AGH

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wyszukiwanie geometryczne, przeszukiwanie obszarów ortogonalnych, quadtree oraz kd-drzewa

Jakub Woś, Wojciech Suski

Grudzień 2022

Spis treści

Rozdział 1

Dokumentacja

1.1 QuadTree

Klasa reprezentująca drzewo ćwiartkowe. Jest modyfikacją koncepcji drzewa binarnego dla dwuwymiarowej przestrzeni. Znajduje się w pliku ./quad_tree/quad_tree.py. Zapewnia możliwość szybkiego zwrócenia wszystkich punktów należących do pewnego obszaru na płaszczyźnie, oraz dodania nowego punktu do struktury.

- **quad_tree(initial_points, x = 0, y = 0, width = 100, height = 100)**
  Tworzy drzewo na danym obszarze zapełniając je dostarczonymi punktami.
  - **initial_points** - zbiór punktów które mają od początku znajdować się w drzewie. Punkty muszą być krotkami(lub listami) w formacie (x, y), gdzie zarówno x jak i y to liczby.
  - **x** - pierwsza współrzędna lewego górnego rogu prostokąta, który ma być obszarem drzewa
  - **y** - druga współrzędna lewego górnego rogu prostokąta, który ma być obszarem drzewa
  - **width** - szerokość prostokąta, który ma być obszarem drzewa
  - **height** - wysokość prostokąta, który ma być obszarem drzewa.

  Może wyrzucić wyjątki:
  - **TypeError('quad_tree cannot handle points of different dimension than 2, given point point')** - jeżeli dowolny, dodawany punkt ma więcej lub mniej niż 2 wymiary, to wyrzuca ten błąd.
  - **ValueError('Given point: point is out of quad_tree bounds')** - jeżeli wrzucany punkt nie należy do obszaru wskazanego przy inicjalizacji, to wyrzuca ten błąd.
  - **ValueError('quad_tree cannot handle duplicate points')** - jeżeli wśród przekazanych punktów istnieją chociażby dwa o takich samych wartościach dla odpowiednich współrzędnych, to zostaje wyrzucony ten błąd. Ta implementacja nie jest w stanie sobie poradzić z powtórzeniami punktów.

- **find(x1, x2, y1, y2)**
  Przeszukuje drzewo w poszukiwaniu punktów z zadanego obszaru, zwraca listę punktów w formacie (x, y), gdzie zarówno x jak i y to liczby.
  - **x1** - pierwsza współrzędna lewego górnego rogu prostokąta będącego przeszukiwanym obszarem
  - **x2** - pierwsza współrzędna prawego dolnego rogu prostokąta będącego przeszukiwanym obszarem
  - **y1** - druga współrzędna lewego górnego rogu prostokąta będącego przeszukiwanym obszarem
  - **y2** - druga współrzędna prawego dolnego rogu prostokąta będącego przeszukiwanym obszarem.

  Może wyrzucić wyjątki:
  - **TypeError('quad_tree does not fully contain given area')** - jeżeli otrzymany obszar nie zawiera się całkowicie w obszarze wskazanym przy inicjalizacji, to wyrzuca ten błąd.
- **add(new_point)**
  Dodaje nowy punkt do struktury.
  - **new_point** - nowy punkt w formacie (x, y), gdzie zarówno x jak i y to liczby.

  Może wyrzucić wyjątki:
  - **TypeError('quad_tree cannot handle points of different dimension than 2, given point point')** - jeżeli dowolny, dodawany punkt ma więcej lub mniej niż 2 wymiary, to wyrzuca ten błąd.
  - **ValueError('Given point: point is out of quad_tree bounds')** - jeżeli wrzucany punkt nie należy do obszaru wskazanego przy inicjalizacji, to wyrzuca ten błąd.
  - **ValueError('quad_tree cannot handle duplicate points')** - jeżeli wśród przekazanych punktów istnieją chociażby dwa o takich samych wartościach dla odpowiednich współrzędnych, to zostaje wyrzucony ten błąd. Ta implementacja nie jest w stanie sobie poradzić z powtórzeniami punktów.

- **visualized_find(x1, x2, y1, y2)**
  Przeszukuje drzewo w poszukiwaniu punktów z zadanego obszaru dodatkowo generując wizualizacje krok po kroku, zwraca listę punktów w formacie (x, y), gdzie zarówno x jak i y to liczby.
  - **x1** - pierwsza współrzędna lewego górnego rogu prostokąta będącego przeszukiwanym obszarem
  - **x2** - pierwsza współrzędna prawego dolnego rogu prostokąta będącego przeszukiwanym obszarem
  - **y1** - druga współrzędna lewego górnego rogu prostokąta będącego przeszukiwanym obszarem
  - **y2** - druga współrzędna prawego dolnego rogu prostokąta będącego przeszukiwanym obszarem.

  Może wyrzucić wyjątki:
  - **TypeError('quad_tree does not fully contain given area')** - jeżeli otrzymany obszar nie zawiera się całkowicie w obszarze wskazanym przy inicjalizacji, to wyrzuca ten błąd.

- **visualized_add(new_point)**
  Dodaje nowy punkt do struktury dodatkowo generując wizualizacje krok po kroku.
  - **new_point** - nowy punkt w formacie (x, y), gdzie zarówno x jak i y to liczby.

  Może wyrzucić wyjątki:
  - **TypeError('quad_tree cannot handle points of different dimension than 2, given point point')** - jeżeli dowolny, dodawany punkt ma więcej lub mniej niż 2 wymiary, to wyrzuca ten błąd.
  - **ValueError('Given point: point is out of quad_tree bounds')** - jeżeli wrzucany punkt nie należy do obszaru wskazanego przy inicjalizacji, to wyrzuca ten błąd.
  - **ValueError('quad_tree cannot handle duplicate points')** - jeżeli wśród przekazanych punktów istnieją chociażby dwa o takich samych wartościach dla odpowiednich współrzędnych, to zostaje wyrzucony ten błąd. Ta implementacja nie jest w stanie sobie poradzić z powtórzeniami punktów.

1.2 KDtree

KDTree to rodzaj zrównoważonego drzewa binarnego. Implementacja znajduje się w pliku ./kd_tree/kd_tree.py

- **KDTree(points: List[Point], dimension: int = 2, vis: bool = False)**
  - **points** - zbiór punktów wejściowych, lista zawierająca punkty do pierwszej budowy drzewa
  - **dimension** - liczba wymiarów punktu, domyślna wartość to 2 wymiarowe punkty
  - **vis** - zmienna wykorzystywana do wizualizacji (można zwizualizować jedynie punkty 2-wymiarowe)

  Dokumentacji pozostałych metod takiech jak find, add czy visualized_find jest idenczyna jak dla Quadtree.

Rozdział 2

Poradnik wykorzystania

2.1 Testy

By przetestować struktury, należy zaimportować funkcje *run_and_dump_tests*() z pliku *./tests/test_code.py* zwracającą słownik z wartościami czasów dla odpowiednich testów i zbiorów. Funkcja przyjmuje 3 argumenty konfiguracyjne:

- **easy** - wartość logiczna reprezentująca wybór zbiorów testowych
- **with_repeats** - wartość logiczna reprezentująca czy testy mają zostać wykonane z powtórzeniami, których ilość określona jest w pliku *./config.py*
- **as_csv** - wartość logiczna reprezentująca czy testy mają zostać zapisane w pliku o formacie csv. Wygenerowane pliki csv dla każdego typu zbioru zapiszą się w folderze *./tests/csv*.
- **as_json** - wartość logiczna reprezentująca czy testy mają zostać zapisane w pliku o formacie .json. Wygenerowany plik json zostanie zapisany w folderze *./tests*.

Poniżej przedstawiliśmy przykładowe wywołanie:

```
from tests.test_code import run_and_dump_tests
run_and_dump_tests(easy=False, with_repeats=True, as_csv=True)
```

2.2 QuadTree i KDTree

Poniżej przedstawione zostaną przykładu z użyciem QuadTree, ale użycie obu struktur jest analogiczne - wystarczy zmienić zaimportowany plik i konstruktor struktury.

2.2.1 Tworzenie struktury

Przykładowy kod:

```
from quad_tree.quad_tree import quad_tree
init_points = [( 11 , 12 ) , ( 12 , 45 ) , (18 , 68 ) , ( 19 , 62 ) , ( 20 , 25 ) ,
               ( 23 , 17 ) , ( 23 , 68 ) , ( 24 , 51 ) , ( 25 , 98 ) , ( 28 , 73 )]
tree_area = (0, 0, 100, 100)
qt = quad_tree ( points, *tree_area )
```

2.2.2 Przeszukiwanie obszaru zadanego przez x_1, x_2, y_1, y_2

Przykładowy kod:

```
qt.find(x1, x2, y1, y2)
```

2.2.3 Dodanie nowego punktu do struktury

Przykładowy kod:

```
new_point = (3, 4)
qt.add(new_point)
```

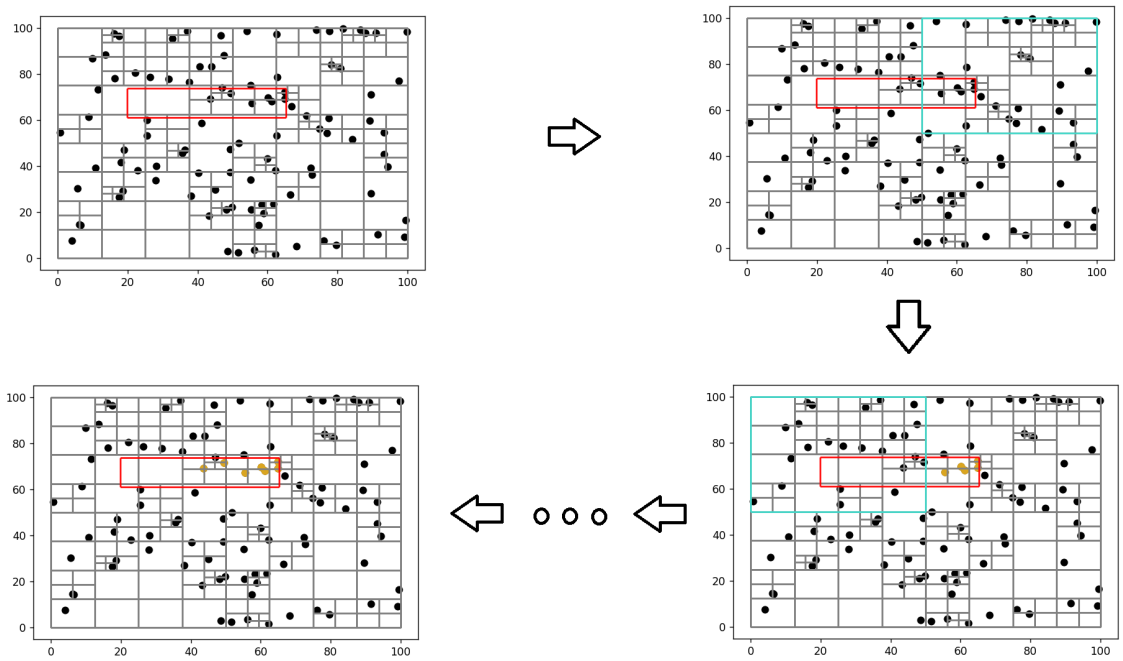
2.2.4 Przeszukiwanie obszaru zadanego przez x_1, x_2, y_1, y_2 z wizualizacją

