Dokumentacja techniczna projektu:

Dzielenie i przeszukiwanie płaszczyzny za pomocą KD-Tree oraz Quad-Tree

Wojciech Łoboda, Krzysztof Pęczek

Spis treści

Wymagania techniczne

Środowisko oraz narzędzia:
- System operacyjny: Windows 10 x64
- Procesor: Intel Core i7-7700HQ 2.80GHz
- Pamięć RAM: 8 GB
- Wykorzystany język: python v3.10.2
- Wykorzystane biblioteki:

Drugie środowisko:
- System operacyjny: Fedora Linux 36
- Procesor: Intel(R) Core(TM) i5-8600 CPU @ 3.10GHz
- Pamięć RAM: 8GB
- Język oraz biblioteki takie same jak powyżej

Wykorzystane klasy

Point

moduł: utils.geometry.py

Klasa **Point** (w module utils.geometry) odpowiada za reprezentację punktu. Daje możliwości reprezentacji punktu jako string oraz rzutowania obiektu klasy **Point** na obiekt typu **Tuple[float, float]**. Metody i pola klasy **Point:**

__eq__(self, other: Point) -> bool

Zapewnia możliwość porównywania punktów które są dokładnie takie same.

__str__(self) -> str

Metoda używana do reprezentacji punktu jako string

follows(self, other: Point) -> bool

Metoda przyjmuje punkt i zwraca prawdę jeśli x oraz y punktu **self** są większe niż x i y punktu **other**.

precedes(self, other: Point) -> bool

Metoda podobna do follows: zwraca prawdę jeżeli współrzędne **self** są mniejsze niż współrzędne **other**

__iter__(self)

Zwraca obiekt iterowalny. Dzięki temu możliwe jest rzutowanie na typ **Tuple[float, float]**

__getitem__(self, key: int)

Zależenie od key zwraca współrzędną.
Dzięki tej metodzie można używać operatora **[]**

Rect

moduł: utils.geometry.py

Klasa **Rect** (w module utils.geometry) wreprezentuje prostokąt. Daje wiele metod pomocniczych pozwalających na zwięzłe zapisywanie niektórych dłuższych funkcji w algorytmach. Metody i pola klasy **Rect**:

Rect(lower_left: Point, upper_right: Point)
: Konstruktor tworzący prostokąt. Przyjmuje lewy-dolny oraz prawy-górny róg prostokąta.

__eq__(self, other: Rect) -> bool
: Metoda pozwalająca na porównywanie ze sobą dokładnie równych prostokątów.

__str__(self)
: Metoda pozwalająca na reprezentację prostokąta jako string.

intersects(self, other: Rect) -> bool
: Metoda zwracająca prawdę jeśli dwa prostokąty się przecinają.

contains_point(self, point: Point) -> bool
: Metoda zwracająca prawdę jeśli prostokąt zawiera dany punkt

contains_rectangle(self, rect: Rect) -> bool
: Metoda zwracająca prawdę jeśli prostokąt **rect** mieści się w prostokącie **self**

divide_vertically(self) -> Tuple[Rect, Rect]
: Metoda dzieląca prostokąt na dwa prostokąty (lewy i prawy) o równych rozmiarach.

divide_horizontally(self) -> Tuple[Rect, Rect]
: Metoda dzieląca prostokąt na dwa prostokąty (górny i dolny) o równych rozmiarach.

Visualiser

moduł: utils.visualiser.py

Klasa posiadająca statyczne metody potrzebne do wyświetlania wizualizacji algorytmów. Metody klasy **Visualiser**:

visualise_points(points: List[Point])

Otwiera wizualizację punktów na płaszczyźnie

visualise_build(points: List[Point], tree)

Otwiera wizualizację budowy określonego algorytmu. Argument points przyjmuje listę punktów, na których ma być zbudowana struktura podana w argumencie **tree**.

visualise_result(points: List[Point], rect: Rect, tree)

Otwiera wizualizację wykonania algorytmu wyszukiwania wszystkich punktów w obszarze określonym w argumencie rect. Do argumentu tree przekazujemy klasę struktury, która ma być wizualizowana.

KDtree

moduł: kdtree.py

Klasa implementująca strukturę kdtree.

Konstruktor:

def __init__(self, P:List[Point]) - Konstruktor struktury przyjmuje jako argument listę obiektów typu Point (P) i inicjuje strukturę kdtree.

