Dokumentacja

Michał Dobranowski Wiktor Perczak

31 grudnia 2023

Spis treści

1 Część techniczna

1.1 Wymagania

Do używania programu potrzebny jest język `Python` w wersji conajmniej `3.10`. Ponadto wykorzystywane są również moduły:

- `numpy 1.26.2`
- `bitalg` napisany przez koło naukowe BIT do wizualizacji danych: `https://github.com/aghbit/Algorytmy-Geometryczne`

1.2 Moduł `geometry`

1.2.1 `Rectangle`

Klasa Rectangle implementuje prostokąt, oraz wiele operacji, które można na nim wykonać:

- `def __eq__(self, other: Rectangle) -> bool` – metoda sprawdzająca, czy dwa prostokąty są sobie równe (czy współrzędne ich wierzchołków są identyczne)
- `def __and__(self, other: Rectangle) -> Rectangle | None` – metoda zwracająca część wspólną dwóch prostokątów, lub `None`, jeśli ona nie istnieje
- `def __contains__(self, item: Rectangle | Point) -> bool` – metoda sprawdzająca czy jeden prostokąt w pełni zawiera się w drugim

1.3 Moduł `tree`

Zawiera klasę abstrakcyjną `Tree`, po której dziedziczą `Quadtree` oraz `KdTree`. Posiada dwie metody:

- `def __init__(self, points: list[Point])` – konstruktor klasy
- `def find(self, rectangle: Rectangle) -> list[Point]` – zwraca listę punktów, które znajdują się w zadanym prostokącie

1.4 Moduł `quadtree`

1.4.1 `_Node`

Każdy obiekt `_Node` trzyma informację o:

- `self.rectangle` – prostokącie, za który odpowiada dany wierzchołek drzewa
- `self.min_x, self.max_x, self.min_y, self.max_y` – ekstremach prostokąta
- `self.mid_x, self.min_y` – wartości w środku prostokąta
- `self.children` – dzieciach danego wierzchołka
- `self.leaf_node` – współrzędnych punktu, jeśli wierzchołek jest liściem

1.4.2 Quadtree

Klasa `Quadtree` umożliwia budowanie drzewa czwórkowego oraz odpowiadanie na zapytania o punkty wewnątrz danego prostokąta. Posiada następujące metody:

- `def __init__(self, points: list[Point])` – tworzy pierwszy (największy) prostokąt, który zostaje zapisany w korzeniu drzewa (`self.root`). Nastepnie wywołuje z korzenia metodę `__construct_subtree`, która konstruuje drzewo czwórkowe.

- `__construct_subtree(self, node: _Node, points: list[Point])` – konstruuje drzewo czwórkowe, dzieląc kolejne prostokąty na cztery ćwiartki oraz rodzielając odpowiednio punkty na cztery zbiory. Jeżeli dojdzie do prostokąta, wewnątrz którego jest tylko jeden punkt, to obecnie rozważany wierzchołek (`_Node`) staje się liściem drzewa.

  Złożoność: $\mathcal{O}(hn)$, gdzie h to wysokość drzewa czwórkowego. Dzięki standardowi liczb zmiennoprzecinkowych jest ograniczone przez $\Omega(\log n)$ (a jeśli punkty w P są rozłożone równomiernie, to $\Theta(\log n)$).

- `def __find(self, node: _Node, rectangle: Rectangle, res: list[Point])` – metoda, która służy do znajdywania punktów wewnątrz zadanego prostokąta. Rekurencyjnie znajduje prostokąty, które w pełni zawierają się w prostokącie z zapytania i dodaje liście poniżej do odpowiedzi.

- `def find(self, rectangle: Rectangle) -> list[Point]` – wywołuje metodę `__find()` i zwraca listę, punktów które mieszczą się w zadanym prostokącie. Złożoność: $\mathcal{O}(hn)$.

1.5 Moduł `kd_tree`

1.5.1 `_Node`

Każdy obiekt `_Node` trzyma informację o:

- `self.left` – lewym dziecku wierzchołka
- `self.right` – prawym dziecku wierzchołka
- `self.rectangle` – prostokącie, który powstał przez ograniczenia przodków
- `self.leafs` – liście wszystkich liści, którę są potomkami danego wierzchołka
- `self.leaf_point` – współrzędnych punktu, jeśli dany wierzchołek jest liściem

1.5.2 `KdTree`

Klasa `KdTree` umożliwia budowanie drzewa oraz odpowiadanie na zapytania 2D, gdyż taki był temat zadania. Kod można jednak bardzo łatwo przekształcić, tak aby umożliwiał operacje na dowolnej liczbie wymiarów. Klasa `KdTree` posiada następujące metody:

