Raport

Bartłomiej Jańczak

# Opis zadania

1. Wyszukiwanie Euklidesowego otoczenia Eps i k sąsiadów dla wszystkich (lub zadanej liczby) wektorów i wszystkich (lub zadanej liczby) wymiarów:
   1. bez wykorzystania nierówności trójkąta
   2. z sortowaniem zbioru wektorów/indeksu na zbiór wektorów oraz:
      1. z wykorzystaniem nierówności trójkąta i 1 lub więcej wektorów referencyjnych (np. [0,0,0,1,0,0], [0,0,0,1/2,0,0], [0,0,0,1 1/2,0,0], [0,0,0,10,0,0]),
      2. z wykorzystaniem rzutu(ów) na os(ie),
      3. z wykorzystaniem wektorów referencyjnych i dodatkowych rzutów na osie,
2. Wyszukiwanie kosinusowego otoczenia Eps i k sąsiadów dla wszystkich (lub zadanej liczby) wektorów i wszystkich (lub zadanej liczby) wymiarów poprzez przekształcenie oryginalnego problemu do problemu z odległością Euklidesowa:
   1. bez wykorzystania nierówności trójkąta,
   2. z sortowaniem zbioru wektorów/indeksu na zbiór wektorów oraz:
      1. z wykorzystaniem nierówności trójkąta i 1 lub więcej wektorów referencyjnych (np. [0,0,0,1,0,0], [0,0,0,1/2,0,0], [0,0,0,1 1/2,0,0], [0,0,0,10,0,0]),
      2. z wykorzystaniem rzutu(ów) na os(ie),
      3. z wykorzystaniem wektorów referencyjnych i dodatkowych rzutów na osie
3. Klasyfikacja KNN z sortowaniem zbioru wektorów/indeksu na zbiór wektorów oraz:
   1. z uplasowaniem się liniowym/binarnym oraz:
      1. wykorzystaniem punktu/ów referencyjnych do szacowania odległości,
      2. wykorzystaniem rzutu/(ów) na os do szacowania odległości,
   2. bez uplasowania się oraz:
      1. z wykorzystaniem punktu/ów referencyjnych do szacowania odległości,
      2. z wykorzystaniem rzutu/ów na oś do szacowania odległości,
4. Wyszukiwanie euklidesowego/cosinusowego otoczenia Eps i k sąsiadów dla wszystkich (lub zadanej liczby) wektorów i wszystkich (lub zadanej liczby) wymiarów przy pomocy drzewa vantage point:
   1. z wykorzystaniem mediany jako kryterium podziału,
   2. z wykorzystaniem sąsiednich wartości mediany jako kryterium podziału.

# Założenia

//TODO?

# Opis formy danych wejściowych i wyjściowych

Na dane wejściowe składają się 3 rodzaje plików:

* Plik parametrów uruchomienia aplikacji – plik tekstowy zawierający rekordy postaci „*nazwa\_parametru*=*wartość\_parametru*”,
* Plik parametrów uruchomienia algorytmu – plik tekstowy zawierający rekordy postaci „*nazwa\_parametru*=*wartość\_parametru*”,,
* Plik danych wejściowych – plik tekstowy zawierający zbiór wektorów w formacie gęstym lub rzadkim.

Na dane wyjściowe składają się 3 rodzaje plików:

* Raport wykonania uruchomienia algorytmu – plik tekstowy zawierający:
  + wartości parametrów, z którymi uruchomiono algorytm,
  + czasy wykonania poszczególnych kroków algorytmu,
  + wyniki wykonania algorytmu.
* Raport wykonania uruchomienia aplikacji – plik \*.csv, zawierający dane raportów z wykonania uruchomień algorytmów z wyłączeniem wyników wykonania algorytmu. Każdy rekord pliku \*.csv odpowiada pewnemu raportowi z uruchomienia algorytmu w ramach jednego uruchomienia aplikacji.
* Oczyszczony raport wykonania uruchomienia aplikacji – plik \*.csv, zawierający dane raportów z wykonania uruchomień algorytmów z wyłączeniem wyników wykonania algorytmu. Każdy rekord pliku \*.csv odpowiada serii (ilość powtórzeń) uruchomień algorytmu. Uruchomienie algorytmu opisane jest plikiem parametrów uruchomienia algorytmu. Wielkość serii opisana jest parametrem **test\_repeats** w pliku parametrów uruchomienia aplikacji. W rekordzie pliku \*.csv każda wartość odpowiadająca czasowi wykonania pewnego kroku K algorytmu A w serii S jest tworzona w następujący sposób: spośród wszystkich wartości czasowych wykonania K w A w S usuwane są jedna maksymalna i jedna minimalna wartość czasowa, wartość średnia z pozostałych czasowych wartości stanowi wartość czasową wykonania pewnego kroku K algorytmu A w rekordzie pliku \*.csv.

# Dokumentacja użytkowa

Rozdział ten zawiera opisy:

1. struktury plików, z których korzysta implementacja,
2. pliku parametrów uruchomienia aplikacji,
3. pliku parametrów uruchomienia algorytmu,
4. znaczenia czasów wykonania dla uruchomień poszczególnych algorytmów,
5. przygotowanych zbiorów danych,
6. uruchomienia aplikacji.

## Struktura plików

Do dokumentacji załączono plik *Implementacja.zip* o następującej strukturze folderów:

* Implementacja:
  + algorithms\_engine\_properties
  + code
    - hdr
    - src
  + datasets
  + logs
  + properties

Zawartość folderów opisuje Tabela 1 .

|  |  |
| --- | --- |
| Nazwa folderu | Zawartość folderu |
| implementacja | Foldery **algorithms\_engine\_properties**, code, **datasets**, **logs**, **properties** oraz plik wykonywalny implementacja.exe. |
| algorithms\_engine\_properties | Plik parametrów **algorithms\_engine\_properties.txt**. Opis pliku parametrów uruchomienia aplikacji znajduje się w rozdziale [Rozdział 4.2](#_Plik_parametrów_uruchomienia_1). |
| code | Foldery **hdr** i **src**. |
| hdr | Pliki nagłówkowe **\*.h**. |
| src | Pliki źródłowe **\*.cpp**. |
| datasets | Pliki ze zbiorami danych **\*.txt**. |
| logs | Pliki raportów wykonania algorytmu **\*.txt**, oraz pliki raportów wykonania aplikacji **\*.csv**. Krótki opis plików raportu znajduje się w rozdziale [Rozdział 3](#_Opis_formy_danych). |
| properties | Pliki parametrów uruchomienia algorytmu. Opis pliku parametrów uruchomienia algorytmu znajduje się w rozdziale [Rozdział 4.3](#_Plik_parametrów_uruchomienia). |

Tabela 1: Opis zawartości folderów pliku implementacja.zip

Plik wykonywalny *implementacja.exe* jest programem implementującym odmiany algorytmów, spełniającym wymagania wymienione w rozdziale [Rozdział 1](#_Opis_zadania). Program skompilowany jest dla środowiska WIN32 z podniesioną flagą LARGEADDRESSAWARE, która umożliwia aplikacji użycie do 3GB pamięci wirtualnej. Uruchomienia wykonywane są na środowisku Windows 7 x64 z procesorem Intel® i7™ 950 z dostępną pamięcią RAM równą 6GB.

## Plik parametrów uruchomienia aplikacji

W folderze **algorithms\_engine\_properties** znajduje się plik parametrów uruchomienia aplikacji *algorithms\_engine\_properties.txt*. Tabela 2 opisuje znaczenie poszczególnych parametrów pliku parametrów.

|  |  |
| --- | --- |
| Nazwa parametru | Opis |
| alfa | Wartość współczynnika, przez który przemnażane są wartości wymiarów znormalizowanych wektorów (Normalizacja wektorów wykonywana jest w fazie przetwarzania danych gdy stosowana jest odległość kosinusowa). |
| test\_repeats | Liczba uruchomień algorytmu opisanego plikiem parametrów uruchomienia algorytmu. Parametr ten pozwala na wielokrotne powtórzenie serii uruchomień. |

Tabela 2: Opis znaczenia parametrów pliku algorithms\_engine\_properties.txt

## Plik parametrów uruchomienia algorytmu

W folderze **properties** znajdują się pliki uruchomienia algorytmu. Każdy plik parametrów definiuje uruchomienie algorytmu. Program kolejno wczytuje pliki parametrów i dla każdego z nich wykonuje zdefiniowany algorytm. Tabela 3 opisuje znaczenie poszczególnych parametrów pliku parametrów.

