Projekt - Temat 9 - Dokumentacja

Rafał Piwowar
Jan Smółka

1. Aspekty techniczne projektu

1.1. Ogólny zarys programu

Program stanowi realizację zadania projektowego polegającego na implementacji dwóch struktur danych, użytecznych przy operacjach na obiektach geometrycznych – drzew czwórkowych oraz k-d drzew.
Implementacji dokonano w języku Julia, z wykorzystaniem zewnętrznych pakietów wymienionych w sekcji „Wymagana techniczne".

1.2. Moduły

1.2.1. Moduły główne

GeometricSearch.jl – główny moduł programu. Udostępnia użytkownikowi wszystkie przeznaczone dla niego funkcje programu,
Main.jl – moduł pomocniczy. Zawiera propozycję użycia funkcji dostępnych dla użytkownika, wraz z krótkimi opisami.

1.2.2. Implementacje – moduły niedostępne dla użytkownika

BasicDatatypes.jl, FileIO.jl, Generators.jl, Interactive.jl, PlotPreprocess.jl, Queries.jl, QueryProcessing.jl, TreeVisualization.jl, QuadNode.jl, QuadTree.jl, KDTreeNode.jl, KD_Tree.jl

1.3. Wymagania techniczne

Dla osiągnięcia pełnego współdziałania konieczne jest posiadanie kompilatora języka Julia, wraz z pakietami *Plots.jl* oraz *Statistics.jl*. Wskazane jest również dysponowanie środowiskiem programistycznym wspierającym działanie biblioteki Plots, np. Visual Studio Code z wtyczką języka Julia.

2. Instrukcja dla użytkownika

2.1. Korzystanie z funkcji

Korzystanie ze wszystkich funkcji przeznaczonych dla użytkownika jest możliwe po imporcie modułu *GeometricSearch.jl,* po uprzednim dodaniu bieżącego katalogu do ścieżki poszukiwań interpretera.

```
julia> push!(LOAD_PATH, @__DIR__)
julia> using GeometricSearch
```

2.1.1. Poszczególne funkcje

2.1.1.1. Operowanie na istniejącym zbiorze punktów

2.1.1.1.1. Zbiór wczytany z pliku tekstowego

Istnieje możliwość wczytania zbioru punktów z pliku tekstowego, którego rekordy mają postać par liczb oddzielonych przecinkiem, reprezentujących współrzędne. Wczytywanie odbywa się za pomocą funkcji *readPointsFromFIle,* która jako argument przyjmuje napis będący nazwą pliku.

```
julia> points = readPointsFromFile("plik.txt")
```

2.1.1.1.2. Zbiór punktów jako lista

Funkcje wchodzące w skład pakietu przyjmują zbiór punktów jako listę dwuelementowych list zawierających współrzędne będące liczbami rzeczywistymi. Można użyć obiektu typu Array{Array{Float64, 1}, 1}, jednak pakiet zawiera również wygodne aliasy: Point jako Array{Float64, 1} oraz PointList jako Array{Point, 1}, opisane w dalszej części dokumentacji.

```
julia> points = [[1.0, 1.0], [2.0, 2.0], [3.0, 3.0]]
```

2.1.1.2. Generowanie losowego zbioru punktów

2.1.1.2.1. Rozkład jednostajny – równomierne rozmieszczenie

Funkcja *generateUniformPoints* pozwala wygenerować zbiór równomiernie rozmieszczonych punktów o współrzędnych z danego zakresu. Jako argumenty przyjmuje liczbę punktów oraz dwa zakresy liczb rzeczywistych, odpowiednio dla współrzędnej x i y, zapisane w standardowej teoriomnogościowej notacji przedziałów otwartych.

```
julia> points = generateUniformPoints(100, (0.0, 10.0), (1.0, 5.0))
```

2.1.1.2.2. Rozkład normalny – skupiska punktów

Funkcja *generateClusterPoints* pozwala wygenerować skupiska punktów, losowanych z dwuwymiarowego rozkładu normlanego. Jako argumenty przyjmuje liczbę punktów na skupisko oraz dowolną liczbę par, reprezentujących parametry rozkładu – wartość oczekiwaną, będącą punktem oraz odchylenie standardowe, będące dodatnią liczbą rzeczywistą.

```
julia> points = generateClusterPoints(100, ([0.0, 0.0], 1.0), ([1.0, 2.0], 5.0))
```

