Wyszukiwanie geometryczne Quadtree oraz KD-tree

przeszukiwanie obszarów ortogonalnych

Hubert Kukla, Maksymilian Siemek

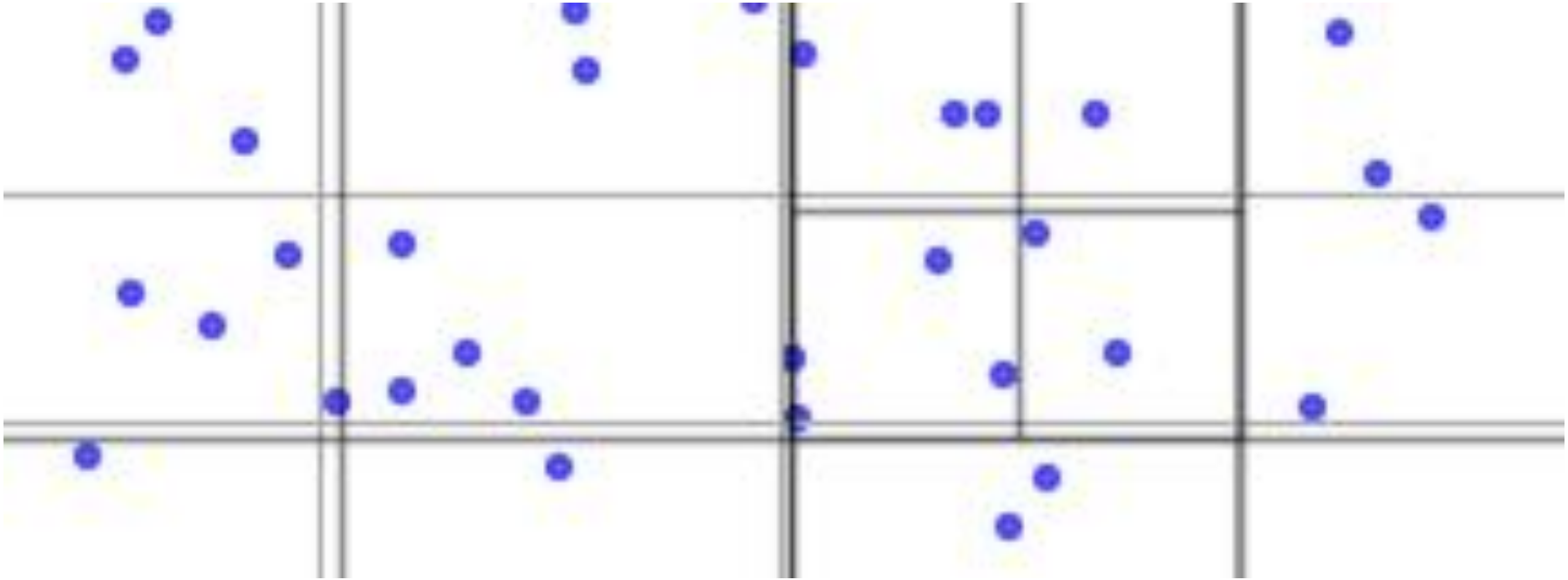
Opis projektu

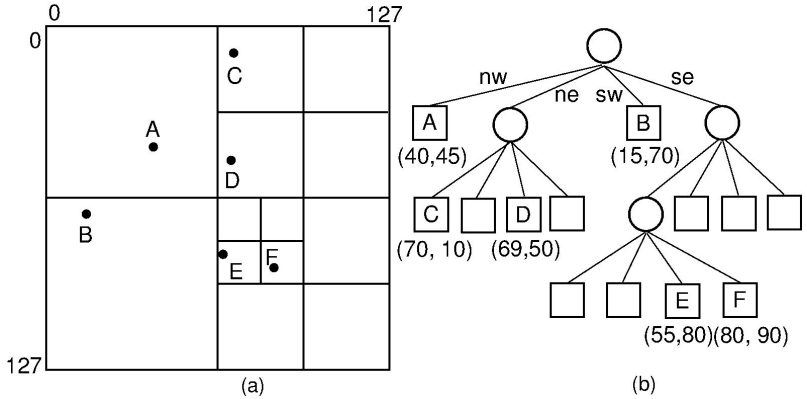
Dane: zbiór punktów P na płaszczyźnie. Zadanie polega na tym, aby dla podanych wartości x_1, x_2, y_1, y_2 znaleźć wszystkie punkty $q \in P$ spełniające warunki: $x_1 \leq q_x \leq x_2$ oraz $y_1 \leq q_y \leq y_2$.

