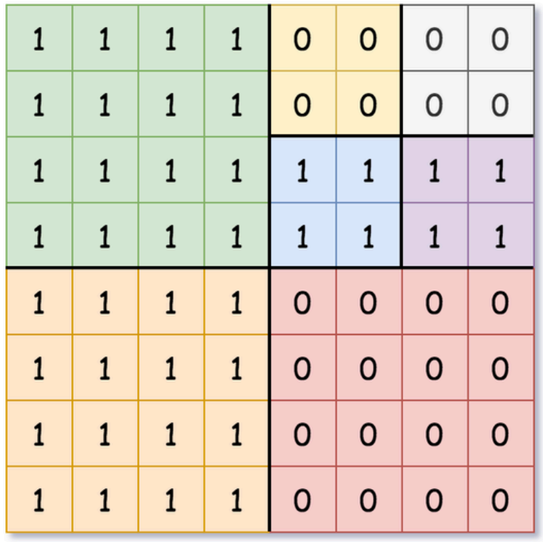
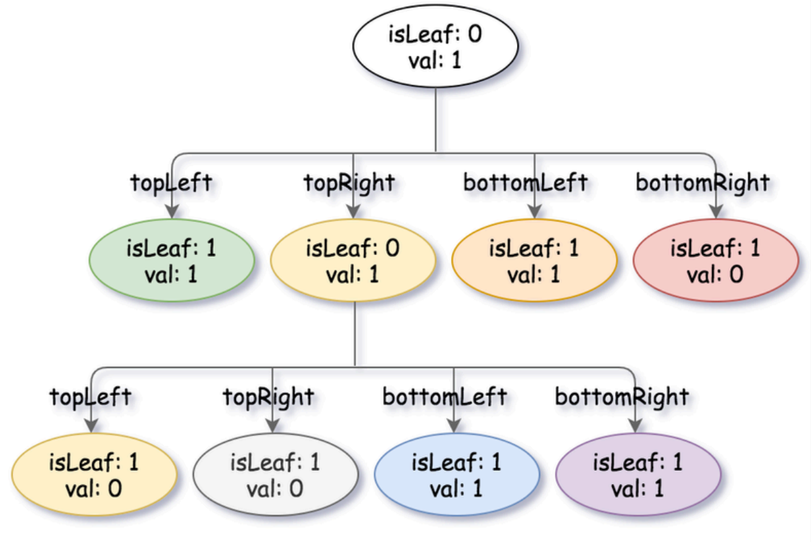
Algorytmy geometryczne - Quad Trees oraz KD Trees





Powyższa prezentacja została stworzona przez Szymona Tyburczy oraz Marka Swakonia.

Drzewa

Podstawowe informacje na temat struktur drzewiastych

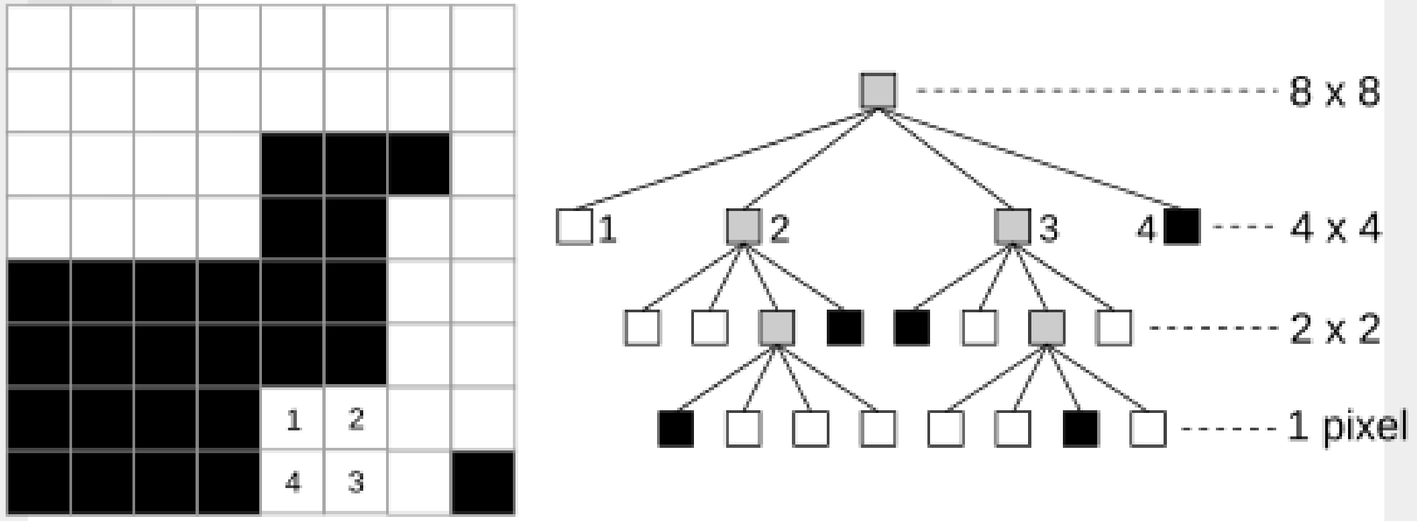
Wstęp do zagadnień związanych z drzewami

Drzewo to struktura danych składająca się z wierzchołków (węzłów) i krawędzi, przy czym krawędzie łączą wierzchołki w taki sposób, iż istnieje zawsze dokładnie jedna droga pomiędzy dowolnymi dwoma wierzchołkami.

Wierzchołki w drzewach przedstawiamy w postaci warstwowej, tzn. każdy wierzchołek w drzewie znajduje się na jakimś poziomie. Poziom wierzchołka w drzewie jest równy długości drogi łączącej go z korzeniem. Korzeń drzewa jest na poziomie 0. Wysokość drzewa równa jest maksymalnemu poziomowi drzewa, czyli długości najdłuższej spośród ścieżek prowadzących od korzenia do poszczególnych liści drzewa. Wierzchołki mogą posiadać rodzica, który jest umieszczony na wyższym poziomie oraz dzieci, które są umieszczone na niższym poziomie. Niektóre dzieci nie posiadają własnych dzieci i są liśćmi. Dzieci jednego rodzica nazywamy rodzeństwem. Wierzchołki, które nie posiadają ani jednego dziecka nazywamy liśćmi.

