Rok akademicki 2012/2013

Politechnika Warszawska

Wydział Elektroniki i Technik Informacyjnych

Instytut Informatyki



praca dyplomowa magisterska

inż. Bartłomiej Jańczak

Tytuł pracy dyplomowej

Opiekun pracy

prof. dr hab. Marzena Kryszkiewicz

Ocena:

Podpis Przewodniczącego

Komisji Egzaminu Dyplomowego

Kierunek: Informatyka

Specjalność: Inżynieria Systemów Informatycznych

Data urodzenia: 1988.03.06

Data rozpoczęcia studiów: 2007.10.01

Życiorys

Podpis studenta

EGZAMIN DYPLOMOWY

Złożył egzamin dyplomowy w dniu 20\_\_ r

z wynikiem

Ogólny wynik studiów:

Dodatkowe wnioski i uwagi Komisji:

STRESZCZENIE

Streszczenie pracy w języku polskim.

Słowa kluczowe:

THESIS TITLE IN ENGLISH

Summary in English.

Keywords:

Spis treści

[1. Wprowadzenie 1](#_Toc354957537)

[1.1. Przegląd literatury 2](#_Toc354957538)

[1.2. Motywacja i cel pracy 3](#_Toc354957539)

[1.3. Układ pracy 3](#_Toc354957540)

[2. Miary odległości i podobieństwa 4](#_Toc354957541)

[2.1. Metryki odległości 4](#_Toc354957542)

[2.2. Miara odległości Kosinusowej 5](#_Toc354957543)

[2.3. Wyznaczanie kosinusowego sąsiedztwa za pomocą sąsiedztwa opartego na odległości Euklidesowej 7](#_Toc354957544)

[3. Grupowanie gęstościowe 10](#_Toc354957545)

[3.1. Algorytm DBSCAN 11](#_Toc354957546)

[3.2. Wyszukiwanie k-najbliższych sąsiadów 16](#_Toc354957547)

[4. Szacowanie odległości 17](#_Toc354957548)

[4.1. Wykorzystanie nierówności trójkąta 17](#_Toc354957549)

[4.2. Wykorzystanie indeksu metrycznego 21](#_Toc354957550)

[5. Użyte algorytmy 26](#_Toc354957551)

[5.1. Zastosowanie nierówności trójkąta w algorytmach gęstościowego grupowania danych 26](#_Toc354957552)

[5.2. Zastosowanie rzutowania w algorytmach gęstościowego grupowania danych 36](#_Toc354957553)

[5.3. Zastosowanie indeksu metrycznego w algorytmie wyszukiwania k sąsiadów 37](#_Toc354957554)

[6. Szczegóły implementacji 42](#_Toc354957555)

[7. Badania eksperymentalne 42](#_Toc354957556)

[Bibliografia 17](#_Toc354957557)

# 

# 1. Wprowadzenie

Współczesne systemy komputerowe agregują i generują ogromną ilość danych zawierających cenną dla biznesu trudno odkrywalną wiedzę. Jej znajdowaniem zajmuje się dziedzina informatyki zwana odkrywaniem wiedzy. Mimo, że jest ona stosunkowo młoda, to stworzyła wiele technik eksploracji danych, które dzięki swojej skuteczności oraz wydajności znalazły szerokie praktyczne zastosowanie w rozwiązywaniu problemów związanych z analizą danych.

Zasadniczą przyczyną rychłego rozwoju odkrywania wiedzy jest spopularyzowanie wydajnych metod pozyskiwania i gromadzenia informacji. Zjawisko to nie byłoby możliwe bez postępu technologicznego w dziedzinie urządzeń agregujących dane i systemów bazodanowych oraz dzięki upowszechnieniu urządzeń umożliwiających automatyczną rejestrację sposobu ich wykorzystania. Z punktu widzenia konsumenta można tu wyszczególnić kody kreskowe, karty płatnicze oraz szeroko pojęte urządzenia mobilne. Kolejnym wartym uwagi źródłem danych jest sieć Internet, w której możliwa jest rejestracja wielu czynności korzystających z niej użytkowników, którzy ponadto dobrowolnie umieszczają w niej wiele informacji o sobie, przykładowo na portalach społecznościowych.

Wyżej wymienione zjawiska mają wpływ na osiąganie ogromnych rozmiarów przez współczesne zbiory danych. Tempo ich wzrostu jest szybsze niż przewidywano jeszcze kilka lat temu. Ich gromadzenie i przechowywanie na nośnikach pamięci masowej nie stanowi problemów dla współczesnych systemów, natomiast działanie na takiej ilości danych, pomimo stale wzrastającej mocy obliczeniowej komputerów, wciąż jest wyzwaniem dla dzisiejszej informatyki. Zbiory danych same w sobie nie stanowią wielkiej wartości, jednakże rozsądnie wykorzystane mogą stać się cennym źródłem szczególnej wiedzy. Nierzadko użyteczna wiedza ukryta jest między pewnymi składowymi danych, więc do jej odkrycia konieczne są właściwe algorytmy. Zagadnieniom tym poświęcona jest dziedzina informatyki zwana eksploracją danych, której ideą jest wykorzystanie komputera do znajdowania ukrytych dla człowieka wartościowych prawidłowości w danych zgromadzonych w dużych repozytoriach.

Odkrywanie wiedzy jest procesem złożonym, na który najczęściej składają się następujące etapy:

* analiza danych – poznanie charakteru danych i określenie celu eksploracji,
* selekcja danych – czyszczenie, weryfikacja poprawności i wybór danych, które zostaną poddane dalszej analizie,
* transformacja danych – przekształcenie danych do odpowiedniej postaci, określenie strategii wobec danych niepełnych,
* eksploracja danych – ekstrakcja wiedzy z danych,
* interpretacja wyników – logiczna i graficzna wizualizacja wyników, wybór najbardziej interesującej wiedzy, wnioskowanie.

Kluczową fazą procesu odkrywania wiedzy jest eksploracja danych. Do zasadniczych metod eksploracji danych należą:

* grupowanie,
* klasyfikacja,
* odkrywanie asocjacji,
* regresja.

Każda z metod ujawnia różnego rodzaju korelacje pomiędzy danymi, z czego wynika ich odmienne zastosowanie.

W tej pracy skoncentrowałem się na zagadnieniu grupowania danych, które określane jest jako wyznaczanie zbiorów obiektów podobnych przy zachowaniu właściwości maksymalizacji podobieństwa obiektów należących do tych samych grup i minimalizacji podobieństwa obiektów należących do innych grup.

## 1.1. Przegląd literatury

Grupowanie danych jest popularną metodą o wielu zastosowaniach, dlatego nie trudno o jej opis w literaturze. W przypadku algorytmów, na których skupiłem się w niniejszej pracy wyjątkowo przydatne okazały się artykuły naukowe.

Jednym z najpopularniejszych algorytmów gęstościowego grupowania danych jest DBSCAN [[1](#MEs66)] stanowiący często punkt odniesienia dla porównań z innymi algorytmami gęstościowych grupowań. [TODO]

Nową koncepcją zwiększenia wydajności wyżej wymienionych algorytmów jest wykorzystanie nierówności trójkąta do redukcji liczby kosztownych operacji wyznaczania podobieństwa obiektów. Na przykładzie algorytmu k-środków [[2](#CEl03)] przedstawiane już były próby wykorzystania nierówności trójkąta w algorytmach grupowania danych. Natomiast po raz pierwszy została ona użyta w celu porządkowania dostępu do danych w algorytmach gęstościowego grupowania TI-DBSCAN [[3](#Kry10)] i TI-NBC [[4](#MKr11_2)]. Dokonano również badania wpływu liczby punktów referencyjnych i strategii ich wyboru na efektywność tych algorytmów [Wawer??].

## 1.2. Motywacja i cel pracy

Grupowanie danych to proces powszechnie stosowany w porządkowaniu produktów, segmentacji klientów, organizacji obiektów czy rozpoznawaniu i analizie obrazów. Procesy te wymieniane są pośród kluczowych elementów, na których bazuje szeroko rozumiana sztuczna inteligencja. We współczesnym świecie algorytmy grupowania danych znajdują coraz szersze zastosowanie. Ich popularność rozpala zainteresowanie naukowców, którzy opracowują coraz sprawniejsze algorytmy lub modyfikują istniejące, które dotychczas wydawały się optymalne. Nierzadko zdarza się, że usprawnienia po wielokroć zwiększają wydajność dotychczasowych rozwiązań, co z kolei umożliwia przetwarzanie zbiorów danych z większą liczbą obiektów bądź atrybutów. Niekiedy może to oznaczać sposobność użycia tych algorytmów w nieosiągalnych dotychczas obszarach.

Jednym z najnowszych pomysłów na zwiększenie wydajności algorytmów grupowania danych jest zastosowanie nierówności trójkąta. [TODO]

Celem pracy jest … [TODO]

## 1.3. Układ pracy

W kolejnych rozdziałach zamiennie będę posługiwał się terminami wektor i punkt.[TODO]

# 2. Miary odległości i podobieństwa

Podobieństwo jest pojęciem fundamentalnym w niemal każdej dziedzinie naukowej. Przykładowo, w matematyce, geometryczne metody oceny podobieństwa wykorzystywane są do określania przystawania jak również w dziedzinach pokrewnych takich jak trygonometria. W biologii molekularnej ważnym problemem jest mierzenie podobieństwa par białek. Zbiory rozmyte wykształciły własne miary podobieństwa znajdujące zastosowanie na polach zarządzania, medycyny czy meteorologii. Przegląd wszystkich zastosowań podobieństwa jest samym w sobie wdzięcznym tematem na pracę dyplomową. W niniejszym rozdziale skupię się na wybranych miarach podobieństwa wektorów, tj. metrykach odległości oraz metryce podobieństwa kosinusowego.