Celem projektu jest zaimplementowanie odpowiednich struktur danych — quadtree oraz drzewa kd — które umożliwiają szybkie wykonywanie takich zapytań. Program ma pełnić funkcję dydaktyczną: pomagać w zrozumieniu procesu budowy struktur oraz sposobu obsługi zapytań.

Dodatkowo należy przygotować analizę porównawczą obu rozwiązań.

QuadTree





Quadtree to drzewiasta struktura danych służąca do hierarchicznego dzielenia przestrzeni dwuwymiarowej. Polega na rozcinaniu obszaru na cztery równe części (ćwiartki), a następnie – w razie potrzeby – dalszym, wielokrotnym podziale każdej z nich na kolejne ćwiartki.

Taka struktura sprawdza się szczególnie dobrze m.in. w:

- przetwarzaniu obrazów,
- tworzeniu i zagęszczaniu siatek (np. w grafice lub symulacjach),
- efektywnym wykrywaniu kolizji.

Struktura Quadtree

```
class QuadTree:  16 usages  ± Hubert Kukla *

    def __init__(self, boundary: Rect, capacity: int = 4, max_depth: int = 16, depth: int = 0):  ± Hubert Kukla *
        self.boundary = boundary
        self.capacity = capacity
        self.max_depth = max_depth
        self.depth = depth
        self.points: List[Point] = []
        self.divided: bool = False
        self.nw: Optional["QuadTree"] = None
        self.ne: Optional["QuadTree"] = None
        self.sw: Optional["QuadTree"] = None
        self.se: Optional["QuadTree"] = None
```

Budowa QuadTree

```python
def insert(self, p: Point) -> bool:  6 usages  Hubert Kukla *
    if not self.boundary.contains_point(p):
        return False
    if self.is_leaf():
        if len(self.points) < self.capacity or self.depth >= self.max_depth:
            self.points.append(p)
            return True
        self.subdivide()
        old_points = self.points
        self.points = []
        for op in old_points:
            child = self._child_for_point(op)
            child.insert(op)
        child = self._child_for_point(p)
        return child.insert(p)
    child = self._child_for_point(p)
    return child.insert(p)
```

- Budowanie QuadTree polega na wstawianiu kolejnych punktów do drzewa metodą `insert(p)` (zaczynamy od korzenia).
- `insert` najpierw sprawdza, czy punkt leży w obszarze węzła: `boundary.contains_point(p)`
  - jeśli nie → zwraca False.
- Następnie sprawdzamy, czy węzeł jest liściem: `is_leaf()`
  - `is_leaf()` zwraca True, gdy węzeł nie jest podzielony (`divided == False`).
  - Liście przechowują punkty w `self.points`.
- Jeśli węzeł jest liściem i ma miejsce (`len(points) < capacity`) lub osiągnięto `max_depth` → dodajemy punkt do `points` i zwracamy True.
- Jeśli liść jest pełny → wywołujemy `subdivide()`
  - `subdivide()` dzieli obszar na 4 ćwiartki i tworzy dzieci: `nw`, `ne`, `sw`, `se`, ustawia `divided = True`.
  - stare punkty z liścia są przenoszone do dzieci: wybieramy dziecko funkcją `_child_for_point(p)` i wywołujemy `insert` w tym dziecku.
- Gdy węzeł jest już podzielony, `insert` zawsze wybiera odpowiednie dziecko przez `_child_for_point(p)` i rekurencyjnie wstawia punkt niżej.