Quad Trees

Drzewo czwórkowe (ang. quadtree) – struktura danych będąca drzewem, używana do podziału dwuwymiarowej przestrzeni na mniejsze części, dzieląc ją na cztery równe ćwiartki, a następnie każdą z tych ćwiartek na cztery kolejne itd.



Bitmapa reprezentowana przez drzewo czwórkowe

Quad Trees - struktura

Drzewa czwórkowe działaja jak zwykłe drzewa

Korzeń (root): Reprezentuje całą przestrzeń, którą chcemy podzielić.
Gałęzie (branches): Są węzłami wewnętrznymi, które dzielą przestrzeń na cztery ćwiartki.
Liście (leaves): Reprezentują najmniejsze podzielone jednostki, które zawierają konkretne dane.

02

Podział przestrzeni:

- Węzeł dzieli przestrzeń na cztery części: północno-zachodnią, północno-wschodnią, południowo-zachodnią i południowo-wschodnią.
- Każda ćwiartka może być dalej podzielona, jeśli zawiera więcej danych niż określony próg.

Taki system umożliwia szybkie wyszukiwanie i manipulowanie danymi w dużych zbiorach przestrzennych.

Quad Tree - działanie

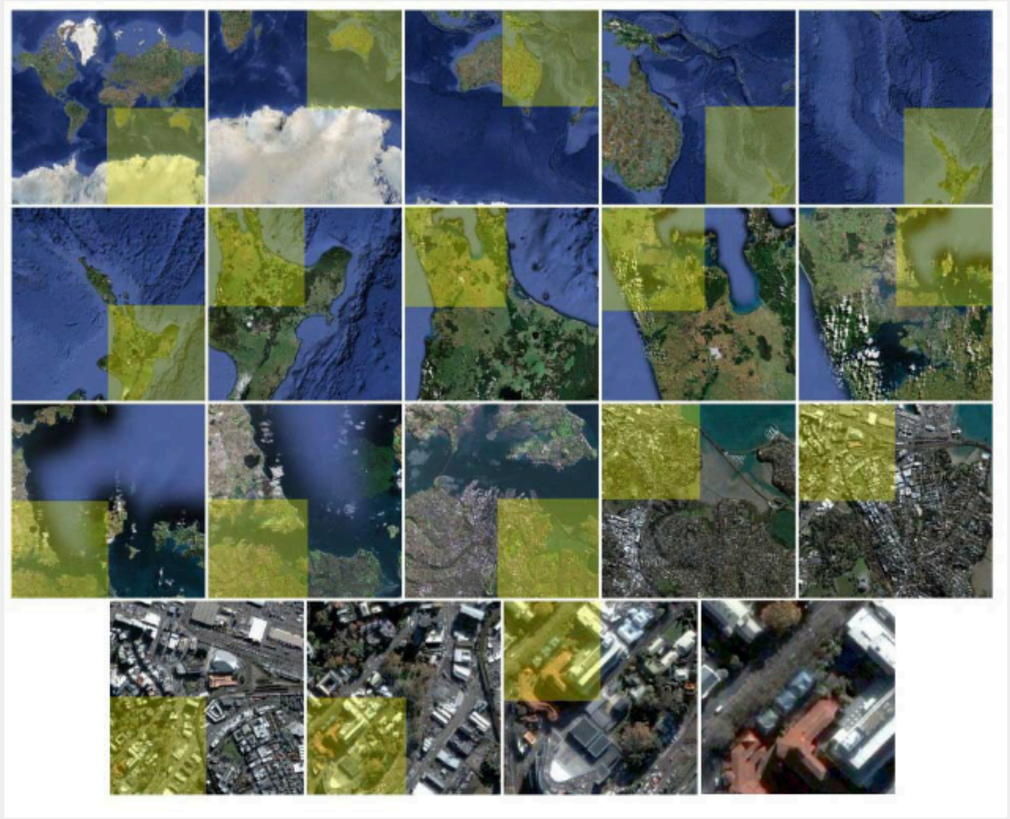
Działanie Quad Tree można opisać w kilku krokach:

- **Inicjalizacja: Cała przestrzeń jest reprezentowana jako pojedynczy węzeł (korzeń).**
- **Podział: Jeśli dane w węźle przekraczają określony próg (np. liczba obiektów), przestrzeń jest dzielona na cztery równe części.**
- **Rekurencja: Proces podziału jest powtarzany dla każdej ćwiartki, aż dane w każdej z nich będą łatwe do zarządzania.**
- **Wyszukiwanie: Aby znaleźć dane, przeszukujemy tylko te ćwiartki, które mogą zawierać poszukiwany element.**

Zastosowania Quad Tree

Quad Tree jest szeroko stosowane w różnych dziedzinach, takich jak:

- Grafika komputerowa:
- Ułatwia renderowanie scen 2D i 3D poprzez optymalizację podziału przestrzeni.
- Stosowany w grach do zarządzania obiektami na mapie.
- Mapy cyfrowe i GIS:
- Umożliwia przechowywanie i wyszukiwanie danych geograficznych w dużych skalach.
- Google Maps wykorzystuje Quad Tree do efektywnego ładowania map.
- Przetwarzanie obrazów:
- Kompresja obrazów przez dzielenie ich na mniejsze części.
- Analiza obrazów, np. identyfikacja obiektów.
- Symulacje i modelowanie:
- Zarządzanie danymi przestrzennymi w modelach symulacyjnych.