Publiczne metody:

def querry_range(self, rect: Rect) -> List[Point] - Metoda przyjmuje jako argument obiekt typu Rect reprezentujący prostokąt (rect) i zwracająca listę punktów należących do struktury które znajdują się w prostokącie.

def visualize_build(points : List[Point]) -> List[Scene] - Statyczna metoda służąca do wizualizowania tworzenia struktury kdtree. Przyjmuje jako argument listę punktów (points) i zwraca listę obiektów typy Scene.

def visualize_querry(points: List[Point], rect: Rect) -> List[Scene] - Statyczna metoda służąca do wizualizacji pojedynczego zapytania o punkty należące do prostokąta. Jako argumenty przyjmuje listę obiektów Point (points) oraz prostokąt zadany jako obiekt Rect (rect), zwraca listę obiektów typu Scene

QuadTree

moduł: quad_tree.py

Klasa implementująca strukturę quad tree. Metody i pola klasy **QuadTree**:

__init__(self, bounding_box: Rect)

Konstruktor tworzący strukturę. Bounding box określa prostokąt zawierający punkty dla tego drzewa

insert(self, point: Point)

Dodaje punkt do drzewa lub rekurencyjnie dodaje punkt do jego poddrzew

querry_range(self, rect: Rect)

Zwraca listę punktów znajdujących się w obszarze rect

_visualise_tree(self, color: str) -> LinesCollection

Prywatna metoda zwracająca linie podziału wizualizujące drzewo jako LinesCollection.

visualise_build(points: List[Point]) -> List[Scene]

Zwraca listę scen, które wizualizują kolejne kroki budowy drzewa z podanych punktów.

_visualise_insert(self, points: List[Point], i: int, scenes: List[Scene], lines_vis: LinesCollection)

Metoda prywatna dodająca do obecnych kroków wizualizacji kroki dodania i-tego punktu do drzewa.

visualise_querry(points: List[Point), rect: Rect) -> List[Scene]

Metoda zwracająca listę scen wizualizujących kolejne kroki algorytmu znajdowania wszystkich punktów w zadanym przez argument rect obszarze.

_visualise_queerry(self, rect, scenes: List[Scene], solution: List[Point], all_points: List[Point], line_vis: LinesCollection) -> List[Point]

Prywatna metoda rekurencyjna zwracająca poprawny wynik algorytmu ale przy okazji dodająca do argumentu scenes kolejne kroki algorytmu.

FileHandler

Moduł: utils/files.py

Klasa obsługująca odczyt i zapis zbiorów punktów do pliku.

Publiczne metody:

def save_points_to_file(points : List[Point], name : str) - Statyczna metoda która zapisuje listę punktów do pliku. Jako argumenty przyjmuje listę obiektów typu Point (points) oraz nazwę pliku. (name).

def get_saved_points(name : str) -> List[Point] - Statyczna metoda która odczytuje listę punktów zapisanych w pliku o nazwie (name) i zwracająca listę obiektów typu Point. Jako argument metoda przyjmuje nazwę pliku.

Generator

Moduł: utils/generator.py

Publiczne metody:

normal_distribution(bounds: Rect, total: int)
: Statyczna metoda zwraca punkty ulokowane na płaszczyźnie zgodnie z rozkładem normalnym. Przyjmuje prostokąt **rect**, który określa na jakim maksymalnie obszarze generowane są punkty oraz argument **total**, który określa ile punktów ma być wygenerowanych.

on_rectangle(bounds: Rect, total: int)
: Statyczna metoda zwracająca punkty na brzegu prostokąta. Przyjmuje prostokąt **rect**, który określa na jakim maksymalnie obszarze generowane są punkty oraz argument **total**, który określa ile punktów ma być wygenerowanych.

in_rectangle(bounds: Rect, total: int)
: Statyczna metoda zwracająca punkty z rozkładem jednostajnym na określonym obszarze. Przyjmuje prostokąt **rect**, który określa na jakim maksymalnie obszarze generowane są punkty oraz argument **total**, który określa ile punktów ma być wygenerowanych.

rectangle_outliers(cluster_bounds: Rect, total_clustered: int, total_outliers=10)
: Statyczna metoda tworząca jedno gęste skupisko **total_clustered** punktów w obszarze **cluster_bounds** oraz **total_outliers** puntków poza tym obszarem.