|  |  |
| --- | --- |
| Nazwa parametru | **Opis** |
| algorithm\_name | Nazwa algorytmu przyjmująca wartości: *\****dbscan** – algorytm DBSCAN. Wymaga zdefiniowania parametrów: *eps*, *min\_pts*.  **\*dbscan\_points\_elimination** – algorytm DBSCAN z eliminacją rozpatrzonych punktów. Wymaga zdefiniowania parametrów: *eps*, *min\_pts*.  \***ti\_dbscan** – algorytm TI-DBSCAN. Wymaga zdefiniowania parametrów: *eps*, *min\_pts*,  *reference\_point*, *reference\_point\_format*, *use\_dataset\_index\_access*.  \***ti\_dbscan\_ref** – algorytm TI-DBSCAN wykorzystujący wiele punktów referencyjnych do szacowania odległości. Wymaga zdefiniowania parametrów: *eps*, *min\_pts*,  *reference\_point*, *reference\_point\_format, use\_dataset\_index\_access*.  \***ti\_dbscan\_ref\_projection** – algorytm TI-DBSCAN wykorzystujący wiele punktów referencyjnych oraz rzuty na osie do szacowania odległości.  Wymaga zdefiniowania parametrów: *eps*, *min\_pts*, *reference\_point*, *reference\_point\_format*, *use\_dataset\_index\_access,* *projection\_dimensions*, *projection\_source\_sequence*. \***k\_neighborhood** – algorytm K-NEIGHBORHOOD. Sąsiedzi wyznaczani metodą brute-force liczenia odległości na zasadzie ‘każdy z każdym’. Wymaga zdefiniowania parametrów: *k*, *use\_dataset\_index\_access, classification\_subset\_factor*, *use\_binary\_placement*.  \***ti\_ k\_neighborhood** – algorytm TI-K-NEIGHBORHOOD. Wymaga zdefiniowania parametrów: *k*, *reference\_point*, *reference\_point\_format*, *use\_dataset\_index\_access, classification\_subset\_factor*, *use\_binary\_placement*.  \***ti\_k\_neighborhood\_ref** – algorytm TI-K-NEIGHBORHOOD wykorzystujący wiele punktów referencyjnych do szacowania odległości. Wymaga zdefiniowania parametrów: *k*, *reference\_point*, *reference\_point\_format, use\_dataset\_index\_access, classification\_subset\_factor*, *use\_binary\_placement*. \***ti\_k\_neighborhood\_ref\_projection** – algorytm TI-K-NEIGHBORHOOD wykorzystujący wiele punktów referencyjnych oraz rzuty na osie do szacowania odległości. Wymaga zdefiniowania parametrów: *k*, *reference\_point*, *reference\_point\_format*, *use\_dataset\_index\_access, classification\_subset\_factor*, *use\_binary\_placement*, *projection\_dimensions*, *projection\_source\_sequence*. \***vp\_tree** – algorytm wyszukujący k najbliższych sąsiadów. Buduje a następnie przeszukuje VANTAGE POINT TREE. Wymaga zdefiniowania parametrów: *k*, *p\_sample\_index*, *s\_sample\_index*, *classification\_subset\_factor*, *use\_binary\_placement,*  *search\_method,* use\_boundaries. |
| eps | Promień sąsiedztwa. |
| min\_pts | Minimalna liczba punktów w grupie. |
| k | Liczba sąsiadów. |
| use\_cosine\_similarity | Flaga definiująca rodzaj wyszukiwanego otoczenia przyjmująca wartości:  *\****true** – jeśli wyszukiwanie kosinusowego otoczenia, *\****false** – jeśli wyszukiwanie euklidesowego otoczenia. |
| dataset\_file\_format | Flaga definiująca format wczytywanych danych przyjmująca wartości:  *\****dense** – jeśli wczytywane wektory zapisane są w formacie gęstym,  *\****sparse** – jeśli wczytywane wektory zapisane są w formacie rzadkim. |
| dataset\_file\_path | Ścieżka dostępu do pliku ze zbiorem punktów, plik powinien się znajdować w katalogu **datasets**. |
| dataset\_dimension | Liczba definiująca największy wymiar każdego punktu jaki zostanie wczytany do pamięci programu. Wymiary punktu większe niż *dataset\_dimension* nie zostaną wczytane do pamięci programu. |
| dataset\_dimension\_value\_treshold | Współczynnik wyznaczający poziom istotności wartości wymiaru punktu. Jeśli wartość dowolnego wymiaru punktu będzie niższa od *dimension\_value\_treshold* to temu wymiarowi zostanie przypisana wartość 0. |
| dataset\_elements\_number | Liczba punktów zbioru *dataset\_file\_path*, które zostaną wczytane do pamięci programu jako zbiór punktów. |
| dataset\_internal\_format | Flaga definiująca wewnętrzny sposób przechowywania punktów w programie przyjmująca wartości: \***dense** – punkt przechowywany jest w formacie gęstym,  \***sparse** – punkt przechowywany jest w formacie rzadkim |
| reference\_point | Definicja punktów referencyjnych. Punkty referencyjne mogą być definiowane zarówno w formacie gęstym jak i rzadkim. Format według, którego *reference\_point*  będzie parsowany przechowywany jest w parametrze *reference\_point\_format.* Punkty referencyjne przechowywane są w programie w formacie zgodnym z parametrem *internal\_representation.*  Funkcje:  **[max]** – punkt referencyjny o maksymalnych wartościach poszczególnych wymiarów w danym zbiorze danych. ([max, max, …, max])  **[min]** – punkt referencyjny o minimalnych wartościach poszczególnych wymiarów w danym zbiorze danych. ([min, min, …, min])  **[n]** – punkt referencyjny postaci [n,n,…,n]  **[(a,b)\*]** – punkt referencyjny, tworzony na zasadzie powtarzania wzorca aż do maksymalnego wymiaru. Przykładowo dla: [(a,b)\*] punkt referencyjny byłby postaci [a,b,a,b,a,b,a,b,a,b,…],  a dla [(a,b,c,d)\*] punkt referencyjny byłby postaci [a,b,c,d,a,b,c,d,a,b,c,d,a,b,c,d,…]. **[rand]** – punkt referencyjny o losowych i nie większych niż maksymalne wartościach poszczególnych wymiarów w danym zbiorze danych.  **max** – maksymalna wartość wymiaru w danym zbiorze danych.  Przykładowe użycie dla przestrzeni dwuwymiarowej:  format gęsty:  [max,1] ,  [max,max] ⬄ [max].  format rzadki: (1 odpowiada pierwszemu wymiarowi)  [1,max,2,1],  [1,max,2,max] ⬄ [max].  **min** - minimalna wartość wymiaru w danym zbiorze danych.  Przykładowe użycie dla przestrzeni dwuwymiarowej: (analogicznie do max).  *reference\_point* może definiować więcej niż jeden punkt referencyjny, przykładowo: reference\_point = [0][min][max] |
| reference\_point\_format | Flaga definiująca format definicji *reference\_point* przyjmująca wartości: **\*dense** – definicja *reference\_point* w formacie gęstym,  **\*sparse** – definicja *reference\_point* w formacie rzadkim. |
| use\_dataset\_index\_access | Flaga definiująca sposób dostępu do zbioru wektorów przyjmująca wartości: **\*true** – dostęp do zbioru wektorów przez indeks. Jeśli wymagane jest sortowanie zbioru to sortowany jest indeks,  **\*false** – bezpośredni dostęp do zbioru wektorów lub dostęp do zbioru przez indeks jeśli wymaga tego implementacja (algorytmy DBSCAN z nierównością trójkąta). Jeśli wymagane jest sortowanie zbioru to sortowany jest bezpośrednio zbiór wektorów. |
| projection\_dimension | Numery wymiarów oddzielone przecinkami, na które mogą być rzutowane punkty w celu szacowania odległości (1 odpowiada pierwszemu wymiarowi, brak wymiaru zerowego 0). |
| projection\_source\_sequence | Kolejność kryteriów, według których realizowane będzie przycinanie (pruning) punktów w celu szacowania odległości.  Przykładowo, mając dane:  *reference\_point*=[0][max][min]  *projection\_dimensions*=1,2  *projection\_source\_sequence*=d1,r2,r3,r1,d2  Zbiór punktów zostanie posortowany względem wartości kryterium.  Kryterium dla każdego punktu wyznaczane jest na podstawie *projection\_source\_sequence*.  Sposób obliczania kryterium:  Kryterium stanowi wektor wartości: \*(d1) wartość pierwszego wymiaru punktu,  \*(r2) odległość punktu do drugiego punktu referencyjnego, \*(r3) odległość punktu do trzeciego punktu referencyjnego, \*(r1) odległość punktu do pierwszego punktu referencyjnego, \*(d2) wartość drugiego wymiaru punktu.  W czasie przycinania kolejnymi kryteriami branymi pod uwagę będą:   * wartość pierwszego wymiaru zdefiniowanego w liście *projection\_dimensions* (d1=1) * odległość do drugiego punktu referencyjnego zdefiniowanego w liście *reference\_point* (r2=[max]), * odległość do trzeciego punktu referencyjnego zdefiniowanego w liście *reference\_point* (r3=[min]), * odległość do pierwszego punktu referencyjnego zdefiniowanego w liście *reference\_point* (r1=[0]), * wartość drugiego wymiaru zdefiniowanego w liście *projection\_dimensions* (d2=2) |
| classification\_subset\_factor | Definiuje, co który rekord zbioru danych ma zostać wybrany do zbioru, które będzie podlegał klasyfikacji. |
| use\_placement | Flaga definiująca sposób wybrania najlepszego odpowiednika punktu klasyfikowanego w zbiorze punktów *dataset\_file\_path* przyjmująca wartości:  **\*true** – z zastosowaniem wyszukiwania binarnego,  **\*false** – z zastosowaniem wyszukiwania liniowego. |
| p\_sample\_index | Maksymalna liczba elementów zbioru randomSampleP wykorzystywanego do znalezienia vantage point. |
| s\_sample\_index | Maksymalna liczba elementów zbioru randomSampleD wykorzystywanego do znalezienia vantage point. |
| search\_method | Zmienna definiująca metodę poszukiwania sąsiadów w drzewie vantage point. Dostępne metody to: **\*range** – wyszkiwanie zakresowe (wymaga zdefiniowanie parametr *eps*), **\*k\_neighborhood** – wyszukiwanie k sąsiadów (wymaga zdefiniowania parametru (*k\_neighborhood*). |
| use\_boundaries | Flaga definiująca sposób wyszukiwania sąsiadów w drzewie VP przyjmująca wartości: **\*true** – wyszukiwanie z wykorzystaniem wartości sąsiednich mediany (największej wartości mniejszej od mediany i najmniejszej wartości większej od mediany),  **\*false** – wyszukiwanie z wykorzystaniem mediany. |