Implementation of quadtrees within Google Maps. Note the yellow quadrant on each image correspond to the next image when zooming. ©2009 Google - Imagery ©2009 TerraMetrics, DigitalGlobe, GeoEye

Zalety i wady Quad Tree

Zalety Quad Tree

- **Efektywne zarządzanie danymi: Umożliwia łatwe podzielenie dużej ilości danych na mniejsze, bardziej zorganizowane części.**
- **Szybki dostęp: Dzięki hierarchii, wyszukiwanie danych w konkretnym obszarze jest szybkie i efektywne.**
- **Wszechstronność: Może być stosowane w różnych dziedzinach, od gier komputerowych po systemy GIS.**
- **Optymalizacja pamięci: Quad Tree przechowuje tylko istotne dane w odpowiednich miejscach, co minimalizuje zużycie pamięci.**

Wady

- **Złożoność implementacji: Wymaga starannego zaprojektowania algorytmu, zwłaszcza w przypadku dynamicznie zmieniających się danych.**
- **Problemy z pamięcią: Dla bardzo szczegółowych danych Quad Tree może zajmować dużo pamięci, gdy przestrzeń jest podzielona na wiele małych ćwiartek.**
- **Nierównomierne rozmieszczenie danych: Jeśli dane są skupione w jednym miejscu, Quad Tree może stać się nieefektywne.**

Implementacja Quad Tree

W naszej implementacji algorytmu Quad Tree używamy 3 klas.

- Pierwszą klasą jest klasa Point która przechowuje nam punkt na płaszczyźnie
- Drugą klasą jest klasa QuadTreeNode posiadającą konstruktor granic obszaru
- Ostatnią klasą jest QuadTree w której zaimplementowana jest reszta metod.

Klasa Point zawiera w sobie konstruktor punktu na płaszczyźnie
który tworzy punkt o współrzędnych (x, y) oraz metode __repr__
zwracającą reprezentację tekstową punktu w postaci Point(x, y).

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

    def __repr__(self):
        return f"Point({self.x}, {self.y})"
```

Implemetnacja Quad Tree

Klasa QuadTreeNode to klasa
reprezentująca pojedynczy węzeł w drzewie QuadTree.
Każdy taki węzeł odpowiada pewnemu
prostokątnemu obszarowi w przestrzeni 2D
i przechowuje informacje o punktach
należących do tego obszaru. Węzeł może być:

- Liściem – przechowuje punkty bez podziału na podobszary.
- Węzłem wewnętrznym – podzielony na cztery podobszary (ćwiartki) z odnośnikami do dzieci.

W kodzie klasy QuadTreeNode uściślamy przedziały danego węzła
• lista points przechowuje punkty w tym węźle (tylko jeśli jest liściem bo jeśli nie jest liściem to ma dzieci i nie ma punktów)
• nw, ne, sw, se określają podział na 4 części zgodnie z zasadą tworzenia QuadTree
• Na samym końcu znajduje się informacja czy węzeł jest liściem

Legenda oznaczeń:
x_min (float): Minimalna wartość współrzędnej x dla obszaru korzenia.
y_min (float): Minimalna wartość współrzędnej y dla obszaru korzenia.
x_max (float): Maksymalna wartość współrzędnej x dla obszaru korzenia.
y_max (float): Maksymalna wartość współrzędnej y dla obszaru korzenia.

```python
class QuadTreeNode:
    """
    Węzeł w drzewie QuadTree przechowujący:
      - granice obszaru (x_min, y_min, x_max, y_max),
      - listę punktów (jeśli jest liściem),
      - odnośniki do dzieci: NW, NE, SW, SE.
    """

    def __init__(self, x_min, y_min, x_max, y_max):
        self.x_min = x_min  # lewy dolny rog wezla
        self.y_min = y_min  # lewy dolny rog wezla
        self.x_max = x_max  # prawy gorny rog wezla
        self.y_max = y_max  # prawy gorny rog wezla

        # Punkty przechowywane w tym węźle (tylko jeśli jest liściem bo jeśli nie jest liściem to ma dzieci i nie ma punktów )
        self.points = []

        # Dzieci: kolejność (NW, NE, SW, SE), podział na 4 części zgodnie z zasadą tworzenia Quad Tree
        self.nw = None
        self.ne = None
        self.sw = None
        self.se = None

        # Informacja, czy węzeł jest liściem (ma dzieci czy nie)
        self.is_leaf = True
```

Implementacja QuadTree

Klasa QuadTree

Klasa QuadTree to klasa główna algorytmu zawierająca wiele metod

Na samym początku możemy znaleźć konstruktor klasy w którym tworzymy korzeń drzewa.

```
class QuadTree:
    """
    Klasa zarządzająca QuadTree:
      - posiada korzeń (root),
      - operacje insert (wstawianie punktu),
      - operacje query_range (wyszukiwanie punktów w zadanym prostokącie).
    """

    def __init__(self, x_min, y_min, x_max, y_max, capacity=4):
        """
        Tworzy korzeń drzewa z zadaną pojemnością liścia (capacity),
        która określa maksymalną liczbę punktów przed podziałem.
        """
        self.root = QuadTreeNode(x_min, y_min, x_max, y_max)
        self.capacity = capacity  # maksymalna liczba punktów w węźle przed podziałem czyli 4
```

Implementacja QuadTree

Metoda insert oraz metoda insert_point

Metoda insert_point to publiczna metoda wstawiania obiektu klasy Point do całego drzewa, „właściwe” wstawianie dzieje się w metodzie insert(node, point)

Metoda insert

insert(node, point), która:
Sprawdza, czy node jest liściem (node.is_leaf).
Jeśli jest liściem i ma mniej niż 4 punkty dodaje punkt do node.points.
Jeśli jest liściem, ale przekroczył limit (capacity = 4), wówczas:
dzieli węzeł (subdivide(node) na 4 dzieci,
przenosi wszystkie dotychczasowe punkty w node.points do odpowiednich dzieci, wstawia także nowy punkt do właściwego dziecka.
Jeśli node nie jest liściem, od razu rekurencyjnie kieruje punkt do któregoś z podwęzłów (NW, NE, SW, SE) w zależności od współrzędnych.
Złożoność: insert_point - O(logN)

```python
def insert_point(self, x, y):
    """
    Publiczna metoda wstawiania punktu (x, y) do całego drzewa.
    """
    p = Point(x, y)
    self.insert(self.root, p)
```