on_polyline(polyline: List[Tuple[float, float]], total)
: Statyczna metoda tworząca **total** punktów na krzywej łamanej. Argument **polyline** to lista punktów określająca kolejne punkty łamanej. Za pomocą tej metody zaimplementowana jest metoda **on_rectangle.**

def generate_grid(bounds: Rect, total: int) -> List[Point] - Statyczna metoda generująca listę punktów tworzących siatkę. Jako argumenty metoda przyjmuje obiekt typu Rect (bounds) reprezentujący prostokąt wewnątrz którego generowana będzie siatka oraz liczbę punktów do wygenerowania. Metoda zwraca listę obiektów typu Point. Siatka generowana jest za pomocą funkcji meshgrid z biblioteki numpy.

def generate_cross(bounds: Rect, total: int) -> List[Point] - Statyczna metoda generująca listę punktów na znajdujących się na krzyżu. Jako argument metoda przyjmuje obiekt typu Rect (bounds) reprezentujący prostokąt wewnątrz którego wygenerowane będą punkty oraz liczba punktów do wygenerowania (total). Punkty generowana są losowo przy pomocy funkcji uniform z biblioteki random.

def generate circle(center: Point, rad: float, total: int) -> List[Point] - Statyczna metoda generująca listę punktów znajdujących się na okręgu. Jako argument metoda przyjmuje obiekt Point (center) reprezentujący środek okręgu, promień okręgu (rad) typu float, oraz liczbę punktów do wygenerowania (total). Metoda zwraca listę obiektów typu Point. Przyjęta parametryzacja okręgu:

$C(t) = (\cos\frac{\pi}{2}t, \sin\frac{\pi}{2}t)$ gdzie t $\in$ [0, 4]

TimeTest

Moduł: utils/test.py

Publiczne metody:

def test_trees(test_name : str, path = '') - Statyczna metoda przyjmuje jako argument nazwę testu oraz opcjonalnie ścieżke do katalog, wykonująca pomiary czasu działania algorytmu i zapisuje wynik do pliku csv o nazwie test_name_nazwa_zbioru.csv w katalogu z zadanej ścieżki lub w katalogu z programem. Struktury są testowanie dla kazdego zbioru danych przy różnej ilości wygenerowanych punktów. Czas działania podany jest w sekundach.

Przykłady wykorzystania

Generowanie i zapisywanie zbiorów do pliku:

```python
from utils.generator import Generator as gen
from utils.files import FileHandler
from utils.geometry import Rect


def main():
    points = gen.normal_distribution(Rect((0.0, 0.0), (1.0, 1.0)), 1000)
    FileHandler.save_points_to_file(points, "example.json")


if __name__ == "__main__":
    main()
```

Odczytywanie zbiorów z pliku i ich wizualizacja

```python
from utils.files import FileHandler
from utils.visualizer import Visualizer


def main():
    points = FileHandler.get_saved_points("example.json")
    Visualizer.visualize_points(points)

if __name__ == "__main__":
    main()
```

Wizualizacja kroków algorytmu inicjalizacji struktur:

```python
from utils.files import FileHandler
from utils.visualizer import Visualizer
from kdtree import KDtree
from quad_tree import QuadTree


def main():
    points = FileHandler.get_saved_points("example.json")

    #for QuadTree
    Visualizer.visualize_build(points, QuadTree)

    #for KDTree
    Visualizer.visualize_build(points, KDtree)

if __name__ == "__main__":
    main()
```

Wizualizacja kroków algorytmu dla pojedynczego zapytania:

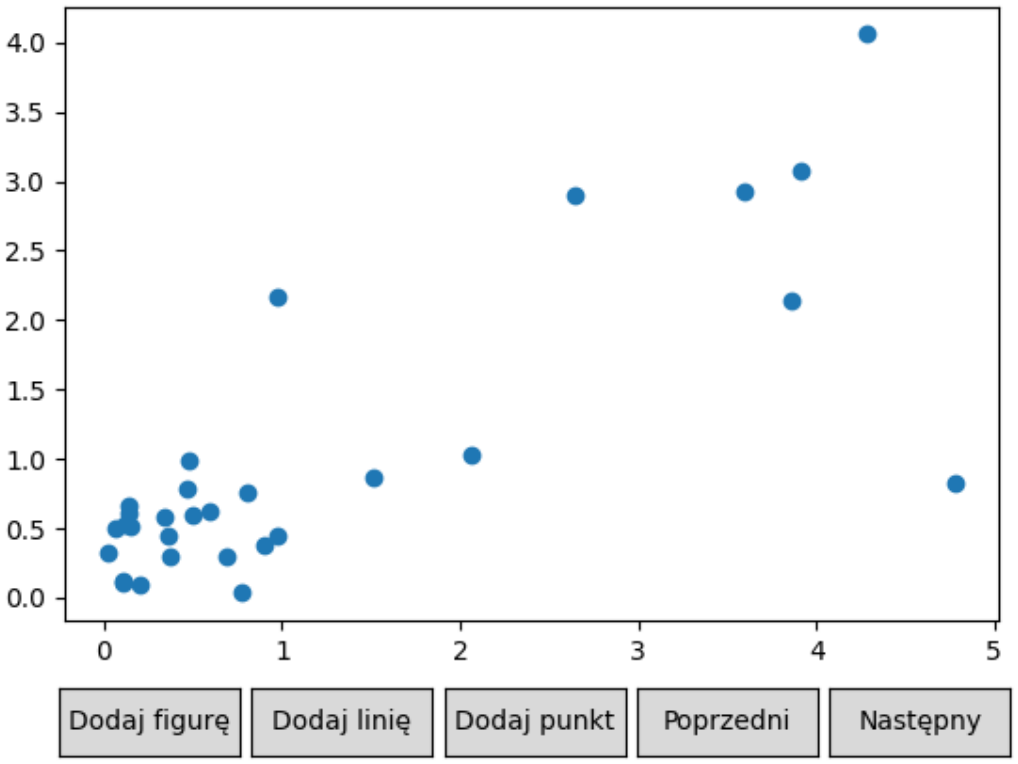
```python
from utils.files import FileHandler
from utils.visualizer import Visualizer
from utils.geometry import Rect
from kdtree import KDtree
from quad_tree import QuadTree


def main():
    points = FileHandler.get_saved_points("example.json")

    #for QuadTree
    Visualizer.visualize_result(points, Rect((0.5, 0.5), (1.0, 1.0)), QuadTree)

    #for KDTree
    Visualizer.visualize_result(points, Rect((0.5, 0.5), (1.0, 1.0)), KDtree)

if __name__ == "__main__":
    main()
```

Testowanie struktur

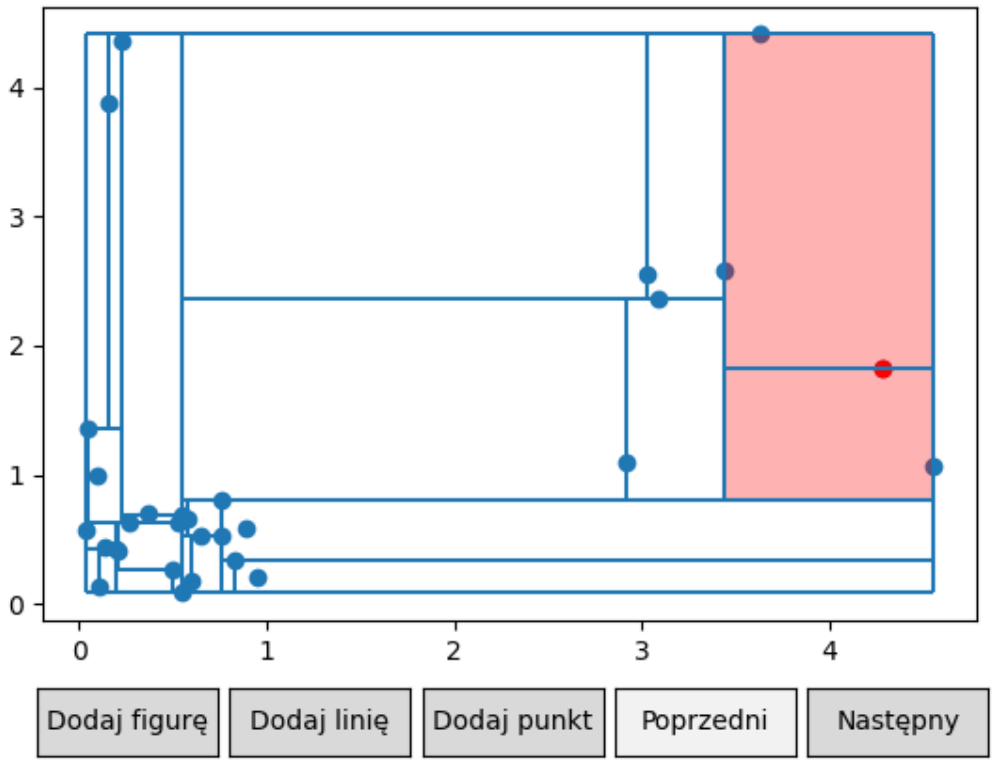
```python
from utils.test import TimeTest


def main():
    TimeTest.test_trees('test1')

if __name__ == "__main__":
    main()
```