- `def __init__(self, points: list[Point])` – konstruktor klasy, do którego przekazuje się listę punktów, a następnie konstruowane jest kd-drzewo. Tworzony jest też parametr `self.root`, dzięki któremu ma się dostęp do całego drzewa.
- `def build_tree(self, points: list[Point], depth: int, rectangle: Rectangle) -> _Node` – metoda, która rekurencyjnie buduje kd-drzewo. W każdej instancji dzieli dany zbiór punktów na dwa zbiory względem współrzędnej podziału, którą jest mediana (wyliczana algorytmem `quick_select`). Następnie wywołuje się rekurencyjnie z dzieci wierzchołka, który jest obecnie rozważany. W liściach zapisuje punkty z zadanego zbioru. Złożoność: $\mathcal{O}(n \log n)$.
- `def __find(self, node: _Node, rectangle: Rectangle, res: list[Point])` – metoda, która służy do znajdywania punktów wewnątrz zadanego prostokąta. Jeśli prostokąt z danego wierzchołka w pełni zawiera się, w prostokącie z zapytania to dodaje do odpowiedzi wszystkie liście poniżej. Jeśli zawiera się tylko częściowo, wywołuje się rekurencyjnie z jego dzieci.
- `def find(self, rectangle: Rectangle) -> list[Point]` – wywołuje metodę `__find()` i zwraca listę, punktów które mieszczą się w zadanym prostokącie. Złożoność: $\mathcal{O}(\sqrt{n} + k)$, gdzie k to liczba znalezionych punktów.

1.6 Moduł `quick_select`

Służy do zwracania mediany nieposortowanego zbioru punktów w czasie $\mathcal{O}(n)$. Działanie jest identyczne do sortowania przez scalanie, dzieli zbiór na dwa zbiory względem zadanej wartości, aż odnajdzie medianę. Posiada trzy funkcje:

- `def quick_select(points: list[Point], l: int, r: int, k: int, depth: int) -> Point` – zwraca medianę zbioru punktów, względem danego wymiaru (`depth: int`)
- `def partition(points: list[Point], l: int, r: int, depth: int) -> int` – porządkuje punkty na dwa zbiory: mniejsze od punktu porządkującego, lub większe od niego i zwraca indeks pierwszego punktu ze zbioru punktów większych od punktu porzadkującego
- `def rand_partition(points: list[Point], l: int, r: int, depth: int) -> int` – losowo wybiera punkt porządkujący, a następnie wywołuje funkcję `partition()`

2 Część użytkownika

W tej sekcji zaprezentowane są przykładowe użycia klas `Quadtree` i `KdTree`: skontruowanie drzewa oraz znalezienie punktów, leżących wewnątrz zadanego prostokąta.

2.1 Użycie Quadtree

```python
from quadtree import Quadtree
from geometry import Rectangle

points = [(0, 0), (1.5, 1), (2, 3), (2, 0), (0.5, 1.5)]
rectangle = Rectangle(0, 1.5, 1, 3)

tree = Quadtree(points)
print(tree.find(rectangle))
```

Odpowiedź dla przykładu: [(0.5, 1.5), (1.5, 1)].

2.2 Użycie KdTree

```python
from kd_tree import KdTree
from geometry import Rectangle

points = [(0, 0), (1.5, 1), (2, 3), (2, 0), (0.5, 1.5)]
rectangle = Rectangle(0, 1.5, 1, 3)

tree = KdTree(points)
print(tree.find(rectangle))
```

3 Wizualizacja

Do wizualizacji wykorzystano narzędzie koła naukowego BIT (instrukcja jak je pobrać: `https://github.com/aghbit/Algorytmy-Geometryczne`). Napisano dwie klasy `QuadtreeVis` oraz `KdTreeVis`, które umożliwiają:

- podział przestrzeni na prostokąty
- zobrazowanie zbioru znalezionych punktów dla danego zapytania
- wygenerowanie pliku GIF, który wizualizuje kolejne kroki algorytmu

3.1 Wizualizacja `QuadtreeVis`

Klasa `QuadtreeVis` dziedziczy po `Quadtree` dodatkowo dodając elementy wizualizacji.