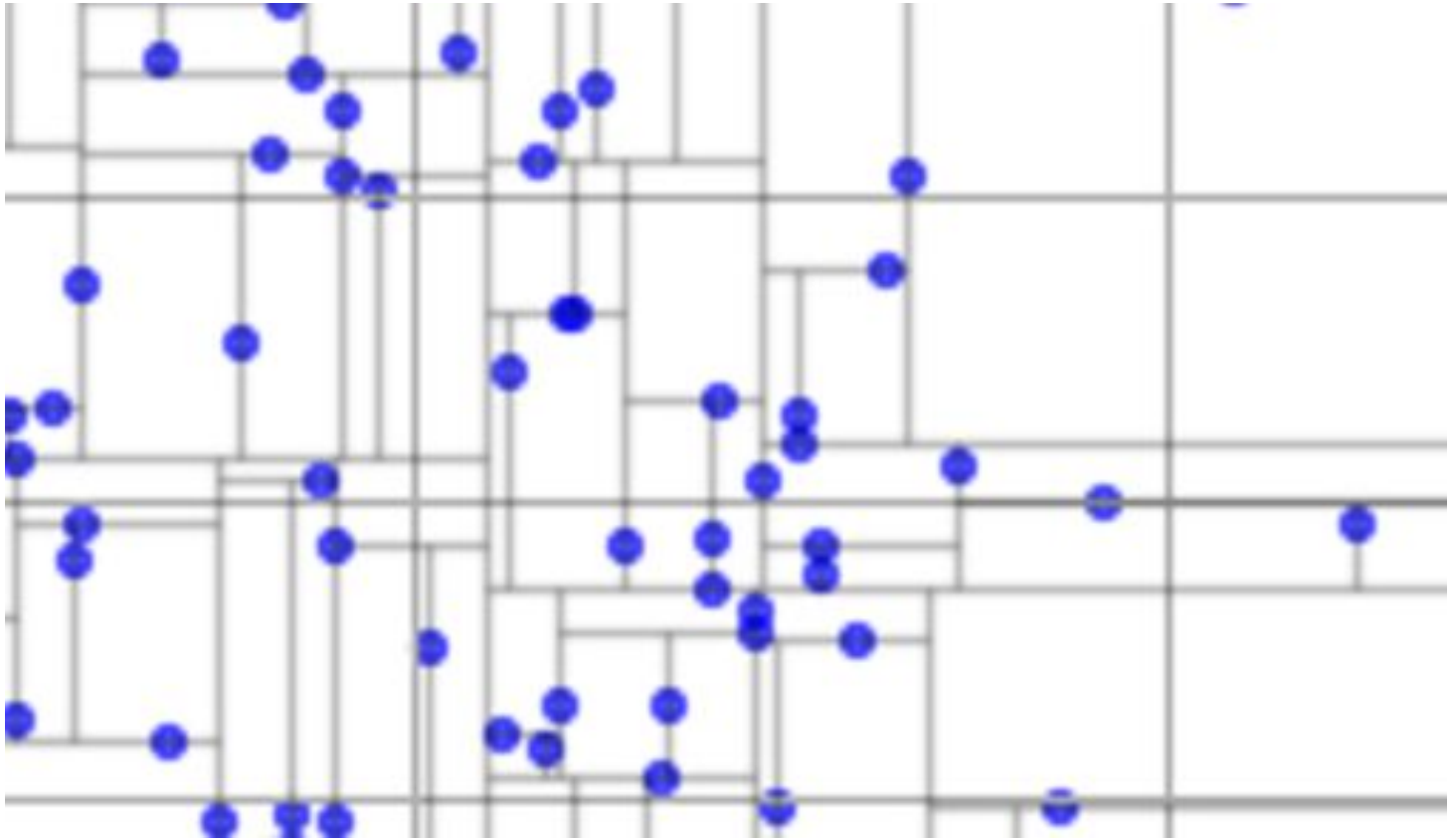
Przeszukiwanie Quadtree

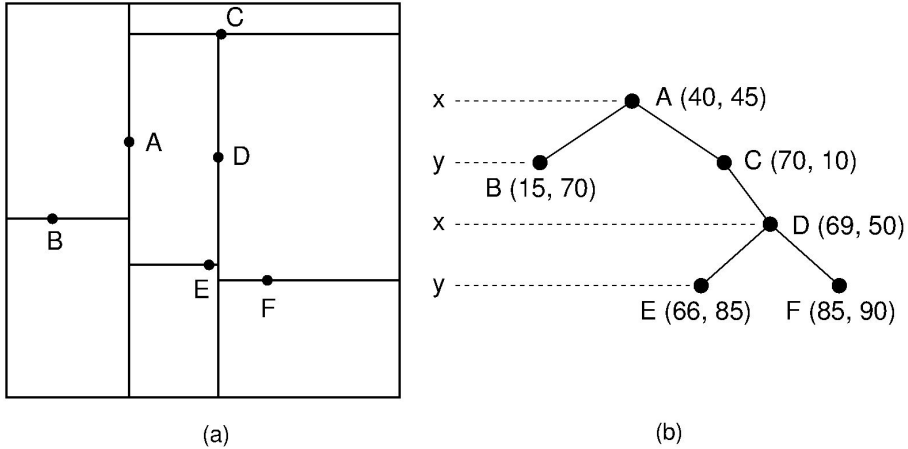
```python
def query(self, range_rect: Rect, found: Optional[List[Point]] = None) -> List[Point]:  5 usages  Hubert Kukla *
    if found is None:
        found = []
    if not self.boundary.intersects(range_rect):
        return found
    if self.is_leaf():
        for p in self.points:
            if range_rect.contains_point(p):
                found.append(p)
        return found
    self.nw.query(range_rect, found)
    self.ne.query(range_rect, found)
    self.sw.query(range_rect, found)
    self.se.query(range_rect, found)
    return found
```

`query(range_rect, found=None)`

- `range_rect` to prostokąt zapytania (obszar, w którym szukamy punktów).
- Na początku sprawdzamy, czy w ogóle warto schodzić niżej:
  - `boundary.intersects(range_rect)`
  - jeśli **nie przecina** → kończymy dla tego węzła (nic tu nie znajdziemy).
- Potem sprawdzamy `is_leaf()`:
  - `is_leaf()` = **True**, gdy węzeł nie jest podzielony (`divided == False`).
- Jeśli to **liść**:
  - sprawdzamy wszystkie punkty w `self.points`
  - dla każdego punktu robimy `range_rect.contains_point(p)`
  - jeśli tak → dodajemy do listy wyników `found`.
- Jeśli to **węzeł podzielony**:
  - wywołujemy rekurencyjnie `query` w dzieciach: `nw`, `ne`, `sw`, `se`
  - (ale tylko te poddrzewa, których obszar przecina `range_rect`, przejdą dalej przez `intersects`).
- `found` to lista, do której dopisujemy wyniki (jak bufor). Jeśli nie jest podana, funkcja tworzy ją sama na początku i zwraca na końcu.

KD-Tree





KD-drzewo (k-d tree) to binarna, drzewiasta struktura danych służąca do hierarchicznego dzielenia przestrzeni k-wymiarowej . W Twojej implementacji przestrzeń jest dzielona naprzemiennie względem osi X i Y: na danym poziomie wybierana jest oś podziału, punkty są sortowane po tej osi, a mediana zostaje zapisana w węźle. Punkty „mniejsze” trafiają do lewego poddrzewa, a „większe” do prawego, co zwykle daje drzewo zbliżone do zbalansowanego.

