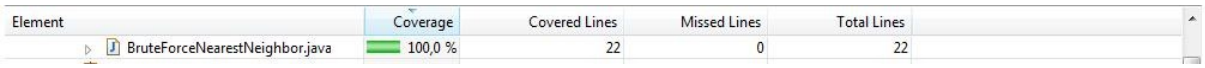
Testy Białoskrzynkowe

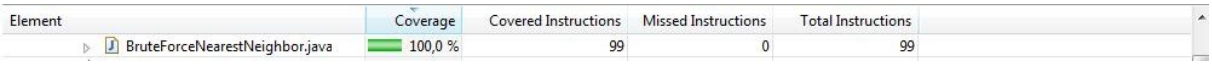
1. **BruteForceNearestNeighbor:**
   Koncepcja przyjętych testów polegała na jak największym pokryciu kodu, przy odpowiednio dobranych danych.

   W rezultacie po wykonaniu testów otrzymaliśmy 100% pokrycie linii:



Element	Coverage	Covered Lines	Missed Lines	Total Lines
BruteForceNearestNeighbor.java	100,0 %	22	0	22

   o czym świadczy powyższy obrazek, a także 100% pokrycie instrukcji:



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
BruteForceNearestNeighbor.java	100,0 %	99	0	99

   Nasze testy w wystarczający sposób pokrywają kod badanego algorytmu, świadczą one o poprawnym wykonaniu algorytmu dla testowanych danych.

   Dane użyte w algorytmie to:
   - IMultiPoint points[] – tablica punktów o rozmiarze 100. Punkty umieszczone zostały na linii prostej x=0, ze zmiennym y od 0 do 99.
   - BruteForceNearestNeighbor instance – badany algorytm.
   - IMultiPoint x – punkt od którego szukamy najbliższego punktu (10,10)
   - IMultiPoint expResult – oczekiwany wynik, punkt o współrzędnych (0,10)
   - IMultiPoint result – otrzymany wynik działania metody nearest algorytmu BruteForceNearestNeighbor.

   Algorytm dla powyższych danych zwrócił wynik oczekiwany. Oprócz badania działania metody nearest przetestowano obsługę wyjątków zaimplementowanych w konstruktorze algorytmu.

   Testy przeprowadzone zostały w programie eclipse, a miara kodu możliwa była, dzięki wyposażeniu programu w specjalną wtyczkę (EclEmma Java Code Coverage 2.2.1)

2. **Struktura kdTree i klas wykorzystywanych w kodzie algorytmu:**
   Testy przeprowadzone zostały z myślą uzyskania jak największego pokrycia kodu. Skorzystaliśmy z testów zaproponowanych przez autora algorytmu, które uzupełniliśmy o własne testy tak, aby kod był pokryty w jak największym stopniu.

   Pokrycie kodu:
   a) Przed dodaniem testów:

- Hypercube test:

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Hypercube.java	76,8 %	413	125	538

Element	Coverage	Covered Lines	Missed Lines	Total Lines
Hypercube.java	75,2 %	76	25	101

Testy Hypercube zaproponowane przez autora mają stosunkowo wysokie (powyżej 75%) pokrycie kodu

- Hyperpoint test:

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Hyperpoint.java	99,4 %	358	2	360

Element	Coverage	Covered Lines	Missed Lines	Total Lines
Hyperpoint.java	98,5 %	64	1	65

Autor zaproponował testy klasy Hyperpoint z bardzo dużym pokryciem kodu. Pominął zaledwie 2 instrukcje.