Legenda wizualizacji - przykłady

Wizualizacja zbioru punktów

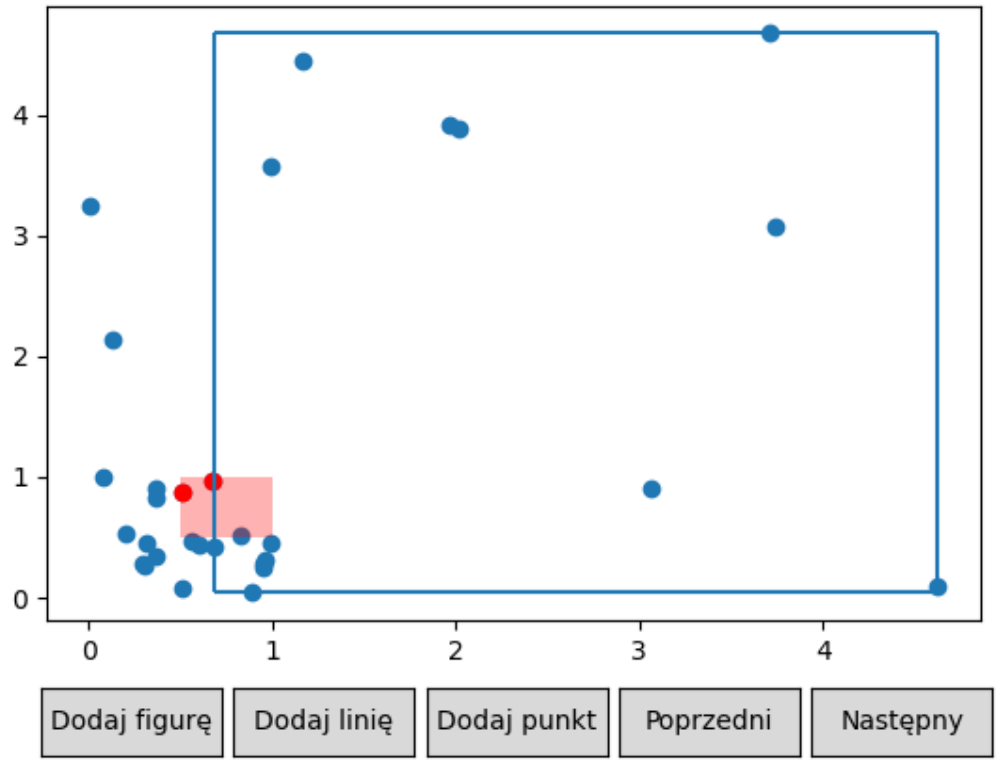


Wizualizacja kroków algorytmu tworzącego strukturę KDTree



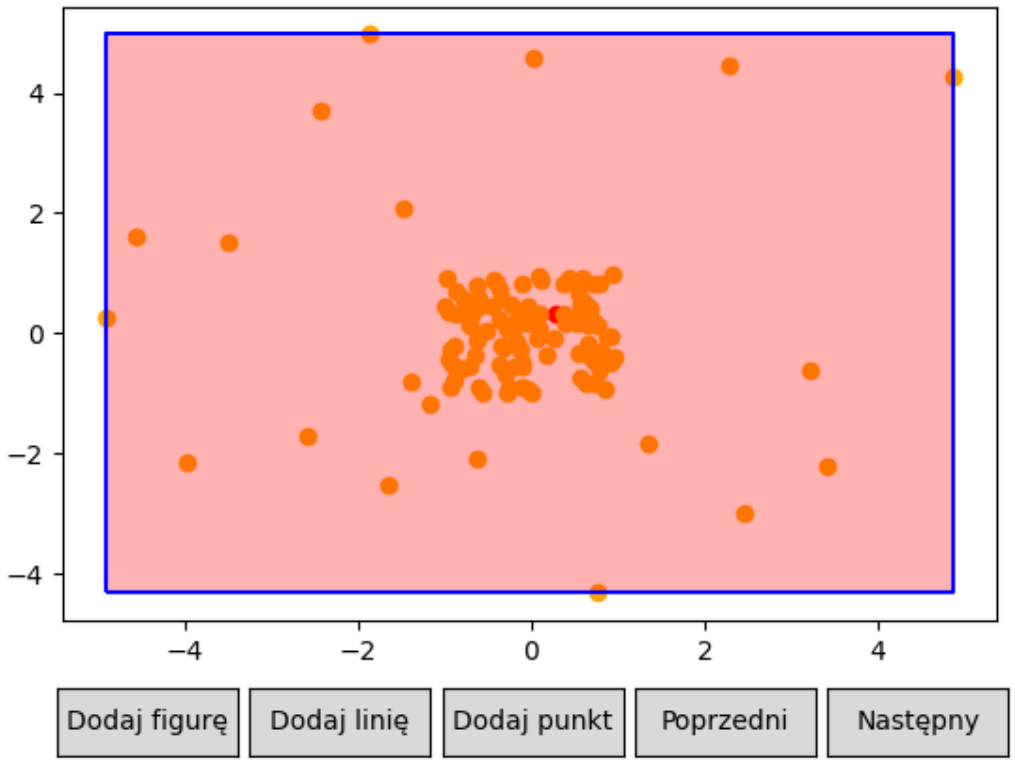
Niebieskie linie wyznaczają podział płaszczyzny w trakcie kolejnych kroków algorytmu, czerwony punkt wyznacza prostą podziału aktualnie rozpatrywanego obszaru. Obszar aktualnie rozpatrywany jest zaznaczony na czerwono.