Po wywołaniu metody visualized_find analogicznym do poniższego wyskoczy nowe okno, w którym przyciskami 'Następny' oraz 'Poprzedni' znajdującymi się w prawym dolnym rogu okna można zobaczyć kolejne kroki wykonywane przez alogrytm. Znaczenie obiektów w wizualizacji:

- **czerwony prostokąt** - wprowadzony jako argument meotody find obszar przeszukiwany
- **cyjanowy prostokąt** - aktualnie rozpatrywany obszar przez algorytm
- **pomarańczowe punkty** - znalezione punkty znajdujące się w strukturze, które zostały sklasyfikowane do podzboru punktów znajdującego się w obszarze przeszukiwanym
- **szare prostokąty** - podzielenie przestrzeni przez strukture (w tym przypadku QuadTree)
- **czarne punkty** - punkty znajdujące się w strukturze

Przykładowy kod:

```
qt.visualized_find(x1, x2, y1, y2)
```



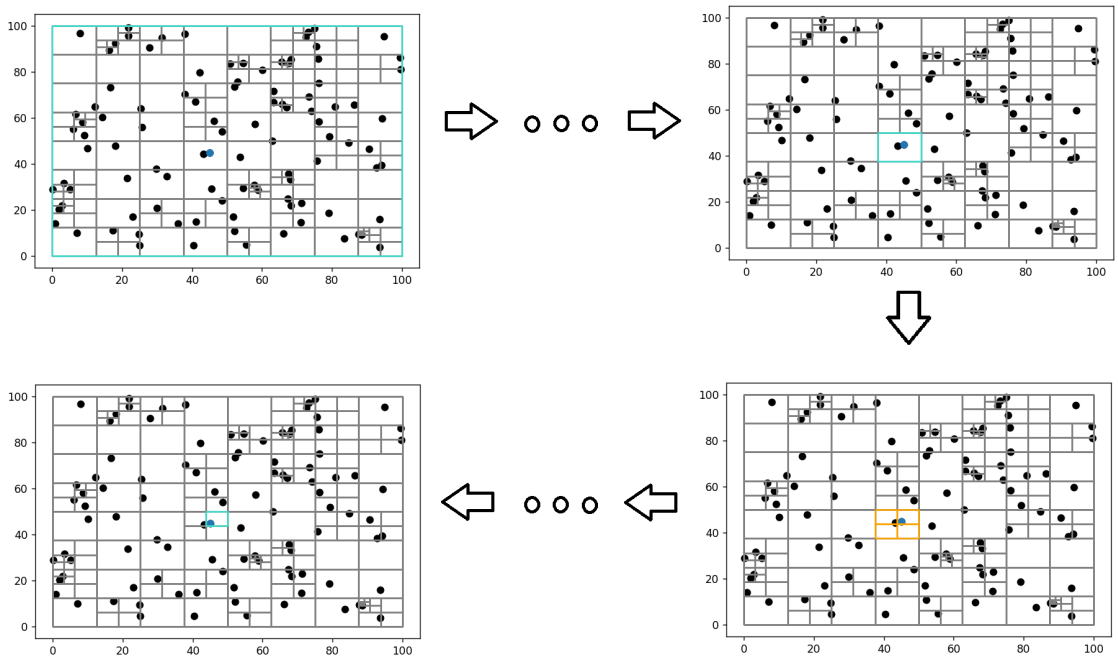
2.2.5 Dodanie nowego punktu do struktury z wizualizacją

Po wywołaniu metody visualized_add analogicznym do poniższego wyskoczy nowe okno, w którym przyciskami 'Następny' oraz 'Poprzedni' znajdującymi się w prawym dolnym rogu okna można zobaczyć kolejne kroki wykonywane przez alogrytm. Znaczenie obiektów w wizualizacji:

- **cyjanowy prostokąt** - aktualnie rozpatrywany obszar przez algorytm
- **niebieski punkt** - punkt, wprowadzony jako argument (aktualnie dodawany punkt)
- **pomarańczowe prostokąty** - prostokąty utworzone przez podzielenie obszaru ze struktry
- **szare prostokąty** - podzielenie przestrzeni przez strukture (w tym przypadku QuadTree)
- **czarne punkty** - punkty znajdujące się w strukturze

Przykładowy kod:

```
new_point = (3, 4)
qt.visualized_add(new_point)
```



2.3 KDTree

Poniżej przedstawione zostaną przykładu z użyciem KDTree

2.3.1 Tworzenie struktury

Przykładowy kod:

```
from  kd_tree.kd_tree_utils import *
from kd_tree.geo_types
init_points = [( 11 , 12 ) , ( 12 , 45 ) , (18 , 68 ) , ( 19 , 62 ) , ( 20 , 25 ) ,
               ( 23 , 17 ) , ( 23 , 68 ) , ( 24 , 51 ) , ( 25 , 98 ) , ( 28 , 73 )]
tree_area = Rectangle(0, 100, 0, 100)
kdtree = KDTree( points, dimension = 2, vis = False)
```

2.3.2 Przeszukiwanie obszaru zadanego przez x_1, x_2, y_1, y_2

Przykładowy kod:

```
kdtree.find(x1, x2, y1, y2)
```

2.3.3 Dodanie nowego punktu do struktury

Przykładowy kod:

```
new_point = (3, 4)
kdtree.add(new_point)
```

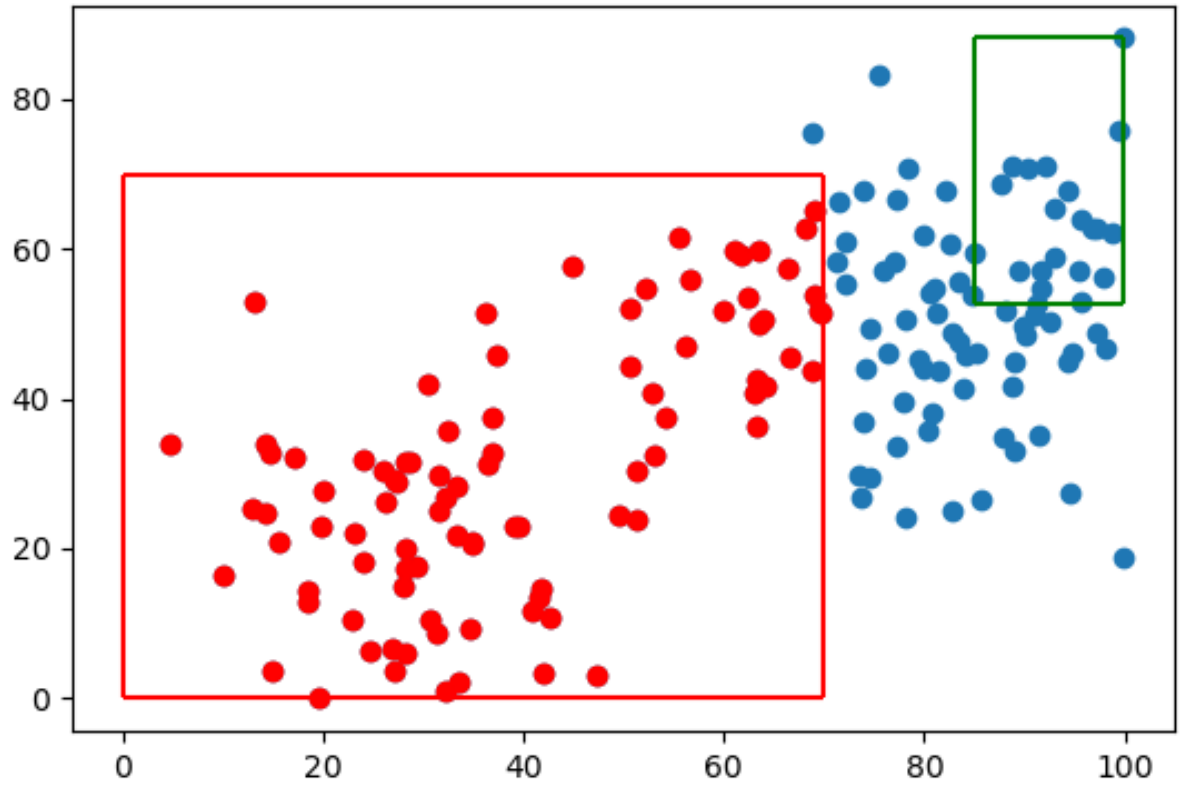
2.3.4 Przeszukiwanie obszaru zadanego przez x_1, x_2, y_1, y_2 z wizualizacją

Po wywołaniu metody visualized_find analogicznym do poniższego wyskoczy nowe okno, w którym przyciskami 'Następny' oraz 'Poprzedni' znajdującymi się w prawym dolnym rogu okna można zobaczyć kolejne kroki wykonywane przez alogrytm. Znaczenie obiektów w wizualizacji:

- **czerwony prostokąt** - wprowadzony jako argument meotody find obszar przeszukiwany
- **zielony prostokąt** - aktualnie rozpatrywany obszar przez algorytm
- **czerwone punkty** - znalezione punkty znajdujące się w strukturze, które zostały sklasyfikowane do podzboru punktów znajdującego się w obszarze przeszukiwanym
- **niebieskie punkty** - punkty znajdujące się w strukturze