## 2.1. Metryki odległości

Metryką odległości (lub krócej odległością) w zbiorze wektorów jest miara podobieństwa , która spełnia następujące warunki dla dowolnych wektorów , oraz w :

1. ;
2. ;
3. ;

Warto zauważyć, że istnieje wiele miar odległości. W zależności od zastosowania, w danym przypadku, jedne miary mogą być stosowniejsze niż inne. Najpopularniejszą miarą odległości jest *odległość Euklidesowa*. Odległość Euklidesowa między punktami i oznaczana jest jako , i definiowana jako:

Gdy stosowana jest odległość Euklidesowa to otoczenie punktu przyjmuje sferyczny kształt.

Innym przykładem popularnej miary odległości jest *odległość Manhattan*. Odległość Manhattan między punktami i oznaczana jest jako , i definiowana w następujący sposób:

Gdy stosowana jest odległość Manhattan to otoczenie punktu przyjmuje prostokątny kształt.

Zarówno odległość Manhattan jak i odległość Euklidesowa są szczególnymi przypadkami *odległości Minkowskiego*. Odległość Minkowskiego rzędu między punktami i oznaczana jest jako , i definiowana jako:

Dla uproszczenia, bez straty ogólności, w swoich rozważaniach będę posługiwał się odległością euklidesową jako metryką odległości.

W dalszych podrozdziałach, na podstawie [[5](#MKr12)], przedstawiono miarę odległości Kosinusowej oraz wyznaczanie kosinusowego sąsiedztwa za pomocą sąsiedztwa opartego na odległości Euklidesowej.

## 2.2. Miara odległości Kosinusowej

W wielu aplikacjach, w szczególności odkrywających wiedzę w danych tekstowych, chemii ,czy inżynierii biomedycznej, *miara podobieństwa kosinusowego* stosowana jest w celu znajdowania obiektów podobnych danemu.

W dalszej części pracy zakładam, że obiekty reprezentowane są przez wektory przestrzeni wymiarowej. Każdy wektor rozumiany jest jako sekwencja komponentów , gdzie komponent jest wartością -tego wymiaru , gdzie . Wektor o wszystkich wymiarach równych 0 będzie nazywany *wektorem zerowym*, w inny przypadku będzie nazywany *wektorem niezerowym*.

Miara podobieństwa kosinusowego między wektorami i oznaczana jest jako i definiowana w następujący sposób:

**Przykład 1**. Na Rys. 1 przedstawiono trzy wektory , i . Należy zwrócić uwagę, że odległość między i jest większa niż odległość Euklidesowa między i . Z drugiej strony, w sensie podobieństwa kosinusowego, jest bardziej podobne do niż , ponieważ kosinus kąta między i () jest większy niż kosinus kąta między i ().

W Tab. 1 zamieszczono wartości podobieństwa kosinusowego między wektorami , i . Można zauważyć, że dla podobieństwa kosinusowego warunki oraz i nie są spełnione.



Rys. 1. Odległość euklidesowa i podobieństwo kosinusowe

Tab. 1. Podobieństwo kosinusowe wektorów z rys. 1.

|  |  |
| --- | --- |
| **(u,v)** | **cosSim(u,v)** |
| (p,q) | 0,965 |
| (p,r) | 0,196 |
| (q,r) | 0,447 |

Z powyższego przykładu płyną następujące wnioski:

1. Nierówność trójkąta nie jest spełniona dla dla dowolnego zbioru wektorów.
2. Nierówność trójkąta nie jest spełniona dla dla dowolnego zbioru wektorów.
3. Nierówność trójkąta nie jest spełniona dla dla dowolnego zbioru wektorów.

Ponieważ podobieństwo kosinusowe między niezerowymi wektorami i opiera się wyłącznie na kącie zawartym między nimi i nie zależy od ich długości, stąd obliczanie może być wyznaczone w oparciu o znormalizowane wektory i , tj. i . Z powyższego spostrzeżenia wynikają poniższe własności:

1. ;
2. ;
3. .

## 2.3. Wyznaczanie kosinusowego sąsiedztwa za pomocą sąsiedztwa opartego na odległości Euklidesowej

Artykuł [[5](#MKr12)] dowodzi, że kosinusowe sąsiedztwo[[1]](#footnote-2) w zbiorze wektorów może zostać wyznaczone za pomocą odpowiedniego sąsiedztwa opartego na odległości Euklidesowej w zbiorze wektorów składającym się z α-znormalizowanych[[2]](#footnote-3) wektorów z . Stąd, autorka artykułu proponuje następujące podejście do wyznaczania sąsiedztwa opartego na podobieństwie kosinusowym.

W pierwszej kolejności początkowy zbiór wektorów transformowany jest do - zbioru α-znormalizowanych wektorów z . Następnie kosinusowe -sąsiedztwo (lub kosinusowe k-sąsiedztwo) w zbiorze ustanawiane jest jako oparte na odległości Euklidesowej -sąsiedztwo (lub alternatywnie k-sąsiedztwo) w zbiorze , gdzie . Warto zwrócić uwagę, że w przeciwieństwie do kosinusowego -sąsiedztwa, -sąsiedztwo spełnia własność nierówności trójkąta.

**Przykład 2.** W Tab. 2 zdefiniowano a na Rys. 2 przedstawiono przykładowy zbiór . Na Rys. 3 zamieszczono zbiór α-znormalizowanych wektorów z , gdzie . Warto zauważyć, że znormalizowany wektory zbioru mają długość równa , a punkty opisywane wektorami zbioru układają się na okręgu o środku w punkcie i promieniu .



Rys. 2. Przykładowy zbiór wektorów

Tab. 2. Przykładowy zbiór wektorów

|  |  |  |
| --- | --- | --- |
| **Wektor** |  |  |
|  | 4.20 | 4.00 |
|  | 5.90 | 3.90 |
|  | 2.80 | 3.50 |
|  | 1.10 | 3.00 |
|  | 0.00 | 2.40 |
|  | 2.40 | 2.00 |
|  | 1.50 | 0.50 |
|  | 1.00 | 1.50 |



Rys. 3. Zbiór wektorów α-znormalizowanych

Załóżmy, że chcemy znaleźć wektory najbardziej podobne do wektora względem kosinusowego podobieństwa. Z Rys. 2 widać, że są nimi wektory i . Analogicznie przedstawia się relacja między α-znormalizowanymi wektorami , i , tj. , i . Najbardziej podobnymi wektorami do , względem kosinusowego podobieństwa, są wektory i . Co więcej, wektory i są również najbardziej podobne do względem podobieństwa opartego na odległości euklidesowej, co zaznaczono na Rys. 3 błękitną przerywaną linią.

# 3. Grupowanie gęstościowe

Istnieje wiele rozwiązań problemu grupowania danych, czyli wyznaczania zbiorów obiektów podobnych przy zachowaniu właściwości maksymalizacji podobieństwa obiektów należących do tych samych grup i minimalizacji podobieństwa obiektów z różnych grup. Popularnym przykładem miary podobieństwa jest odległość Euklidesowa klasyfikująca obiekty leżące blisko siebie jako podobne, jednak większość algorytmów jest niezależna od przyjętej miary podobieństwa. Mnogość zastosowań grupowania, częstokroć o odmiennych wymaganiach co do rezultatu oraz specyficznych danych wejściowych (np. o różnej liczności, rozkładzie bądź liczbie atrybutów), prowadzi do dużej liczby wyspecjalizowanych algorytmów. W każdym z nich można doszukać się wad oraz zalet, jednakże nie znaleziono dotychczas uniwersalnego algorytmu. Często trudno porównywać algorytmy grupowania danych, ponieważ, ze względu na charakterystyczne podejście do rozwiązywanego problemu, różnią się one nie tylko sposobem grupowania, ale także definicją grupy.

Najpopularniejsza klasyfikacja algorytmów grupowania dzieli je na algorytmy oparte na podziale i algorytmy hierarchiczne. W przypadku pierwszej klasy kluczowym elementem jest znalezienie najlepszego podziału zbioru na z góry zadaną liczbę możliwie najbardziej jednorodnych grup. Początkowy podział odpowiednio ze zdeterminowaną strategią optymalizowany jest w kolejnych iteracjach zgodnie z przyjętą funkcją celu. Przykładami metod podziału są algorytmy k-średnich i k-medoidów. Wynikiem drugiej klasy algorytmów grupowania jest dendrogram[[3]](#footnote-4) - drzewo, które iteracyjnie dzieli zbiór danych na coraz to mniejsze podzbiory dopóki każdy podzbiór składa się z jednego obiektu. W takiej hierarchii każdy węzeł drzewa reprezentuje klaster zbioru danych. W dendrogramie, relacja między węzłami a ich przodkami odpowiada relacji między podgrupami a grupami. Dendrogramy mogą być tworzone od liści w górę do korzenia (*podejście aglomeracyjne*) lub od korzenia w dół do liści (*podejście podziału*) poprzez scalanie lub podział klastrów z każdym krokiem algorytmu. Obie wymienione klasy algorytmów grupowania posiadają pewne wady. W przeciwieństwie do algorytmów opartych na podziale algorytmy hierarchiczne nie oczekują arbitralnie zadanej liczby klastrów, jednakże wymagają zdefiniowania *warunku zakończenia* wskazującego kiedy proces podziału lub scalania powinien się zakończyć.

Wyniki wyżej wymienionych metod rzadko odpowiadają oczekiwaniom. Taki stan rzeczy można tłumaczyć nienaturalnym dla człowieka mechanizmem grupowania. Gdyby zlecić człowiekowi zadanie pogrupowania punktów dwuwymiarowej przestrzeni, to okazałoby się, że nie dzieliłby on zbioru hierarchicznie na kolejne podzbiory czy też nie próbowałby podzielić go na z góry określoną liczbę podzbiorów. Ludzie z łatwością rozpoznają klastry o dowolnych kształtach oraz szum. Głównym powodem, dla którego rozpoznajemy klastry jest fakt, iż wewnątrz każdego z nich można wyszczególnić pewną gęstość punktów znacznie wyższą niż gęstość punktów poza klastrem. Zatem, do grupy należą punkty leżące w obszarze o gęstości wyraźnie większej niż punkty leżące w obszarze otaczającym ją. Tak zdefiniowanemu pojęciu metody grupowania najbliżej jest algorytmom gęstościowym.