```python
def insert(self, node, point):
    """
    Wstawia punkt do drzewa, zaczynając od podanego węzła `node`.
    """
    # Jeśli węzeł jest liściem
    if node.is_leaf:
        # Jeśli jest jeszcze miejsce, to dodaj punkt
        if len(node.points) < self.capacity:
            node.points.append(point)
            return
        else:
            # Przekroczono pojemność -> podział
            self.subdivide(node)

            # Przenieś istniejące punkty do odpowiednich dzieci
            while node.points:
                p = node.points.pop()
                self.insert_into_child(node, p)

            # Teraz wstawiany punkt również ląduje w odpowiednim dziecku
            self.insert_into_child(node, point)

    else:
        # Jeśli węzeł nie jest liściem -> rekurencyjnie wstaw do dziecka
        self.insert_into_child(node, point)
```

Implementacja QuadTree

Metoda insert_into_child(self, node, point):
Pomocnicza metoda określająca, do którego dziecka węzła należy wstawić dany punkt.
Parametry:
• node (QuadTreeNode): Węzeł, którego dzieci przeszukujemy.
• point (Point): Punkt do wstawienia.
Oblicz współrzędne środka prostokąta reprezentowanego przez węzeł:
Na podstawie położenia punktu:
Jeśli lewy górny róg podziału , wstaw do dziecka NW.
Jeśli prawy górny róg podziału , wstaw do dziecka NE.
Jeśli lewy dolny róg podziału, wstaw do dziecka SW.
Jeśli prawy dolny róg podziału, wstaw do dziecka SE.
Rekurencyjnie wywołaj insert na odpowiednim dziecku.
Złożoność:insert_into_child - O(1)

```python
def insert_into_child(self, node, point):
    """
    Pomocnicza funkcja określająca, do którego dziecka węzła wstawić dany punkt. obliczamy srodek prostokata
    """
    mid_x = (node.x_min + node.x_max) / 2.0
    mid_y = (node.y_min + node.y_max) / 2.0

    # W zależności od położenia punktu wybieramy NW, NE, SW lub SE
    if point.x <= mid_x and point.y >= mid_y:
        # NW
        self.insert(node.nw, point)
    elif point.x > mid_x and point.y >= mid_y:
        # NE
        self.insert(node.ne, point)
    elif point.x <= mid_x and point.y < mid_y:
        # SW
        self.insert(node.sw, point)
    else:
        # SE
        self.insert(node.se, point)
```

Implementacja QuadTree

Metoda subdivide(self, node)
Dzieli węzeł na cztery mniejsze prostokątne obszary, tworząc dzieci NW, NE, SW i SE.
Parametry:
• node (QuadTreeNode): Węzeł do podzielenia.
Oblicza środek prostokąta i tworzy cztery nowe węzły reprezentujące podobszary
• NW ()
• NE ()
• SW ()
• SE ()
Ustaw nowo utworzone węzły jako dzieci node.
Ustaw flagę is_leaf na False bo skoro ma dzieci to to nie liść.

Złożoność:subdivide - O(1)

```python
def subdivide(self, node):
    """
    Dzieli węzeł (node) na 4 dzieci: NW, NE, SW, SE.
    """
    mid_x = (node.x_min + node.x_max) / 2.0
    mid_y = (node.y_min + node.y_max) / 2.0

    # Tworzymy 4 podwęzły
    # NW
    node.nw = QuadTreeNode(
        x_min=node.x_min,
        y_min=mid_y,
        x_max=mid_x,
        y_max=node.y_max
    )
    # NE
    node.ne = QuadTreeNode(
        x_min=mid_x,
        y_min=mid_y,
        x_max=node.x_max,
        y_max=node.y_max
    )
    # SW
    node.sw = QuadTreeNode(
        x_min=node.x_min,
        y_min=node.y_min,
        x_max=mid_x,
        y_max=mid_y
    )
    # SE
    node.se = QuadTreeNode(
        x_min=mid_x,
        y_min=node.y_min,
        x_max=node.x_max,
        y_max=mid_y
    )

    node.is_leaf = False
```

Implementacja QuadTree

Algorytm change wywoływany w algorytmie query zamienia punkty klasy Point() na floaty.
Przyjmuje liste znalezionych punktów.
Złożoność: change - O(K)
Iteracja przez listę punktów: O(K), gdzie K to liczba punktów wynikowych.

```
def change(result):
    new_result = []
    for i in range(len(result)):
        new_result.append((result[i].x, result[i].y))
    return new_result
```

Algorytm query() to publiczna metoda do wyszukiwania punktów w prostokącie. Korzysta z algorytmu query_range() opisanego na kolejnym slajdzie. Zwraca liste punktów “k” znalezionych takich, że spełniają warunki zadania tj. x1 <= kx <= x2 oraz y1 <= ky <=y2.

```
def query(self, x1, y1, x2, y2):
    """
    Publiczna metoda wyszukiwania punktów w prostokącie (x1, y1, x2, y2).
    Zwraca listę znalezionych punktów.
    """
    found_points = []
    self.query_range(self.root, x1, y1, x2, y2, found_points)
    return change(found_points)
```

Implementacja QuadTree

Algorytm query_range() przyjmuje węzeł od którego zaczynamy poszukiwanie, lewy dolny róg prostokąta oraz prawy górny róg prostokąta, found_points to lista, do której dodawane są znalezione punkty.

Algorytm w działaniu:
Jeśli obszar węzła nie nachodzi na zadany prostokąt, zakończ (brak wyników).
Jeśli węzeł jest liściem:
Przejrzyj wszystkie punkty w węźle.
Dodaj punkty znajdujące się w prostokącie do found_points.

W innym przypadku jeśli węzeł posiada dzieci, rekurencyjnie przeszukaj każde dziecko

```python
def query_range(self, node, x1, y1, x2, y2, found_points):  #
    """
    Wyszukuje punkty w zadanego prostokąta [x1, x2] x [y1, y2],
    zaczynając od węzła `node`.