Tabela 3: Opis znaczenia parametrów pliku properties.txt

## Wyniki uruchomień programu

Wyniki uruchomień programu znajdują się w folderze **logs**. Każdemu poprawnemu uruchomieniu algorytmu odpowiada plik raportu o nazwie *run\_report%\_data\_i\_czas\_uruchomienia%.txt*. Raport zawiera:

* wartości parametrów programu,
* czasy wykonania poszczególnych etapów algorytmów,
* wynik wykonania algorytmu.

Tabela 4 opisuje znaczenie rekordów raportu dotyczących czasów wykonania w zależności od użytego algorytmu.

|  |  |  |
| --- | --- | --- |
| Algorytm | **Nazwa czasu** | **Opis** |
|  | Dataset read | Czas wczytywania zbioru punktów z pliku zbioru punktów. |
|  | Reference point calculation | Czas obliczania punktów referencyjnych. |
|  | Normalization | Czas normalizacji punktów. |
| DBSCAN | Algorithm execution | Czas wykonania algorytmu. |
| DBSCAN\_POINTS\_ELIMINATION | Algorithm execution | Czas wykonania algorytmu. |
| TI-DBSCAN | Algorithm execution | Czas wykonania algorytmu. Algorithm execution = Clustering + Distance + Sorting( + Building index); |
|  | Clustering | Czas wykonania grupowania. |
|  | Distance | Czas obliczania odległości punktów zbioru punktów do punktu referencyjnego. |
|  | Building index | Tworzenie indeksu dostępu do danych. |
|  | Sorting | Czas sortowania zbioru punktów według kryterium odległości do punktu referencyjnego. |
| TI-DBSCAN-REF | Algorithm execution | Czas wykonania algorytmu. Algorithm execution = Clustering + Distance + Sorting; |
|  | Clustering | Czas wykonania grupowania. |
|  | Distance | Czas obliczania odległości punktów zbioru punktów do każdego z punktów referencyjnych. |
|  | Sorting | Czas sortowania zbioru punktów według kryteriów odległości do punktów referencyjnych. |
| TI-DBSCAN-REF-PROJECTION | Algorithm execution | Czas wykonania algorytmu. Algorithm execution = Clustering + Distance + Sorting; |
|  | Clustering | Czas wykonania grupowania. |
|  | Distance | Czas obliczania odległości punktów zbioru punktów do każdego z punktów referencyjnych oraz wykonania projekcji. |
|  | Sorting | Czas sortowania zbioru punktów według kryteriów odległości do punktów referencyjnych oraz projekcji. |
| K-NEIGHBORHOOD | Algorithm execution | Czas wykonania algorytmu. |
| TI-K-NEIGHBORHOOD | Algorithm execution | Czas wykonania algorytmu. Algorithm execution = Clustering + Distance + Sorting + (Building index); |
|  | Clustering | Czas znajdowania k sąsiadów. |
|  | Distance | Czas obliczania odległości punktów zbioru punktów do punktu referencyjnego oraz, w przypadku klasyfikacji, odległości punktów zbioru punktów klasyfikowanych do punktu referencyjnego. |
|  | Building index | Tworzenie indeksu dostępu do danych. (Wartość zliczana jeśli algorytm wykonywany jest z opcją dostępu do danych przez indeks) |
|  | Sorting | Czas sortowania zbioru punktów według kryterium odległości do punktu referencyjnego. |
|  | Positioning | Czas pozycjonowania. |
| TI-K-NEIGHBORHOOD-REF | Algorithm execution | Czas wykonania algorytmu. Algorithm execution = Clustering + Distance + Sorting( + Building index); |
|  | Clustering | Czas znajdowania k sąsiadów. |
|  | Distance | Czas obliczania odległości punktów zbioru punktów do każdego z punktów referencyjnych oraz, w przypadku klasyfikacji, odległości punktów zbioru punktów klasyfikowanych do każdego z punktów referencyjnych. |
|  | Building index | Tworzenie indeksu dostępu do danych. (Wartość zliczana jeśli algorytm wykonywany jest z opcją dostępu do danych przez indeks) |
|  | Sorting | Czas sortowania zbioru punktów według kryteriów odległości do punktów referencyjnych. |
|  | Positioning | Czas pozycjonowania . |
| TI-K-NEIGHBORHOOD-REF-PROJECTION | Algorithm execution | Czas wykonania algorytmu. Algorithm execution = Clustering + Distance + Sorting( + Building index); |
|  | Clustering | Czas znajdowania k sąsiadów. |
|  | Distance | Czas obliczania odległości punktów zbioru punktów do każdego z punktów referencyjnych oraz wykonania projekcji a także, w przypadku klasyfikacji, odległości punktów zbioru punktów klasyfikowanych do każdego z punktów referencyjnych oraz wykonania projekcji. |
|  | Building index | Tworzenie indeksu dostępu do danych. (Wartość zliczana jeśli algorytm wykonywany jest z opcją dostępu do danych przez indeks) |
|  | Sorting | Czas sortowania zbioru punktów według kryteriów odległości do punktów referencyjnych oraz projekcji. |
|  | Positioning | Czas pozycjonowania (Pozycjonowanie wykonywane jest w przypadku klasyfikacji). |
| VP-TREE | Algorithm execution | Czas wykonania algorytmu. Algorithm execution = Clustering + Index building; |
|  | Clustering | Czas znajdowania k sąsiadów lub otoczenia eps. |
|  | Building index | Czas budowania drzewa vantage point z punktów zbioru punktów. |

Tabela 4: Opis rekordów raportu dotyczących czasów wykonania w zależności od algorytmu

Wartości czasów podawane są w sekundach.

Każdemu poprawnemu uruchomieniu aplikacji odpowiada plik raportu o nazwie: *ultimate\_run\_report%\_data\_i\_czas\_uruchomienia%.csv*. Raport zawiera:

* wartości parametrów programu,
* czasy wykonania poszczególnych etapów algorytmów,

dla wszystkich uruchomień algorytmów w danym uruchomieniu programu (tj. dla wszystkich plików parametrów w folderze **properties**).

Raport ten jest podsumowaniem uruchomienia aplikacji i zbiera najważniejsze dane z uruchomień algorytmów. Raport można uruchomić w arkuszu kalkulacyjny i dzięki dostępnym narzędziom swobodnie porównywać dane raportów uruchomień algorytmów.

## Zbiory danych

W folderze **datasets** znajdują się pliki ze zbiorami wektorów. Tabela 5 opisuje właściwości przykładowych przygotowanych zbiorów wektorów.

|  |  |  |  |
| --- | --- | --- | --- |
| Nazwa zbioru | **Liczba wektorów zbioru** | **Liczba grup zbioru** | **Liczba wymiarów** |
| dense\_d2\_r3000.txt | 3000 | 20 | 2 |
| dense\_d2\_r62556\_sequoia.txt | 62556 |  | 2 |
| dense\_d2\_r100000\_birch1.txt | 100000 |  | 2 |
| dense\_d15\_r10126.txt | 10126 |  | 15 |
| dense\_d56\_r96367\_cup98.txt | 96367 |  | 56 |
| dense\_d55\_r581012\_covtype.txt | 581012 |  | 55 |
| sparse\_d126373\_r8580\_karypis\_sport.txt | 8580 |  | 126373 |
| sparse\_d126373\_r4069\_karypis\_reviews.txt | 4069 |  | 126373 |
| sparse\_d126373\_r2301\_karypis\_hitech.txt | 2301 |  | 126373 |

Tabela 5: Opis przykładowych przygotowanych zbiorów punktów

Pliki nazywane są według konwencji *FormatPliku****\_d****LiczbaWymiarów****\_r****LiczbaRekordów\_NazwaZbiowu****.txt***.