W podrozdziałach przedstawiono opis algorytmu DBSCAN na podstawie [[1](#MEs66)] oraz opis wyszukiwania k-najbliższych sąsiadów.

## 3.1. Algorytm DBSCAN

DBSCAN, czyli Density Based Spatial Clustering of Applications with Noise jest jednym z najpopularniejszych algorytmów gęstościowego grupowania danych. Zaproponowany w 1996 roku wciąż jest sztandarowym algorytmem gęstościowym będącym punktem odniesienia w wielu pracach naukowych dotyczących tematyki grupowania oraz prezentujących nowe rozwiązania lub algorytmy.

Oprócz grup, czyli zbioru punktów o dużej gęstości punktów, DBSCAN rozpoznaje również szum, do którego należą punkty leżące w obszarze o małej gęstości. Algorytm ten wymaga podania jedynie dwóch parametrów wejściowych, które opisują najmniejszy klaster będący obiektem zainteresowania. Jest to promień wokół danego punktu, wewnątrz którego to promienia znajduje się minimalna liczba punktów. Para parametrów i stanowi intuicyjną definicję najmniejszej gęstości, tym samym definiując minimalną liczność wykrywanych grup.

Podstawowym pojęciem używanym w kontekście algorytmu DBSCAN jest *otoczenie epsilonowe* oznaczane przez i definiowane jako zbiór takich punktów zbioru , które są różne od i nie bardziej odległe od niż , czyli:

W DBSCAN wyróżnia się dwa rodzaje punktów wchodzące w skład klastra: punkty wewnątrz grupy zwane *punktami rdzeniowymi* oraz punkty leżące na obrzeżach klastra, *punkty brzegowe*.

*Punktem rdzeniowym* nazywamy taki punkt , którego otoczenie epsilonowe zawiera wymaganą liczbę punktów, czyli:

Mówimy, że punkt jest *bezpośrednio gęstościowo osiągalny* z punktu względem i , jeżeli należy do otoczenia epsilonowego oraz jest punktem rdzeniowym:

Relacja *bezpośredniej gęstościowej osiągalności* jest symetryczna tylko dla punktów rdzeniowych. Na Rys. 4 przedstawiono klaster, w którym zaznaczono pewien punkt brzegowy , punkt rdzeniowy . Okręgami zaznaczono otoczenie epsilonowe równe , a wynosi 5. Rysunek prezentuje asymetryczny przypadek, w którym jest bezpośrednio gęstościowo osiągalny z , natomiast nie jest bezpośrednio gęstościowo osiągalny z .



Rys. 4. Przykład ilustrujący punkty rdzeniowe i brzegowe

Mówimy, że punkt jest *gęstościowo osiągalny* z punktu względem i , jeżeli istnieje sekwencja punktów takich, że jest *bezpośrednio gęstościowo osiągalny* z . Relacja ta to kanoniczne rozszerzenie relacji bezpośredniej gęstościowej osiągalności, jest tranzytywna lecz nie jest symetryczna. Z tego powodu została wprowadzona symetryczna relacja gęstościowego połączenia.

Mówimy, że punkt jest *gęstościowo połączony* z punktem względem i , jeżeli istnieje punkt taki, że i są gęstościowo osiągalne z względem i .

Na Rys. 5 przedstawiono klaster, w którym zaznaczono pewien punkt brzegowy , punkt rdzeniowy , okręgami zaznaczono otoczenia epsilonowe Eps pewnych punktów, a wynosi 5. Analiza rysunku pozwala zauważyć, że punkt jest gęstościowo osiągalny z , natomiast jest gęstościowo osiągalny z .



Rys. 5. Przykład ilustrujący relację gęstościowej osiągalności

Na Rys. 6 przedstawiono klaster, w którym zaznaczono punkty brzegowe i oraz punkt . Okręgami wyznaczono otoczenia epsilonowe punktów, a wynosi 5. Studium rysunku pozwala spostrzec, że punkty i są gęstościowo osiągalne z , czli punkty i są gęstościowo połączone.



Rys. 6. Przykład ilustrujący relację gęstościowej łączności

Wszystkie terminy niezbędne do przedstawienia gęstościowego pojęcia grupy zostały już wprowadzone. Niech jest zbiorem punktów. Grupą względem i nazywamy niepusty zbiór spełniający następujące warunki:

1. : jeśli i jest gęstościowo osiągalne z względem i , wtedy .
2. : jest gęstościowo połączone z względem i .

Niech będą grupami zbioru punków względem i , . *Szumem* nazywamy podzbiór punktów zbioru nie należących do żadnej z grup czyli:

Algorytm DBSCAN iteruje wejściowy zbiór punktów i uruchamia procedurę wyznaczania nowej grupy *ExpandCluster* dla każdego punktu, który nie został jeszcze przypisany do którejś z grup lub zidentyfikowany jako szum. *ExpandCluster* w pierwszej kolejności wyznacza otoczenie epsilonowe danego punktu i buduje nową grupę, jeśli ów punkt jest punktem rdzeniowym, w przeciwnym przypadku oznacza go jako szum. Proces tworzenia nowej grupy rozpoczyna się od dodania do niej punktów należących do otoczenia epsilonowego danego punktu. Następnie wszystkie punkty epsilonowego sąsiedztwa[[4]](#footnote-5) dodawane są do zbioru *seeds* zawierającego punkty, które potencjalnie mogą rozszerzyć budowaną grupę. Algorytm iteruje zbiór *seeds* wyznaczając epsilonowe otoczenie dla każdego jego punktu. Jeżeli dany punkt jest punktem rdzeniowym, to wszystkie punkty należące do jego otoczenia epsilonowego, nie mające przypisanej żadnej grupy również dodawane są do nowoutworzonej grupy. Te z nich, które nie są oznaczone jako szum dodawane są do zbioru *seeds*. Na Wydruk 1 wyżej opisany algorytm został zapisany w formie pseudokodu.

Analiza kodu pozwala zauważyć, że algorytm DBSCAN jest deterministyczny z dokładnością do punktów brzegowych. Nie uwzględnia on, że punkty brzegowe znajdujące się między leżącymi blisko siebie grupami mogą należeć do kilku z grup. Taka sytuacja została przedstawiona na Rys. 7.



Rys. 7. Ilustracja sytuacji, w której przynależność do jednej z grup (czerwonej bądź zielonej) punktu brzegowego zależy od kolejności w jakiej DBSCAN będzie badał punkty

**DBSCAN** (SetOfPoints, Eps, MinPts)

// SetOfPoints is UNCLASSIFIED

ClusterId := nextId(NOISE);

**for** i from 1 TO SetOfPoints.size **do**

Point := SetOfPoints.get(i);

**if** Point.ClId = UNCLASSIFIED **then**

**if** ExpandCluster(SetOfPoints, Point, ClusterId, Eps, MinPts) **then**

ClusterId := nextId(ClusterId);

**endif**;

**endif**;

**endfor**;

**end**; //DBSCAN

**ExpandCluster** (SetOfPoints, Point, ClId, Eps, MinPts) : Boolean

seeds := SetOfPoints.regionQuery(Point, Eps);

**if** seeds.size < MinPts THEN // no core point

SetOfPoints.changeClId(Point, NOISE);

**return** false;

**else** // all points in seeds are density reachable from Point

SetOfPoints.changeClId(seeds, ClId);

seeds.delete(Point);

**while** seeds <> Empty **do**

currentP := seeds.first();

result := SetOfPoints.regionQuery(currentP, Eps);

**if** result.size >= MinPts **then**

**for** I from 1 to result.size **do**

resultP := result.get(i);

**if** resultP.ClId IN (UNCLASSIFIED, NOISE) **then**

**if** resultP.ClId = UNCLASSIFIED **then**

seeds.append(resultP);

**endif**;

SetOfPoints.changeClId(resultP, ClId);

**endif**;

**endfor**;

**endif**;

seeds.delete(currentP);

**endwhile**;

**return** true;

**endif**;

**end**; //ExpandCluster

Wydruk 1. Zapis algorytmu DBSCAN w formie pseudokodu

Wynik wykonania algorytmu DBSCAN zależy od kolejności przeglądania punktów, ponieważ punkt brzegowy zakwalifikowany do pewnej grupy, w rezultacie rozbudowy kolejnych grup, może zostać przypisany do innych grup. Problem ten może zostać rozwiązany poprzez przechowywanie w każdym punkcie zamiast jednego identyfikatora *clusteId* zbioru identyfikatorów. Jednakże podobnie do autorów algorytmu, problem ten uznaję za pomijalny.

## 3.2. Wyszukiwanie k-najbliższych sąsiadów

Kolejnym podejściem do zagadnienia gęstościowego grupowania jest poszukiwanie k najbliższych sąsiadów. Poszukiwanie k najbliższych sąsiadów jest zagadnieniem optymalizacyjnym znajdowania najbliższych punktów w przestrzeni metrycznej. Problem ten definiowany jest w następujący sposób.

Niech będzie punktem zbioru , a odległość między punktami i będzie wyrażana jako . Zbiór wszystkich punktów w , które są różne od i bliższe niż będzie oznaczany przez ; czyli:

K sąsiedztwo punktu , oznaczane przez , jest definiowane jako zbiór wszystkich punktów w D, gdzie , takich, że liczba punktów różnych od i bliższych niż jest mniejsza niż ; czyli:

W większości przypadków k sąsiedztwo wyznaczane jest w n wymiarowej przestrzeni Euklidesowej a odległość mierzona jest odległością Euklidesową lub odległością Manhattan. Problem wyszukiwania najbliższych sąsiadów pojawia się na wielu polach, wśród których znajdują się: rozpoznawanie wzorców, sekwencjonowanie DNA, systemy rekomendacji, analiza skupień.

Istnieje niemało metod rozwiązań problemu k najbliższych sąsiadów. Użyteczność oraz jakość tych algorytmów determinowana jest przez złożoność czasową zapytań jak również koszt utrzymania potrzebnych struktur danych. Najprostszym z nich jest obliczanie odległości punktu zapytania do wszystkich punktów zbioru , śledząc dotychczasowo najlepszych punktów.