2.1.1.3. Tworzenie drzew

Poszczególne struktury dla wskazanego zbioru punktów można utworzyć poprzez wywołanie funkcji *Quadtree* lub *KDTree* z argumentem będącym listą punktów. Funkcje zwracają drzewa.

```
julia> tree = Quadtree(points)
julia> tree = KDTree(points)
```

2.1.1.4. Typy danych

Program nie wprowadza żadnych typów danych. Udostępnia jedynie aliasy, pozwalające na łatwiejszą manipulację danymi. Przyjęto następujące reprezentacje wraz z aliasami, odpowiednio dla punktu, listy punktów, przedziału oraz prostokąta:

```
Point = Array{Float64, 1}
PointList = Array{Point, 1}
Interval = Tuple{Float64, Float64}
Square = Tuple{Point, Point, Point, Point}
```

Przy czym jako punkt rozumiana jest tablica dwuelementowa. Dalsze elementy będą ignorowane, a użycie tablicy o mniejszej liczbie elementów poskutkuje nieprzewidzianymi błędami wykonania.

2.1.1.5. Struktury danych

Każde z drzew reprezentowane jest złożoną strukturą. Jest niemodyfikowalna, zawartość pól można jedynie odczytać. Z punktu widzenia użytkownika istotne są pola *depth* oraz *construction_time*, zawierające głębokość drzewa i czas jego konstrukcji. Dostarczają informacji na temat zrównoważenia i optymalności implementacji.

2.1.1.6. Deklaracja ograniczeń

Wyszukiwanie punktów odbywa się przy uprzednim zadeklarowaniu ograniczeń na ich współrzędne. Ograniczenia reprezentowane są przez obiekty klasy *Constraint*. Można tworzyć je za pomocą funkcji *Constraint*, podając jako argumenty górne i dolne ograniczenia na współrzędne x i y.

```
julia> constraint1 = Constraint(0.1, 2.5, 3.75, 4.2)
```

2.1.1.7. Wyszukiwanie punktów

Do wyszukiwania punktów spełniających podane ograniczenie służy funkcja *solveSatisfy*, która jako argumenty przyjmuje drzewo czwórkowe oraz ograniczenie. Funkcja zwraca listę punktów spełniających ograniczenie oraz czas wykonania.

```
julia> result, execution_time = solveSatisfy(tree, constraint1)
```

2.1.1.8. Wyświetlanie wyników

Wyniki wyszukiwania można wyświetlić, wraz ze wszystkimi zadanymi punktami, za pomocą funkcji *plotEverything!*, podając jako argumenty wszystkie punkty oraz te, które spełniają ograniczenie. Argumentem opcjonalnym jest ciąg znaków *filename*. Jeśli został on podany, obraz z wykresem zostanie zapisany do pliku o takiej nazwie.

```
julia> plotEverything!(points, result, "nazwa_pliku")
```

2.1.1.9. Wizualizacja budowy drzewa

Funkcja *visualize!* Umożliwia podgląd podziałów dokonanych podczas konstrukcji drzewa. Jako argument przyjmuje drzewo. Wyświetla podziały na już istniejącym wykresie.

```
julia> visualize!(tree)
```

2.1.1.10. Generowanie opisów

Dla danego wyniku poszukiwań można wyświetlić słowny opis, zawierający informację o liczbie zadanych punktów, czasie konstrukcji drzewa, liczbie znalezionych punktów oraz czasie wykonania zapytania. Służy do tego funkcja *commentResults!*. Jako argumenty przyjmuje liczbę wszystkich punktów, czas budowy drzewa, liczbę znalezionych punktów oraz czas wykonania zapytania.

```
julia> commentResult!(n, tree.construction_time, length(result), execution_time)
```

2.1.1.11. Zapis wyników

2.1.1.11.1. Wyniki wyszukiwania

Listę punktów, będącą wynikiem wyszukiwania, można zapisać do pliku tekstowego za pomocą funkcji *saveResultToFile!*, podając jako argumenty napis reprezentujący nazwę pliku, do którego ma zostać dokonany zapis oraz listę punktów. Jako argument opcjonalny można podać również ograniczenie, którego spełnialność była testowana w tym wyszukiwaniu. Umożliwia to zapisanie w pierwszej linii pliku informacji o przynależności punktów. Punkty zostaną zapisane w kolejnych liniach jako pary liczb oddzielone przecinkiem.

```
julia> saveResultToFile!("plik", result, constraint)
```

2.1.1.11.2. Wizualizacja

Wykres powstały w czasie użytkowania programu można zapisać do pliku *png* przy użyciu funkcji *savePlotToFile!*, podając jako argument nazwę pliku, do którego ma nastąpić zapis.