Przykładowy kod na zwizualizowanie procesu tworzenia quadtree (na losowo wygenerowanym zbiorze):

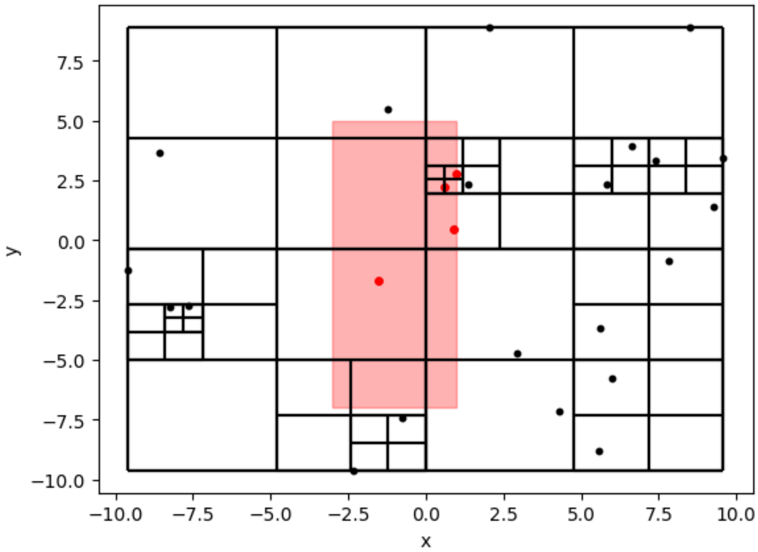
```
from quadtree_visualization import QuadtreeVis
from geometry import Rectangle
from numpy.random import seed, uniform

seed(0)

def generate_uniform_points(left, right, n):
    return list(zip(uniform(left, right, size=n), uniform(left,
    ↪  right, size=n)))


points = generate_uniform_points(-10, 10, 25)
rectangle = Rectangle(-3, 1, -7, 5)
quadtree = QuadtreeVis(points)
quadtree.add_grid()
quadtree.find(rectangle)
quadtree.vis.show()
```

Wykres wygenerowany przez ten program:



Aby wygenerować plik GIF do wyświetlania kolejnych kroków algorytmu, należy zamiast: `quadtree.vis.show()`, użyć `quadtree.vis.show_gif()`.

3.2 Wizualizacja KdTreeVis

Klasa KdTreeVis działa identycznie jak klasa KdTree, ale dodatkowo dodaje elemnty wizualizacji, zarówno podczas konstruowania drzewa jak i odpowiadania na zapytanie.

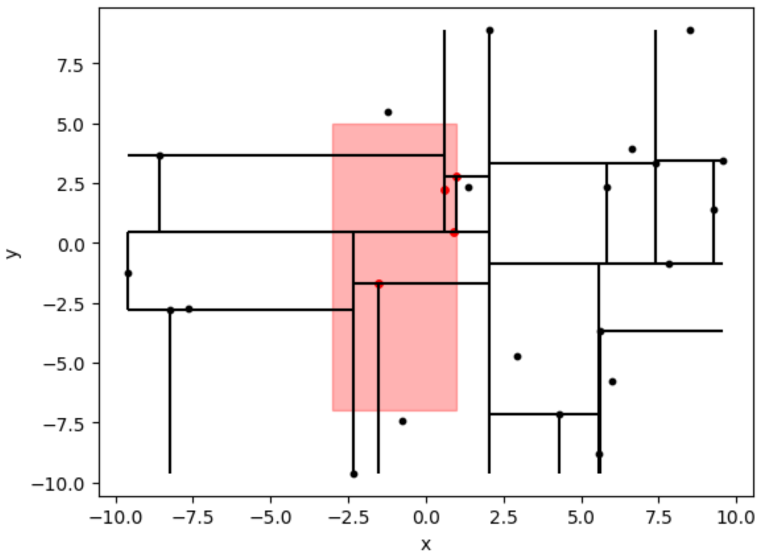
Przykładowe użycie klasy KdTreeVis dla losowo wygenerowanego zbioru punktów:

```
from kd_tree_visualization import KdTreeVis
from geometry import Rectangle
from numpy.random import seed, uniform

seed(0)

def generate_uniform_points(left, right, n):
    return list(zip(uniform(left, right, size=n), uniform(left,
    ↪  right, size=n)))


points = generate_uniform_points(-10, 10, 25)
rectangle = Rectangle(-3, 1, -7, 5)
kd_tree = KdTreeVis(points)
kd_tree.find(rectangle)
kd_tree.vis.show()
```



Aby wygenerować plik GIF do wyświetlania kolejnych kroków algorytmu, należy zamiast: kd_tree.vis.show(), użyć kd_tree.vis.show_gif().

4 Bibliografia

1. dr inż. Barbara Głut Wykład – wyszukiwanie geometryczne.
2. Prof. Dr. Kevin Buchin wykłady: Link do prezentacji o quadtree
3. Clifford Stein, Charles E. Leiserson, Ron Rivest i Thomas H. Cormen: ”Wprowadzenie do algorytmów”