Taka struktura sprawdza się szczególnie dobrze m.in. w:

- wydajnym filtrowaniu punktów w prostokątnym zakresie
- wyszukiwaniu sąsiadów i zapytaniach przestrzennych (np. kNN — ogólnie dla kd-tree),
- porządkowaniu danych geometrycznych w aplikacjach grafiki, GIS i symulacji.

Budowa KD-Tree

```python
def _build(self, points: List[Point], depth: int) -> Optional[KDNode]:  3 usages  Maksymilian Siemek *
    if not points:
        return None

    axis = depth % 2

    if axis == 0:
        points.sort(key=lambda p: p.x)
    else:
        points.sort(key=lambda p: p.y)

    median_idx = len(points) // 2

    return KDNode(
        point=points[median_idx],
        left=self._build(points[:median_idx], depth + 1),
        right=self._build(points[median_idx + 1:], depth + 1)
    )
```

Budowanie KDTree

- Start: `_build(points, depth=0)`.
- Jeśli `points` puste → `None`.
- `axis = depth % 2` (0 = X, 1 = Y).
- Sortuj punkty po osi (`x` albo `y`).
- Weź medianę (`len(points)//2`) jako punkt węzła.
- Rekurencyjnie buduj:
    - `left` z punktów < mediana
    - `right` z punktów > mediana
    - `depth+1`.
    - `>= val` → idź w `right`.