Przykładowy kod:

```
kdtree.visualized_find(x1, x2, y1, y2)
```



Rozdział 3

Sprawozdanie

3.1 Informacje sprzętowe

Dane były generowane przy pomocy laptopa z procesorem Ryzen 7 5800H, z zasobem pamięci RAM w wielkości 32GB, systemem Windows 10 oraz językiem Python w wersji 3.10.6. Biblioteki którymi się wspomogliśmy w implementacji i wizualizacji algorytmów to:

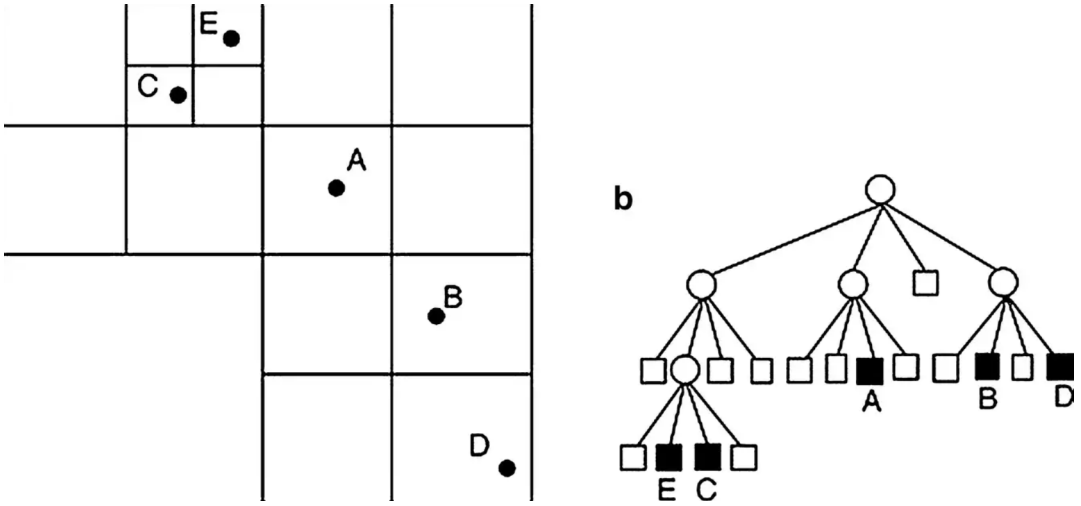
- *json* do zapisu danych
- *numpy* do działań na macierzach
- *time* do pomiaru czasu wykonania algorytmu
- *timeit* do pomiaru czasu wykonania algorytmu
- *itertools* do działań na listach wielowymiarowych
- *copy* do kopiowania struktur danych
- *random* do generowania danych losowych
- *pandas* do generowania plików csv z danymi

Wizualizacja była generowana przy pomocy udostępnionego nam w ramach zajęć narzędzia.

3.2 Wstęp teoretyczny

3.2.1 Quadtree

Jest to 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. Każdy węzeł w drzewie ma przypisany obszar, z którego przechowywuje dane. Przy dostatecznie dużej ilości punktów korzeń drzewa ma cztery dzieci każde odpowiadające jednej ćwiartce obszaru przypisanego korzeniowi. Każde z dzieci korzenia ma dzieci itd.



Quadtree ilustracja

Dodawanie danych do quadtree

Dodawanie punktu do drzewa polega na znalezieniu takiej ścieżki od korzenia by kończyła się na takim liściu w drzewie by nie przechowywał on jeszcze danych o żadnym punkcie oraz by obszar mu przydzielony zawierał dany punkt. Taki liść może nie istnieć, wtedy tworzy się go przez podzielenie innego węzła. Po znalezieniu ścieżki dodajemy informacje o danym punkcie do każdego węzła na tej ścieżce.

Przeszukiwanie quadtree

Szukanie punktów z zadanego obszaru w drzewie quadtree polega na znalezieniu jak największej ilości węzłów o jak największym przydzielonym obszarze, takim by całkowicie zawierał się on w obszarze przeszukiwanym. Po znalezieniu takich węzłów dane w nich przechowywane dodajemy do zbioru wynikowego oraz uzupełniamy go o przefiltrowane dane (tak by spełniały warunki wyszukiwania) z węzłów, których obszary są jak najmniejsze oraz przecinają się z obszarem przeszukiwanym (nie zawierając się w nim). Złożoność obliczeniowa przeszukania quadtree wynosi $O(dl)$, gdzie d - głębokość drzewa, l – liczba liści reprezentujących prostokąty częściowo zawierające się w przeszukiwanym obszarze.

Tworzenie quadtree

Tworzenie drzewa polega na dodaniu kolejno wszystkich punków, które mają znaleźć się w drzewie. Złożoność czasowa, jak i pamięciowa tworzenia quadtree zależy stricte od jego głębokości (ona zależy natomiast od rozmieszczenia punktów na płaszczyźnie) oraz ilości punktów leżących na płaszczyźnie i wynosi $O((d+1)n)$, gdzie d – głębokość drzewa, n – liczba punktów.

Podsumowanie

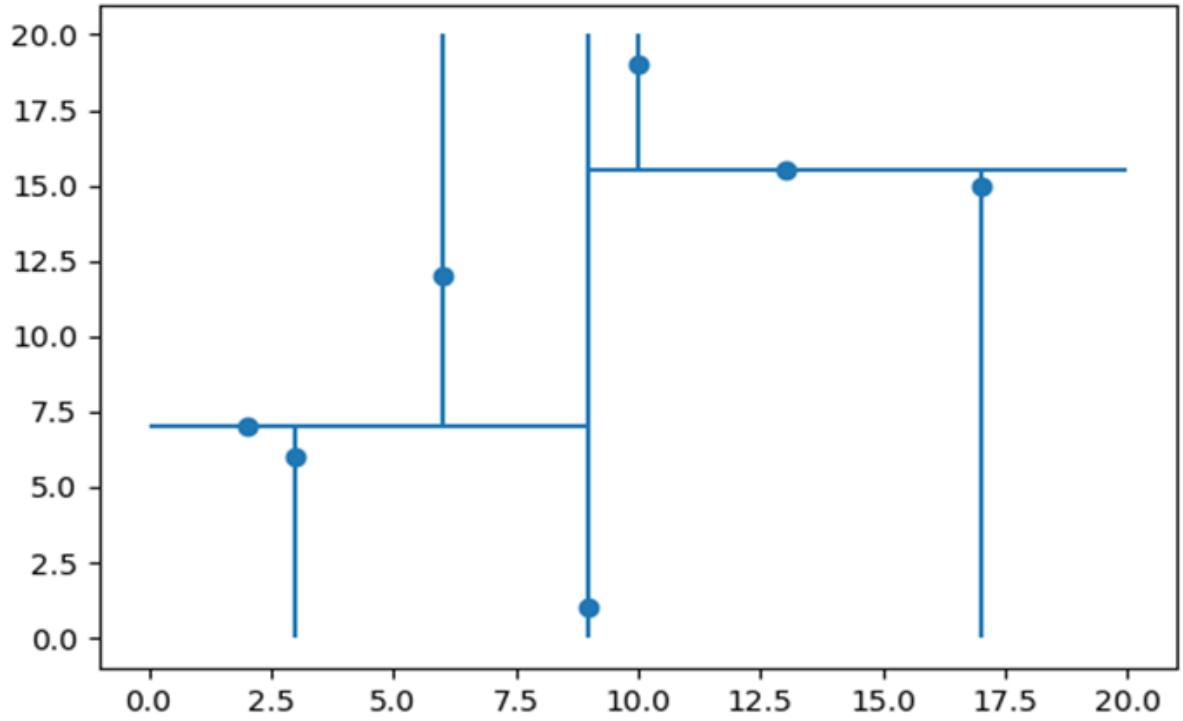
Pomimo niezbyt zadawalającej pesymistycznej złożoności ($O(n)$) działań na quadtree, jest ona nieporównywalnie szybsza od liniowego podejścia do problemu przy większości zbiorów danych. Różnica jest jednak najbardziej widoczna przy zbiorach których dane (punkty) są rozmieszczone dość równomiernie na płaszczyźnie.

3.2.2 KDtree

Struktura danych, będąca wariantem drzew binarnych, używana do podziału przestrzeni. Drzewa kd są przydatne do tworzenia struktur w niektórych zastosowaniach, takich jak wyszukiwanie najbliższych sąsiadów lub znajdowanie punktów w prostokątnych obszarach. Każdy węzeł wewnętrzny tworzy hiperpłaszczyznę podziału, podział ten odbywa się względem mediany punktów które reprezentuje dany węzeł, mediana jako punkt zostaje zapisana w tym węźle natomiast reszta jest podzielona na dwie części i odpowiednio przekierowana do lewego i prawego dziecka. Podział następuje po kolejnych wymiarach punktu i jest zależny od obecnej głębokości w drzewie.