    Wyniki dokłada do listy `found_points`.
    """
    # Jeśli nie ma przecięcia obszarów, to kończymy bo
    if not self.intersects(node, x1, y1, x2, y2):
        return

    # Jeśli węzeł jest liściem, to sprawdź wszystkie punkty
    if node.is_leaf:
        for p in node.points:
            if (x1 <= p.x <= x2) and (y1 <= p.y <= y2):
                found_points.append(p)
    else:
        # Przeszukaj rekurencyjnie dzieci
        self.query_range(node.nw, x1, y1, x2, y2, found_points)
        self.query_range(node.ne, x1, y1, x2, y2, found_points)
        self.query_range(node.sw, x1, y1, x2, y2, found_points)
        self.query_range(node.se, x1, y1, x2, y2, found_points)
```

```python
def find_end_points(points):
    x_min = float('inf')
    x_max = float('-inf')
    y_min = float('inf')
    y_max = float('-inf')

    for x, y in points:
        x_min = min(x_min, x)
        x_max = max(x_max, x)
        y_min = min(y_min, y)
        y_max = max(y_max, y)
    return x_min, y_min, x_max, y_max
```

find_end_points nie dotyczy już bezpośrednio żadnej z klas. Zadaniem tej funkcji jest wyszukanie granic dla korzenia QuadTree
Złożoność: find_end_points - O(M)
Znajduje minimalne i maksymalne wartości współrzędnych punktów iterując przez M punktów.

```python
def CreateQuad(points_to_insert):
    qt = QuadTree(find_end_points(points_to_insert)[0], find_end_points(points_to_insert)[1],
                  find_end_points(points_to_insert)[2], find_end_points(points_to_insert)[3], capacity=4)
    for (x, y) in points_to_insert:
        qt.insert_point(x, y)
    return qt
```

Funkcja CreateQuad buduje QuadTree używajac odpowiednich przedziałów oraz wstawia wszystkie punkty w stworzone drzewo.

M: Oznacza liczbę punktów, które są wstawiane do drzewa QuadTree. To dane wejściowe funkcji.
N: Oznacza liczbę węzłów w drzewie QuadTree. Liczba węzłów zależy od tego, jak głęboko drzewo zostało podzielone.

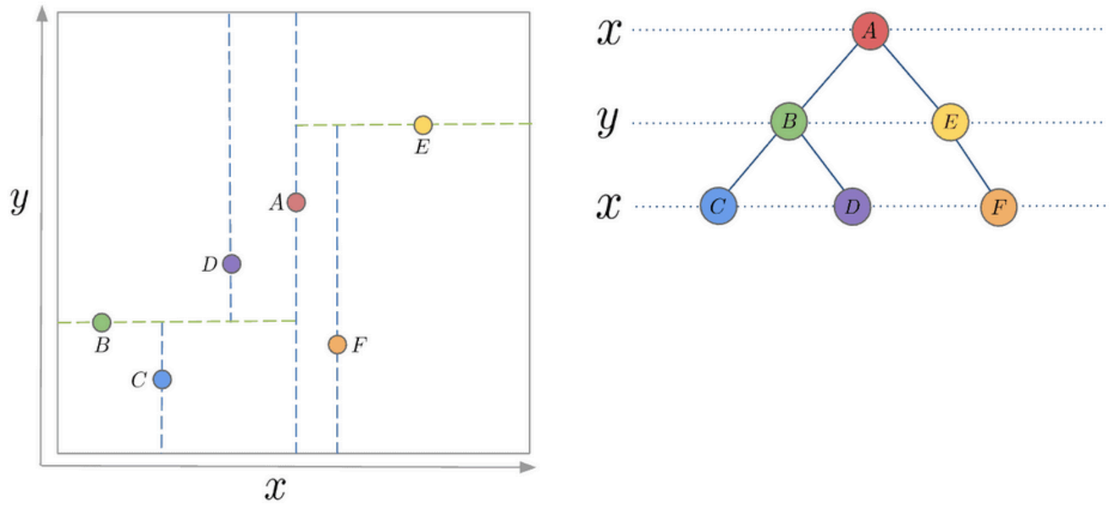
Złożoność: CreateQuad - O(MlogN)
1. Szukamy granic obszaru O(M)
2. Tworzy QuadTree O(1)
3. Wstawia każdy punkt O(MlogN)
Zatem :O(M+MlogN)=O(MlogN)