## Uruchomienie

Aby program poprawnie zapisywał i odczytywał pliki, powinny one być umieszczone w następujący sposób:

Obok pliku **implementacja.exe**, na tym samym poziomie drzewa katalogów, powinny znajdować się następujące katalogi:

* **datasets** – zawierający pliki zbiorów danych,  
  Żadna linia pliku zbioru punktów nie powinna rozpoczynać się białym znakiem.  
  Kolejne wartości opisujące punkt powinny być oddzielone dokładnie jedną spacją.
* **logs** – zawierający pliki raportów wykonania (ścieżka względem pliku *implementacja.exe* \logs\),
* **properties** – zawierający pliki parametrów uruchomienia algorytmu(ścieżka względem pliku *implementacja.exe* \properties\).  
  W każdym pliku parametrów po nazwie parametru powinien znajdować się dokładnie znak ‘=’ a następnie wartość parametru.
* **algorithms\_engine\_properties** – zawierający plik parametrów uruchomienia aplikacji (ścieżka względem pliku *implementacja.exe* \algorithms\_engine\_properties\)  
  W pliku parametrów po nazwie parametru powinien znajdować się dokładnie znak ‘=’ a następnie wartość parametru.

Jeżeli pliki parametrów nie będą znajdowały się w odpowiednim katalogu (properties\) program będzie działał niepoprawnie.

Jeżeli katalog **logs** nie zostanie utworzony, pliki logów/raportów przebiegu wykonania programu nie zostaną zapisane.

Istnienie katalogu **datasets** nie jest wymagane, ponieważ w pliku parametrów podawana jest ścieżka do pliku ze zbiorem danych. Jednakże dla spójności rozwiązania, wszystkie pliki z danymi umieszczane są w folderze **datasets**.

Aby uruchomić implementację należy:

* + - dokonać odpowiednich zmian w plikach parametrów,
    - uruchomić plik *implementacja.exe*.

Po poprawnym uruchomieniu implementacji zostanie wydrukowany raport do folderu **logs**. W celu zapoznania się z wynikami uruchomienia należy przejrzeć raport wykonania.

# Wnioski

Przycinanie (pruning) zdecydowanie lepiej sprawdza się, jeśli pierwsze kryterium stanowi odległość do punktu referencyjnego.

//TODO

# Badania eksperymentalne

W przeprowadzonych przeze mnie eksperymentach badałem wydajność wyszukiwania k-sąsiedztwa oraz epsilonowego sąsiedztwa z zastosowaniem nierówności trójkąta oraz wpływ sposobu ich zaimplementowania. Implementacja algorytmów została wykonana w języku C++ w oparciu o pseudokody z artykułów [1], [4], [5].

Eksperymenty przeprowadziłem na środowisku Windows 7 x64 z procesorem Intel® i7™ 950 z dostępną pamięcią RAM równą 6GB. Aplikacja, którą posłużyłem się do zebrania wyników eksperymentów została skompilowana dla środowiska WIN32 z podniesioną flagą LARGEADDRESSAWARE, która umożliwia użycie do 3GB pamięci wirtualnej.

Eksperymenty uruchamiałem jako proces o priorytecie ‘Wysoki’ aby uniknąć wywłaszczenia procesu z rdzenia zmniejszając tym samym ryzyko losowego zakłócenia wyników.

Zauważyłem, że kolejne uruchomienia danego algorytmu z tymi samymi parametrami, nazwijmy je serią, wykonują się w różnym czasie. Ze względu na charakter wspomnianych różnic wyszczególniłem dwa przypadki. Pierwszy, w którym różnice czasów wykonań algorytmów w danej serii są niewielkie oraz drugi, w którym owe różnice są znaczące. W pierwszym przypadku powodem odchyleń są losowe zakłócenia. W drugim przypadku zauważyłem pewną prawidłowość, według której pierwsze uruchomienie algorytmu w serii trwa najdłużej z wszystkich, kolejne trochę krócej a różnice czasów trwania następnych uruchomień odpowiadają tym z przypadku pierwszego. Dzieje się tak ponieważ cache procesora (w moim przypadku 8MB, L3) z każdym wykonaniem algorytmu w serii działa sprawniej do chwili osiągnięcia pewnej sprawności granicznej dla danej serii. W efekcie czas wykonania algorytmu skraca się aż do momentu gdy różnice czasów kolejnych wykonań przypominają te z przypadku pierwszego. Aby umniejszyć wpływ wyżej opisanych zjawisk na zbieranie wyników czasowych poszczególnych kroków algorytmów ich pomiar dokonywany jest w następujący sposób: spośród wszystkich wykonań w serii odrzucane są największe i najmniejsze wartości czasowe a z pozostałych wielkości obliczana jest wartość średnia.

# Dane testowe

W eksperymentach wykorzystałem zbiory danych stosowane w dziedzinowej literaturze, których odmienne charakterystyki pozwoliły na sprawdzenie działania algorytmów w różnych warunkach. Zamieszczone poniżej opisy dotyczą zbiorów, które wykorzystałem do przeprowadzenia eksperymentów, na których oparłem zaprezentowane w kolejnych podrozdziałach wnioski.

Covtype [7] jest zbiorem udostępnianym przez US Forest Service. Jego pełna nazwa to Forest Cover Type. Baza ta zawiera informacje na temat gatunków drzew w amerykańskich lasach. Zbiór posiada 581012 rekordów w formacie gęstym o 55 atrybutach. 10 pierwszych atrybutów to wartości zmiennych ilościowych, kolejne 44 atrybuty posiadają wartości 0 lub 1, z kolei ostatni determinuje jeden z siedmiu gatunków drzew. Należy zwrócić uwagę, że każdy z rekordów spośród atrybutów binarnych przyjmuje wartość 1 dokładnie dwa razy, a wartość zero 42 razy. W przypadku 10 pierwszych atrybutów zróżnicowanie wartości jest znacznie większe, z których pierwsze dwa posiadają najszerszą dziedziną (ponad 7000), a pozostałe charakteryzuje znacznie mniejsza różnorodność i zakres.

KDD-Cup98 [8] jest zbiorem opublikowanym podczas Drugiego Międzynarodowego Konkursu Odkrywania Wiedzy i Eksploracji Danych, który miał miejsce podczas KDD-98, Czwartej Międzynarodowej Konferencji Odkrywania Wiedzy i Eksploracji Danych w 1998 roku. Zbiór posiada 96367 rekordów w formacie gęstym o 56 atrybutach o wartościach będącymi liczbami naturalnymi. Najszerszą dziedziną [0, 6000] posiadają dwa atrybuty, najwęższą dziedzinę [0, 13] również tylko dwa atrybuty, natomiast aż 44 atrybuty posiadają taką samą dziedzinę [0, 99].

Karypis\_sport jest zbiorem danych wchodzącymi w skład bazy zbiorów [9] udostępnianych na stronie profesora George Karypis z Department of Computer Science & Engineering , University of Minesota. Zbiór karypis\_sport zawiera 8580 rekordów w formacie rzadkim o 126373 atrybutach przyjmujących wartości będące liczbami naturalnymi. Baza ta zawiera informacje o dokumentach dotyczących sportu.

Karypis\_review jest zbiorem danych wchodzącym w ten sam skład bazy zbiorów co karypis\_sport [9]. Zbiór ten zawiera 4069 rekordów w formacie rzadkim o 126373 atrybutach przyjmujących wartości będące liczbami naturalnymi. Baza ta zawiera informacje o dokumentach dotyczących recenzji.