# 4. Szacowanie odległości

Szacowanie odległości jest przybliżonym określaniem jej wartości. Działanie to pozwala uniknąć wielokrotnego jej obliczania między pewnym wektorem a wszystkimi wektorami danego zbioru wektorów . W następujących podrozdziałach opisałem użyte przeze mnie metody szacowania odległości między wektorami. Indeks metryczny został opisany na podstawie artykułu [[6](#PYa)].

## 4.1. Wykorzystanie nierówności trójkąta

Na początek warto przypomnieć nierówność trójkąta.

**Twierdzenie 1.** Dla dowolnych wektorów , i :

Stąd, odległości i do arbitralnie wybranego wektora (czyli różnica ) zapewniają pesymistyczne oszacowanie odległości między i . Owa pesymistyczna odległość między wektorami i w odniesieniu do wektora będzie oznaczana jako , czyli:

Wektor niezbędny do wyznaczania pesymistycznego oszacowania będzie nazywany *wektorem referencyjnym*. Oczywiście, wartość pesymistycznego oszacowania zależy od wyboru wektora referencyjnego.

Załóżmy, że dla każdego rozważanego wektora odległość do punktu referencyjnego została już obliczona. W takiej sytuacji określenie pesymistycznego oszacowania odległości między wektorami i jest niezwykle szybkie, jako że wymaga jedynie odjęcia uprzednio obliczonych wartości od . Jeżeli szukane są wektory nie dalsze od danego wektora niż , to gdy pesymistyczne oszacowanie jest większe niż , wtedy odległość wektora od jest również większa niż , czyli:

W takim przypadku nie jest konieczne obliczanie odległości między wektorami i , którego złożoność zależy liniowo od liczby wymiarów, aby upewnić się, że jest większa od .

Rozważmy wektory , i takie, że . Wtedy, pesymistyczne oszacowanie odległości między wektorami i : . Stąd, implikuje . Zatem, jeśli , to bez żadnych dodatkowych obliczeń wiadomo, że , z czego wynika, że odległość między i jest większa od .

Analogicznie, rozważmy wektory , i takie, że . Wtedy, . Stąd, implikuje . Czyli, jeżeli , wtedy beż żadnych dodatkowych obliczeń wiadomo, że ., z czego wynika, że odległość między i jest większa od .

Powyższe obserwacje prowadzą do następującego wniosku.

**Wniosek 1.** Niech będzie dowolnym wektorem, a zbiorem wektorów posortowanych niemalejąco względem ich odległości do . Niech będzie dowolnym wektorem z , będzie wektorem następującym po w takim, że , a będzie wektorem poprzedzającym w takim, że . Wtedy:

1. i wszystkie wektory następujące po w nie należą do otoczenia epsilonowego w ;
2. i wszystkie wektory poprzedzające w nie należą do otoczenia epsilonowego w .

Zatem, warto jest uporządkować wektory zbioru względem odległości do punktu referencyjnego , ponieważ umożliwia to prostą eliminację potencjalnie licznego podzbioru wektorów nie należących do otoczenia epsilonowego rozpatrywanego wektora.

**Przykład 3.** Niech będzie wektorem referencyjnym o współrzędnych (0,0). Na Rys. 8 przedstawiono zbiór wektorów przestrzeni dwuwymiarowej. Tabela Tab. 3 przedstawia zbiór wektorów uporządkowany niemalejąco względem odległości jego wektorów do wektora referencyjnego . Rozważmy wyznaczenie sąsiedztwa wektora o . Odległość do wektora referencyjnego jest równa 7,07 (). Pierwszym wektorem następującym po wektorze w takim, że jest wektor . Natomiast pierwszym wektorem poprzedzającym wektor w takim, że jest wektor . Przez Wniosek 1, tylko wektory następujące po i poprzedzające w (czyli wektory , , , , ) mogą należeć do . Zatem, , , oraz są jedynymi wektorami, dla których należy obliczyć odległość do wektora w celu właściwego wyznaczenia otoczenia . Przestrzeń potencjalnych sąsiadów wektora wyznaczona w oparciu o wektor referencyjny została oznaczona na Rys. 8 jako pole ograniczone przez okręgi o środkach w . Otoczenie zostało oznaczone na Rys. 8 jako koło o środku w , pokryte szachownicą.

Tab. 3. Zbiór wektorów , wraz z odległościami do wektora i wektora referencyjnego

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Nazwa punktu** | **X** | **Y** | **odl. do R** | **odl. do P** |
| A | 3 | 1 | 3,16 | 4,47 |
| B | 3 | 4 | 5,00 | 0,24 |
| C | 5 | 4 | 6,40 | 1,00 |
| P | 5 | 5 | 7,07 | 0,00 |
| D | 2 | 7 | 7,28 | 3,61 |
| E | 8 | 1 | 8,06 | 5,00 |
| F | 6 | 6 | 8,48 | 1,41 |
| G | 6 | 8 | 10,00 | 3,16 |
| H | 9 | 5 | 10,30 | 4,00 |
| I | 9 | 9 | 12,73 | 5,66 |



Rys. 8. Zbiór wektorów

Nierówność trójkąta można również zastosować w określaniu sąsiedztwa dowolnego wektora w zbiorze wektorów ponieważ problem ten można sprowadzić do wyznaczania otoczenia epsilonowego. Dla każdego wektora , można określić wartość w taki sposób, że . Najmniejsza wartość taka, że , będzie nazywana *promieniem* .

**Twierdzenie 2.** Niech . Wtedy i jest promieniem .

**Twierdzenie 3.** Jeżeli , to .

W praktyce odległość , w zasięgu której gwarantowane jest znalezienie sąsiadów wektora , jest zmniejszana w trakcie obliczania odległości między a kolejnymi wektorami z , różnymi od .

## 4.2. Wykorzystanie indeksu metrycznego

Dany jest skończony podzbiór przestrzeni zwany *zbiorem wektorów*, zadaniem jest zlokalizowanie, dla dowolnego *zapytania* należącego do przestrzeni, elementu zbioru wektorów najbliższego zapytaniu.

Przykładowym narzędziem pozwalającym na rozwiązanie postawionego problemu jest drzewo kd. Drzewo kd jest drzewem BSP[[5]](#footnote-6) tworzonym poprzez rekurencyjną bisekcję zbioru wektorów na podstawie ich położenia względem hiperpłaszczyzny tnącej. W każdym przebiegu rekurencji zbiór wektorów dzielony jest na podzbiory względem mediany rozkładu tworzonego przez rzutowanie zbioru wektorów na k-ty wymiar. Numer wymiaru, na który dokonywane jest rzutowanie, zmienia się cyklicznie i ma taką samą wartość na danym poziomie drzewa kd. Niestety struktura ta podatna jest na przekleństwo wymiaru - gdy liczba wymiarów wzrasta, wyszukiwanie w drzewie kd szybko zaczyna odwiedzać wszystkie węzły drzewa.

Podobnie jak drzewo kd, indeks metryczny jest drzewem BSP. Każdy węzeł indeksu metrycznego dzieli przestrzeń na dwie podprzestrzenie. W procesie podziału zamiast korzystać ze współrzędnych, indeks metryczny posługuje się odległością do wybranego *wektora obserwacyjnego*. Wektory bliskie wektorowi obserwacyjnemu tworzą *lewą/wewnętrzną* podprzestrzeń, podczas gdy *prawa/zewnętrzna* podprzestrzeń składa się z dalszych wektorów. Rekurencyjne stosowanie wyżej opisanego podziału prowadzi do utworzenia drzewa binarnego. Każdy węzeł tego drzewa zawiera punkt obserwacyjny danej przestrzeni, odległość progową, na podstawie której dokonano podziału na podprzestrzenie, oraz wskazania na punkty obserwacyjne podprzestrzeni – swoich potomków.

W procesie budowy indeksu metrycznego przestrzeń metryczna dekomponowana jest przy użyciu sferycznych cięć o środkach w punktach obserwacyjnych. Rozwiązanie to kontrastuje z wykorzystaniem podziału hiperpłaszczyznami w drzewie kd. Obie metody dekompozycji zostały zilustrowane na przykładzie pewnego zbioru punktów przestrzeni dwuwymiarowej na rysunkach Rys. 9 i Rys. 10.



Rys. 9. Dekompozycja przykładowego zbioru punktów za pomocą drzewa kd



Rys. 10. Dekompozycja przykładowego zbioru punktów za pomocą indeksu metrycznego

4.2.1. Wgląd teoretyczny

Dana jest pewna przestrzeń metryczna oraz skończony podzbiór reprezentujący zbiór wektorów, wśród których wyszukiwane jest najbliższe sąsiedztwo. Dla wektora problem najbliższego sąsiedztwa sprowadza się do znalezienia wektora najmniej odległego od i należącego do . Operacja ta będzie dalej oznaczana jako . Ponieważ wektor najbliższy wektorowi może być od niego dość odległy, warto wprowadzić odległość progową , poza którą nie jesteśmy zainteresowani istnieniem sąsiadów . Należy zwrócić uwagę, że w czasie obliczania wartość może być redukowana z każdym kolejnym napotkanym bliższym sąsiadem . Wyszukiwanie sąsiedztwa ograniczane w wyżej wymieniony sposób będzie oznaczane przez .

W dalszej części rozważań załóżmy, że zasięg funkcji odległości przestrzeni jest równy przedziałowi . Ponieważ każdy metryczny zasięg może być sprowadzony do przedziału bez wpływu na relację sąsiedztwa[[6]](#footnote-7), obostrzenie to może zostać wprowadzone bez straty ogólności.

Niech będzie ograniczoną przestrzenią metryczną . Dla danego wektora oraz :

Funkcja jest symetryczna oraz spełnia nierówność trójkąta, stąd , a konsekwencją tej relacji jest implikacja . Czyli, jeśli w procesie poszukiwania napotkano już wektor w odległości od , to w dalszej części poszukiwań nie należy brać pod uwagę elementów, dla których .