Tworzenie KDtree

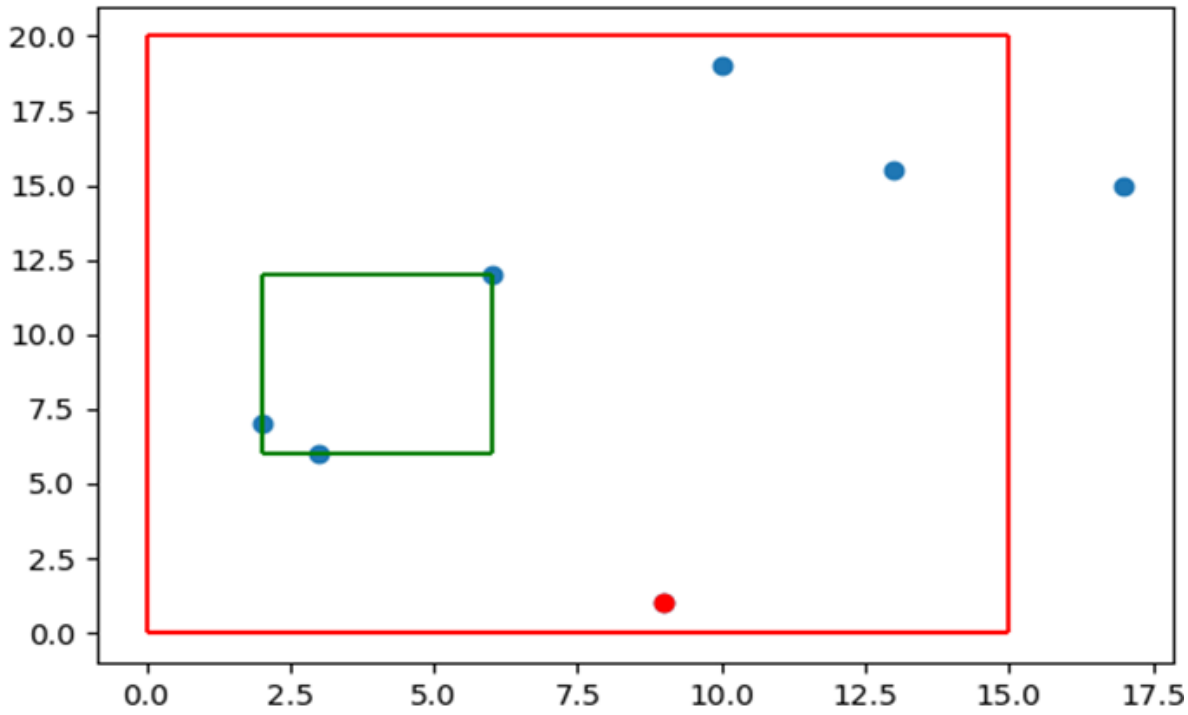
Budowa drzewa kd przebiega w następujący sposób. W każdym węźle dzielimy zbiór punktów na dwie części wzgledem mediany obecnego zbioru. Mediane zapisujemy w obecnym węźle natomiast reszta punktów jest przekazywana niżej do dzieci obecnego węzła, mniejsze punkty względem obecnego wymiaru są przekazane do lewego dziecka, większe do prawego. Wymiar z kolei wybieramy na podstawie obecnej głębokości na jakiej znajdujemy się w drzewie modulo maksymalny wymiar naszych punktów. Powtarzamy tą czynność aż w zbiorze punktów zostanie nam jeden punkt, będzie on liściem naszego drzewa. Wyszukiwanie mediany zrealizowane jest po przez użycie funkcji partition z algorytmu Quicksort, jest to dość szybkie, dzieli zbiór punktów względem mediany, mniejsze po lewej stronie większe po prawej, jak i zwraca jej indeks. Oczekiwany czas działania partition to $O(n)$, jednakże w patologicznych przypadkach może osiągnąć złożoność $O(n^2)$.



KDtree ilustracja zbudowanego drzewa

Przeszukiwanie KDtree

W każdym węźle możemy policzyć jaki region reprezentuje dany węzeł, możliwość tą daje nam przechowywanie wszystkich punktów poddrzewa w obecnym węźle. A więc chcąc znaleźć obszar pomiędzy punktami $(x_1, y_1), (x_2, y_2)$ (ozn. R) możemy natchnąć się na 3 możliwości podczas przeszukiwania kd drzewa. Jeśli region który reprezentuje dany węzeł i szukany region R nie przecinają się to wiemy, że w danym poddrzewie nie mamy czego szukać i wycofujemy się stąd. Jeśli region reprezentowany przez dany węzeł zawiera się w całości w szukanym regionie R to wszystkie punkty z tego poddrzewa powinniśmy zaliczyć jako te których szukamy. Natomiast jeśli region R i region danego węzła przecinają się to wówczas nie jesteśmy wstanie powiedzieć które punkty należą do obecnego obszaru a które nie i musimy zejść głębiej do jego dzieci, wywołując na nich tą samą funkcje.



KDtree ilustracja jednej z faz przeszukiwań

Dodawanie danych do KDtree

Dodawanie polega na znalezieniu odpowiedniego miejsca dla naszego nowego punktu, odbywa się to w podobny sposób jak przy budowie drzewa. Schodzimy w głąb drzewa po kolei porównując wartości odpowiednich punktów po odpowiednim wymiarze. Gdy skończą się węzły drzewa to znajdziemy miejsce na nasz nowy punkt.

Podsumowanie

Pomimo względnie dużego czasu budowy które i tak działa dość dobrze znajdowanie poszczególnych punktów na zadanym obszarze jest dość szybkie ze względu na to iż budowane drzewo jest drzewem binarnym, ponadto implementacja całej struktury nie jest dość skomplikowana i łatwa do modyfikacji.

3.3 Przebieg testów

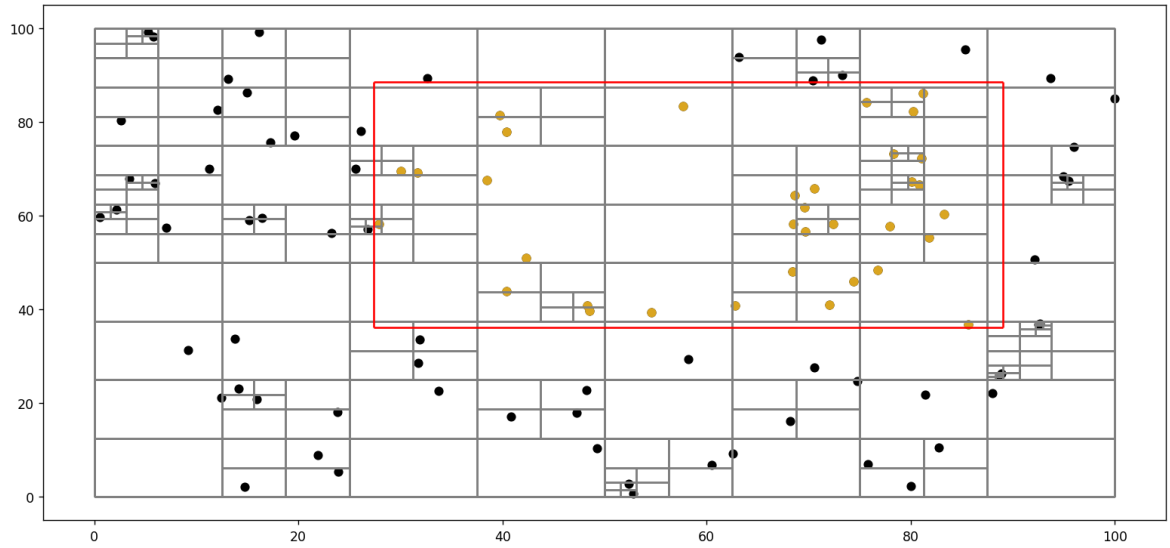
Czasy działań na strukturach umieszczone w tabelach poniżej są średnią arytmetyczną z 20 prób na 20 różnych zbiorach danego typu, każda próba z innym argumentem. Ta metoda testów umożliwia dogłębne przetestowanie struktury pod względem poprawności i porównania czasów wykonania metod każdej ze struktur. Niestety przy testowaniu metody find zauważyliśmy że przy bardzo niskim czasie jej działania (nie rzadko poniżej 0.1ms) może generować pewne nieścisłości, co widać przy testach zbiorów nr 2, 5 i 6 gdzie czasy struktury Quadtree dla 100000 punktów są nieznacznie lepsze niż dla 50000 punktów. Najprawdopodobniej jest to wynik niefortunnie losowo wygenerowanego zbioru lub obszaru przeszukiwań. Głównym aspektem testów nie było dogłębne sprawdzenie złożoności obliczeniowej działań na strukturach,a porównanie czasów pomiędzy obiema strukturami, dlatego nie odrzucamy tych wyników podczas analizy.

3.4 Utworzone zbiory danych

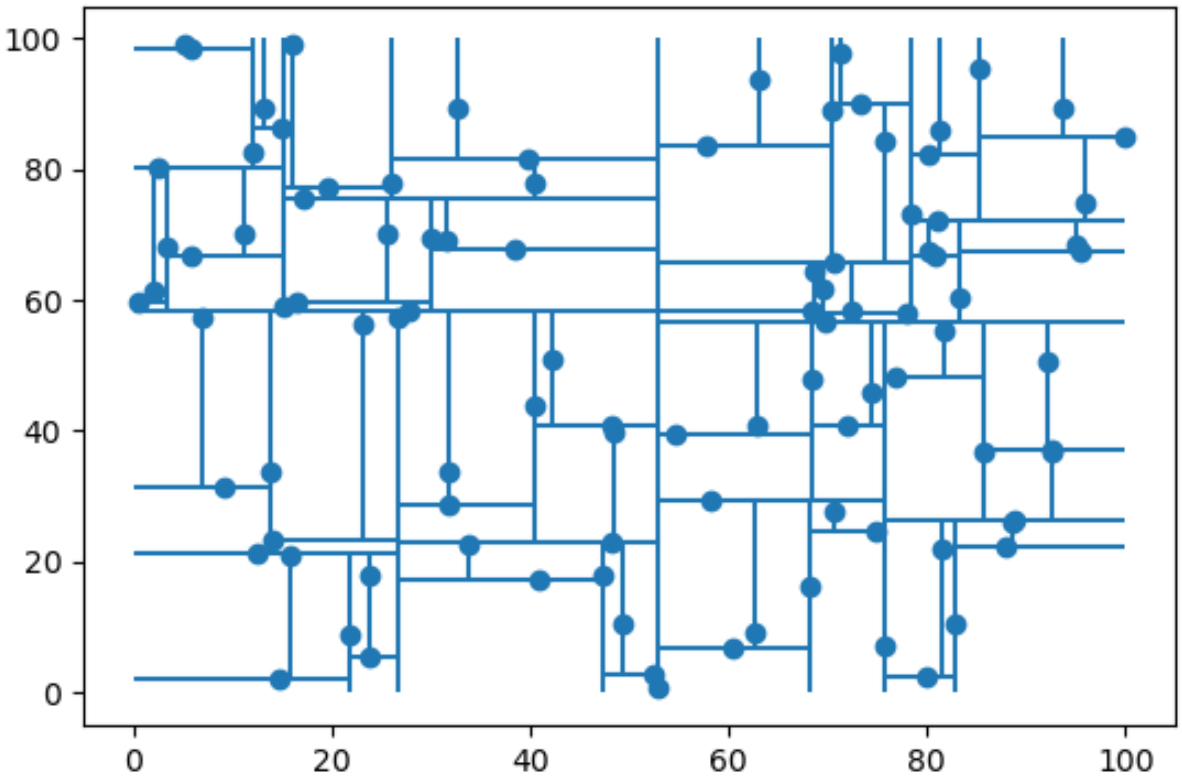
W ramach testów poprawności implementacji oraz złożoności operacji na strukturach danych utworzyliśmy zbiory, które można skategoryzować na 6 typów. Każdy zbiór testowy jest generowany w obrębie kwadratu o boku długości 100 umieszczonego w taki sposób by znajdował się on pierwszej ćwiartce układu współrzędnych, oraz by jego lewy dolny róg leżał na przecięciu obu osi.