# Testy algorytmu K-NEIGHBORHOOD

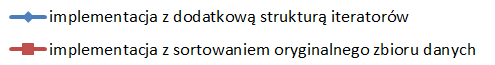
# Testy algorytmu *TI-k-Neighborhood-Index*

# Implementacja algorytmu

Na przykładowych zbiorach danych przetestowałem dwie wersje algorytmu TI-K-NEIGHBORHOOD: korzystającą ze struktury indeksu zbudowanego z iteratorów oraz opartą wyłącznie na oryginalnym zbiorze danych. W pierwszym przypadku sortowaniu poddany zostaje indeks, który następnie używany jest w celu dostępu do zbioru danych. W drugim przypadku sortowany jest zbiór danych i żaden indeks nie jest wykorzystywany w dostępie do jego elementów. W tabeli 6 i na rysunku 1 zamieszczono porównanie obu implementacji.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| zbioór | **liczba punktów** | **TI-K\_NEIGHBORHOOD INDEX** | | | | | | **TI-K\_NEIGHBORHOOD** | | | | | |
| **wykonanie algorytmu** | **klasyfikacja** | **budowa indeksu** | **obliczanie odległości** | **sortowanie** | **plasowanie** | **wykonanie algorytmu** | **klasyfikacja** | **budowa indeksu** | **obliczanie odległości** | **sortowanie** | **plasowanie** |
|
| karypis\_sport | 1000 | 0,206 | 0,204 | 0,000 | 0,000 | 0,000 | 0,001 | 0,235 | 0,206 | N/A | 0,004 | 0,025 | 0,003 |
| 2000 | 0,680 | 0,674 | 0,000 | 0,004 | 0,000 | 0,006 | 0,729 | 0,660 | N/A | 0,004 | 0,065 | 0,006 |
| 4000 | 2,825 | 2,813 | 0,000 | 0,010 | 0,001 | 0,042 | 2,931 | 2,755 | N/A | 0,010 | 0,166 | 0,022 |
| 6000 | 6,887 | 6,870 | 0,000 | 0,016 | 0,001 | 0,096 | 7,122 | 6,570 | N/A | 0,015 | 0,537 | 0,037 |
| 8000 | 13,226 | 13,202 | 0,000 | 0,023 | 0,002 | 0,194 | 13,439 | 12,736 | N/A | 0,024 | 0,680 | 0,079 |
| karypis\_review | 500 | 0,067 | 0,066 | 0,000 | 0,002 | 0,000 | 0,001 | 0,079 | 0,066 | N/A | 0,002 | 0,012 | 0,001 |
| 1000 | 0,250 | 0,246 | 0,000 | 0,003 | 0,000 | 0,006 | 0,279 | 0,247 | N/A | 0,003 | 0,029 | 0,003 |
| 2000 | 1,105 | 1,099 | 0,000 | 0,006 | 0,000 | 0,011 | 1,207 | 1,106 | N/A | 0,006 | 0,096 | 0,007 |
| 3000 | 2,394 | 2,385 | 0,000 | 0,008 | 0,001 | 0,014 | 2,599 | 2,380 | N/A | 0,008 | 0,210 | 0,013 |
| 4000 | 4,347 | 4,334 | 0,000 | 0,011 | 0,001 | 0,026 | 4,636 | 4,303 | N/A | 0,012 | 0,322 | 0,025 |
| covtype | 10000 | 0,548 | 0,391 | 0,000 | 0,153 | 0,003 | 0,176 | 1,366 | 0,184 | N/A | 0,058 | 1,122 | 0,066 |
| 50000 | 7,789 | 5,400 | 0,001 | 2,333 | 0,015 | 1,227 | 78,548 | 4,147 | N/A | 2,089 | 72,281 | 2,147 |
| 100000 | 19,994 | 15,052 | 0,002 | 4,900 | 0,035 | 4,133 | 79,485 | 11,357 | N/A | 4,636 | 63,318 | 4,020 |
| 300000 | 99,572 | 83,380 | 0,004 | 16,079 | 0,130 | 17,136 | 619,503 | 50,367 | N/A | 15,277 | 553,806 | 15,928 |
| 500000 | 261,503 | 234,340 | 0,005 | 26,861 | 0,252 | 29,941 | 1 860,330 | 120,760 | N/A | 25,600 | 1 713,870 | 27,029 |
| cup98 | 10000 | 0,393 | 0,272 | 0,000 | 0,117 | 0,003 | 0,119 | 2,241 | 0,140 | N/A | 0,062 | 2,039 | 0,067 |
| 30000 | 3,076 | 2,197 | 0,000 | 0,865 | 0,010 | 0,932 | 3,229 | 1,191 | N/A | 0,530 | 1,503 | 0,723 |
| 50000 | 7,291 | 5,371 | 0,001 | 1,900 | 0,019 | 2,052 | 60,403 | 3,198 | N/A | 1,555 | 55,625 | 1,648 |
| 70000 | 12,927 | 9,563 | 0,001 | 3,293 | 0,029 | 3,920 | 100,807 | 5,673 | N/A | 2,772 | 92,347 | 3,380 |
| 90000 | 19,646 | 15,316 | 0,002 | 4,185 | 0,044 | 3,746 | 82,883 | 9,064 | N/A | 4,649 | 70,221 | 6,339 |

Tabela 6: Porównanie wydajności implementacji algorytmu *TI-k-Neighborhood-Index* na przykładowych zbiorach danych. Tabela zawiera czasy wykonania (w sekundach) poszukiwań k=5 sąsiadów w przykładowych zbiorach danych z wykorzystanie implementacji algorytmu *TI-k-Neighborhood-Index* dla 10% losowo wybranych punktów zbioru danych.



Rysunek 1: Porównanie wydajności implementacji algorytmu *TI-k-Neighborhood-Index* na przykładowych zbiorach danych. Tabela zawiera czasy wykonania (w sekundach) poszukiwań k=5 sąsiadów w przykładowych zbiorach danych z wykorzystanie implementacji algorytmu *TI-k-Neighborhood-Index* dla 10% losowo wybranych punktow zbioru danych.

Z rezultatów uruchomień wynika, że w porównaniu do implementacji bez indeksu implementacja z indeksem iteratorów radzi sobie z wyszukiwaniem k-sąsiedztwa tym lepiej im więcej rekordów posiada zbiór, w którym szukane jest k-sąsiedztwo. Największa różnica w czasie wykonania algorytmu wystąpiła przy wyszukiwaniu k-sąsiedztwa dla 50000 punktów zbioru covtype. Dla tego konkretnego przypadku implementacja z bezpośrednim dostępem do danych wykonuje się blisko 7 razy wolniej niż implementacja korzystająca z indeksu.

Kluczowym krokiem algorytmu mającym wpływ na powstałą różnicę jest sortowanie punktów. Obie implementacje do sortowania obiektów wykorzystują standardową dla języka C++ funkcję sortowania wektorów, tj. *std::sort*, która oparta jest o algorytm quicksort. Funkcja sortowania biblioteki standardowej korzysta z operacji *swap* zamieniającej miejscami elementy w wektorze, która wywołuje konstruktor kopiujący obiektu będącego elementem sortowanego wektora. W przypadku gdy głęboko kopiowany jest obiekt klasy *Punkt*, to uruchamiane są konstruktory kopiujące wszystkich obiektów będących jego polami, w szczególności kolekcji przechowujących wartości wymiarów. Dlatego, wykonanie konstruktora kopiującego iteratora jest szybsze niż wykonanie konstruktora kopiującego obiektu punktu. Im większy zbiór jest sortowany tym częściej funkcja *swap* jest wywoływana, tym samym częściej wykonywany jest konstruktor kopiujący. W skutek tego, czas sortowania zbioru obiektów punktu jest dłuższy niż czas sortowania zbioru iteratorów. Zatem mimo, że dostęp do nieposortowanych danych przez posortowany zbiór iteratorów jest wolniejszy niż bezpośredni dostęp do posortowanego zbioru, to czas sortowania bezpośrednio zbioru danych jest na tyle duży, że różnice czasowe wynikające ze sposobu dostępu do danych stają się pomijalne. W kolejnych rozważaniach będę się odnosił do algorytmu *TI-k-Neighborhood-Index* jako do implementacji korzystającej z indeksu iteratorów.

# Implementacja struktury punktu

Początkowo naturalnym podejściem wydawało się przechowywanie wartości wymiarów punktu w postaci tablicy lub wektora o liczbie elementów równej liczbie wymiarów przestrzeni danych. Pierwsze eksperymenty z danymi tekstowymi pokazały, że przy zastosowaniu wspomnianego podejścia implementacja indeksu szybko wyczerpywała pamięć RAM maszyny, na której dokonywano eksperymentów. Zachowanie to wynikało z dużej liczby wymiarów przestrzeni danych. Jeden punkt przestrzeni danych tekstowych (105 wymiarowej) zajmował tyle pamięci co mniej więcej 5\*104 punktów przestrzeni dwuwymiarowej.