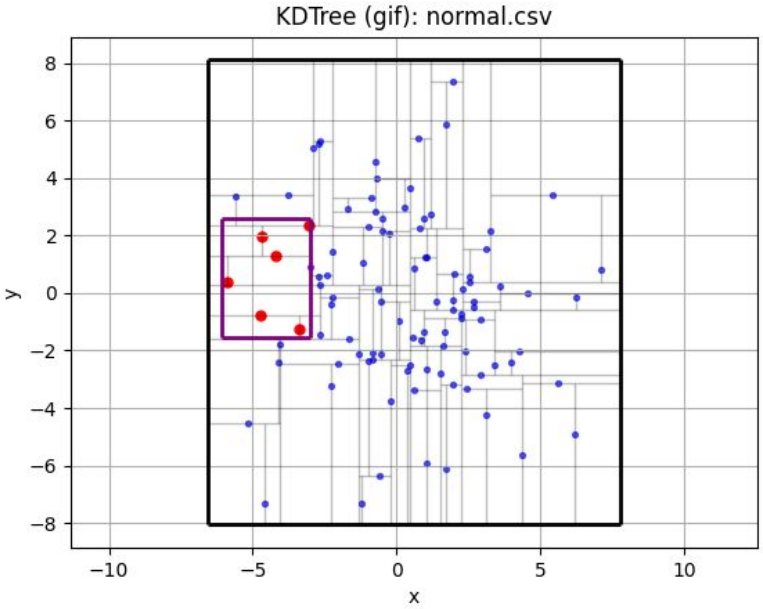
Przeszukiwanie KD-Tree

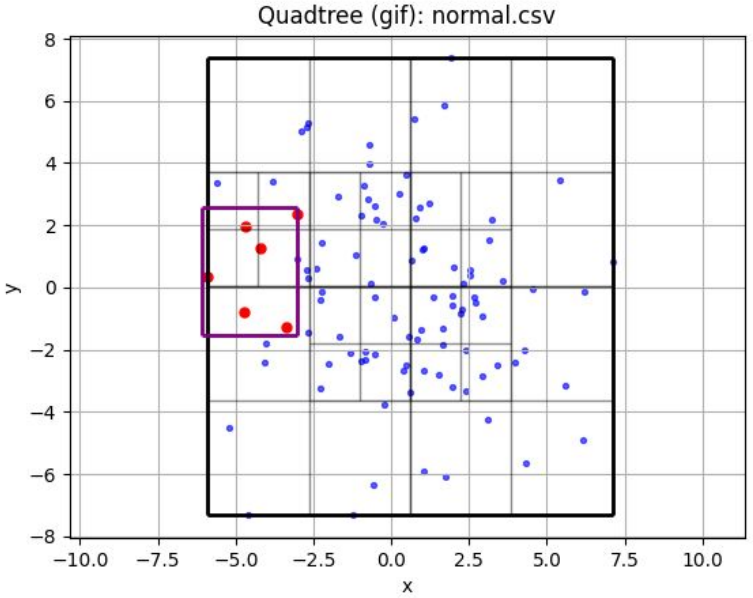
```python
def _search(self, node: KDNode, range_rect: Rect, depth: int, result: List[Point]):  3 usages  Maksymilian Siemek
    if node is None:
        return

    # 1. Sprawdź, czy punkt w bieżącym węźle mieści się w prostokącie
    if range_rect.contains_point(node.point):
        result.append(node.point)

    # 2. Rozstrzygnij, do których dzieci warto zajrzeć
    axis = depth % 2

    # Wartości porównawcze: współrzędna punktu i granice prostokąta
    if axis == 0:  # Oś X
        val = node.point.x
        low = range_rect.left
        high = range_rect.right
    else:  # Oś Y
        val = node.point.y
        low = range_rect.bottom
        high = range_rect.top

    # Jeśli lewa/dolna część prostokąta jest mniejsza od mediany, przeszukaj lewe dziecko
    if low < val:
        self._search(node.left, range_rect, depth + 1, result)

    # Jeśli prawa/górna część prostokąta jest większa lub równa medianie, przeszukaj prawe dziecko
    if high >= val:
        self._search(node.right, range_rect, depth + 1, result)
```

`_search(node, rect, depth, result)`:

- jeśli `node == None` → stop
- jeśli `rect.contains_point(node.point)` → dodaj punkt do `result`
- `axis = depth % 2` (0=X, 1=Y)
- porównaj granice prostokąta z `val` (x albo y punktu węzła):
  - jeśli `low < val` → szukaj w `left`
  - jeśli `high >= val` → szukaj w `right`

Porównanie QuadTree i KD-Tree





Rozkład Normalny

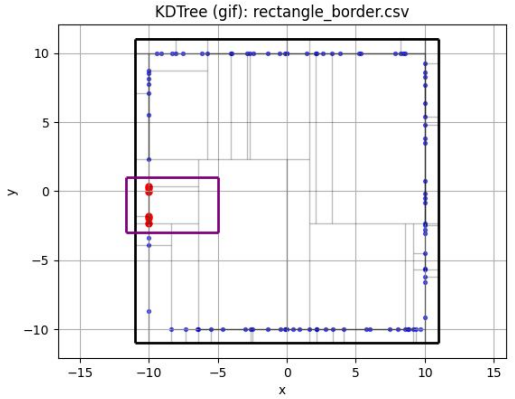
Dataset	PointsNo	QT_FoundPoints	QuadTreeBuildTime	QuadTreeQueryTime	KD_FoundPoints	KDTreeBuildTime	KDTreeQueryTime
rozkład normalny	100	6	0,0005	0,0000	6	0,0002	0,0000
rozkład normalny	1000	137	0,0080	0,0002	137	0,0026	0,0002
rozkład normalny	10000	1127	0,1497	0,0022	1127	0,0582	0,0020
rozkład normalny	100000	17069	1,8060	0,0245	17069	0,8583	0,0249
rozkład normalny	1000000	586556	21,9779	0,8853	586556	13,7860	0,9071

W przypadku rozkładu normalnego punkty są skoncentrowane w centralnej części obszaru, co sprzyja strukturze KDTree przy budowie, ale niekoniecznie podczas wyszukiwania. Jak widać w wynikach, czas budowy KDTree jest krótszy, natomiast QuadTree szybciej wykonuje zapytania (query) dla większych zbiorów danych

Oznacza to, że przy danych skupionych w jednym obszarze QuadTree lepiej radzi sobie z przeszukiwaniem, dlatego w takim przypadku ta struktura jest korzystniejsza.