3.4.1 Zbiór o rozkładzie jednostajnym

Jest to zbiór punktów wygenerowanych przy pomocy funkcji random.uniform. Punkty są jednostajnie rozmieszczone w obrębie kwadratu, jest to najprostszy z wygenerowanych przypadków. Wybraliśmy go właśnie ze względu na prostotę i powszechność jego użycia w testach. Prostokąt dla którego testujemy szukanie zawartych w nim punktów jest losowym prostokątem którego współrzędne są z przedziału (0, 100).



Quadtree, zbiór o rozkładzie jednostajnym



KDTree, zbiór o rozkładzie jednostajnym

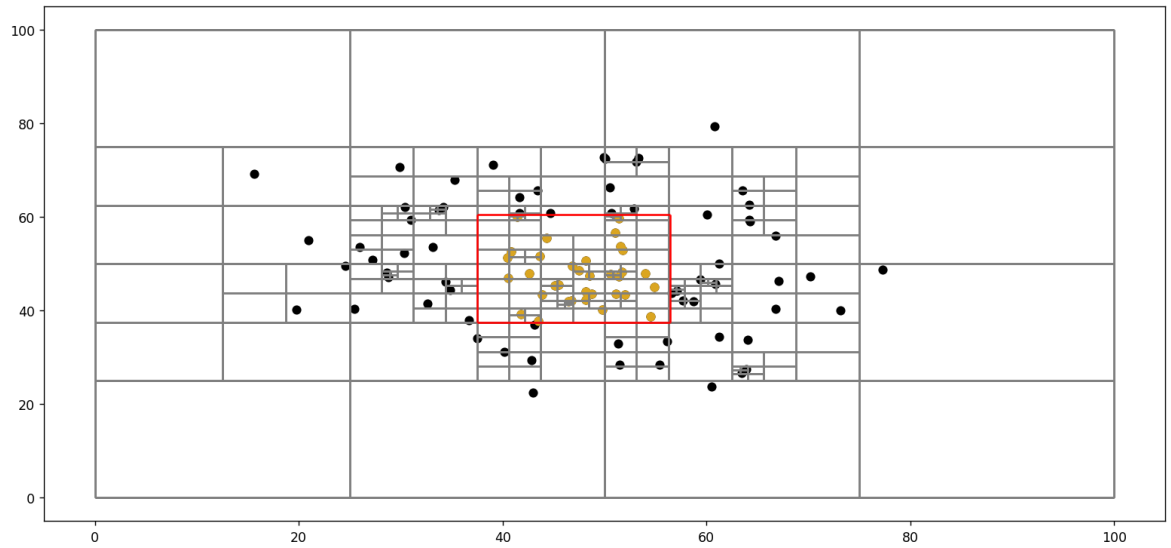
Poniższa tabela ilustruje czasy działania programów dla podanego zbioru, przy danej ilości punktów:

Liczba punktów	Czas tworzenia KDtree [ms]	Czas dodawania punktu KDtree [ms]	Czas przeszukiwania KDtree [ms]	Czas tworzenia Quadtree [ms]	Czas dodawania punktu Quadtree [ms]	Czas przeszukiwania Quadtree [ms]
1000	4.101	2.656	0.009	19.454	2.690	0.031
10000	57.113	24.418	0.011	274.546	6.091	0.048
50000	424.081	133.962	0.018	1732.446	12.576	0.059
100000	995.321	304.823	0.020	3813.153	24.116	0.067

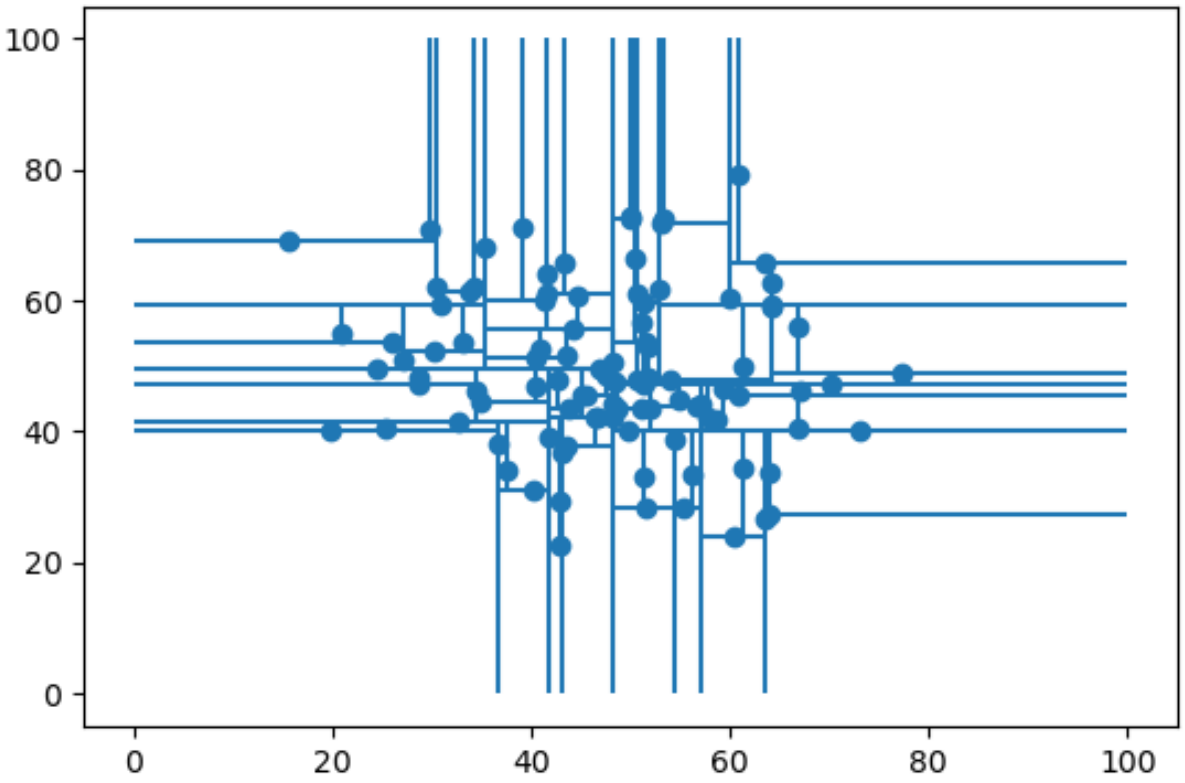
Na podstawie zebranych danych można stwierdzić, że w przypadku budowania, struktura KDtree jest znacznie szybsza. Również analizując czasy przeszukiwań zadanego obszaru w poszukiwaniu punktów KDtree jest szybsze, a stosunkowa różnica w czasie wykonania zapytania to ok 340%, jednak nie można stwierdzić, że z przyrostem ilościu punktów ta proporcja ulega znacznej zmianie. Struktura Quadtree okazuje się jedynie szybsza przy dodawaniu nowego punktu do struktury. Choć czasy dla 1000 punktów są niemalże jednakowe różnica mocno zwiększa się wraz ze wzrostem ilości punktów. Dla zbioru o rozmiarze 10000 czas dodania nowego elementu do KDtree jest ponad 10-krotnie większy.

3.4.2 Zbiór o rozkładzie normalnym

Rozkład punktów w tym zbiorze to rozkład normalny o średniej (50, 50) i odchyleniu standardowym 12.5 ($\frac{1}{8}$ boku kwadratu testowego). Charakteryzuje on tym, że prawie wszystkie punkty skupione są w okolicach punktu $(x, y) = (50, 50)$, który jednocześnie jest punktem przecięcia przekątnych kwadratu testowego. Wybraliśmy ten zbiór by przetestować zarówno czy struktury poprawnie podzielą przestrzeń przy dużym zagęszczeniu punktów, jak również porównać złożoność czasową operacji pomiędzy KDtree i quadtree w tym specyficznym przypadku. Prostokąt dla którego testujemy szukanie zawartych w nim punktów posiada lewy dolny róg o współrzędnych (37.5, 37.5), a współrzędne prawego górnego rogu są losowo generowane ze zbioru (50, 62.5).