```python
def intersects(self, node, x1, y1, x2, y2):
    """
    Sprawdza, czy obszar węzła [x_min, x_max] x [y_min, y_max]
    nachodzi na [x1, x2] x [y1, y2].
    """
    return not (node.x_max < x1 or node.x_min > x2 or
                node.y_max < y1 or node.y_min > y2)
```

Sprawdza, czy obszar prostokąta węzła przecina się z zadanym prostokątem.
Złożoność: intersects - O(1)

KD Trees

Drzewo kd (ang. k-d tree) – struktura danych, służąca do przechowywania danych o k wymiarach, znajdowania punktów należących do zadanego obszaru ortogonalnego oraz do wyszukiwania najbliższych sąsiadów.



Sposób podziału płaszczyzny oraz konstrukcja drzewa

https://www.baeldung.com/cs/k-d-trees

KD Tree - struktura

KD Tree to wariant drzewa binarnego, więc działa na podobnej zasadzie

Korzeń (root): Za jego pomocą mamy dostęp do całej struktury drzewa.
Wierzchołki (Vertex): Węzły wewnętrzne, reprezentujące proste dzielące płaszczyznę na dwie części według współrzędnej zawieranego punktu.
Liście (Leaf): Wierzchołek który nie zawiera dalszych rozgałęzień.

02

Podział przestrzeni:

- Każdy węzeł reprezentuje podział przestrzeni na dwie podprzestrzenie.
- Utworzone w ten sposób podprzestrzenie są dzielone do momentu, aż w podpłaszczyźnie zostanie jeden punkt, który wpisywany jest do liścia.

Taka struktura umożliwia sprawne przeszukiwanie danych o dowolnej liczbie wymiarów.

KD Tree - działanie

Działanie KD Tree można opisać w kilku krokach:

- Inicjalizacja: Cała przestrzeń jest reprezentowana jako pojedynczy węzeł (korzeń).
- Podział: Płaszczyzna jest dzielona na dwie podpłaszczyzny przez prostą o współrzędnej mediany rozpatrywanej płaszczyzny dla danego wymiaru.
- Rekurencja: Proces podziału jest powtarzany dla każdej kolejnej płaszczyzny, do momentu aż płaszczyzna będzie zawierać pojedynczy punkt.
- Wyszukiwanie: Przeszukując drzewo, wybieramy jedynie te poddrzewa, które przecinają się z poszukiwanym obszarem.

Zastosowania KD Tree

Quad Tree jest szeroko stosowane w różnych dziedzinach, takich jak:

- Wyszukiwanie najbliższych punktów w przestrzeni (nearest neighbor search) wykorzystywane w:
- machine learningu
- systemach rekomendacji
- analizie danych przestrzennych
- Przetwarzanie obrazów oraz wideo
- Wyszukiwanie zakresowe

Zalety i wady KD Tree

Zalety

- Efektywne wyszukiwanie najbliższych sąsiadów (KNN), co jest przydatne w rozpoznawaniu obrazów i systemach rekomendacji
- Efektywne wyszukiwanie zakresowe, czyli znajdowanie punktów należących do konkretnego obszaru

Wady

- Course of Dimensionality, przy wyższych liczbach wymiarów, spada efektywność tej struktury, ponieważ wykładniczo rośnie objętość przestrzeni między punktami
- Wrażliwość na wybór linii podziału wzdłuż której dzielona jest przestrzeń. Nieoptymalny wybór takiej prostej może skutkować niezrównoważoną strukturą drzewa
- KD Trees mogą gorzej działać dla danych o silnie nierównomiernym rozkładzie

Implementacja KD Tree

W naszej implementacji algorytmu KD używamy klasy głównej i 2 podklas.

- Klasa główna KDTree zawiera metody budujące i przeszukujące drzewo, metody oraz klasy pomocnicze. Jej podstawowym argumentem jest korzeń, który umożliwia dostęp do struktury drzewa.
- Pierwszą podklasą jest klasa Vertex, zawierająca punkt ze zbioru początkowego oraz lewe i prawe dziecko.
- Drugą podklasą jest klasa Leaf, zawierająca punkt ze zbioru początkowego, ale nie posiadająca dzieci.

```python
class KDtree:
    def __init__(self,points):
        xpoints = self.sort_by_dim(points,0)
        ypoints = self.sort_by_dim(points,1)
        self.root = self.build_tree([xpoints,ypoints],0)
```

```python
class Vertex:
    def __init__(self, point, left=None, right=None):
        self.point = point
        self.left = left
        self.right = right
```

```python
class Leaf:
    def __init__(self, point):
        self.point = point
```

Implementacja KD Tree

Klasa KDTree zawiera 2 główne metody:

Metoda build_tree - wywoływana przy inicjalizacji obiektu klasy KDTree. Tworzy strukturę drzewa kd.

build_tree wywołuje funkcję divide_points, która dzieli zbiór punktów według mediany w zadanym wymiarze i rekurencyjnie wywołuje się dla utworzonych podzbiorów

```python
def build_tree(self, points, depth):
    dim = depth % 2
    if len(points[dim]) == 1:
        return self.Leaf(points[dim][0])
    division = self.divide_points(points, dim)
    if division:
        median, left, right = division
    else:
        return None
    root = self.Vertex(median)
    root.left = self.build_tree(left, depth + 1)
    root.right = self.build_tree(right, depth + 1)
    return root
```

divide_points to metoda pomocnicza dla metody build_tree, dzieląca zbiór punktów na podzbiór lewy i prawy dla zadanego wymiaru

```python
def divide_points(self, points, dim):
    if not points[dim]:
        return None
    if len(points[dim]) == 1:
        return points[dim][0], [], []
    median = points[dim][(len(points[dim]) - 1) // 2][dim]
    median_point = points[dim][(len(points[dim]) - 1) // 2]
    left = []
    right = []
    for i in range(2):
        left_points = [
            point
            for point in points[i]
            if (point[dim] <= median and point != median_point)
        ]
        right_points = [point for point in points[i] if point[dim] > median]
        left.append(left_points)
        right.append(right_points)
    return median_point, left, right
```

Implementacja KD Tree

Klasa KDTree zawiera 2 główne metody:

Metoda search_area wykonuje algorytm przeszukiwania zakresowego

search_area rekurencyjnie przeszukuje drzewo KD, aby znaleźć punkty należące do zadanego obszaru (area). Wywołuje się rekurencyjnie jedynie na dzieciach, które zakresem nakładają się z poszukiwanym obszarem.