Obwód prostokąta

W tym rozkładzie punkty leżą głównie wzdłuż krawędzi prostokąta, więc dane są mocno „wydłużone” i nierównomiernie zajmują przestrzeń. W takim przypadku QuadTree często musi wykonywać więcej podziałów (dużo pustych ćwiartek i głębsze schodzenie), co widać po wyraźnie większym czasie budowy. Z kolei KDTree radzi sobie tu lepiej — zarówno buduje się dużo szybciej, jak i wykonuje zapytania szybciej (query time jest konsekwentnie mniejszy dla KDTree). Dlatego dla punktów rozłożonych na obwodzie prostokąta korzystniejszym wyborem jest KDTree.



Dataset	PointsNo	QT_FoundPoints	QuadTreeBuildTime	QuadTreeQueryTime	KD_FoundPoints	KDTreeBuildTime	KDTreeQueryTime
obwód prostokąta	100	6	0,0008	0,0000	6	0,0002	0,0000
obwód prostokąta	1000	49	0,0109	0,0001	49	0,0021	0,0001
obwód prostokąta	10000	537	0,1656	0,0011	537	0,0332	0,0007
obwód prostokąta	100000	4917	2,6117	0,0104	4917	0,5585	0,0066
obwód prostokąta	1000000	22363	32,8487	0,0408	22363	8,2818	0,0309

Na podstawie poprzednich wyników widać, że wybór struktury powinien zależeć od tego, **jak rozłożone są punkty**.

QuadTree warto wybrać, gdy:

- punkty tworzą **lokalne skupiska** (np. rozkład normalny, „chmura” w jednym obszarze),
- zależy nam na szybkich zapytaniach w sytuacji, gdy wiele punktów leży w tej samej części przestrzeni,
- dane dobrze „pasują” do podziału na ćwiartki (naturalne grupowanie w regionach).

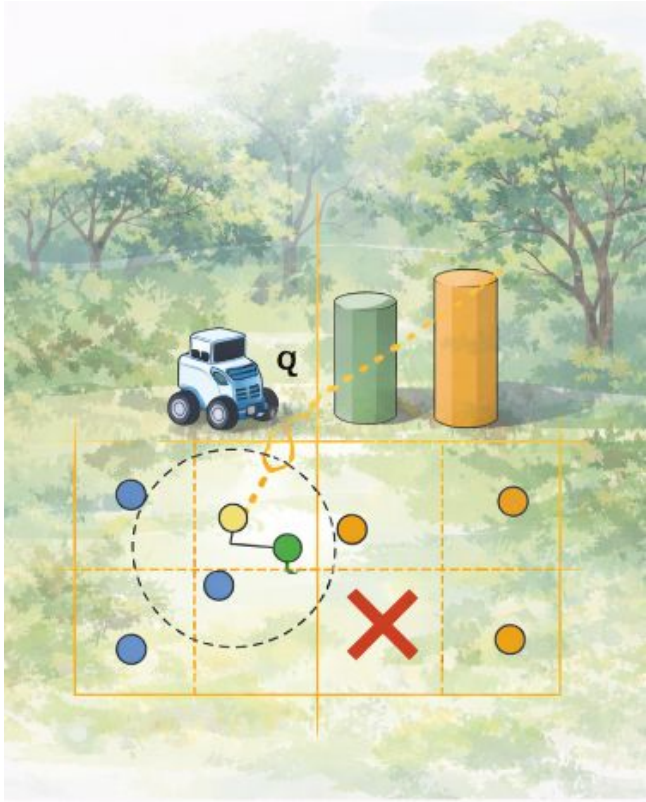
KDTree warto wybrać, gdy:

- punkty układają się **wydłużenie / w pasach / na krawędziach** (np. obwód prostokąta),
- potrzebujemy stabilnie szybkich zapytań dla danych, które nie wypełniają równomiernie obszaru,
- chcemy zwykle też krótszy czas budowy.