Quadtree, zbiór o rozkładzie normalnym



KDTree, zbiór o rozkładzie normalnym

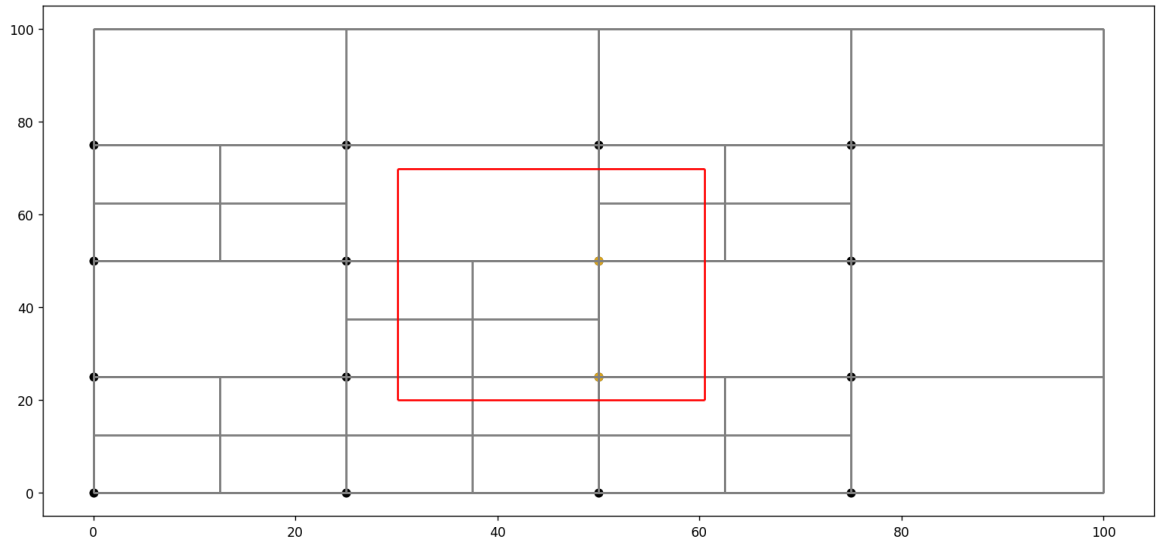
Poniższa tabela ilustruje czasy działania programów dla podanego zbioru, przy danej ilości punktów:

Liczba punktów	Czas tworzenia KDtree [ms]	Czas dodawania punktu KDtree [ms]	Czas przeszukiwania KDtree [ms]	Czas tworzenia Quadtree [ms]	Czas dodawania punktu Quadtree [ms]	Czas przeszukiwania Quadtree [ms]
1000	4.501	3.939	0.013	80.418	6.099	0.041
10000	75.292	37.693	0.018	265.536	9.936	0.045
50000	383.561	205.868	0.071	1974.965	26.945	0.052
100000	1004.680	463.410	0.129	4459.318	41.305	0.046

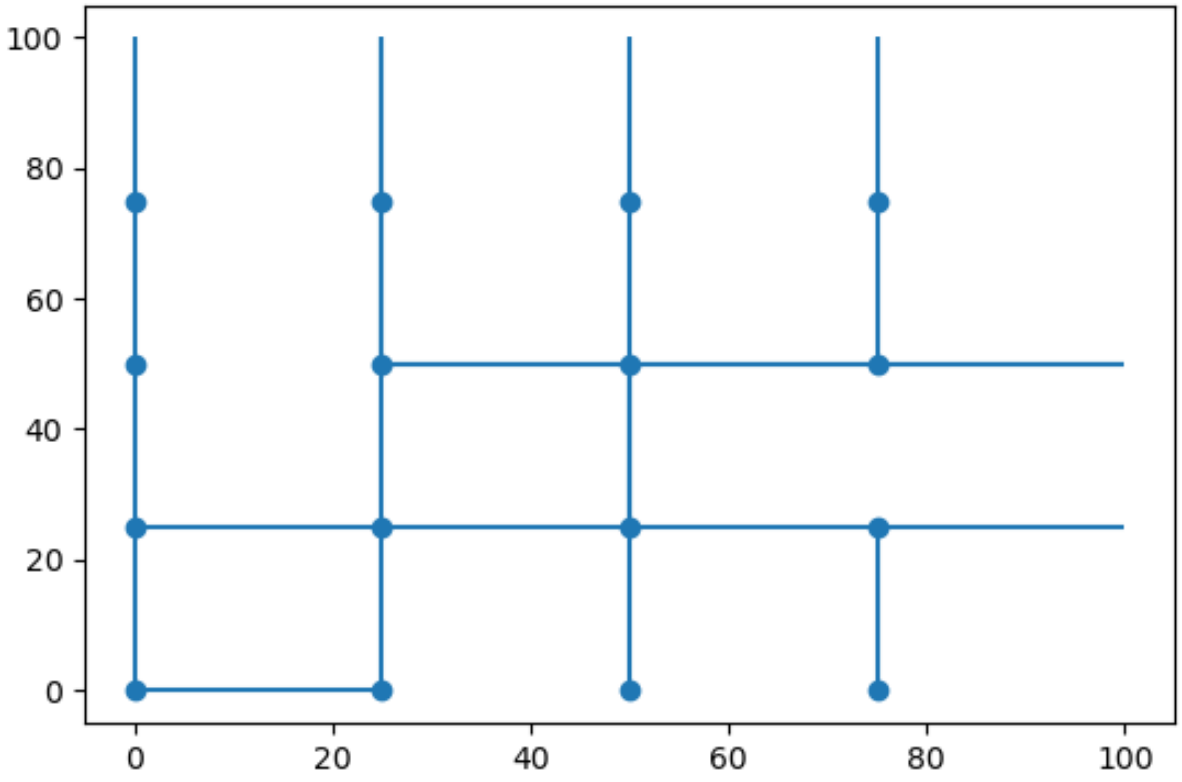
Na podstawie zebranych danych można stwierdzić, że w przypadku budowania, struktura KDtree jest znacznie szybsza. Również analizując czasy przeszukiwań zadanego obszaru w poszukiwaniu punktów KDtree jest szybsze, ale tylko dla ilości punktów do ok 10000. Czasy przeszukiwań struktury Quadtree rosną o wiele wolniej, co w efekcie powoduje że już przy 50000 punktów i powyżej tej ilości to Quadtree okazuje się szybsze. Struktura Quadtree jest również szybsza przy dodawaniu nowego punktu do struktury przy ilości punków większej od 1000. Choć dla 1000 punktów KDtree jest szybsze niemalże 2 krotnie już przy 10000 punktów to Quadtree wypada lepiej, a różnica mocno zwiększa się wraz ze wzrostem ilości punktów. Dla zbioru o rozmiarze 100000 czas dodania nowego elementu do KDtree jest ponad 10-krotnie większy.

3.4.3 Siatka

Jest to zbiór punktów skonstruowany w taki sposób by każdy odległy był od swoich bezpośrednich sąsiadów o równą odległość oraz by punkty przy analogicznym podziale jak w strukturze quadtree leżały na bokach dzielących obszary. Taka charakterystyka zbioru mogłoby generować błędy przy dodawaniu punktów do struktury, jak również przy szukaniu punków z zadanego obszaru. Prostokąt dla którego testujemy szukanie zawartych w nim punktów jest losowym prostokątem którego współrzędne są z przedziału (0, 100).



Quadtree, siatka



KDTree, siatka

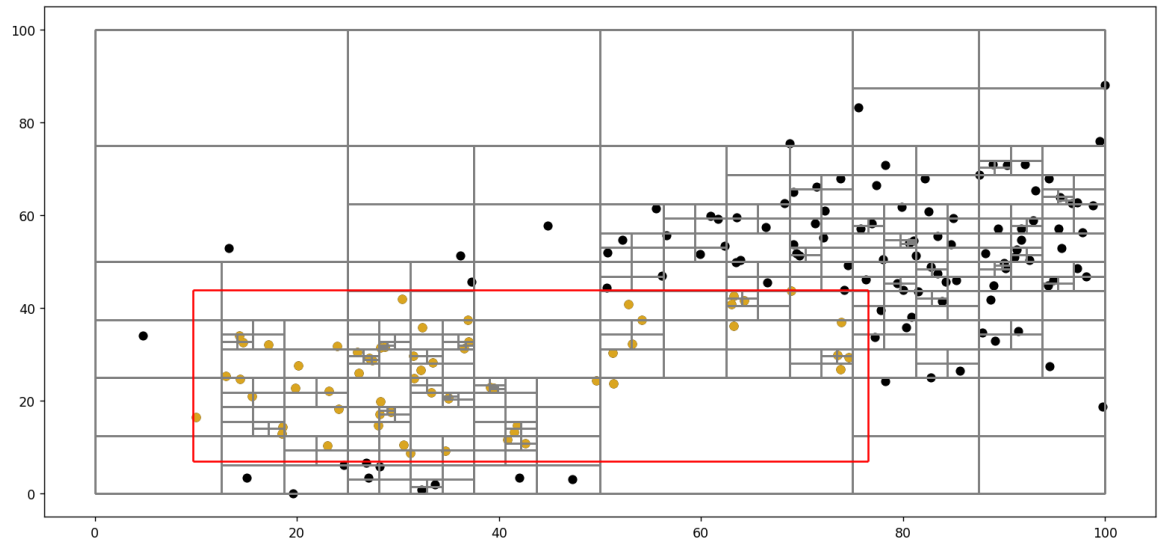
Poniższa tabela ilustruje czasy działania programów dla podanego zbioru, przy danej ilości punktów:

Liczba punktów	Czas tworzenia KDtree [ms]	Czas dodawania punktu KDtree [ms]	Czas przeszukiwania KDtree [ms]	Czas tworzenia Quadtree [ms]	Czas dodawania punktu Quadtree [ms]	Czas przeszukiwania Quadtree [ms]
961	4.501	2.629	0.010	12.203	2.395	0.051
9801	44.410	27.481	0.019	263.634	7.623	0.061
48400	313.667	133.406	0.054	1122.674	13.426	0.070
99856	855.543	258.565	0.133	2692.638	14.128	0.074