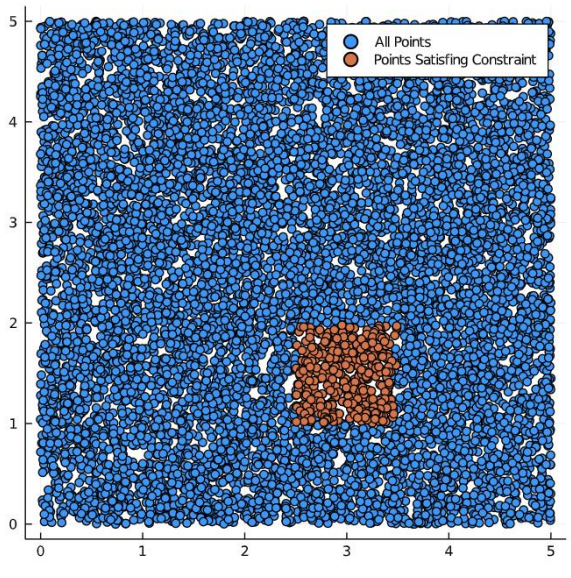
```
julia> savePlotToFile!("wykres")
```

2.1.2. Funkcja *main* – przekrojowa prezentacja działania

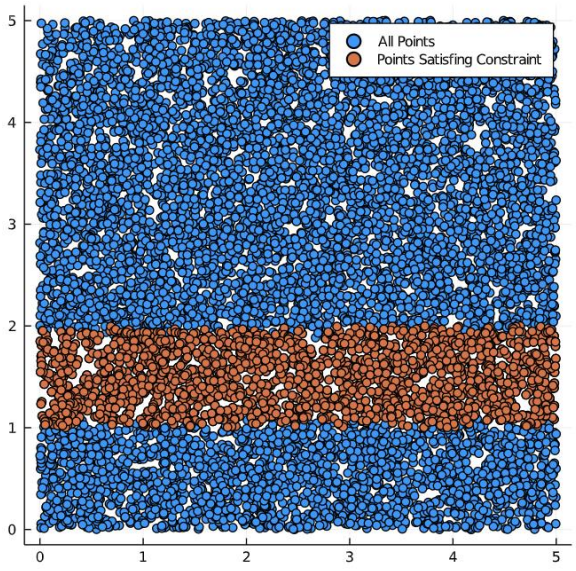
Opisane wyżej funkcje mają zastosowanie przede wszystkim zbiorczo. Ich współdziałanie obrazuje funkcja *main* z pakietu *Main.jl*. Zawiera przykładowe wywołania każdej z dostępnych funkcji. Jej wykonanie poskutkuje wykonanie wszystkich wywołanych wewnątrz funkcji. Niektóre propozycje zapisano jako komentarze, pozostawiając użytkownikowi wybór. Funkcja *main* nie przyjmuje argumentów. Parametry rozkładów, ograniczenia i inne zachowania programu należy zdefiniować wewnątrz funkcji.

```
julia> main()
```

