Przeszukiwanie działa następująco:

- metoda find_points_in korzenia zostaje wywołana z obszarem w którym szukamy
- każdy węzeł zachowuje się tak:
  - jeżeli obszar węzła nie ma wspólnej części z szukanym to zwracamy pustą listę
  - jeżeli obszar węzła całkiem zawiera się w szukanym to zwracamy listę wszystkich punktów poddrzewa tego węzła
  - jeżeli węzeł jest liściem to sprawdzamy czy punkt w liściu należy do szukanego obszaru i odpowiednio zwracamy go lub nic
  - jeżeli nic z powyższego nie zachodzi, to wywołujemy tą metodę na dzieciach węzła i zapisujemy to co zwracają i zwracamy sumę zwróconych zbiorów
- cała lista punktów wraca do samego drzewa, którą już zwracamy użytkownikowi



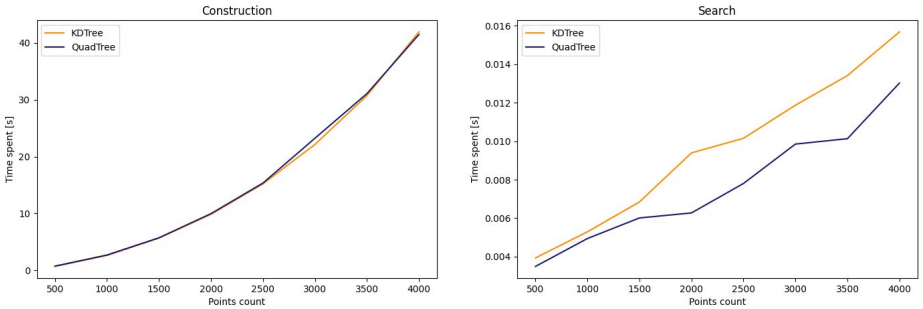
Wizualizacja różnych sytuacji podczas przeszukiwania

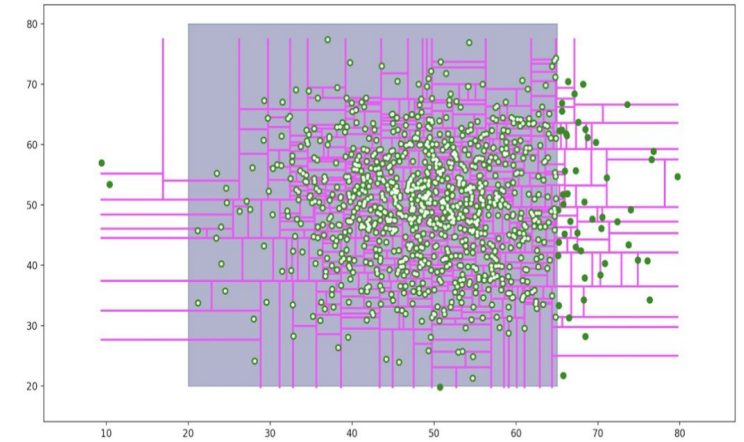
Porównanie QuadTree i KDTree

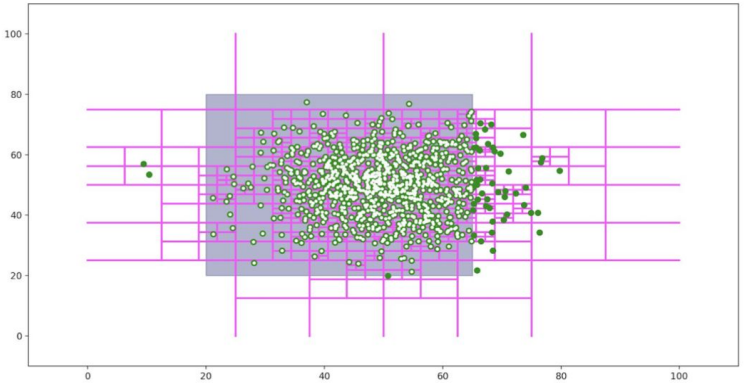
Dużo bardziej obszerne porównanie znajduje się tu

Rozkład normalny

Jest to rozkład statystycznie najczęściej występujący w naturze, jeżeli chcemy, więc przyjrzeć się naszemu problemowi dla najbardziej powszechnych przypadków, powinniśmy wziąć pod lupę właśnie tak wygenerowany zbiór punktów. Jak możemy zauważyć na poniższym wykresie, czas przeszukiwania jest o ok. 15% korzystniejszy na rzecz Quadtree, warto więc w takiej sytuacji wykorzystać tę strukturę.