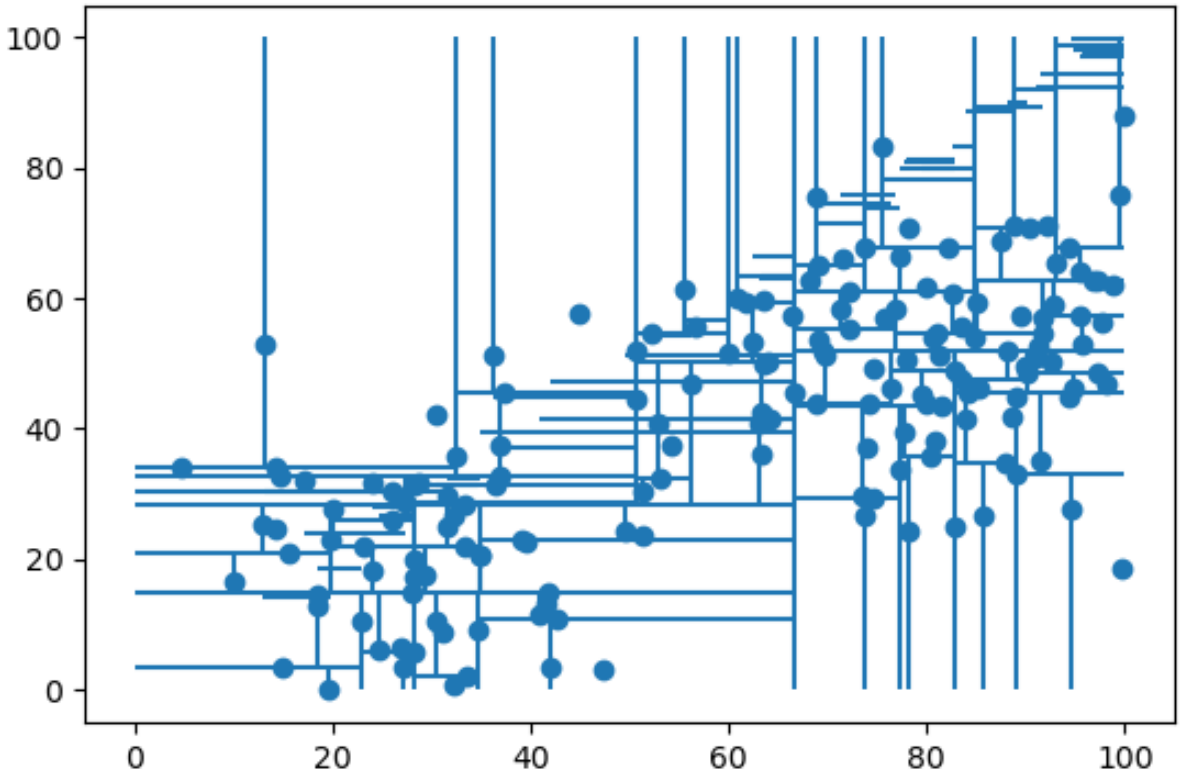
Na podstawie zebranych danych można stwierdzić, że w przypadku budowania, struktura KDtree jest znacznie szybsza. Również analizując czasy przeszukiwań zadanego obszaru w poszukiwaniu punktów KDtree jest szybsze dla ilości punktów do 100000. Czasy przeszukiwań Quadtree rosną o wiele wolniej wraz ze wzrostem ilości punków niż czasy dla struktury KDtree i w efekcie dla 100000 puntków to Quadtree okazuje się być szybsze. Struktura Quadtree jest również szybsza przy dodawaniu nowego punktu do struktury. Choć dla 1000 punktów czasu są niemalże jednakowe to dla zbioru o rozmiarze 100000 czas dodania nowego elementu do KDtree jest ponad 18-krotnie większy.

3.4.4 Wiele skupisk o rozkładzie normalnym

Jest to zbiór podobny do zbioru o rozkładzie normalnym, cechą różniącą te dwa typy zbiorów jest ilość skupisk. W zbiorze o rozkładzie było to jedno skupisko, tutaj jednak są 3. Wybór tej liczby nie ma głębszego znaczenia. Celem testów na zbiorach o tym typie jest sprawdzenie jak struktury radzą sobie z wieloma miejscami gdzie zagęszczenie punktów jest stosunkowo duże w porównaniu do średniej gęstości punktów na jednostkę powierzchni. Prostokąt dla którego testujemy szukanie zawartych w nim punktów jest losowym prostokątem którego współrzędne są z przedziału (0, 100).



Quadtree, wiele skupisk o rozkładzie normalnym



KDTree, wiele skupisk o rozkładzie normalnym

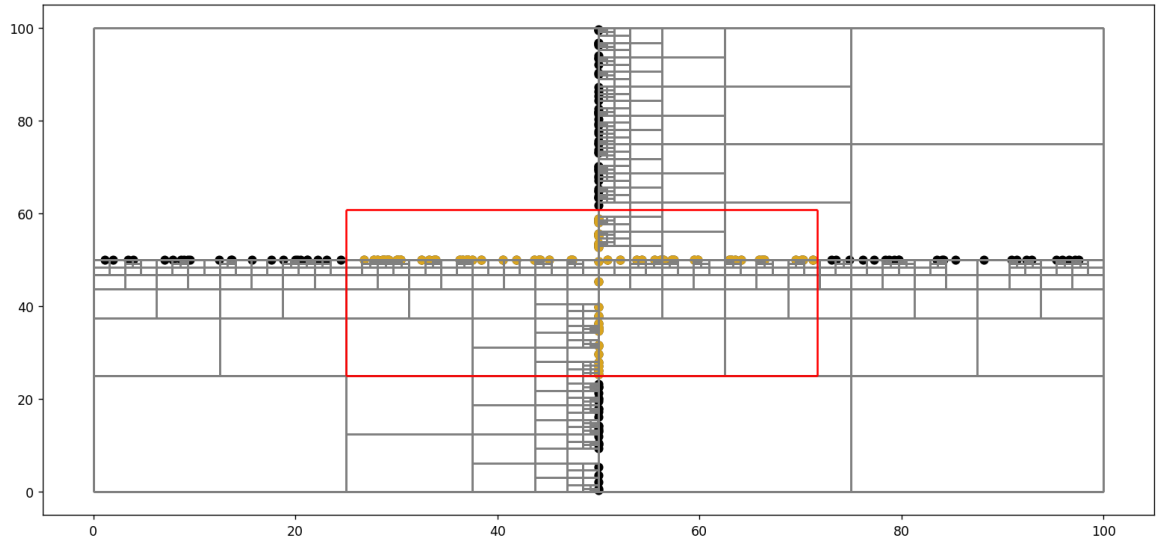
Poniższa tabela ilustruje czasy działania programów dla podanego zbioru, przy danej ilości punktów:

Liczba punktów	Czas tworzenia KDtree [ms]	Czas dodawania punktu KDtree [ms]	Czas przeszukiwania KDtree [ms]	Czas tworzenia Quadtree [ms]	Czas dodawania punktu Quadtree [ms]	Czas przeszukiwania Quadtree [ms]
1000	4.301	2.657	0.012	14.453	3.390	0.046
10000	68.241	23.490	0.020	259.354	6.384	0.039
50000	370.509	121.244	0.053	1560.254	14.001	0.048
100000	748.318	241.864	0.114	3519.927	20.695	0.053

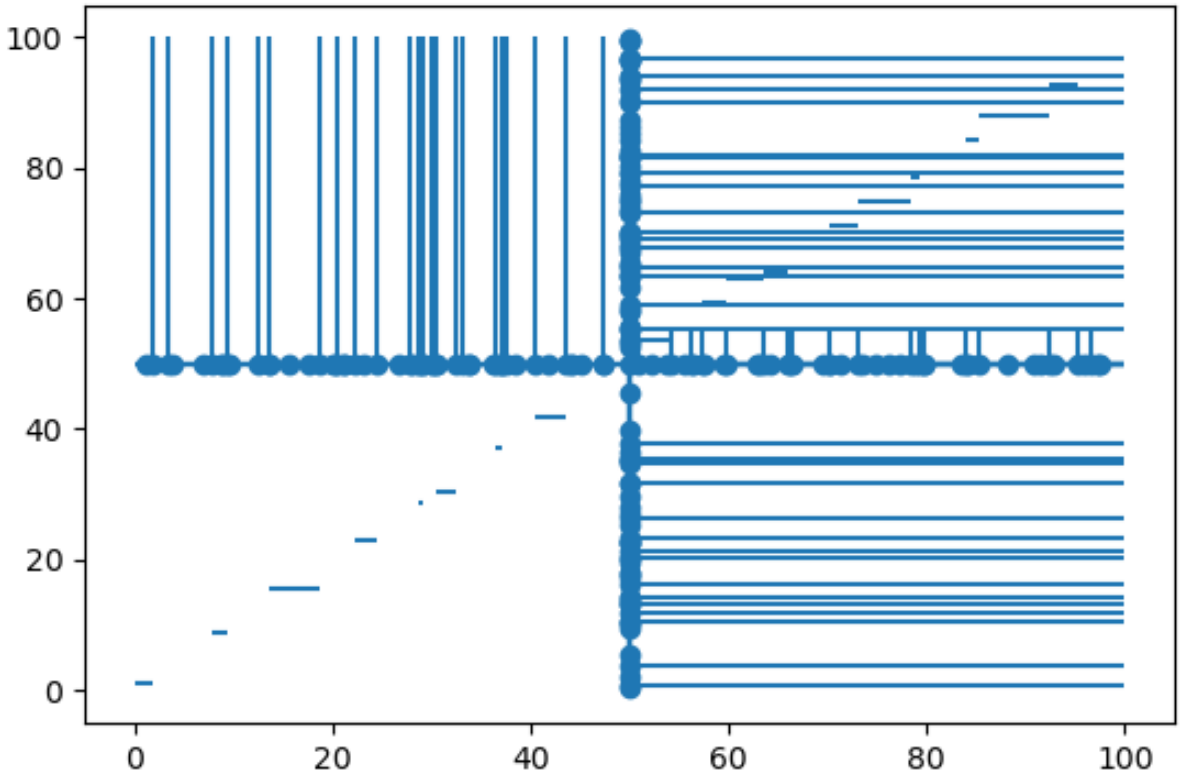
Na podstawie zebranych danych można stwierdzić, że w przypadku budowania, struktura KDtree jest znacznie szybsza. Również analizując czasy przeszukiwań zadanego obszaru w poszukiwaniu punktów KDtree jest szybsze dla ilości punktów do 100000. Czasy przeszukiwań Quadtree rosną o wiele wolniej wraz ze wzrostem ilości punków niż czasy dla struktury KDtree i w efekcie dla 100000 puntków to Quadtree okazuje się być szybsze. Quadtree jest również szybsza przy dodawaniu nowego punktu do struktury. Choć dla 1000 punktów to KDtree jest szybsze to dla większych zbiorów to Quadtree wypada lepiej, dla zbioru o rozmiarze 100000 czas dodania nowego elementu do KDtree jest już ponad 12-krotnie większy.

3.4.5 Krzyż

Zbiór ten składa się z punktów umieszczonych na 2 prostych. Jednej równoległej do osi X, drugiej do osi Y. Proste te przecinają się w punkcie (50, 50). Wybraliśmy ten zbiór ze względu właśnie na jego charakterystyczne rozłożenie punktów. Ma ono za zadanie sprawdzić jak obie struktury poradzą sobie z tak specyficznie rozłożonymi punktami oraz wykryć ewentualnie istniejące błędy w implementacji. Prostokąt dla którego testujemy szukanie zawartych w nim punktów zawiera punkt (50,50), jego lewy dolny róg ma współrzędne (25, 25), a współrzędne prawego górnego rogu są losowo generowane ze zbioru (55, 75).