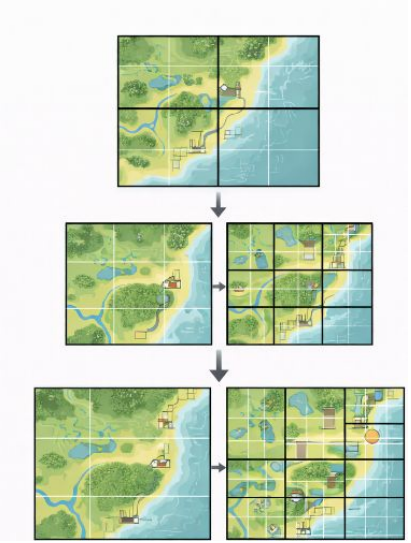
Ciekawostki KD-Tree

KD-tree jest często używane do wyszukiwania najbliższego sąsiada (NN) lub k najbliższych (k-NN), bo pozwala mocno ograniczyć liczbę sprawdzanych punktów.

- Jak działa (intuicja): schodzimy w dół drzewa do regionu, w którym leży punkt zapytania q, znajdujemy pierwszego kandydata, a potem wracając w górę odcinamy całe poddrzewa, które na pewno nie zawierają lepszego wyniku (są „za daleko” od q).
- Zastosowania: robotyka i LiDAR (najbliższa przeszkoda/punkt chmury), grafika i geometria (dopasowanie punktów, kolizje), proste zapytania „najbliższy obiekt” w danych 2D/3D.
- Ograniczenie: działa świetnie w niskich wymiarach (2D/3D), ale w wysokich wymiarach traci przewagę (klątwa wymiarowości).



Ciekawostki QuadTree



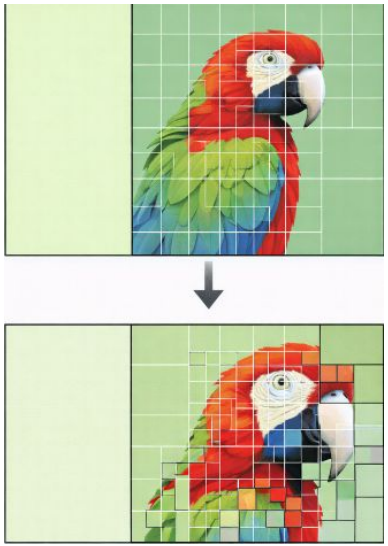
Mapy / GIS:
QuadTree świetnie pasuje do danych przestrzennych, bo dzieli obszar na kafelki: 4 ćwiartki, potem każda znów na 4 itd. Dzięki temu łatwo:

- szybko znaleźć obiekty w danym fragmencie mapy (zapytania obszarowe),
- renderować mapę „warstwami” (różne poziomy szczegółowości przy przybliżaniu/oddalaniu),
- przechowywać dane regionami (mniej pracy poza interesującym obszarem).

Kompresja obrazów:
W kompresji Quadtree dzieli obraz na kwadraty tak długo, aż dany fragment jest „w miarę jednolity” (np. prawie jeden kolor).

- duże jednolite tło → zapisuje się jako jeden duży blok,
- szczegółowe miejsca (krawędzie, tekstury) → dzielone są na mniejsze bloki.

Efekt: mniej danych do zapisania, bo szczegółowość jest tylko tam, gdzie jest potrzebna.



Dziękujemy za uwagę

Bibliografia:

[1] dr inż. Barbara Głut, Algorytmy gemetryczne - wykład.

[2] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.*Computational Geometry: Algorithms and Applications*,Springer, 2008.

[3] Wikipedia

[4] źródła grafik do ciekawostek: ChatGPT