Dla pewnego wektora rozważmy przeciwdziedzinę dziedziny w . Przez oznaczymy medianę dzielącą na i . Pierwszy z tych przedziałów leży wewnątrz sfery , natomiast drugi z nich składa się z punktów leżący na powierzchni oraz poza sferą. Dziedziny pierwszego i drugiego przedziału oznaczymy odpowiednio przez i . Innymi słowy wektor obserwacyjny dzieli zbiór wektorów na podzbiory (lewy/wewnętrzny) i (prawy/zewnętrzny).

Niech oznacza liczność podzbioru a liczność podzbioru . W ogólności niewiele można powiedzieć o relacji między i nie czyniąc żadnych założeń co do natury przestrzeni metrycznej. Wiadomo jednak, że podział wektorów z jest najlepszy gdy , czyli gdy nie więcej niż jeden z wektorów leży na sferze .

Na tym etapie rozważań powinno już być zrozumiałe, że jedne wektory obserwacyjne mogą być lepsze od innych. Jako przykład rozważmy dwuwymiarową przestrzeń unormowaną, w której znajduje się równomiernie rozłożony zbiór wektorów. W zadanej sytuacji należy wybrać w taki sposób aby fragment powstałego wycinka koła zajmował połowę powierzchni przestrzeni. Rozważmy trzy przykładowe wektory obserwacyjne , i . Rozmieszczenie wektorów obserwacyjnych wraz z przynależnymi im liniami podziału zilustrowano na Rys. 11. W Tab. 4znajdują się własności wektorów obserwacyjnych.



Rys. 11. Przykład rozmieszczenia wektorów obserwacyjnych

Tab. 4. Własności przykładowych wektorów obserwacyjnych

|  |  |  |
| --- | --- | --- |
| Wektor obserwacyjny | Promień linii podziału | Długość linii podziału |
|  | 0,7979 | 1,2533 |
|  | 0,5225 | 1,338 |
|  | 0,3989 | 2,5066 |

Wiemy już, że najlepszy wektor obserwacyjny , to taki, dla którego co najwyżej jeden z wektorów leży na sferze . Oczywistym jest, że prawdopodobieństwo położenia wektora na powierzchni jest proporcjonalne do powierzchni , a dla rozpatrywanego przypadku, proporcjonalne do długości linii podziału. Stąd, najlepszym z przykładowych wektorów obserwacyjnych jest wektor , dla którego linia podziału jest najkrótsza. Z powyższego przykładu płynie intuicyjny wniosek, że wektory znajdujące się blisko rogów przestrzeni są najlepszymi wektorami obserwacyjnymi.

# 5. Użyte algorytmy

Analiza złożoności algorytmów przedstawionych w rozdziale 3 prowadzi do wniosku, że największy wpływ na wydajność procesu grupowania ma obliczanie odległości między wektorami. W celu poprawienia wydajności grupowania, czyli ograniczenia liczby obliczanych odległości, można posłużyć się metodami szacowania odległości opisanymi w rozdziale 4. W niniejszym rozdziale opisałem badane przeze mnie odmiany algorytmów grupowania gęstościowego. Odmiany te zostały oparte na uprzednio opisanych algorytmach i metodach szacowania odległości.

## 5.1. Zastosowanie nierówności trójkąta w algorytmach gęstościowego grupowania danych

Użycie nierówności trójkąta jako metody szacowania odległości poprawia wydajność algorytmów gęstościowego grupowania danych. W kolejnych podrozdziałach opisałem jej zastosowanie w algorytmie DBSCAN oraz w wyszukiwaniu k-najbliższych sąsiadów.

5.1.1. Algorytm TI-DBSCAN

W artykule [[3](#Kry10)] po raz pierwszy przedstawiono wykorzystanie nierówności trójkąta jako sposobu zwiększenia wydajności algorytmu DBSCAN. Opublikowaną wersję algorytmu autorzy nazwali TI-DBSCAN. Jego pseudokod został zamieszczony na Wydruk 2 i Wydruk 3. W algorytmie tym, po znanej z DBSCAN inicjalizacji punktów i oznaczeniem ich jako nieprzypisane do żadnej z grup, obliczana jest odległość każdego z punktów do uprzednio wybranego punktu referencyjnego. Następnie w oparciu o te wartości punkty grupowanego zbioru sortowane są niemalejąco.

**algorithm TI-DBSCAN**(D, Eps, MinPts)

D’ := empty set of points;

TI-Init(D);

ClusterId := nextId(NOISE);

**for** each point p in the ordered set D starting from the first point until last point in D **do**

**if** TI-ExpandCluster(D, D’, p, ClusterId, Eps, MinPts) **then**

ClusterId := nextId(ClusterId);

**endif**;

**endfor;**

**return** D’; //D’ is clustered set of points

**end;** //TI-DBSCAN

**function TI-ExpandCluster**(D, D’, p, ClusterId, Eps, MinPts)

seeds := TI-Neighborhood(D, p, Eps);

p.neighborsNr := p. neighborsNr + |seeds|;

**if** p.neighborsNr < MinPts **then**

p.clusterId := NOISE;

**for** each point q in seeds **do**

add p to q.border; q.neighborsNr := q.neighborsNr + 1;

**endfor;**

p.border := ; move p from D to D’; **return** false;

**else**

assign ClusterId to all b in p.border;

assign ClusterId to p; assign ClusterId to all q in seeds;

**for** each point q in seeds **do**

q.neighborsNr := q.neighborsNr + 1;

**endfor;**

p.border := ; move p from D to D’;

**while** |seeds| > 0 **do**

curP := first point in seeds;

curSeeds := TI-Neighborhood(D, curP, Eps);

curP.neighborsNr := curP.neighborsNr + |curSeeds|;

**if** curP.neighborsNr < MinPts **then**

**for** each point q in curSeeds **do**

q.neighborsNr := q.neighborsNr + 1;

**endfor;**

**else**

**for** each point q in curSeeds **do**

q.neighborsNr := q.neighborsNr + 1;

**if** q.clusterId = UNCLASSIFIED **then**

assign ClusterId to q; move q from curSeeds to seeds;

**else**

delete q from curSeeds;

**endif;**

**endfor;**

assign ClusterId to all b in curP.border;

**endif;**

curP.border := ; move curP from D to D’;

delete curP from seeds;

**endwhile;**

**return** true;

**endif;**

**end;** //TI-ExpandCluster

Wydruk 2. Pseudokod algorytmu TI-DBSCAN

**function TI-Init**(D)

rPoint := selectReferencePoint();

**for** each point p in D **do**

p.clusterId := UNCLASSIFIED; p.neighborsNr := 1; p.border := ;

p.dist := distance(p, rPoint);

**end for**

sort D non-decreasingly w.r.t. p.dist;

**end;** //TI-Init

**function TI-Neighborhood**(D, p, Eps)

**return** TI-Backward-Neighborhood(D, p, Eps) TI-Forward-Neighborhood(D, p, Eps);

**end;** //TI-Neighborhood

**function TI-Backward-Neighborhood**(D, p, Eps)

seeds := {};

backwardThreshold := Eps - p.dist;

**for** each point q in the ordered set D starting from the point immediately preceding point p until first point in D **do**

**if** q.dist < backwardThreshold **then**

**break;**

**endif**;

**if** distance(q,p) Eps **then**

append q to seeds;

**endif;**

**endfor**;

**return** seeds;

**end;** // TI-Backward-Neighborhood

**function TI-Forward-Neighborhood**(D, p, Eps)

Seeds := {};

forwardThreshold := Eps + p.dist;

**for** each point q in the ordered set D starting from the point immediately following point p until the last point in D **do**

**if** q.dist > forwardThreshold **then**

**break**;

**endif;**

**if** distance(p,q) Eps **then**

append q to seeds;

**endif;**

**endfor;**

**return** seeds;

**end**; // TI-Forward-Neighborhood

Wydruk 3. Pseudokod algorytmu TI-DBSCAN

Istotną różnica między algorytmem TI-DBSCAN a DBSCAN jest zastosowanie i rozszerzenie opisanej w ?? koncepcji usuwania ze zbioru D przeanalizowanych punktów. Podejście to zakłada, że każdy punkt dodatkowo przechowuje liczbę dotychczas znalezionych sąsiadów oraz zbiór punktów brzegowych. Informacje te pozwalają usuwać z analizowanego zbioru danych D wszystkie zbadane punkty.

Dzięki zastosowanemu rozwiązaniu funkcja TI-ExpandCluster iteruje jedynie po zbiorze dotychczas niezbadanych punktów. Największą korzyścią płynącą z działania na ograniczanym zbiorze punktów występuje w procesie wyznaczania sąsiedztwa punktu, ponieważ możliwe jest uniknięcie wielu obliczeń rzeczywistych odległości między punktami. Algorytm zapewnia, że operacja wyznaczania odległości między dwoma punktami zostanie wykonana najwyżej raz. Jednakże takie zapewnienie nie przychodzi bez ceny, którą jest wzrost zapotrzebowania na pamięć oraz skomplikowania algorytmu.

Kluczową modyfikacją algorytmu TI-DBSCAN względem DBSCAN jest użycie funkcji TI-Neighborhood, która dla zadanego punktu zwraca jego epsilonowe sąsiedztwo. Wynik tej funkcji stanowi sumę teoriomnogościową zbiorów będących rezultatami wywołań funkcji TI-Backward-Neighborhood i TI-Forward-Neighborhood wyszukujących punkty należące do epsilonowego sąsiedztwa danego punktu znajdujące się w indeksie odpowiednio przed i po danym punkcie. Obie funkcje przeglądają indeks odpowiednio w tył i przód, do momentu napotkania punktu, którego odległość do punktu referencyjnego różni się od odległości badanego punktu do punktu referencyjnego o więcej niż wartość Eps. Dalsze przeglądanie punktów indeksu w danym kierunku jest zbędne, ponieważ, zgodnie z teorią przedstawioną w [rozdziale 4.1](#_4.1._Wykorzystanie_nierówności), nie należą one do epsilonowego sąsiedztwa weryfikowanego punktu. Pseudokod dotyczący omówionych funkcji zamieściłem na wydruku 3.