3. Przykładowe wyniki wyszukiwania

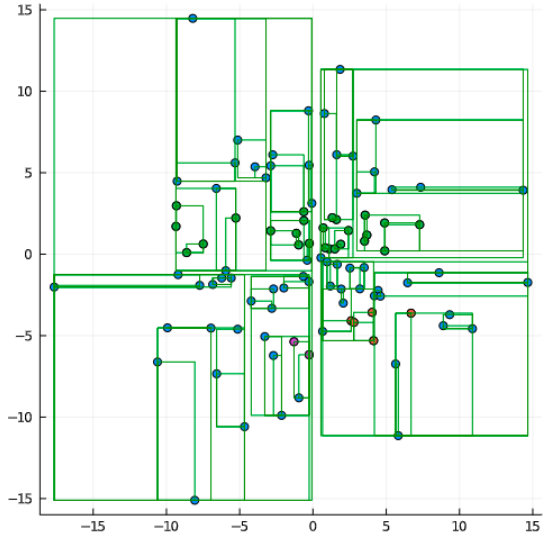


Rysunek 2 Wynik dla ograniczenia zawartego w zbiorze

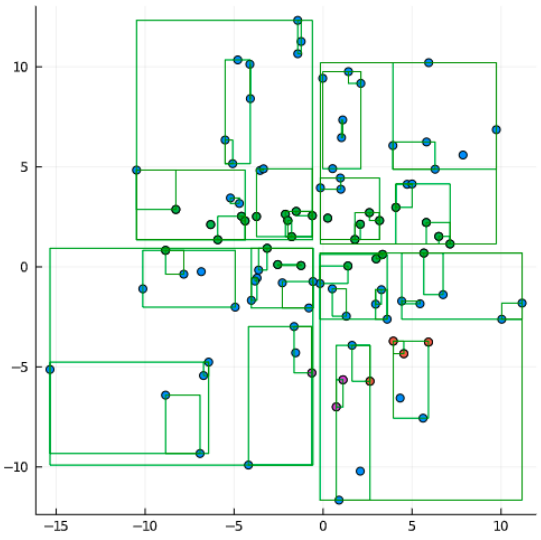


Rysunek 1 Wynik dla ograniczenia wykraczającego poza zbiór

4. Przykładowe wizualizacje podziałów



Rysunek 4 Wizualizacja podziałów w k-d drzewie



Rysunek 3 Wizualizacja podziałów w drzewie czwórkowym

5. Sprawozdanie

5.1. Badany problem

Badając wydajność implementowanych struktur, mierzono czas wyznaczenia zbioru punktów leżących wewnątrz prostokąta normalnego względem obu osi, zdefiniowanego przez podanie przedziałów, w których mieszczą się współrzędne szukanych punktów.

5.2. Testy

Testy polegały na wykonaniu tego samego zapytania na losowych zbiorach różnej liczności oraz pomiarze czasu konstrukcji drzewa i wykonania zapytania.
Wyszukiwano punkty leżące w kwadracie $[2.5, 3.5] \times [1, 2]$, z kwadratu $[1, 5]^2$.
Powtórzono te same testy na drzewach obu rodzajów.

Testy przeprowadzono na komputerze przenośnym z procesorem Intel Core i7, o częstotliwości taktowania 2 GHz. Korzystano ze środowiska programistycznego Visual Studio Code, w systemie operacyjnym Ubuntu 20.4.

5.3. Wyniki

Podsumowania wygenerowano za pomocą funkcji *commentResults!*, otrzymując poniższe raporty.

5.3.1. Drzewo czwórkowe

```
-------------------------------------------
For 100 points:
Tree construted in 0.00053 s
Constraint satisfied by 0 points
Executed in 3.0e-5 s

-------------------------------------------
For 1000 points:
Tree construted in 0.00475 s
Constraint satisfied by 40 points
Executed in 0.00033 s

-------------------------------------------
For 10000 points:
Tree construted in 0.05624 s
Constraint satisfied by 359 points
Executed in 0.00098 s

-------------------------------------------
For 100000 points:
Tree construted in 0.68192 s
Constraint satisfied by 3899 points
Executed in 0.00297 s

-------------------------------------------
For 1000000 points:
Tree construted in 9.42564 s
Constraint satisfied by 39870 points
Executed in 0.00696 s
```

Rysunek 5 Wyniki testów dla drzewa czwórkowego

5.3.2. K-d drzewo

```
-----------------------------------------
For 100 points:
Tree construted in 0.00318 s
Constraint satisfied by 5 points
Executed in 0.01708 s

-----------------------------------------
For 1000 points:
Tree construted in 0.00478 s
Constraint satisfied by 32 points
Executed in 0.00032 s

-----------------------------------------
For 10000 points:
Tree construted in 0.13557 s
Constraint satisfied by 398 points
Executed in 0.00101 s

-----------------------------------------
For 100000 points:
Tree construted in 0.70987 s
Constraint satisfied by 3913 points
Executed in 0.00282 s

-----------------------------------------
For 1000000 points:
Tree construted in 10.80414 s
Constraint satisfied by 40104 points
Executed in 0.0139 s
```

Rysunek 6 Wyniki testów dla k-d drzewa

6. Podsumowanie

Wyszukiwanie w drzewie czwórkowym przebiegło szybciej we wszystkich testach, jednak różnice są zaniedbywalne.

Dla obu struktur obserwowany jest ten sam trend czasu działania.

Nie przeprowadzano testów dla zbiorów powyżej miliona punktów. Jest to niemożliwe ze względu na rekurencyjny charakter implementacji.

7. Bibliografia

https://en.wikipedia.org/wiki/Quadtree

https://en.wikipedia.org/wiki/K-d_tree