```python
def search_area(self, area, root, depth):
    dim = depth % 2
    if root is None:
        return []
    if root.getclass() == self.Vertex:
        left = []
        right = []
        if root.point[dim] >= area[0][dim]:
            left = self.search_area(area, root.left, depth + 1)
        if root.point[dim] <= area[1][dim]:
            right = self.search_area(area, root.right, depth + 1)
        if (area[0][0] <= root.point[0] <= area[1][0] and area[0][1] <= root.point[1] <= area[1][1]):
            return left + [root.point] + right
        else:
            return left + right
    else:
        if (area[0][0] <= root.point[0] <= area[1][0]) and (area[0][1] <= root.point[1] <= area[1][1]):
            return [root.point]
        return []
```

Złożoność obliczeniowa drzewa KD

Złożoności czasowe operacji w strukturze KDTree zależą od:

1. Liczby punktów n,
2. Sortowania użytego do wstępnego posortowania punktów przy inicjalizacji,
3. Głębokości drzewa (w przybliżeniu logn przy równomiernym podziale, a w najgorszym przypadku – O(n).
4. Dla przeszukiwania zakresowego, algorytm jest wrażliwy na wynik, także jego złożoność zależy także od k punktów należących do zadanego obszaru

Inicjalizacja KDTree

- W inicjalizacji drzewa, sortowany jest zbiór punktów. Używamy do tego algorytmu quicksort o złożoności oczekiwanej O(nlogn), a także wywoływana jest metoda build_tree

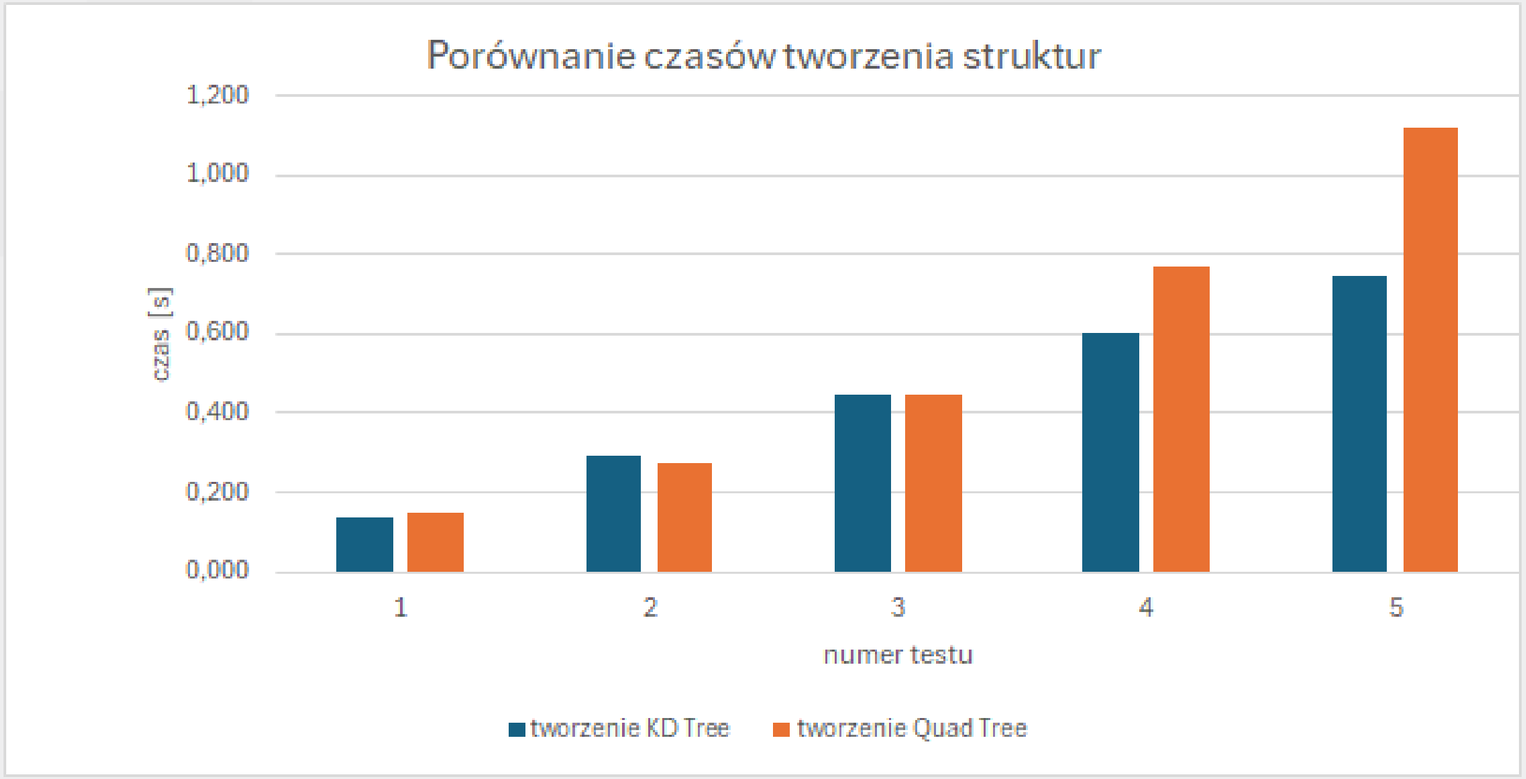
Metoda build_tree

- build_tree wywołuje metodę divide_points, która iteruje po tablicy points, złożoność O(n) w każdym wywołaniu metody
- Następnie dla podzielonych zbiorów metoda wywołuje się dla nich rekurencyjnie, co tworzy głębokość drzewa (przy założeniu że drzewo jest zrównoważone) O(logn).
- Łącznie daje nam to złożoność O(nlogn)

Metoda search_area

- każdy węzeł na danej głębokości jest odwiedzany, jeśli jego obszar przecina się z zadanym obszarem prostokątnym, w najgorszym przypadku odwiedzone zostaną wszystkie n wierzchołki
- Przeglądane są jedynie dzieci, których zakres przecina się z zadanym obszarem, co obniża liczbę odwiedzanych wierzchołków dla typowego przypadku do O(√n)
- Dodawanie punktów k należących do wyniku działa w złożoności O(k)
- Łącznie dla średniego przypadku złożoność wynosi O(√n + k), dla przypadku pesymistycznego O(n)

Porównanie szybkości tworzenia struktur



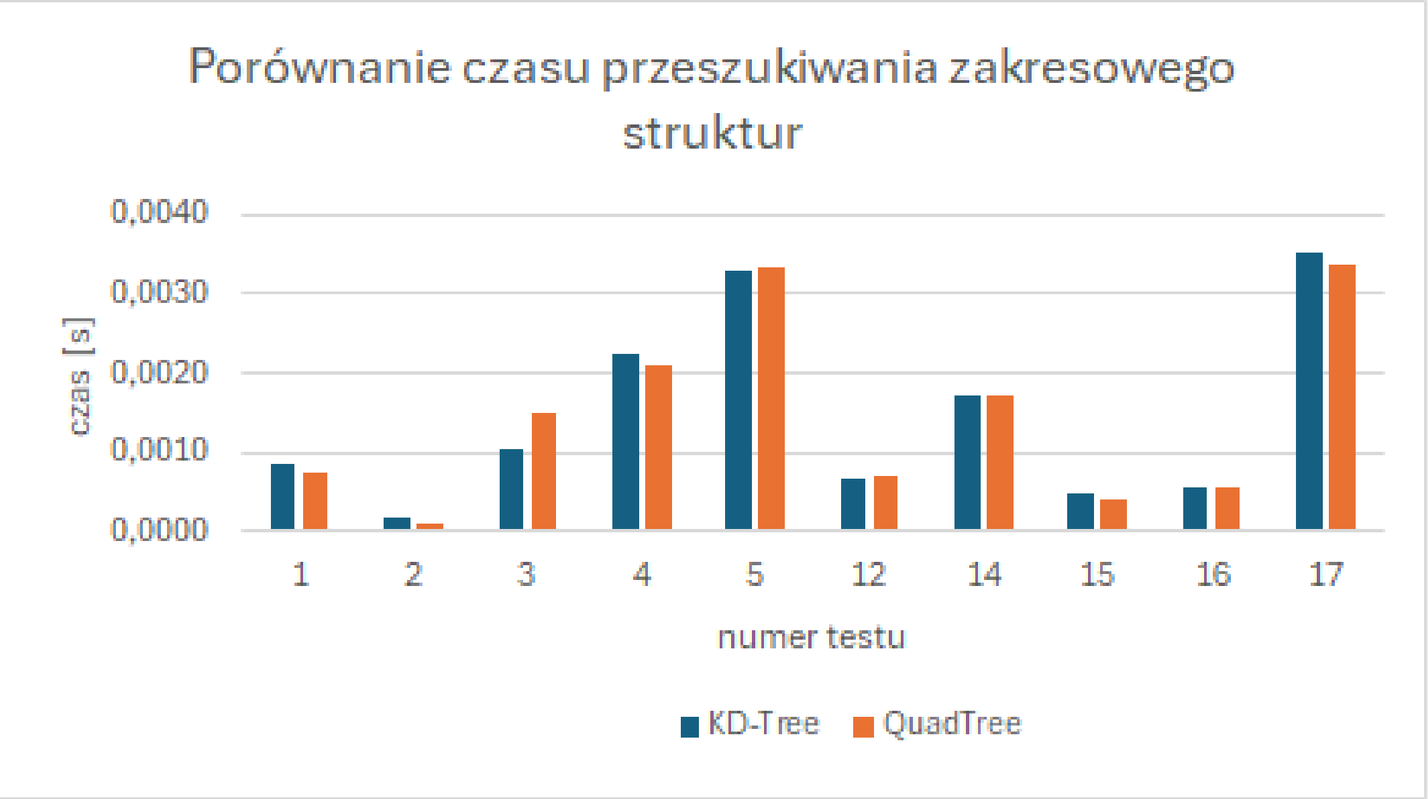
Dane testowe

Zbiory testowe 1-5 zawierają kolejno:

- test 1 - 20,000 punktów