5.1.2. Algorytm TI-DBSCAN-REF

TI-DBSCAN-REF jest odmianą algorytmu TI-DBSCAN opisaną w artykule [[3](#Kry10)] wykorzystującą wiele punktów referencyjnych do estymacji odległości między dwoma punktami. Dodatkowe punkty referencyjne używane są tylko wtedy gdy podstawowy punkt referencyjny, względem którego posortowany jest indeks, nie wystarcza do oszacowania czy dany punkt należy do epsilonowego otoczenia badanego punktu. Rzeczywista odległość między dwoma punktami obliczana jest tylko wtedy gdy żaden z punktów referencyjnych nie pozwala na oszacowanie przynależności do otoczenia epsilonowego. Dodatkowym kosztem wynikającym z posłużenia się wieloma punktami referencyjnymi jest wyznaczanie odległości między nimi a punktami badanego zbioru.

Na Wydruk 4 zamieściłem pseudokod funkcji składających się na algorytm TI-DBSCAN-REF opisany w pracy [[3](#Kry10)] różnych od funkcji algorytmu TI-DBSCAN. Szarym zaznaczeniem wyróżniłem fragmenty pseudokodu, różne od odpowiedniego pseudokodu algorytmu TI-DBSCAN.

**function TI-REF-Init**(D)

rPoints := selectReferencePoints();

**for** each point p in D **do**

p.clusterId := UNCLASSIFIED; p.neighborsNr := 1; p.border := ;

**for** each i := 1..rPoints.size() **do**

p.dists[i] := distance(p,rPoints[i]);

**endfor**;

**end for**

sort D non-decreasingly w.r.t. p.dists[0];

**end;** //TI-REF-Init

**function TI-REF-Neighborhood**(D, p, Eps)

**return** TI-REF-Backward-Neighborhood(D, p, Eps) TI-REF-Forward-Neighborhood(D, p, Eps);

**end;** //TI-REF-Neighborhood

**function TI-REF-Backward-Neighborhood**(D, p, Eps)

backwardThreshold := Eps - p.dists[0]; seeds := {};

**for** each point q in the ordered set D starting from the point immediately preceding point p until first point in D **do**

**if** q.dists[0] < backwardThreshold **then**

**break;**

**endif**;

candidateNeighbor := **true**; i := 1;

**while** candidateNeighbor and (i |p.dists|) **do**

**if** |q.dists[i] – p.dists[i]| > Eps **then**

candidateNeighbor := **false**;

**else**

i := i + 1;

**endif**;

**endwhile;**

**if** candidateNeighbor && (distance(q,p) Eps) **then**

append q to seeds;

**endif;**

**endfor**;

**return** seeds;

**end;** // TI-REF-Backward-Neighborhood

**function TI-Forward-Neighborhood**(D, p, Eps)

forwardThreshold := Eps + p.dists[0]; seeds := {};

**for** each point q in the ordered set D starting from the point immediately following point p until the last point in D **do**

**if** q.dists[0] > forwardThreshold **then**

**break**;

**endif;**

candidateNeighbor := **true**; i := 1;

**while** candidateNeighbor and (i |p.dists|) **do**

**if** |q.dists[i] – p.dists[i]| > Eps **then**

candidateNeighbor := **false**;

**else**

i := i + 1;

**endif**;

**endwhile;**

**if** candidateNeighbor && (distance(p,q) Eps) **then**

append q to seeds;

**endif;**

**endfor;**

**return** seeds;

**end**; // TI-Forward-Neighborhood

Wydruk 4. Pseudokod funkcji algorytmu TI-DBSCAN-REF rożnych od funkcji algorytmu TI-DBSCAN

5.1.3. Algorytm TI-k-Neighborhood-Index

W artykule [[4](#MKr11_2)] zaproponowano wykorzystanie nierówności trójkąta w wyszukiwaniu k-sąsiedztwa. Opisanemu algorytmowi twórcy nadali nazwę TI-k-Neighborhood-Index. Dla k-sąsiedztwa zastosowanie nierówności trójkąta jest bardziej zawiłe niż w przypadku DBSCAN. Skomplikowanie to wynika z faktu, że na początku wykonania algorytmu promień Eps sąsiedztwa nie jest znany, w dodatku należy wyznaczyć go niezależnie dla każdego punktu w oparciu o dystans do k-tego najbliższego sąsiada.

Pierwszym krokiem algorytmu jest wyznaczenie odległości do punktu referencyjnego dla wszystkich punktów zbioru, po czym wykonywane jest jego sortowanie w porządku niemalejącym względem obliczonych odległości. Następnie główna pętla algorytmu tworzy indeks, poprzez wyznaczenie wszystkim punktom ich k-sąsiedztwa za pomocą funkcji TI-k-Neighborhood. Pierwszym etapem wyznaczania k-sąsiedztwa jest szacowanie minimalnej wartości Eps gwarantującej znalezienie k sąsiadów. Punkty znajdujące się w promieniu o oszacowanej wartości tworzą zbiór kandydatów. Faktyczny promień k-sąsiedztwa jest mniejszy bądź równy oszacowanej wartości Eps. Aby wyznaczyć rzeczywiste k-sąsiedztwo, w kolejnym etapie dokonuje się weryfikacji zbioru kandydatów. Zaczynając od badanego punktu posortowany zbiór danych iterowany jest zarówno w przód jak i w tył, do momentu napotkania punktów, których odległość od punktu referencyjnego różni się od odległości badanego punktu do punktu referencyjnego o wartość większą niż Eps. Wartość Eps aktualizowana jest za każdym razem gdy znajdowany jest nowy sąsiad, który wstawiany jest do zbioru kandydatów. Pseudokod algorytmu TI-k-Neighborhood-Index zamieściłem na Wydruk 5, Wydruk 6, Wydruk 7.

**algorithm TI-k-Neighborhood-Index**(D, k)

TI-Init(D);

**for** each point p in ordered set D starting from the first point until last point in D **do**

insert(position od point p, TI-k-Neighborhood(D, p, k))

into k-Neighborhood-Index)

**endfor**;

**end**; //TI-k-Neighborhood-Index

**function TI-Init**(D)

rPoint := selectReferencePoint();

**for** each point p in D **do**

p.dist := distance(p, rPoint);

**endfor;**

sort D non-decreasingly w.r.t. p.dist;

**end;** //TI-Init

Wydruk 5. Pseudokod algorytmu TI-k-Neighborhood-Index

**function TI-k-Neighborhood**(D, p, k)

b := p;

f := p;

backwardSearch := PrecedingPoint(D, b);

forwardSearch := FollowingPoint(D, b);

k-Neighborhood := {};

i := 0;

Find-k-Candidate-Neighbours(D, p, k, i, b, f, k-Neighborhood,

backwardSearch, forwardSearch);

p.Eps := max({e.dist | e k-Neighborhood});

Verify-k-Candidate-Neighbours-Backward(D, p, b, backwardSearch, k,

k-Neighborhood);

Verify-k-Candidate-Neighbours-Forward(D, p, f, forwardSearch, k,

k-Neighborhood);

**return** k-Neighborhood;

**end**; //TI-k-Neighborhood

**function Find-k-Candidate-Neighbours**(D, p, k, i, b, f, k-Neighborhood,

backwardSearch, forwardSearch)

**while** (backwardSearch **and** forwardSearch **and** i < k) **do**

**if** p.dis – b.dist < f.dist – p.dist **then**

dist := distance(b, p);

i := i + 1;

e := (b, dist);

insert e into k-Neighborhood holding it sorted wrt. e.dist;

backwardSearch := PrecedingPoint(D, b);

**else**

dist := distance(f, p);

i := i + 1;

e := (f, dist);

insert e into k-Neighborhood holding it sorted wrt. e.dist;

forwardSearch := FollowingPoint(D, b);

**endif**;

**endwhile**;

**while** backwardSearch **and** i < k) **do**

dist := distance(b, p);

i := i + 1;

e := (b, dist);

insert e into k-Neighborhood holding it sorted wrt. e.dist;

backwardSearch := PrecedingPoint(D, b);

**endwhile**;

**while** forwardSearch **and** i < k) **do**

dist := distance(f, p);

i := i + 1;

e := (f, dist);

insert e into k-Neighborhood holding it sorted wrt. e.dist;

forwardSearch := FollowingPoint(D, b);

**endwhile**;

**end**; //Find-k-Candidate-Neighbours

**function PrecedingPoint**(D, p)

**if** there is a point in D preceding p **then**

p := point immediately preceding p in D;

**return** **true**;

**endif**;

**return false;**

**end**; //PrecedingPoint

Wydruk 6. Pseudokod algorytmu TI-k-Neighborhood-Index

**function FollowingPoint**(D, p)

**if** there is a point in D following p **then**

p := point immediately following p in D;

**return** **true**;

**end if**;

**return false;**

**end**; //FollowingPoint

**function Verify-k-Candidate-Neighbours-Backward**(D, p, b, backwardSearch, k, k-Neighborhood)

**while** backwardSearch **and** ((p.dist – b.dist) p.Eps) **do**

dist := distance(b, p);

e := (b, dist);

**if** dist < p.Eps **then**

i := |{e k-Neighborhood | e.dist = p.Eps}|;

**if** |k-Neighborhood| - i k – 1 **then**

delete each element e with e.dist = p.Eps from k-Neighborhood;

insert e into k-Neighborhood holding it sorted wrt. e.dist;

p.Eps := max({e.dist | e k-Neighborhood});

**else**

insert e into k-Neighborhood holding it sorted wrt. e.dist;

**endif**;

**else**

**if** dist = p.Eps **then**

insert e into k-Neighborhood holding it sorted wrt. e.dist;

**endif;**

**endif;**

backwardSearch := PrecedingPoint(D, b);

**endwhile;**

**end;** //Verify-k-Candidate-Neighbours-Backward

**function Verify-k-Candidate-Neighbours-Forward**(D, p, f, forwardSearch, k, k-Neighborhood)

**while** forwardSearch **and** ((f.dist – p.dist) p.Eps) **do**

dist := distance(f, p);

e := (f, dist);