Na szczęście przechowywanie punktów przestrzeni danych tekstowych można ulepszyć w oparciu o ich cechy charakterystyczne. Kluczową własnością danych tekstowych jest duża liczba wymiarów punktu przestrzeni danych tekstowych, których wartość jest równa 0. Wartość ta nie wnosi żadnej informacji w procesie wyznaczania odległości między dwoma punktami, dlatego postanowiłem jej nie przechowywać. W tym celu zaimplementowałem rzadką reprezentację punktu w postaci listy par <*numer wymiaru*, *wartość wymiaru*>.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| zbioór | **liczba punktów** | ***TI-k-Neighborhood-Index* REPREZENTACJA GĘSTA** | | | | | | ***Ti-k-Neighborhood-Index* REPREZENTACJA RZADKA** | | | | | |
| **wykonanie algorytmu** | **klasyfikacja** | **budowa indeksu** | **obliczanie odległości** | **sortowanie** | **plasowanie** | **wykonanie algorytmu** | **klasyfikacja** | **budowa indeksu** | **obliczanie odległości** | **sortowanie** | **plasowanie** |
|
| karypis\_sport | 1000 | 22,209 | 21,959 | 0,000 | 0,249 | 0,000 | 0,265 | 0,206 | 0,204 | 0,000 | 0,000 | 0,000 | 0,001 |
| 2000 | - | - | - | - | - | - | 0,680 | 0,674 | 0,000 | 0,004 | 0,000 | 0,006 |
| 4000 | - | - | - | - | - | - | 2,825 | 2,813 | 0,000 | 0,010 | 0,001 | 0,042 |
| 6000 | - | - | - | - | - | - | 6,887 | 6,870 | 0,000 | 0,016 | 0,001 | 0,096 |
| 8000 | - | - | - | - | - | - | 13,226 | 13,202 | 0,000 | 0,023 | 0,002 | 0,194 |
| karypis\_review | 500 | 5,710 | 5,585 | 0,000 | 0,125 | 0,000 | 0,125 | 0,067 | 0,066 | 0,000 | 0,002 | 0,000 | 0,001 |
| 1000 | 22,682 | 22,428 | 0,000 | 0,249 | 0,000 | 0,265 | 0,250 | 0,246 | 0,000 | 0,003 | 0,000 | 0,006 |
| 2000 | - | - | - | - | - | - | 1,105 | 1,099 | 0,000 | 0,006 | 0,000 | 0,011 |
| 3000 | - | - | - | - | - | - | 2,394 | 2,385 | 0,000 | 0,008 | 0,001 | 0,014 |
| 4000 | - | - | - | - | - | - | 4,347 | 4,334 | 0,000 | 0,011 | 0,001 | 0,026 |
| covtype | 10000 | 0,548 | 0,391 | 0,000 | 0,153 | 0,003 | 0,176 | 0,349 | 0,286 | 0,000 | 0,062 | 0,000 | 0,052 |
| 50000 | 7,789 | 5,400 | 0,001 | 2,333 | 0,015 | 1,227 | 4,394 | 3,417 | 0,000 | 0,962 | 0,016 | 0,962 |
| 100000 | 19,994 | 15,052 | 0,002 | 4,900 | 0,035 | 4,133 | 16,412 | 12,995 | 0,000 | 3,380 | 0,031 | 3,895 |
| 300000 | 99,572 | 83,380 | 0,004 | 16,079 | 0,130 | 17,136 | 125,773 | 115,939 | 0,005 | 9,453 | 0,141 | 12,755 |
| 500000 | 261,503 | 234,340 | 0,005 | 26,861 | 0,252 | 29,941 | 252,788 | 235,820 | 0,016 | 16,682 | 0,265 | 19,157 |
| cup98 | 10000 | 0,393 | 0,272 | 0,000 | 0,117 | 0,003 | 0,119 | 0,670 | 0,608 | 0,000 | 0,062 | 0,000 | 0,125 |
| 30000 | 3,076 | 2,197 | 0,000 | 0,865 | 0,010 | 0,932 | 5,408 | 4,763 | 0,000 | 0,629 | 0,010 | 1,056 |
| 50000 | 7,291 | 5,371 | 0,001 | 1,900 | 0,019 | 2,052 | 13,458 | 12,054 | 0,000 | 1,367 | 0,021 | 2,002 |
| 70000 | 12,927 | 9,563 | 0,001 | 3,293 | 0,029 | 3,920 | 22,084 | 21,028 | 0,000 | 1,009 | 0,031 | 2,486 |
| 90000 | 19,646 | 15,316 | 0,002 | 4,185 | 0,044 | 3,7463 | 68,249 | 65,275 | 0,000 | 2,891 | 0,052 | 3,952 |

Tabela 7: Porównanie wydajności algorytmu TI-k-Neighborhood-Index w zależności od implementacji punktu na przykładowych zbiorach danych. Tabela zawiera czasy wykonania (w sekundach) poszukiwań k=5 sąsiadów w przykładowych zbiorach danych z wykorzystaniem algorytmu *TI-k-Neighborhood-Index* dla 10% losowo wybranych punktow zbioru danych. Notacja: ‘-‘ rezultat nie został osiągnięty w czasie mneijszym niż 3 godziny lub z powodu wyczerpania pamięci operacyjnej.

Rysunek 2: Porównanie wydajności algorytmu *TI-k-Neighborhood-Index* w zależności od implementacji punktu na przykładowych zbiorach danych. Tabela zawiera czasy wykonania (w sekundach) poszukiwań k=5 sąsiadów w przykładowych zbiorach danych z wykorzystaniem algorytmu *TI-k-Neighborhood-Index* dla 10% losowo wybranych punktow zbioru danych.

W tabeli 7 oraz na rysunku 2 zamieszczono wyniki uruchomień algorytmu *TI-k-Neighborhood-Index* z użyciem zarówno gęstej jak i rzadkiej implementacji punktu.

Dla zbiorów cup98 i covtype, uruchomienia korzystające z implementacji punktu gęstego wykonują się szybciej niż uruchomienia posługujące się implementacją punktu rzadkiego. Odwrotne zachowanie można zauważyć dla danych tekstowych, tj. ziobrów karypis sport i karypis review, w których przypadku uruchomienia korzystające z implementacji punktu rzadkiego wykonują się o dwa rzędy wielkości szybciej niż uruchomienia korzystające z implementacji punktu gęstego. Niestety z powodu wyczerpania pamięci RAM nie można było uzyskać rezultatów badań implementacji gęstej punktu na zbiorach danych tekstowych dla liczby rekordów większej niż 1000.

Z przeprowadzonych badań skonstruowano następujące wnioski:

* w porównaniu z implementacją gęstego punktu stosowanie implementacji rzadkiego punktu w uruchomieniach algorytmów działających na danych tekstowych przyspiesza wykonanie uruchomień oraz zmniejsza wielkość zajętej pamięci operacyjnej,
* w porównaniu z implementacją rzadkiego punktu stosowanie implementacji gęstego punktu w uruchomieniach algorytmów działających na danych gęstych o niewielkim wymiarze przyspiesza wykonanie uruchomień.

W dalszej części raportu badania algorytmów działających na danych tekstowych będą wykonywane z użyciem implementacji rzadkiego punktu.

# Porównanie implementacji *k-Neighborhood-Index*

Przeprowadzone eksperymenty, których rezultaty zostały zamieszczone w tabeli 8 i na rysunku 9 jednoznacznie potwierdzają wzrosty wydajności *TI-k-Neighborhood-Index* i *TI-k-Neighborhood-Index-Ref* w stosunku do algorytmu *Ti-k-Neighborhood*. Wzrost ten jest mniej widoczny dla tanych tekstowych niż dla pozostałych zbiorów.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| zbioór | **liczba punktów** | **K\_NEIGHBORHOOD** | | | | **TI-K\_NEIGHBORHOOD** | | | | **TI-K\_NEIGHBORHOOD\_REF** | | | |
| **wykonanie algorytmu** | **klasyfikacja** | **budowa indeksu** | **obliczanie odległości** | **wykonanie algorytmu** | **klasyfikacja** | **budowa indeksu** | **obliczanie odległości** | **wykonanie algorytmu** | **klasyfikacja** | **budowa indeksu** | **obliczanie odległości** |
|
| karypis\_sport | 1000 | 0,322 | 0,320 | 0,000 | 0,002 | 0,206 | 0,204 | 0,000 | 0,000 | 0,177 | 0,171 | 0,000 | 0,005 |
| 2000 | 1,397 | 1,394 | 0,000 | 0,004 | 0,680 | 0,674 | 0,000 | 0,004 | 0,634 | 0,629 | 0,000 | 0,005 |
| 4000 | 6,013 | 6,005 | 0,000 | 0,008 | 2,825 | 2,813 | 0,000 | 0,010 | 2,699 | 2,678 | 0,000 | 0,016 |
| 6000 | 15,176 | 15,162 | 0,000 | 0,014 | 6,887 | 6,870 | 0,000 | 0,016 | 6,427 | 6,396 | 0,000 | 0,031 |
| 8000 | 30,513 | 30,497 | 0,000 | 0,016 | 13,226 | 13,202 | 0,000 | 0,023 | 11,788 | 11,726 | 0,000 | 0,057 |
| karypis\_review | 500 | 0,093 | 0,092 | 0,000 | 0,001 | 0,067 | 0,066 | 0,000 | 0,002 | 0,062 | 0,062 | 0,000 | 0,000 |
| 1000 | 0,385 | 0,383 | 0,000 | 0,002 | 0,250 | 0,246 | 0,000 | 0,003 | 0,249 | 0,239 | 0,000 | 0,005 |
| 2000 | 2,631 | 2,626 | 0,000 | 0,005 | 1,105 | 1,099 | 0,000 | 0,006 | 1,092 | 1,076 | 0,000 | 0,016 |
| 3000 | 9,182 | 9,175 | 0,000 | 0,007 | 2,394 | 2,385 | 0,000 | 0,008 | 2,386 | 2,366 | 0,000 | 0,015 |
| 4000 | 9,110 | 9,101 | 0,000 | 0,009 | 4,347 | 4,334 | 0,000 | 0,011 | 4,238 | 4,212 | 0,000 | 0,026 |
| covtype | 10000 | 40,334 | 40,115 | 0,000 | 0,219 | 0,548 | 0,391 | 0,000 | 0,153 | 0,520 | 0,276 | 0,000 | 0,229 |
| 50000 | 844,847 | 842,475 | 0,001 | 2,372 | 7,789 | 5,400 | 0,001 | 2,333 | 6,661 | 4,227 | 0,000 | 2,418 |
| 100000 | 4575,721 | 4570,870 | 0,002 | 4,851 | 19,994 | 15,052 | 0,002 | 4,900 | 16,239 | 10,956 | 0,005 | 5,268 |
| 300000 | - | - | - | - | 99,572 | 83,380 | 0,004 | 16,079 | 65,063 | 48,391 | 0,005 | 16,526 |
| 500000 | - | - | - | - | 261,503 | 234,340 | 0,005 | 26,861 | 144,227 | 115,575 | 0,005 | 28,454 |
| cup98 | 10000 | 16,166 | 15,947 | 0,000 | 0,219 | 0,393 | 0,272 | 0,000 | 0,117 | 0,411 | 0,271 | 0,000 | 0,135 |
| 30000 | 164,154 | 163,038 | 0,001 | 1,116 | 3,076 | 2,197 | 0,000 | 0,865 | 2,788 | 1,919 | 0,000 | 0,848 |
| 50000 | 475,852 | 473,576 | 0,001 | 2,276 | 7,291 | 5,371 | 0,001 | 1,900 | 7,062 | 5,070 | 0,000 | 1,971 |
| 70000 | 996,056 | 992,978 | 0,001 | 3,078 | 12,927 | 9,563 | 0,001 | 3,293 | 12,417 | 8,829 | 0,000 | 3,536 |
| 90000 | 1717,047 | 1712,660 | 0,002 | 4,387 | 19,646 | 15,316 | 0,002 | 4,462 | 16,504 | 12,210 | 0,000 | 4,300 |