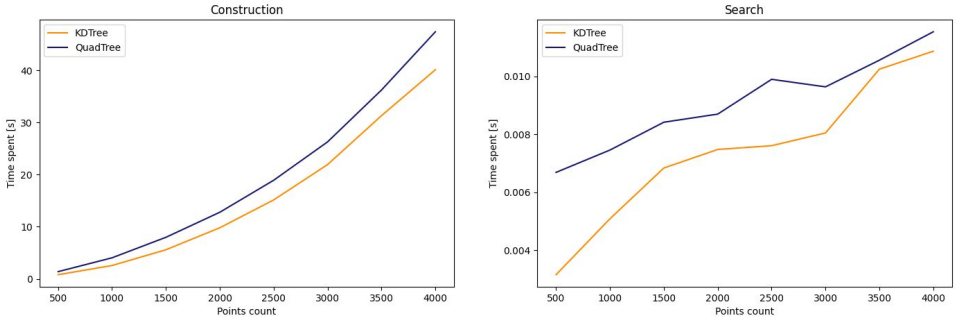


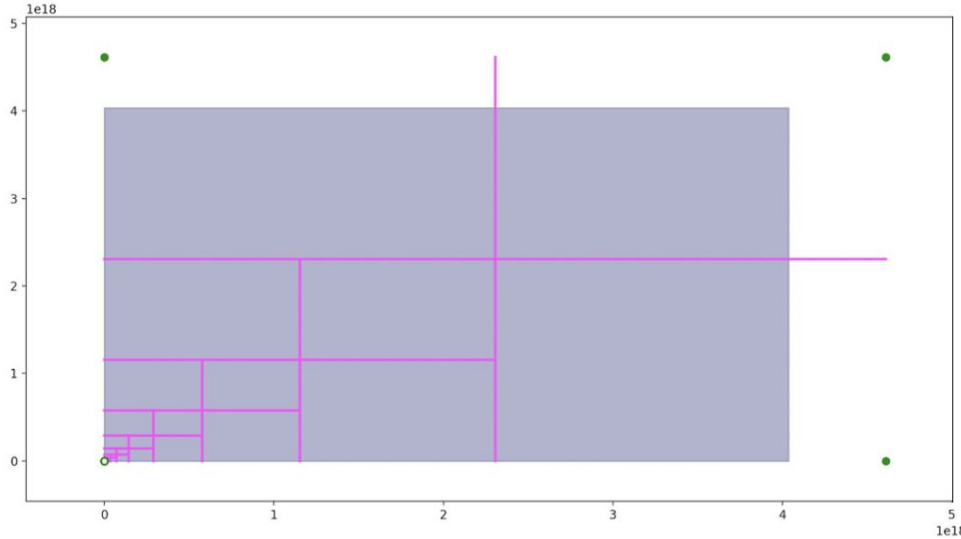
Wartości odstające

KDTree zawsze ma głębokość *logn*, w tym czasie dla QuadTree nie da się jej jednoznacznie ustalić.

Ten zbiór punktów pokazuje jak to wpływa na czas wykonywania operacji. Mamy 3 punkty ze współrzędnymi o wartościach rzędu 2^{62}. Reszta punktów są skoncentrowane w małym prostokącie.

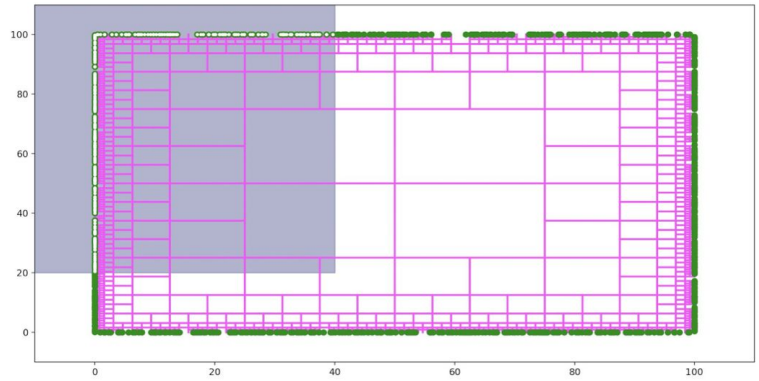
QuadTree musi zejść bardzo głęboko, żeby dojść do skoncentrowanych punktów, gdyż KDTree odcina te 3 punkty w dwóch podziałach.