**if** dist < p.Eps **then**

i := |{e k-Neighborhood | e.dist = p.Eps}|;

**if** |k-Neighborhood| - i k – 1 **then**

delete each element e with e.dist = p.Eps from k-Neighborhood;

insert e into k-Neighborhood holding it sorted wrt. e.dist;

p.Eps := max({e.dist | e k-Neighborhood});

**else**

insert e into k-Neighborhood holding it sorted wrt. e.dist;

**endif**;

**else**

**if** dist = p.Eps **then**

insert e into k-Neighborhood holding it sorted wrt. e.dist;

**endif;**

**endif;**

forwardSearch := FollowingPoint(D, b);

**endwhile;**

**end;** //Verify-k-Candidate-Neighbours-Forward

Wydruk 7. Pseudokod algorytmu TI-k-Neighborhood-Index

5.1.4. Algorytm TI-k-Neighborhood-Index-Ref

Artykuł ?? proponuje odmianę algorytmu TI-k-Neighborhood-Index stosującą wiele punktów referencyjnych. Odmiana ta będzie dalej nazywana TI-k-Neighborhood-Index-Ref. Zastosowanie wielu punktów referencyjnych pozwala na przyspieszenie weryfikacji zbioru kandydatów do k sąsiedztwa. Usprawnienie tą metodą wcześniejszych kroków algorytmu nie jest możliwe, ponieważ bez szacunkowej znajomości Eps nie można zaostrzyć warunku wykluczania elementów ze zbioru kandydatów. Dodatkowy warunek spełnienia nierówności trójkąta został dodany w funkcjach Verify-k-Candidate-Neighbours-Backward i Verify-k-Candidate-Neighbours-Forward. Pseudokod algorytmu TI-k-Neighborhood-Index-Ref zamieściłem na wydrukach. Szarym kolorem wyróżniłem fragmenty różne od odpowiedniego pseudokodu algorytmu TI-k-Neighborhood-Index.

**function TI-REF-Init**(D)

rPoints := selectReferencePoints();

**for** each point p in D **do**

**for** each i := 1..rPoints.size() **do**

p.dists[i] := distance(p,rPoints[i]);

**endfor**;

**endfor;**

sort D non-decreasingly w.r.t. p.dists[1];

**end;** //TI-REF-Init

**function Is-Candidate-Neighbor-By-Additional-Reference-Points**(p, q,

Eps)

candidateNeighbor := **true**;

i := 2;

**while** candidateNeighbor **and** (i |p.dists|) **do**

**if** |q.dists[i] – p.dists[i]| > Eps **then**

candidateNeighbor := **false**;

**else**

i := i + 1;

**endif**;

**endwhile**;

**return** candidateNeighbor;

**end**; //Is-Candidate-Neighbor-By-Additional-Reference-Points

Wydruk 8. Pseudokod funkcji algorytmu TI-k-Neighborhood-Index-Ref różnych od funkcji algorytmu TI-k-Neighborhood-Index

**function Verify-k-Candidate-Neighbours-Forward**(D, p, f, forwardSearch, k, k-Neighborhood)

**while** forwardSearch **and** ((f.dists[1] – p.dists[1]) p.Eps) **do**

**if** Is-Candidate-Neighbor-By-Additional-Reference-Points(p, b, p.Eps)

**then**

dist := distance(f, p);

e := (f, dist);

**if** dist < p.Eps **then**

i := |{e k-Neighborhood | e.dist = p.Eps}|;

**if** |k-Neighborhood| - i k – 1 **then**

delete each element e with e.dist = p.Eps from k-Neighborhood;

insert e into k-Neighborhood holding it sorted wrt. e.dist;

p.Eps := max({e.dist | e k-Neighborhood});

**else**

insert e into k-Neighborhood holding it sorted wrt. e.dist;

**endif**;

**else**

**if** dist = p.Eps **then**

insert e into k-Neighborhood holding it sorted wrt. e.dist;

**endif;**

**endif;**

**endif;**

forwardSearch := FollowingPoint(D, b);

**endwhile;**

**end;** //Verify-k-Candidate-Neighbours-Forward

**function Verify-k-Candidate-Neighbours-Backward**(D, p, b, backwardSearch, k, k-Neighborhood)

**while** backwardSearch **and** ((p.dists[1] – b.dists[1]) p.Eps) **do**

**if** Is-Candidate-Neighbor-By-Additional-Reference-Points(p, b, p.Eps)

**then**

dist := distance(b, p);

e := (b, dist);

**if** dist < p.Eps **then**

i := |{e k-Neighborhood | e.dist = p.Eps}|;

**if** |k-Neighborhood| - i k – 1 **then**

delete each element e with e.dist = p.Eps from k-Neighborhood;

insert e into k-Neighborhood holding it sorted wrt. e.dist;

p.Eps := max({e.dist | e k-Neighborhood});

**else**

insert e into k-Neighborhood holding it sorted wrt. e.dist;

**endif**;

**else**

**if** dist = p.Eps **then**

insert e into k-Neighborhood holding it sorted wrt. e.dist;

**endif;**

**endif;**

**endif;**

backwardSearch := PrecedingPoint(D, b);

**endwhile;**

**end;** //Verify-k-Candidate-Neighbours-Backward

Wydruk 9. Pseudokod funkcji algorytmu TI-k-Neighborhood-Index-Ref różnych od funkcji algorytmu TI-k-Neighborhood-Index

## 5.2. Zastosowanie rzutowania w algorytmach gęstościowego grupowania danych

Alternatywnym sposobem usprawnienia algorytmów gęstościowego grupowania jest zastosowanie rzutowania na wymiar. Łatwo zauważyć, że dla każdego wymiaru , i każdych dwóch wektorów i prawdziwe jest, że:

Stąd, jeśli , wtedy ; czyli .

Obserwacja ta prowadzi następującego twierdzenia.

**Twierdzenie 5.** Niech będzie indeksem wymiaru , gdzie , i jest zbiorem wektorów posortowanych niemalejąco względem wartości -tego wymiaru. Niech , będzie takim wektorem następującym po w , że i będzie takim wektorem poprzedzającym w , że . Wtedy:

* i wszystkie wektory następujące po w nie należą do ,
* i wszystkie wektory poprzedzające w nie należą do .

Zgodnie z twierdzeniem 2 opisanym w rozdziale 4.1 twierdzenie 5 można zastosować również do wyszukiwania k sąsiedztwa.

5.2.1. Algorytm DBSCAN-PROJECTION

Modyfikacja przedstawionego uprzednio algorytmu TI-DBSCAN stosująca rzutowanie na wymiar będzie w dalszej części pracy nazywana DBSCAN-PROJECTION. W stosunku do bazowego algorytmu zmianie ulega jedynie funkcja inicjalizująca dane, której pseudokod przedstawiłem na Wydruk 10. Kolorem szarym wyróżniłem fragmenty różne od odpowiedniego kodu algorytmu TI-DBSCAN.

**function Init**(D)

pDimension := selectProjectionDimension();

**for** each point p in D **do**

p.clusterId := UNCLASSIFIED; p.neighborsNr := 1; p.border := ;

p.dist := p[pDimension];

**end for**

sort D non-decreasingly w.r.t. p.dist;

**end;** //TI-Init

Wydruk 10. Pseudokod funkcji algorytmu DBSCAN-PROJECTION różnych od funkcji algorytmu TI-DBSCAN

5.2.2. Algorytm k-Neighborhood-Index-Projection

W dalszej części pracy modyfikacja przedstawionego uprzednio algorytmu TI-k-Neighborhood-Index stosująca rzutowanie na wymiar będzie nazywana k-Neighborhood-Index-Projection. Podobnie jak w przypadku DBSCAN-PROJECTION zmianie uległa jedynie funkcja inicjalizująca dane, której pseudokod przedstawiłem na ??. Kolorem szarym wyróżniłem fragmenty różne od odpowiedniego kodu algorytmu TI-k-Neighborhood-Index.

**function Init**(D)

pDimension := selectProjectionDimension();

**for** each point p in D **do**

p.dist := p[pDimension];

**end for**

sort D non-decreasingly w.r.t. p.dist;

**end;** //TI-Init

Wydruk 11. Pseudokod funkcji algorytmu k-Neighborhood-Index-Projection różnych od funkcji algorytmu TI-k-Neighborhood-Index

## 5.3. Zastosowanie indeksu metrycznego w algorytmie wyszukiwania k sąsiadów

Zastosowanie indeksu metrycznego pozwala na sprawne wyszukiwanie najbliższego sąsiada danego wektora w pewnym zbiorze wektorów. W oparciu o implementację zaproponowaną w artykule ?? stworzyłem algorytm gęstościowego grupowania danych poszukujący k najbliższych sąsiadów, który wykorzystuje indeks metryczny. Algorytm ten nazwałem k-Neighborhood-Index-Vp-Tree, jego pseudokod zamieściłem na wydrukach Wydruk 12, Wydruk 13 i Wydruk 14.

Pierwszym krokiem algorytmu jest budowa indeksu metrycznego w oparciu o zbiór wektorów D. Korzeń indeksu odnosi się do całej rozpatrywanej przestrzeni wektorów. Jego punkt obserwacyjny dzieli przestrzeń na lewą i prawą podprzestrzeń, które odpowiadają lewemu i prawemu potomkowi korzenia. Każdy kolejny węzeł drzewa nawiązuje do coraz to mniejszych podprzestrzeni. Niech najmniejsza wartość większą od mediany będzie nazywana prawym ograniczeniem, a największą wartość mniejszą od mediany będzie nazywana lewym ograniczeniem. W przedstawionym algorytmie w węźle indeksu metrycznego oprócz wektora obserwacyjnego przechowywana jest wartość mediany oraz lewe i prawe ograniczenie w celu opisania metrycznej relacji między punktem obserwacyjnym a lewą i prawą podprzestrzenią. Algorytm budowy indeksu metrycznego korzysta z funkcji *select\_vp*, której celem jest obieranie lepszych niż losowe wektorów obserwacyjnych. Funkcja ta w sposób stochastyczny konstruuje zbiór kandydatów do wektora obserwacyjnego. Następnie, dla każdego wektora zbioru kandydatów losowo konstruowany jest podzbiór przestrzeni, dla którego wyznaczana jest mediana oraz odchylenie standardowe. Spośród zbioru kandydatów , wybierany jest ten o największym odchyleniu standardowym.