Tabela 8: Porównanie wydajności algorytmu implementacji k-Neighborhood-Index zbiorach danych. Tabela zawiera czasy wykonania (w sekundach) poszukiwań k=5 sąsiadów w przykładowych zbiorach danych dla 10% losowo wybranych punktow zbioru danych. Notacja: ‘-‘ rezultat nie został osiągnięty w czasie mneijszym niż 3 godziny lub z powodu wyczerpania pamięci operacyjnej.



Rysunek 3: Porównanie wydajności algorytmu implementacji k-Neighborhood-Index zbiorach danych. Tabela zawiera czasy wykonania (w sekundach) poszukiwań k=5 sąsiadów w przykładowych zbiorach danych dla 10% losowo wybranych punktów zbioru danych.

# Testy algorytmu *VP-Tree-Inedex*

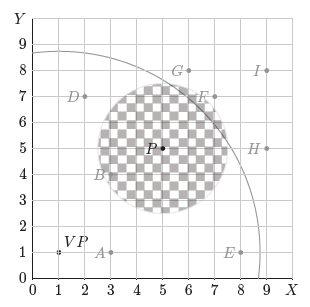
# Implementacja algorytmu *VP-Tree-Index*

Na przykładowych zbiorach danych przetestowano dwie metody wyszukiwania k-sąsiedztwa w oparciu o indeks *vp-drzewa*: pierwszą, której kryterium przeszukiwania kolejnych gałęzi *vp-drzewa* stanowi mediana odległości punktów do punktu *vp*, zwaną dalej *metodą mediany*; drugą, której kryterium przeszukiwania kolejnych gałęzi *vp-drzewa* stanowią największa z odległości punktów do punktu *vp* mniejsza od mediany i najmniejsza z odległości punktów do punktu *vp* większa od mediany, zwaną dalej *metodą ograniczeń*.W tabeli 10 i na rysunku 4 zamieszczono czasy uruchomień algorytmu *VP-Tree-Index* dla obu metod wyszukiwania k-sąsiedztwa.Z rezultatów badań wynika, że *metoda ograniczeń* zapewnia szybsze wyszukiwanie k-sąsiedztwa niż *metoda mediany*. Największa różnica w czasach wykonania algorytmów występuje dla wyszukiwania k-sąsiedztwa spośród 500000 punktów zbioru covtype. Implementacja korzystająca z *metody ograniczeń* wykonuje się blisko 1.5 raza szybciej niż implementacja stosująca *metodę mediany*.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| zbioór | **liczba punktów** | ***Vp-Tree-Index* MEDIANA** | | | ***Vp-Tree-Index* OGRANICZENIA** | | |
| **wykonanie algorytmu** | **klasyfikacja** | **budowa indeksu** | **wykonanie algorytmu** | **klasyfikacja** | **budowa indeksu** |
|
| karypis\_sport | 1000 | 0,614 | 0,241 | 0,373 | 0,296 | 0,218 | 0,078 |
| 2000 | 1,566 | 0,883 | 0,678 | 0,988 | 0,822 | 0,166 |
| 4000 | 4,981 | 3,747 | 1,229 | 3,874 | 3,510 | 0,369 |
| 6000 | 10,281 | 8,483 | 1,793 | 8,689 | 8,107 | 0,577 |
| 8000 | 15,954 | 15,039 | 0,905 | 15,538 | 14,768 | 0,774 |
| karypis\_review | 500 | 0,392 | 0,084 | 0,308 | 0,270 | 0,078 | 0,192 |
| 1000 | 0,911 | 0,318 | 0,593 | 0,676 | 0,296 | 0,379 |
| 2000 | 2,532 | 1,385 | 1,145 | 1,981 | 1,326 | 0,660 |
| 3000 | 4,658 | 3,089 | 1,561 | 3,629 | 2,949 | 0,681 |
| 4000 | 7,350 | 5,462 | 1,882 | 5,840 | 5,247 | 0,593 |
| covtype | 10000 | 1,849 | 0,366 | 1,491 | 0,855 | 0,298 | 0,561 |
| 50000 | 9,009 | 3,029 | 6,008 | 5,083 | 2,241 | 2,770 |
| 100000 | 27,536 | 10,687 | 17,158 | 14,576 | 7,703 | 6,501 |
| 300000 | 126,302 | 67,943 | 58,494 | 107,732 | 70,103 | 36,422 |
| 500000 | 351,608 | 219,689 | 126,527 | 253,721 | 180,813 | 68,562 |
| cup98 | 10000 | 2,490 | 0,494 | 1,983 | 0,728 | 0,307 | 0,421 |
| 30000 | 8,107 | 4,054 | 4,014 | 4,174 | 2,812 | 1,366 |
| 50000 | 19,648 | 11,432 | 8,076 | 10,649 | 7,766 | 2,450 |
| 70000 | 29,294 | 18,886 | 10,485 | 21,992 | 16,790 | 3,710 |
| 90000 | 44,735 | 29,800 | 15,018 | 39,182 | 27,149 | 13,156 |

Tabela 9: Porównanie wydajności algorytmu *VP-Tree-Index* w zależności od implementacji metody wyszukiwania w *vp-*drzewie na przykładowych zbiorach danych. Tabela zawiera czasy wykonania (w sekundach) poszukiwań k=5 sąsiadów w przykładowych zbiorach danych z wykorzystaniem algorytmu *VP-Tree-Index* dla 10% losowo wybranych punktow zbioru danych

Rysunek 4: Porównanie wydajności algorytmu VP-Tree-Index w zależności od implementacji metody wyszukiwania w vp-drzewie na przykładowych zbiorach danych. Tabela zawiera czasy wykonania (w sekundach) poszukiwań k=5 sąsiadów w przykładowych zbiorach danych z wykorzystaniem algorytmu *VP-Tree-Index* dla 10% losowo wybranych punktow zbioru danych

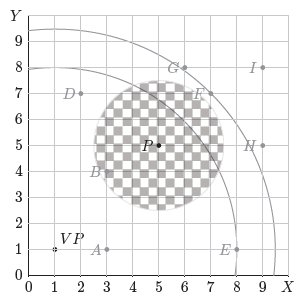
Powodem różnic w czasie wykonania obu implementacji jest kryterium wyszukiwania.

Rysunek

Następujące wyjaśnienie jest oparte na Eps-sąsiedztwie bez straty wartości merytorycznej dla k-sąsiedztwa ponieważ problem k-sąsiedztwa można sprowadzić do problemu Eps-sąsiedztwa.