- TwoDTest:
Instrukcje:

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
HorizontalNode.java	88,1 %	52	7	59
VerticalNode.java	88,1 %	52	7	59
TwoDNode.java	41,3 %	147	209	356
TwoDTree.java	32,3 %	84	176	260

linie:

Element	Coverage	Covered Lines	Missed Lines	Total Lines
HorizontalNode.java	92,9 %	13	1	14
VerticalNode.java	92,9 %	13	1	14
TwoDNode.java	41,4 %	41	58	99
TwoDTree.java	37,3 %	31	52	83

Testy zawarte w tym pliku testowym pokrywają ok. 90% kodu klas HorizontalNode i VerticalNode, gorzej natomiast (ok. 40%) TwoDNode i TwoDTree. Autor także przetestował klasę TwoDPoint ze 36% pokryciem kodu, czego nie widać na powyższych ilustracjach.
Wnioskując powyższy test jest niewystarczający, dlatego z całą pewnością wymaga uzupełnienia.

- KDTreeTest:
Instrukcje:

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
CounterKDTree.java	100,0 %	27	0	27
KDTree.java	88,1 %	296	40	336
DimensionalNode.java	79,7 %	521	133	654
KDTraversal.java	74,1 %	40	14	54

Linie:

Element	Coverage	Covered Lines	Missed Lines	Total Lines
CounterKDTree.java	100,0 %	10	0	10
KDTree.java	87,2 %	95	14	109
DimensionalNode.java	75,2 %	118	39	157
KDTraversal.java	66,7 %	16	8	24

Testy autora zawarte w tym pliku mają dość duże pokrycie kodu, jednak można było uzyskać większe

- KDExtendedTest
  Autor w tym pliku testuje wyłącznie konstruktor TwoDPoint i to w znikomym stopniu.

- Razem:
  Instrukcje:

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
CounterKDTree.java	100,0 %	27	0	27
HorizontalNode.java	88,1 %	52	7	59
VerticalNode.java	88,1 %	52	7	59
KDTree.java	88,1 %	296	40	336
DimensionalNode.java	79,7 %	521	133	654
KDTraversal.java	74,1 %	40	14	54
TwoDNode.java	41,3 %	147	209	356
TwoDTree.java	32,3 %	84	176	260

Linie:

Element	Coverage	Covered Lines	Missed Lines	Total Lines
CounterKDTree.java	100,0 %	10	0	10
HorizontalNode.java	92,9 %	13	1	14
VerticalNode.java	92,9 %	13	1	14
KDTree.java	87,2 %	95	14	109
DimensionalNode.java	75,2 %	118	39	157
KDTraversal.java	66,7 %	16	8	24
TwoDNode.java	41,4 %	41	58	99
TwoDTree.java	37,3 %	31	52	83

Klasy testowane we wszystkich plikach testowych zaproponowanych przez autora razem osiągają dość wysokie wyniki (poza 3 ostatnimi klasami z dołączonej ilustracji)

b) Po dodaniu testów:
- Hypercube test

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Hypercube.java	100,0 %	538	0	538

Element	Coverage	Covered Lines	Missed Lines	Total Lines
Hypercube.java	100,0 %	101	0	101

Po dodaniu odpowiednich testów udało nam się uzyskać 100% pokrycie kodu dla klasy Hypercube. Przetestowaliśmy inne wersje konstruktorów klasy Hypercube (nie testowane przez autora), a także metody takie jak setLeft, setRight, intersect oraz contain w taki sposób, aby uzyskać jak największe pokrycie kodu. Dwie pierwsze metody nie były testowane przez autora. W metodzie intersect brakowało przypadku, gdy punkt znajdował się poniżej lewego zasięgu i powyżej prawego.
Metoda contain wymagała prawie całkowitego przetestowania.
Aby uzyskać 100% pokrycie kodu dla tej metody musieliśmy stworzyć dwie różne Hypercube (h1 i h2), takie, że jedna nie zawierała w sobie drugiej, oraz sprawdzić czy h1 zawiera w sobie h1 (h1.contains(h1) - true) i czy h1 zawiera w sobie h2 (h1.contains(h2) – false).

- Hyperpoint test:

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Hyperpoint.java	100,0 %	360	0	360

Element	Coverage	Covered Lines	Missed Lines	Total Lines
Hyperpoint.java	100,0 %	65	0	65

Udało nam się uzyskać 100% pokrycia kodu dla klasy Hyperpoint dzięki przetestowaniu działania tylko jednej metody (equals) i to tylko raz. Dopisany test polegał na sprawdzeniu czy 2 różne typy punktów (Hyperpoint hp oraz TwoDPoint hp2) o różnych współrzędnych przy porównaniu rzeczywiście zwrócą wynik false.