Kolejnym krokiem algorytmu k-Neighborhood-Index-Vp-Tree jest wyszukanie k sąsiadów, przy użyciu uprzednio zbudowanego indeksu, dla każdego wektora zbioru zapytań Q. K sąsiedztwo przechowywane jest w formie kolekcji par: (odległość między wektorem p a zapytaniem q; wektor p). Dla każdego wektora zbioru zapytań Q kolekcja k sąsiedztwa   
*k-Neighborhood* inicjalizowana jest k parami (maksymalna wartość odległości; null), a następnie przekazywana jest wraz zapytaniem i korzeniem indeksu metrycznego do funkcji *search-k-Neighborhood*. Funkcja ta przegląda w głąb indeks metryczny w poszukiwaniu dla zapytania k sąsiedztwa, które zapisywane jest w kolekcji *k-Neighborhood*.

W stworzonym algorytmie proponuję dwie metody przeszukiwania indeksu metrycznego w poszukiwaniu k sąsiedztwa. Pierwszą, która korzysta z mediany jako z kryterium przeszukiwania poddrzew indeksu. Jej pseudokod przedstawiłem na Wydruk 13. Oraz drugą, która wykorzystuje ograniczenie górne i dolne w celu wyboru poddrzewa do przeszukania. Jej pseudokod zamieściłem na Wydruk 14.

**algorithm k-Neighborhood-Index-Vp-Tree** (D, Q, k)

vp\_tree := make-Vp-Tree(D);

**for** each point q in Q **do**

k-Neighborhood := ;

**for** i := 1..k **do**

insert(MAX\_DOUBLE, null) into k-Neighborhood;

**endfor**;

search-k-Neighborhood(vp\_tree, q, k-Neighborhood)

insert(position of point q, k-Neighborhood) into k-Neighborhood-Index)

**endfor**;

**end**; //k-Neighborhood-Index-Vp-Tree

**function make-Vp-Tree**(D)

**if** **then**

**return** ;

**endif**;

vantage\_potint := select-Vp(D);

new(node);

node.p := vantage\_potint;

node.μ :=;

node.leftLimit := ;

node.rightLimit := ’

:= {dD - vantage\_point | distance(vantage\_point,d) };

:= {dD - vantage\_point | distance(vantage\_point,d) };

node.left := make-Vp-Tree();

node.right := make-Vp-Tree();

**return** node;

**end;** //mave-Vp-Tree

**function select-Vp**(D)

P := random sample of D;

best\_spread := 0;

best\_vp := null;

**for** each point p in P **do**

S := random sample of D;

μ := ;

spread := ;

**if** spread > best\_spread **then**

best\_spread := spread;

best\_p := p;

**endif;**

**endfor;**

**return** best\_p;

**end; /**/select-Vp

Wydruk 12. Pseudokod funkcji algorytmu k-Neighborhood-Index-Vp-Tree

**function search-k-Neighborhood**(vp\_tree\_p, p, k-Neighborhood)

**if** p is null **then**

**return**;

**endif**;

tau := get distance to the most distant neighbor from k-Neighborhood;

dist := distance(vp\_tree\_p.p, p);

leftBoundary := dis – tau;

rightBoundary := dist + tau;

**if** dist tau **do**

erase the most distant neighbor from k-Neighborhood;

insert(dist, vp\_tree\_p.p) into k-Neighborhood;

**endif**;

**if** rightBoundary vp\_tree\_p.μ **then**

search-k-Neighborhood(vp\_tree\_p.right, p, k-Neighborhood);

**endif**;

**if** leftBoundary vp\_tree\_p.μ **then**

search-k-Neighborhood(vp\_tree\_p.left, p, k-Neighborhood);

**endif**;

**end**; //search-k-Neighborhood

Wydruk 13. Pseudokod funkcji wyszukującej k sąsiadów danego punktu w indeksie metrycznym z wykorzystaniem mediany

**function search-k-Neighborhood**(vp\_tree\_p, p, k-Neighborhood)

**if** p is null **then**

**return**;

**endif**;

tau := get distance to the most distant neighbor from k-Neighborhood;

dist := distance(vp\_tree\_p.p, p);

leftBoundary := dis – tau; rightBoundary := dist + tau;

**if** dist tau **do**

erase the most distant neighbor from k-Neighborhood;

insert(dist, vp\_tree\_p.p) into k-Neighborhood;

**endif**;

**if** leftBoundary vp\_tree\_p.leftLimit **then**

**if** rightBoundary vp\_tree\_p.rightLimit **then**

leftBuffer := vp\_tree\_p.leftLimit – leftBoundary;

rightBuffer := rightBoundary – tree.rightLimit;

**if** leftBuffer < rightBuffer **do**

search-k-Neighborhood(vp\_tree\_p.left, p, k-Neighborhood);

search-k-Neighborhood(vp\_tree\_p.right, p, k-Neighborhood);

**else**

search-k-Neighborhood(vp\_tree\_p.right, p, k-Neighborhood);

search-k-Neighborhood(vp\_tree\_p.left, p, k-Neighborhood);

**endif**;

**else**

search-k-Neighborhood(vp\_tree\_p.left, p, k-Neighborhood);

**endif**;

**else**

**if** rightBoundary vp\_tree\_p.rightLimit **then**

search-k-Neighborhood(vp\_tree\_p.right, p, k-Neighborhood);

**endif**;

**endif**;

**end**; //search-k-Neighborhood

Wydruk 14. Pseudokod funkcji wyszukującej k sąsiadów danego punktu w indeksie metrycznym z wykorzystaniem prawego i lewego ograniczenia

# 6. Szczegóły implementacji

W niniejszym rozdziale przedstawię zaprojektowane i stworzone przeze mnie oprogramowanie służące do analizy algorytmów. Oprogramowanie to jest przeznaczone dla systemu operacyjnego Microsoft Windows i implementuje wszystkie algorytmy omówione w poprzednich rozdziałach.

**Architektura**

Zasadniczym celem implementacji jest analiza oraz testy opisanych uprzednio algorytmów, ze szczególnym zwróceniem uwagi na możliwość usprawnienia ich przez zastosowanie nierówności trójkąta lub odległości kosinusowej. Mnogość odmian implementowanych algorytmów wymusiła zastosowanie rozbudowanej hierarchii dziedziczenia w architekturze silnik algorytmów. Dziedziczenie umożliwiło uniknięcie powtarzania wspólnego kodu w implementacjach modyfikacji algorytmów, co pozwoliło na ujednolicenie wspólnych funkcji i procesów, a co najistotniejsze zmniejszyło ryzyko wpływu różnic implementacyjnych na badaną wydajność algorytmu.

**Wykorzystane technologie**

Oprogramowanie zostało napisane w całości w języku C++ w środowisku Microsoft Visual Studio. Język C++ łączy zaletę obiektowości z wydajnością, nie występują w nim niekontrolowane operacje czyszczenia pamięci (ang. garbage collection) jak w Java, które w przypadku algorytmów operujących na dużych ilościach danych mogą zakłócić wyniki testów. W celu zapewnienia efektywnej realizacji operacji przetwarzania danych takich jak wczytywanie, implementacja algorytmów, funkcji pomocniczych i zapisywanie wykorzystywana jest standardowa biblioteka szablonów STL.

**Interfejs użytkownika**

# 7. Badania eksperymentalne

# Bibliografia

x

|  |  |
| --- | --- |
| [1] | Ester M., Kriegel H.P., Sander J., and Xu X., "A Density-Based Algorithm of Discovering Clusters in LargeSpatial Database with Noise," , Portland, 1966. |
| [2] | Elkan C., "Using the triangle inequality to accelerate k-Means," 2003. |
| [3] | Kryszkiewicz M. and Lasek P., "TI-DBSCAN: Clustering with DBSCAN by Means of the Triangle Inequality," 2010. |
| [4] | Kryszkiewicz M. and Lasek P., "A Neighborhood Based Clustering by Means of the Triangle Inequality and Reference Points," 2011. |
| [5] | Kryszkiewicz M., "Determining Cosine Similarity Neighborhoods by Means of the Euclidean Distance," 2012. |
| [6] | Kryszkiewicz M., "Efficient Determination of Neighborhoods Defined in Terms of Codine Similarity Measure," 2011. |
| [7] | Yanilos P., "Data Struvtures and Algorithms of Nearest Neighbor Search in General Metric Spaces,". |

x

1. kosinusowe sąsiedztwo jest sąsiedztwem opartym na podobieństwie kosinusowym [↑](#footnote-ref-2)
2. α-znormalizowany wektor to znormalizowany wektor przemnożony przez stałą α [↑](#footnote-ref-3)
3. Dendrogram to diagram stosowany do prezentacji związków między elementami lub grupami elementów w kształcie przypominający drzewo. [↑](#footnote-ref-4)
4. Epsilonowe sąsiedztwo jest epsilonowym otoczeniem danego punktu bez niego samego. [↑](#footnote-ref-5)
5. BSP – (ang. Binary Search Partitioning) metoda dokonująca rekurencyjnego podziału przestrzeni na podprzestrzenie za pomocą hiperpłaszczyzn. Podział ten tworzy reprezentację obiektów w przestrzeni zwaną drzewem BSP. Wyszukiwanie w drzewie BSP jest wyszukiwaniem binarnym. [↑](#footnote-ref-6)
6. Ograniczone przestrzenie metryczne mogą zostać w prosty sposób przeskalowane. Nieograniczone przestrzenie metryczne mogą zostać dostosowane dzięki zastosowaniu wzoru: [↑](#footnote-ref-7)