- test 2 - 40,000 punktów
- test 3 - 60,000 punktów
- test 4 - 80,000 punktów
- test 5 - 100,000 punktów

Punkty zostały wygenerowane losowo z kwadratu ograniczonego punktami (-10000,-10000), (10000,10000)

Porównanie szybkości przeszukiwania struktur



Dane testowe

Zbiory testowe 1-5 mają parametry analogiczne jak zbiory na których zostały przeprowadzone testy czasowe budowania obu struktur.

Pozostałe zbiory widoczne na wykresie:

- test 12,14,15 - równomiernie rozmieszczone punkty na kwadracie
- test 16,17 - punkty równomiernie rozmieszczone na innym czworoboku

Podsumowanie pomiarów czasowych dla zaimplementowanych struktur

Budowanie struktur

Analiza:

- KD-Tree: Czas tworzenia struktury rośnie w miarę zwiększania się liczby punktów, ale wciąż jest stosunkowo szybki.
- QuadTree: Czas budowy jest generalnie dłuższy niż w przypadku KD-Tree, a różnice stają się większe przy większych zbiorach danych. Dla testu 5 czas budowy QuadTree jest wyraźnie dłuższy.

Wnioski:

- KD-Tree ma przewagę w czasie budowy przy większej liczbie punktów. Różnice te nie są dramatyczne, ale są zauważalne, zwłaszcza przy większych zestawach danych.
- QuadTree buduje się wolniej, szczególnie w przypadku większych zbiorów danych.

Przeszukiwanie struktur

Analiza:

- KD-Tree: Czas wyszukiwania w KD-Tree wykazuje tendencję wzrostu wraz ze wzrostem liczby punktów.
- QuadTree: Czas wyszukiwania w QuadTree również rośnie z większymi danymi, ale w większości przypadków jest on porównywalny z czasami wykonywania KDTree, czasami jest nieznacznie szybszy.

Wnioski:

- QuadTree dla większości testów jest nieznacznie szybsze, chociaż różnice nie są duże.

Wykorzystane źródła

- https://www.researchgate.net/figure/mplementation-of-quadtrees-within-Google-Maps-Note-the-yellow-quadrant-on-each-image_fig3_252774444
- https://en.wikipedia.org/wiki/Quadtree
- https://mkramarczyk.zut.edu.pl/?cat=M&l=QUAD
- https://www.geeksforgeeks.org/quad-tree/
- https://www.youtube.com/watch?v=UQ-1sBMV0v4
- https://www.youtube.com/watch?v=OJxEcs0w_kE
- Mark de Berg "Geometria Obliczeniowa - Algorytmy i Zastosowania"
- https://en.wikipedia.org/wiki/K-d_tree
- https://www.youtube.com/watch?v=Glp7THUpGow&pp=ygUGa2R0cmVI
- https://www.baeldung.com/cs/k-d-trees
- https://medium.com/@isurangawarnasooriya/exploring-kd-trees-a-comprehensive-guide-to-implementation-and-applications-in-python-3385fd56a246

Thank You

Dziękujemy za czas poświęcony na naszą prezentacje