Quadtree, krzyż



KDTree, krzyż

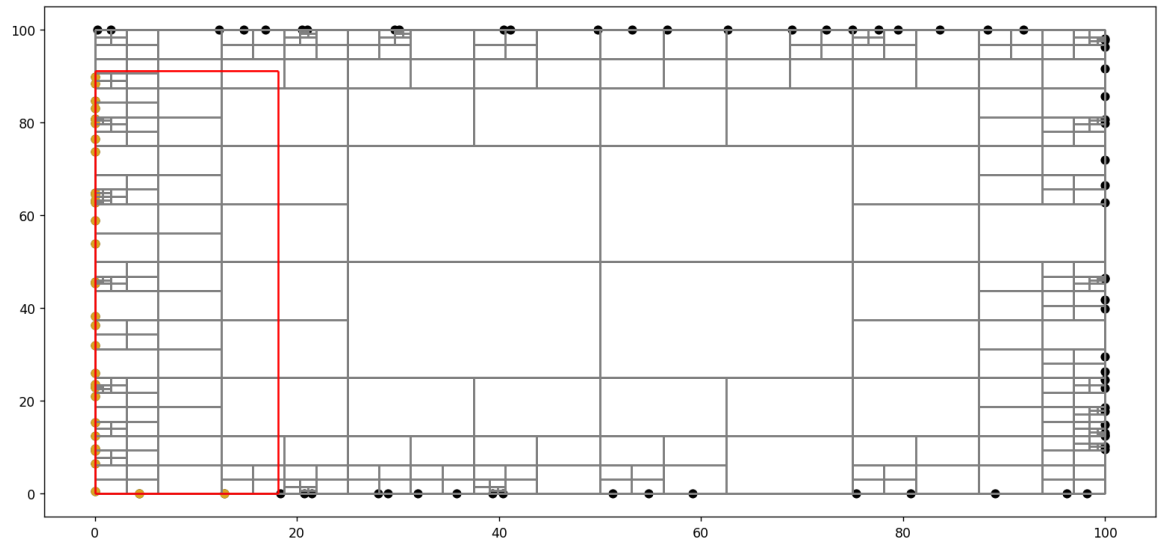
Poniższa tabela ilustruje czasy działania programów dla podanego zbioru, przy danej ilości punktów:

Liczba punktów	Czas tworzenia KDtree [ms]	Czas dodawania punktu KDtree [ms]	Czas przeszukiwania KDtree [ms]	Czas tworzenia Quadtree [ms]	Czas dodawania punktu Quadtree [ms]	Czas przeszukiwania Quadtree [ms]
1000	5.701	4.502	0.019	106.624	5.460	0.028
10000	140.682	40.724	0.019	572.005	9.875	0.027
50000	481.509	217.888	0.066	4039.674	24.643	0.031
100000	1305.524	481.176	0.134	8986.784	43.999	0.030

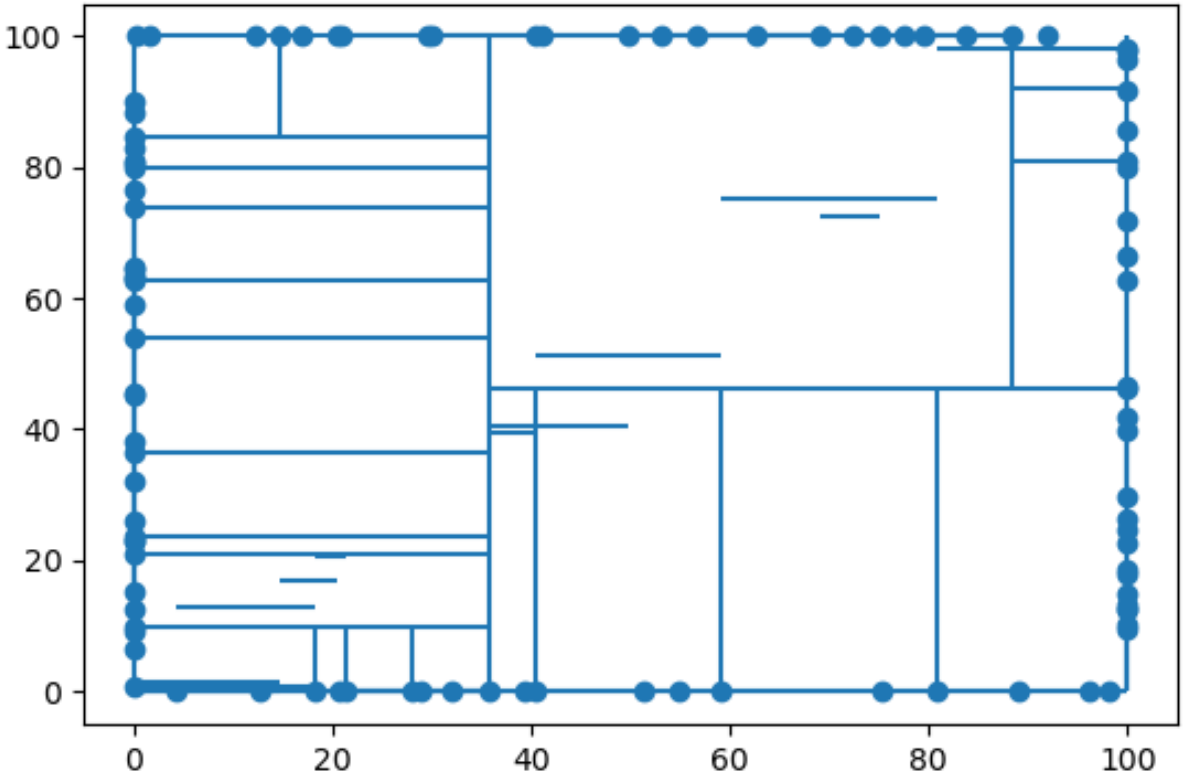
Na podstawie zebranych danych można stwierdzić, że w przypadku budowania, struktura KDtree jest znacznie szybsza. Również analizując czasy przeszukiwań zadanego obszaru w poszukiwaniu punktów KDtree jest szybsze dla ilości punktów do 100000. Czasy przeszukiwań Quadtree rosną o wiele wolniej wraz ze wzrostem ilości punków niż czasy dla struktury KDtree i w efekcie już dla 50000 puntków to Quadtree okazuje się być szybsze. Struktura Quadtree okazuje się również być szybsza przy dodawaniu nowego punktu do struktury. Choć dla 1000 punktów to KDtree jest szybsze to dla większych zbiorów to Quadtree wypada lepiej, dla zbioru o rozmiarze 100000 czas dodania nowego elementu do KDtree jest już ponad 10-krotnie większy.

3.4.6 Prostokąt

Zbiór ten składa się z punktów jednostajnie umieszczonych na bokach testowego prostokąta. Argumentacja wyboru tego zbioru jest analogiczna jak dla zbioru nr 6. Prostokąt dla którego testujemy szukanie zawartych w nim punktów posiada lewy dolny róg o współrzędnych (0, 0), a współrzędne prawego górnego rogu są losowo generowane ze zbioru (0, 100).



Quadtree, prostokąt



KDTree, prostokąt

Poniższa tabela ilustruje czasy działania programów dla podanego zbioru, przy danej ilości punktów:

Liczba punktów	Czas tworzenia KDtree [ms]	Czas dodawania punktu KDtree [ms]	Czas przeszukiwania KDtree [ms]	Czas tworzenia Quadtree [ms]	Czas dodawania punktu Quadtree [ms]	Czas przeszukiwania Quadtree [ms]
1000	5.852	3.461	0.009	25.656	3.531	0.026
10000	330.089	23.959	0.012	466.926	7.299	0.031
50000	482.032	128.465	0.018	3525.077	16.514	0.035
100000	1029.129	279.073	0.022	8031.801	27.606	0.032

Na podstawie zebranych danych można stwierdzić, że w przypadku budowania, struktura KDtree jest znacznie szybsza. Również analizując czasy przeszukiwań zadanego obszaru w poszukiwaniu punktów KDtree jest szybsze. Struktura Quadtree okazuje się jedynie być szybsza przy dodawaniu nowego punktu do struktury. Choć dla 1000 punktów to KDtree jest nieznacznie szybsze to dla większych zbiorów to Quadtree wypada lepiej, dla zbioru o rozmiarze 100000 czas dodania nowego elementu do KDtree jest już ponad 10-krotnie większy.

3.5 Wnioski

Na podstawie testów przeprowadzonych dla wyżej przeanalizowanych zbiorów, możemy stwierdzić, że programy przez nas zaimplementowane działają prawidłowo i poprawnie wyznaczają podzbiór punktów należących do zadanej płaszczyzny. W każdym przetestowanym zbiorze budowa struktury QuadTree była wolniejsza, jednak dodawanie nowego punktu do struktury w większości przypadków było bardziej optymalne dla struktury Quadtree. Wyjątkiem były jedynie niektóre zbiory liczące 1000 punktów. Do przeszukiwania zbiorów o bardzo dużych punktowych zagęszczeniach zdecydowanie lepszym wyborem będzie Quadtree, co można zobaczyć na przykładach zbiorów nr 2 i 4 przy zbiorach liczących ponad 50000 punktów. Również do znajdywania podzbioru punktów zbioru którego rozkład punktów na płaszczyźnie zbliżony jest do krzyża oraz liczebność punktów przekracza 10000 warto rozważyć Quadtree. KDtree jednak lepiej sprawdzi się do przeszukiwania obszarów zbiorów dowolnego typu liczących do 10000 punktów. Dodatkowo KDtree będzie lepszym wyborem niezależnie od rozmiaru zbioru przy zbiorach o rozkładzie jednostajnym (zbiór nr 1) lub zbiorach gdzie punkty głównie rozłożone są na krawędziach prostokąta (zbiór nr 6).

Rozdział 4

Bibliografia

Umieściliśmy tu informacje na temat źródeł, z których korzystaliśmy by zebrać wiedze na temat struktur oraz niektóre grafiki.

4.1 QuadTree

- Angielskojęzyczna Wikipedia
- Polska Wikipedia
- OpenDSA Data Structures and Algorithms Modules Collection

4.2 KDtree

- Angielskojęzyczna Wikipedia
- Przykładowa implementacja przeszukiwania i wstawiania do KD drzewa. Geeksforgeeks