Wizualizacja kroków algorytmu szukającego punktów w danym obszarze w KDTree

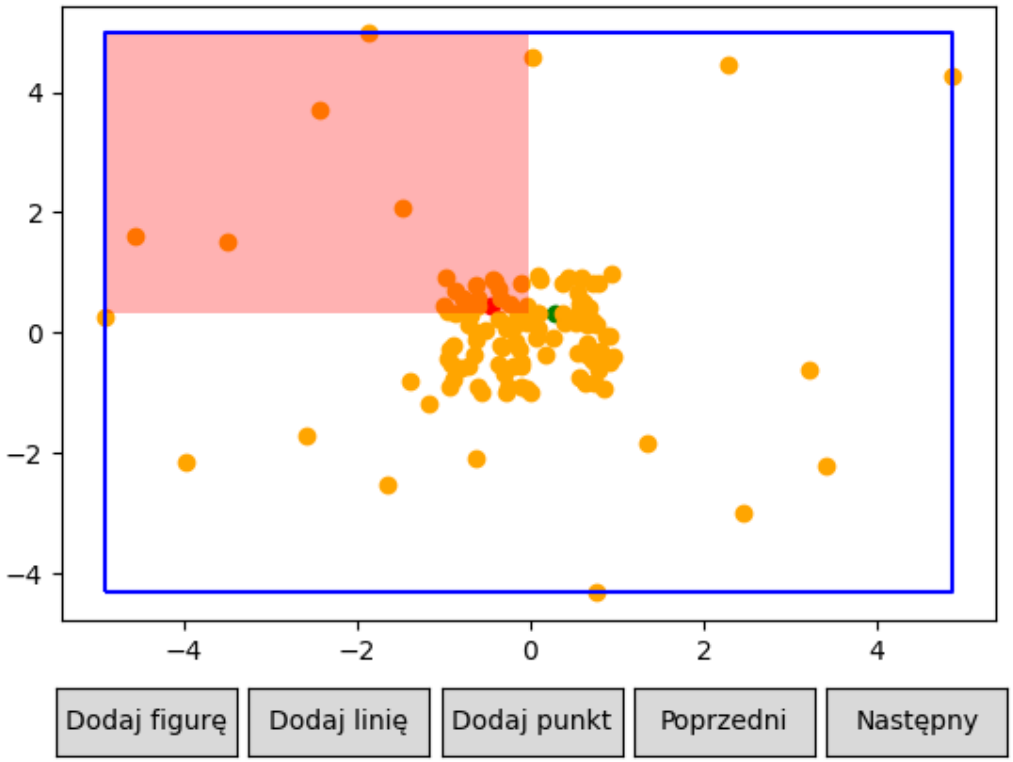


Na czerwono zaznaczony jest obszar w którym szukamy punktów. Niebieska ramka wyznacza aktualnie rozpatrywany zakres, punkty zaznaczone na czerwono to punkty znalezione przez algorytm w zadanym obszarze,