- TwoDTest:
Instrukcje:

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
HorizontalNode.java	100,0 %	59	0	59
VerticalNode.java	100,0 %	59	0	59
TwoDTree.java	97,7 %	254	6	260
TwoDNode.java	82,9 %	295	61	356

Linie:

Element	Coverage	Covered Lines	Missed Lines	Total Lines
HorizontalNode.java	100,0 %	14	0	14
VerticalNode.java	100,0 %	14	0	14
TwoDTree.java	97,6 %	81	2	83
TwoDNode.java	75,8 %	75	24	99

Po dopisaniu przez nas testów uzyskaliśmy znacznie większe pokrycie kodu niż autor. W pierwszej kolejności zajęliśmy się przetestowaniem klasy TwoDPoint. W efekcie czego otrzymaliśmy 100% pokrycie kodu dla tej klasy.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
TwoDPoint.java	100,0 %	176	0	176

Element	Coverage	Covered Lines	Missed Lines	Total Lines
TwoDPoint.java	100,0 %	36	0	36

Przetestowaliśmy działanie dwóch nietestowanych wcześniej konstruktorów (od ciągu znaków, a także drugiego punktu), metodę equals dla niepokrytej wcześniej części kodu (czyli przypadku, gdy punkty różnią się miedzy sobą składową y, a następnie różniące się składową x). Dodatkowo przetestowaliśmy działanie takich metod jak toString(), dimensionality(), getCoordinate(), distance(), oraz hashCode().

W kolejnym kroku testowaliśmy klasę TwoDTree w wyniku czego otrzymaliśmy 97.6% pokrycia linii i 97.7 % pokrycia instrukcji. Niepokryte linie są nieosiągalne i stanowią dodatkowe zabezpieczenie badanego algorytmu. Do badania tej klasy stworzyliśmy 5 różnych punktów (typu TwoDPoint). W pierwszej kolejności badaliśmy obsługę wyjątków rzucanych podczas wykonywania metod insert, oraz parent dla

parametru null. Algorytm zachował się tak jak powinien i wyrzucił wyjątek. Następnie w sposób wystarczający (do pokrycia kodu) badaliśmy równomiernie metodę insert oraz parent, działanie algorytmu nearest oraz toString i updateRectangle. Wszystkie testy przebiegły według oczekiwań.

Zwiększenie pokrycia kodu dla TwoDNode jest wynikiem testowania algorytmu TwoDTree.

- KDTreeTest:
  Instrukcje:

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
CounterKDTree.java	100,0 %	27	0	27
KDTree.java	96,7 %	325	11	336
DimensionalNode.java	82,9 %	542	112	654
KDTraversal.java	74,1 %	40	14	54

  Linie:

Element	Coverage	Covered Lines	Missed Lines	Total Lines
CounterKDTree.java	100,0 %	10	0	10
KDTree.java	97,2 %	106	3	109
DimensionalNode.java	80,9 %	127	30	157
KDTraversal.java	66,7 %	16	8	24

  Pokrycie kodu uzyskane dzięki dopisaniu przez nas testów jest oczywiście, tak jak zamierzaliśmy większe od tych, które podał autor.
  W pierwszej kolejności zbadaliśmy dwie niebadane wcześniej możliwości metody nearest – gdy Root=null i target=null. W obu tych przypadkach algorytm zadziałam zgodnie z oczekiwaniem i zwrócił null.
  W następnym kroku zajęliśmy się badaniem metody toString() – gdy nie ma korzenia i gdy on jest.
  Następnie przebadaliśmy takie metody jak count i height, które działają zgodnie z oczekiwaniami.
  Nieprzetestowane linie kodu wynikają z tego, że są nieosiągalne i stanowią zabezpieczenie kodu w przypadku niezadziałania wcześniejszej instrukcji.
  Zwiększenie pokrycia kodu w DimensionalNode jest wynikiem przeprowadzonych testów dla KDTree.