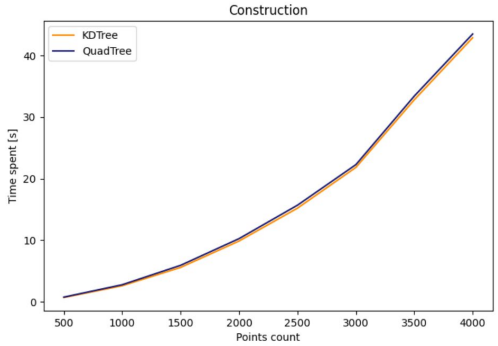


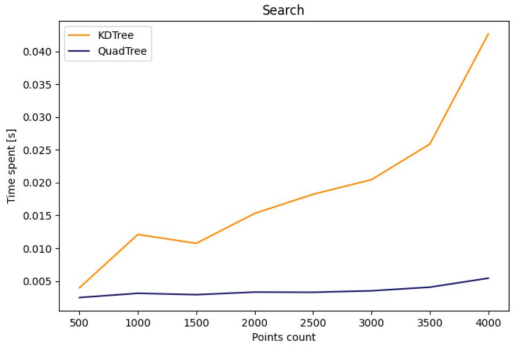


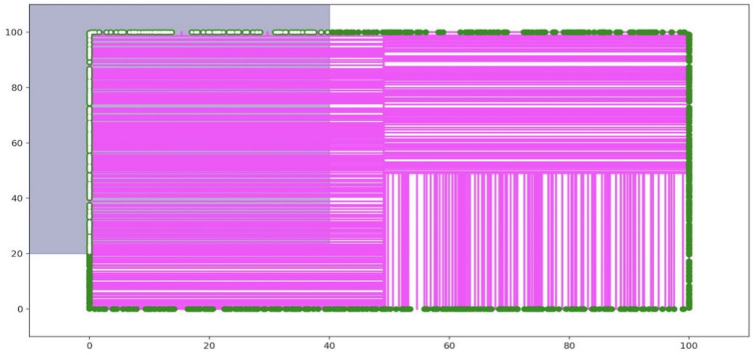
Zbiór punktów pokrywający krawędzie prostokąta

Zbiór ten składa się z dwóch par prostych równoległych do siebie (jedna para do osi OX, druga do osi OY). Zbiór ten ujawnia problem ryzyka nieskończonej rekurencji dla struktury KD-tree. Zbiór ten uwydatnia także wady KD-tree w tego typu strukturach, złożonych z niewielkiej ilości prostych. Poniższy wykres demonstruje ogromną przewagę szybkości Quadtree nad KD-tree w tym zbiorze.