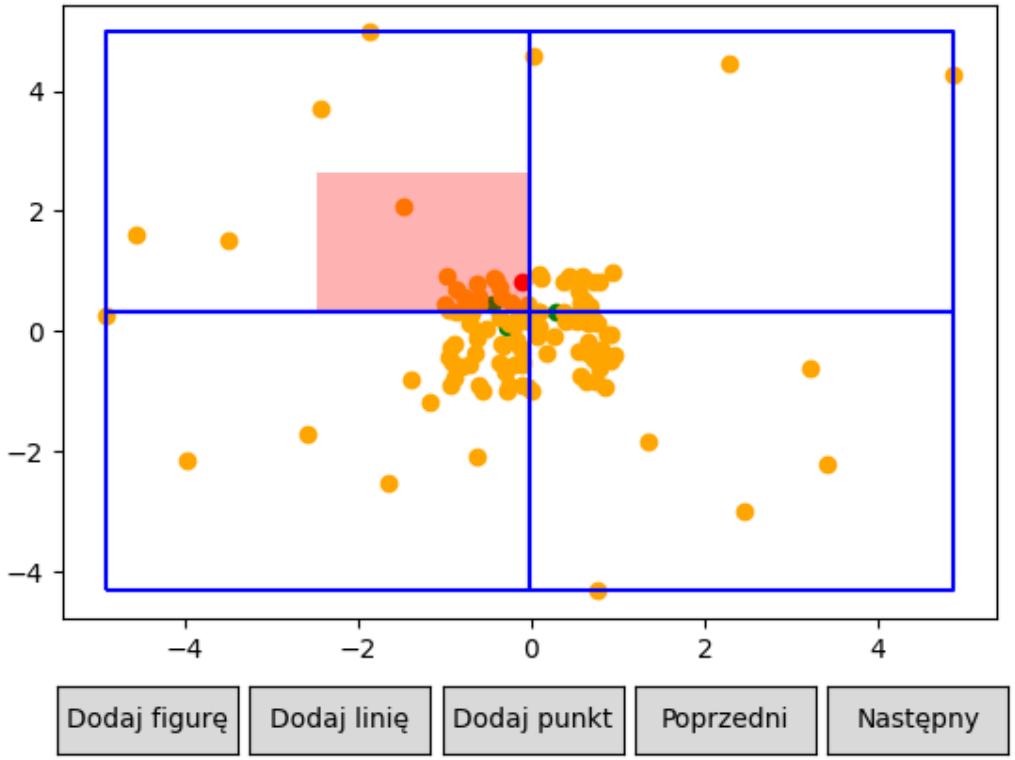
Wizualizacja tworzenia struktury QuadTree