- KDExtendedTest
  Nie przeprowadziliśmy tu dodatkowych testów, ponieważ badanie klasy TwoDPoint zostało przeprowadzone przez nas w TwoDTest, dlatego uznaliśmy (po uzyskaniu 100% pokrycia kodu) ,że kolejne testowanie jest zbyteczne.

- Razem

Instrukcje:

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
CounterKDTree.java	100,0 %	27	0	27
HorizontalNode.java	100,0 %	59	0	59
VerticalNode.java	100,0 %	59	0	59
TwoDTree.java	97,7 %	254	6	260
KDTree.java	96,7 %	325	11	336
DimensionalNode.java	82,9 %	542	112	654
TwoDNode.java	82,9 %	295	61	356
KDTraversal.java	74,1 %	40	14	54

Linie:

Element	Coverage	Covered Lines	Missed Lines	Total Lines
CounterKDTree.java	100,0 %	10	0	10
HorizontalNode.java	100,0 %	14	0	14
VerticalNode.java	100,0 %	14	0	14
TwoDTree.java	97,6 %	81	2	83
KDTree.java	97,2 %	106	3	109
DimensionalNode.java	80,9 %	127	30	157
TwoDNode.java	75,8 %	75	24	99
KDTraversal.java	66,7 %	16	8	24

Jak widać na powyższych ilustracjach udało nam się osiągnąć znacznie większe pokrycie kodu niż miało to miejsce w początkowej wersji testów. Uważamy, że przeprowadzone przez nas testy są wystarczające.
Testy przeprowadzone zostały w programie eclipse, a miara kodu możliwa była, dzięki wyposażeniu programu w specjalną wtyczkę (EclEmma Java Code Coverage 2.2.1)

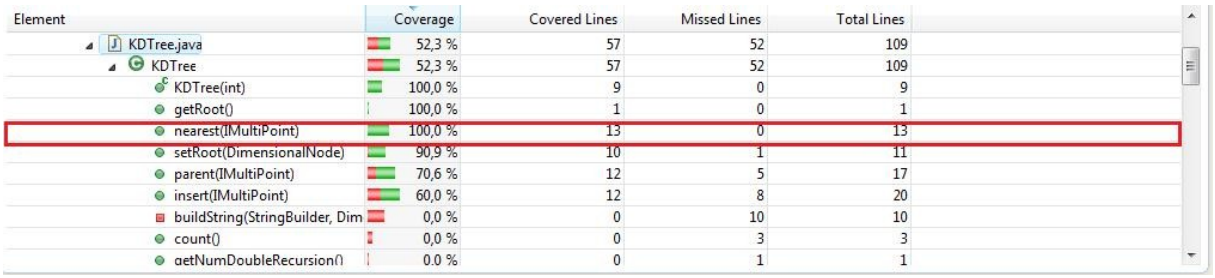
3. Algorytm NearestNeighbor działający na kdTree

Testy przeprowadzone zostały z myślą jak największego pokrycia kodu, a także wykrycia podstawowych błędów w działaniu algorytmu znajdowania najbliższego sąsiada opartego na kdTree.

Miara pokrycia kodu:
Instrukcje:

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
KDTree.java	54,5 %	183	153	336
KDTree	54,5 %	183	153	336
KDTree(int)	100,0 %	26	0	26
getRoot()	100,0 %	3	0	3
nearest(IMultiPoint)	100,0 %	46	0	46
setRoot(DimensionalNode)	97,6 %	41	1	42
parent(IMultiPoint)	67,4 %	31	15	46
insert(IMultiPoint)	60,0 %	36	24	60
buildString(StringBuilder, Dim	0,0 %	0	27	27
count()	0,0 %	0	9	9
getNumDoubleRecursion()	0,0 %	0	3	3

Linie:



Element	Coverage	Covered Lines	Missed Lines	Total Lines
KDTree.java	52,3 %	57	52	109
KDTree	52,3 %	57	52	109
KDTree(int)	100,0 %	9	0	9
getRoot()	100,0 %	1	0	1
nearest(IMultiPoint)	100,0 %	13	0	13
setRoot(DimensionalNode)	90,9 %	10	1	11
parent(IMultiPoint)	70,6 %	12	5	17
insert(IMultiPoint)	60,0 %	12	8	20
buildString(StringBuilder, Dim	0,0 %	0	10	10
count()	0,0 %	0	3	3
getNumDoubleRecursion()	0,0 %	0	1	1

W czerwonej ramce zaznaczone jest pokrycie kodu dla interesującej nas metody klasy KDTree jaką jest nearest(IMultiPoint). Jak możemy zauważyć przeprowadzone przez nas testy pokrywają kod w 100% co do instrukcji i linii kodu.

Dane użyte w algorytmie to:
- KDTree kdt – drzewo KD – na nim wykonywana jest metoda nearest, w środku (w strukturze drzewa) znajdują punkty.
- DimensionalNode n – węzeł drzewa używany jako korzeń.
- TwoDPoint[] points – tablica przechowująca punkty, które będą znajdowały się w drzewie. Wykorzystane do badania, czy algorytm działa poprawnie, dla struktury wielu punktów.
- Hyperpoint hp - punkt o wymiarze 3 umieszczony w drzewie. Do badania wpływu wielowymiarowości na działanie algorytmu
- Hyperpoint hp2 - punkt o wymiarze 4 umieszczony w drzewie. Do badania wpływu wielowymiarowości na działanie algorytmu

Przebieg testów:
Na początku w metodzie testowej nearestTest() badane jest podstawowe działanie metody nearest. Badamy tu początkowo konstruktor klasy KDTree (czy wyrzuca wyjątek dla d<1), następnie czy metoda nearest zwraca poprawne wyniki dla przypadków, gdy:
- nie ma korzenia (root = null). Wynik null.
- mamy korzeń, ale wykonujemy metodę nearest(null) – wynik null.
- mamy korzeń i wykonujemy metodę nearest dla punktu, który jest korzenień – w wyniku otrzymujemy ten sam punkt, czyli tak jak powinno być. Powyższe 3 przypadki zapewniają nam zbadanie metody nearest 92.3% pokryciem kody ze względu na linie(jedna linia pominięta) oraz 95.7% pokryciem kodu ze względu na instrukcje(2 pominięte).

W kolejnej metodzie testowej (nearestTest1()) badane jest działanie algorytmu w drzewie, które zawiera 100 punktów. Punkty te są specjalnie dobrane (aby ułatwić nam oszacowanie czy algorytm działa poprawnie) w następujący sposób. Składowa x jest zawsze stała i równa 10, a składowa y zmienia się od 0 do 99 z krokiem 1) Badamy tu działanie algorytmu dla kilku punktów i sprawdzamy, czy wynik jest zgodny z oczekiwanym. Nasze testy przebiegły pomyślnie zgodnie z oczekiwaniami. Po wykonaniu tej metody testowej uzyskujemy 100% pokrycie kodu (linii i instrukcji).

W trzeciej metodzie testowej badamy wpływ dodania do struktury drzewa różnych punktów (w sensie różno wymiarowości). Zauważamy, że po dodaniu do drzewa dwóch punktów o różnych wymiarach (hp i hp2) i szukając najbliższego punktu od 0,0 zostaje wyrzucony wyjątek mówiący o tym, że nie zgadzają się wymiary punktów. Następnie zostaje wywołana metoda nearest od punktu hp2 w wyniku czego otrzymujemy ten właśnie punkt. Dla wywołanej metody nearest dla punktu hp zostaje wyrzucony wyjątek.

Testy przeprowadzone zostały w programie eclipse, a miara kodu możliwa była, dzięki wyposażeniu programu w specjalną wtyczkę (EclEmma Java Code Coverage 2.2.1)