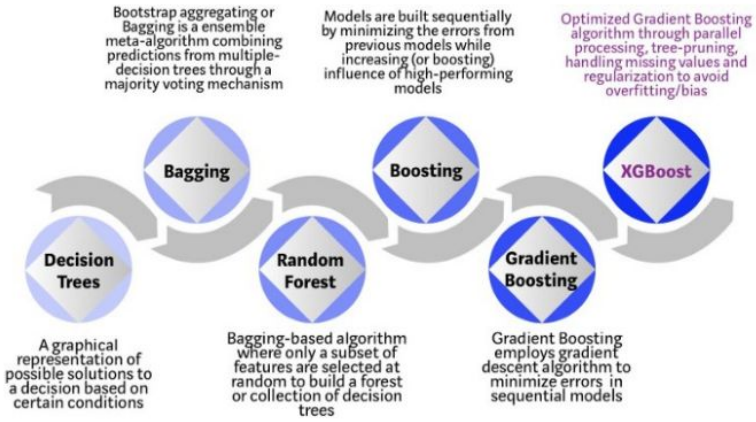


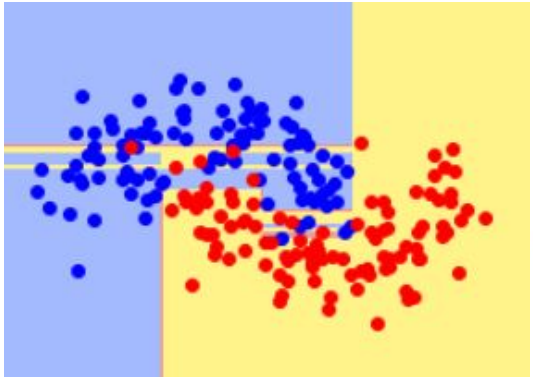
Ciekawostki

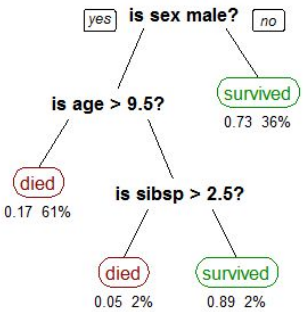
Jednym z najbardziej popularnych i używanych modeli w klasycznym uczeniu maszynowym jest XGBoost.

W prawym górnym rogu jest rysunek jego ewolucji.

Jego podstawą są drzewa decyzyjne. Są to drzewa, które dzielą k-wymiarową przestrzeń na obszary, do których przypisujemy klasy. Odpowiednio dostając nowy punkt i chcąc go zaklasyfikować, patrzymy do którego obszaru on należy.





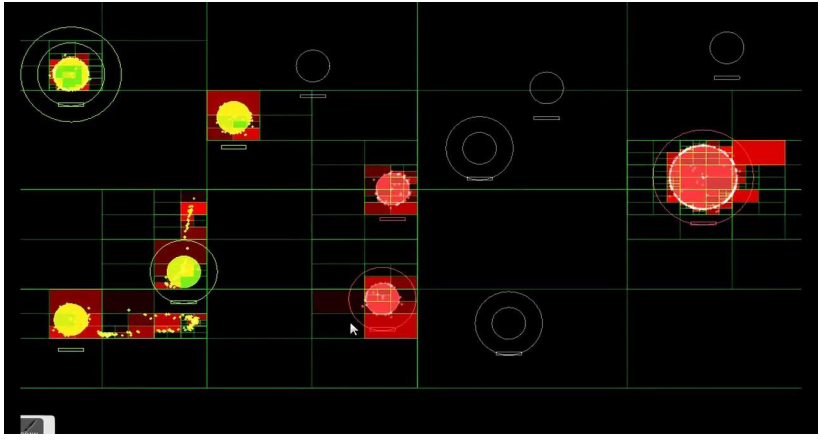


Czy coś nam przypomina taki podział? :)

Ciekawostki

Quadtree znajduje szerokie zastosowanie przede wszystkim w kompresji obrazu, im głębsze drzewo, tym jakość będzie lepsza, lecz w tym tkwi zaleta quadtree, nie w każdym miejscu musimy kopać tak samo głęboko.

Strukturę tą wykorzystuje się także w grach komputerowych do wykrywania kolizji. Dzięki budowie quadtree możemy uwzględnić daną kolizję nie zmieniając całości drzewa, a tylko jego fragment.



Wnioski

Jak widzimy, każde drzewo ma swoje zastosowania, które są spowodowane pewnymi własnościami:

QuadTree jest zazwyczaj bardzo szybki dla zbalansowanych zbiorów i nie ma pewnych wyjątkowych sytuacji przy których jest bardzo niewydajny.

KDTree ma największą zaletę w wielowymiarowości i dodatkowych operacjach (np. szukanie najbliższego sąsiada), które zapewniają popularność tej struktury.

Przedziałowe przeszukiwanie w 2D przestrzeni jest bardzo wydajne na obu strukturach, ale wymaga dodatkowego zastanowienia się nad postacią zbioru punktów jak pokazaliśmy ciut wcześniej, żeby uwzględnić własności drzew, i odpowiednio wybrać najbardziej odpowiednie.

Dziękujemy

Projekt został zrealizowany przez:
Konrad Krzemiński
Mykola Haltiuk