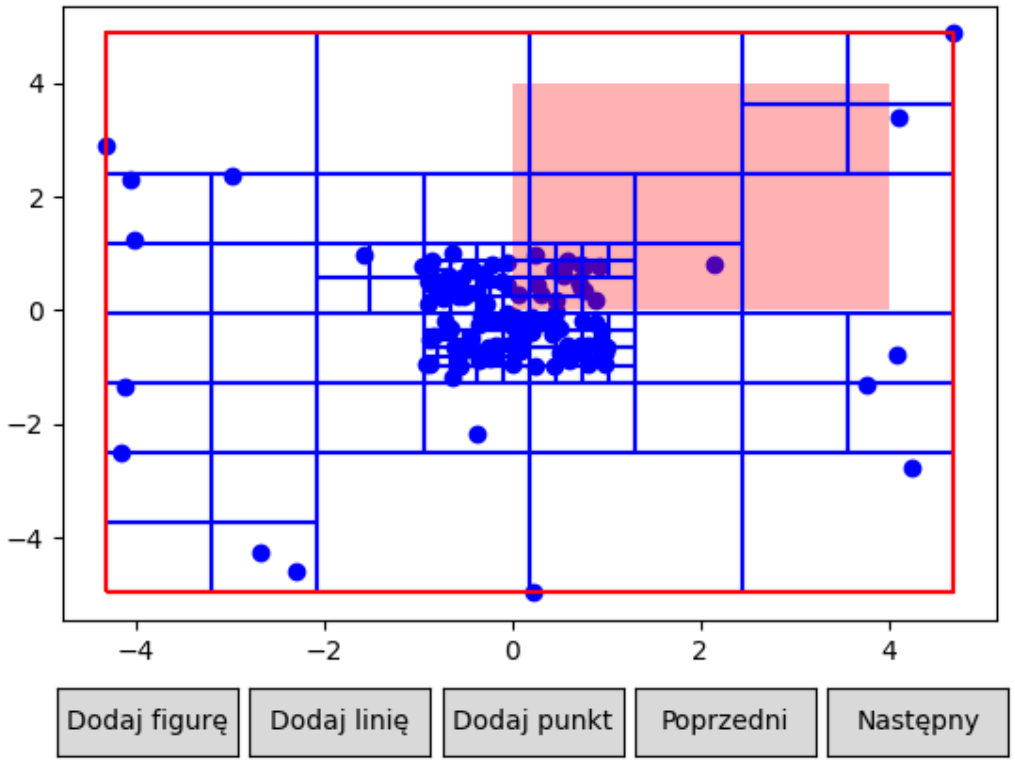
Kolorem pomarańczowym oznaczone są punkty, które jeszcze nie zostały przetworzone.
Kolorem czerwony oznaczony jest punkt, który obecnie jest przetwarzany.
Kolorem niebieskim oznaczone są linie obecnej struktury drzewa



Jasno czerwonym przezroczystym prostokątem zaznaczono obecnie rozważane poddrzewo do którego wkładany jest czerwony punkt.

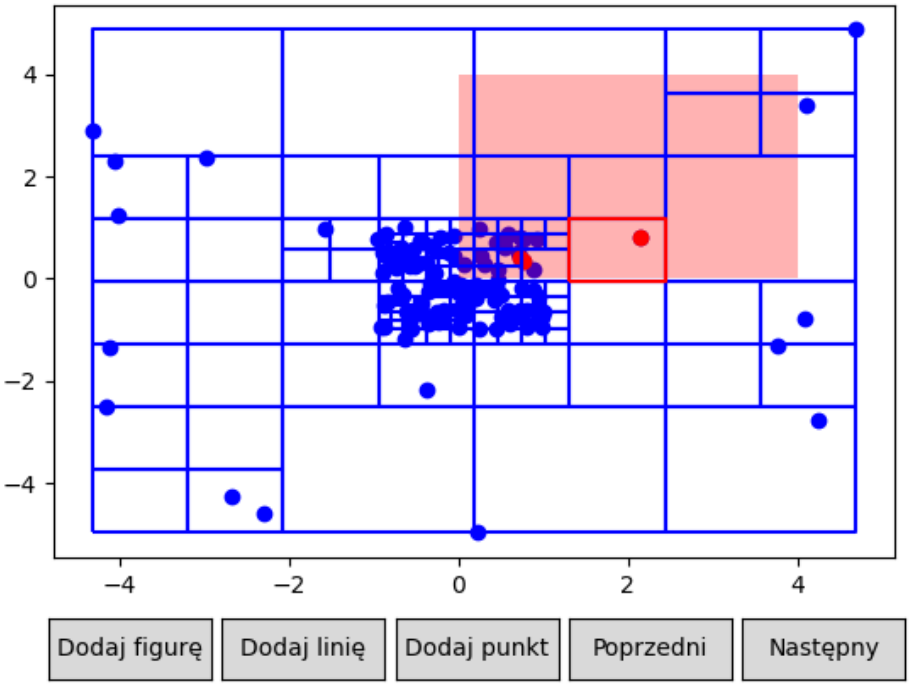


Wizualizacja kroków algorytmu wyszukującego punkty w danym obszarze za pomocą struktury Quad Tree



Punkty oznaczone kolorem niebieskim są punktami jeszcze nie przetworzonymi. Czerwona obwódka oznacza obecnie rozważane poddrzewo. Na powyższym rysunku jest to całe drzewo.

Czerwony półprzezroczysty prostokąt oznacza obszar przeszukiwany w celu rozwiązania zadania. Niebieskie linie oznaczają linie podziału struktury QuadTree



Czerwona linia przekreślająca poddrzewo oznacza, że to poddrzewo nie przecina się z przeszukiwanym obszarem i dlatego jest ignorowane w dalszym szukaniu.