Na rysunku 4 przedstawiono sytuację znajdowania Eps-sąsiedztwa punktu P (obszar pokryty czarno-białą kratą) w węźle VP *vp-drzewa metodą mediany*. Fragmentem okręgu zaznaczono medianę odległości punktów od punktu VP. Algorytm znajdowania eps-sąsiedztwa punktu P *metodą mediany* odwiedzając węzeł VP *vp-drzewa* oblicza odległość distance(VP, P) a następnie:

* przeszukuje lewe poddrzewo *vp-drzewa* jeśli distance(VP,P)-Eps<*mediana* i distance(VP,P)+Eps<=*mediana.*
* przeszukuje prawe poddrzewo *vp-drzewa* jeśli distance(VP,P)+Eps>=*mediana* i distance(VP,P)-Eps>=*mediana.*
* przeszukuje lewe i prawe poddrzewo *vp-drzewa* jeśli distance(VP,P)-Eps<*mediana* i distance(VP,P)+Eps>=*mediana.*

Zatem w przypadku przedstawionym na rysunku 4 *metoda mediany* przeszuka zarówno lewe jak i prawe poddrzewo mimo, że żaden z punktów prawego poddrzewa (F, G, H, I) nie należy do Eps-sąsiedztwa punktu P.

Rysunek 5 przedstawia sytuację znajdowania Eps-sąsiedztwa punktu P (obszar pokryty czarno-białą kratą) w węźle VP *vp-drzewa metodą ograniczeń.* Fragmentami okręgów zostały oznaczone odległości bhl (najmniejsza z odległości punktów do punktu VP większa od mediany) i blh (największa z odległości punktów do punktu VP mniejsza od mediany). Algorytm znajdowania eps-sąsiedztwa punktu P  *metodą ograniczeń* odwiedzając węzeł VP *vp-drzewa* obilcza odległość distance (VP,P) a następnie:

Rysunek

* przeszukuje lewe poddrzewo *vp-drzewa* jeśli distance(VP,P)-Eps<=blh i distance(VP,P)+Eps<bhl.
* przeszukuje prawe poddrzewo *vp-drzewa* jeśli distance(VP,P)+Eps>=bhl i distance(VP,P)-Eps>blh,
* przeszukuje lewe i prawe poddrzewo *vp-drzewa* jeśli distance(VP,P)+Eps>=bh; i distance(VP,P)-Eps<=blh.

Zatem w przypadku przedstawionym na rysunku 5 *metoda ograniczeń* przeszuka jedynie lewe poddrzewo *vp-drzewa*.

*Metoda ograniczeń* szybciej znajduje k-sąsiedztwo niż *metoda mediany*, ponieważ dzięki znajomości blh i bhl pozwala w pewnych przypadkach wcześniej zdecydować o braku konieczności eksploracji aktualnie rozpatrywanej gałęzi *vp-drzewa* niż *metoda mediany*.

# Implementacja struktury punktu

Podobnie jak w przypadku algorytmu *TI-k-Neighborhood-Index* przetestowałem wydajność *VP-Tree-Index* w zależności od implementacji punktu. W tabeli 11 oraz na rysunku 7 zamieszczono wyniki uruchomień algorytmu *VP-Tree-Index* z użyciem zarówno gęstej jak i rzadkiej implementacji punktu.

Wnioski płynące z wykonanych badań są takie same jak w przypadku algorytmu *TI-k-Neighborhood-Index* w rozdziale 6.2.1.2*.*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| zbioór | **liczba punktów** | ***VP-Tree-Index* REPREZENTACJA GĘSTA** | | | ***VP-Tree-Index* REPREZENTACJA RZADKA** | | |
| **wykonanie algorytmu** | **klasyfikacja** | **budowa indeksu** | **wykonanie algorytmu** | **klasyfikacja** | **budowa indeksu** |
|
| karypis\_sport | 1000 | 28,153 | 22,407 | 5,808 | 0,296 | 0,218 | 0,078 |
| 2000 | - | - | - | 0,988 | 0,822 | 0,166 |
| 4000 | - | - | - | 3,874 | 3,510 | 0,369 |
| 6000 | - | - | - | 8,689 | 8,107 | 0,577 |
| 8000 | - | - | - | 15,538 | 14,768 | 0,774 |
| karypis\_review | 500 | 8,398 | 5,673 | 2,735 | 0,270 | 0,078 | 0,192 |
| 1000 | 28,465 | 22,657 | 5,808 | 0,676 | 0,296 | 0,379 |
| 2000 | - | - | - | 1,981 | 1,326 | 0,660 |
| 3000 | - | - | - | 3,629 | 2,949 | 0,681 |
| 4000 | - | - | - | 5,840 | 5,247 | 0,593 |
| covtype | 10000 | 0,855 | 0,298 | 0,561 | 1,181 | 0,416 | 0,765 |
| 50000 | 5,083 | 2,241 | 2,770 | 6,926 | 3,000 | 3,973 |
| 100000 | 14,576 | 7,703 | 6,501 | 15,501 | 8,574 | 6,984 |
| 300000 | 107,732 | 70,103 | 36,422 | 92,888 | 65,031 | 28,034 |
| 500000 | 253,721 | 180,813 | 68,562 | 250,568 | 179,249 | 71,318 |
| cup98 | 10000 | 0,728 | 0,307 | 0,421 | 2,044 | 1,388 | 0,656 |
| 30000 | 4,174 | 2,812 | 1,366 | 11,679 | 9,537 | 2,153 |
| 50000 | 10,649 | 7,766 | 2,450 | 33,129 | 29,411 | 3,744 |
| 70000 | 21,992 | 16,790 | 3,710 | 46,878 | 41,204 | 5,731 |
| 90000 | 39,182 | 27,149 | 13,156 | 81,823 | 73,710 | 8,039 |

Tabela 10: Porównanie wydajności algorytmu *VP-Tree-Index* w zależności od implementacji punktu na przykładowych zbiorach danych. Tabela zawiera czasy wykonania (w sekundach) poszukiwań k=5 sąsiadów w przykładowych zbiorach danych z wykorzystaniem algorytmu *VP-Tree-Index* dla 10% losowo wybranych punktow zbioru danych. Notacja: ‘-‘ rezultat nie został osiągnięty w czasie mneijszym niż 3 godziny lub z powodu wyczerpania pamięci operacyjnej.

Rysunek 7: Porównanie wydajności algorytmu *VP-Tree-Index* w zależności od implementacji punktu na przykładowych zbiorach danych. Tab ela zawiera czasy wykonania (w sekundach) poszukiwań k=5 sąsiadów w przykładowych zbiorach danych z wykorzystaniem algorytmu *VP-Tree- Index* dla 10% losowo wybranych punktow zbioru danych

# *VP-Tree-Index* vs *TI-k\_Neighborhood-Index*

Na przykładowych zbiorach danych przetestowano algorytmy *VP-Tree-Index* oraz *TI-k-Neighborhood-Index*.

W tabeli 12 i na rysunku 8 zamieszczono czasy poszukiwań k-sąsiedztwa przez oba algorytmy.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| zbioór | **liczba punktów** | ***VP-Tree-Index*** | | | ***TI-k-Neighborhood-Index*** | | |
| **wykonanie algorytmu** | **klasyfikacja** | **budowa indeksu** | **wykonanie algorytmu** | **klasyfikacja** | **budowa indeksu** |
|
| arypis\_sport | 1000 | 0,296 | 0,218 | 0,078 | 0,206 | 0,204 | 0,000 |
| 2000 | 0,988 | 0,822 | 0,166 | 0,680 | 0,674 | 0,000 |
| 4000 | 3,874 | 3,510 | 0,369 | 2,825 | 2,813 | 0,000 |
| 6000 | 8,689 | 8,107 | 0,577 | 6,887 | 6,870 | 0,000 |
| 8000 | 15,538 | 14,768 | 0,774 | 13,226 | 13,202 | 0,000 |
| karypis\_review | 500 | 0,270 | 0,078 | 0,192 | 0,067 | 0,066 | 0,000 |
| 1000 | 0,676 | 0,296 | 0,379 | 0,250 | 0,246 | 0,000 |
| 2000 | 1,981 | 1,326 | 0,660 | 1,105 | 1,099 | 0,000 |
| 3000 | 3,629 | 2,949 | 0,681 | 2,394 | 2,385 | 0,000 |
| 4000 | 5,840 | 5,247 | 0,593 | 4,347 | 4,334 | 0,000 |
| covtype | 10000 | 0,855 | 0,298 | 0,561 | 0,548 | 0,391 | 0,000 |
| 50000 | 5,083 | 2,241 | 2,770 | 7,789 | 5,400 | 0,001 |
| 100000 | 14,576 | 7,703 | 6,501 | 19,994 | 15,052 | 0,002 |
| 300000 | 107,732 | 70,103 | 36,422 | 99,572 | 83,380 | 0,004 |
| 500000 | 253,721 | 180,813 | 68,562 | 261,503 | 234,340 | 0,005 |
| cup98 | 10000 | 0,728 | 0,307 | 0,421 | 0,393 | 0,272 | 0,000 |
| 30000 | 4,174 | 2,812 | 1,366 | 3,076 | 2,197 | 0,000 |
| 50000 | 10,649 | 7,766 | 2,450 | 7,291 | 5,371 | 0,001 |
| 70000 | 21,992 | 16,790 | 3,710 | 12,927 | 9,563 | 0,001 |
| 90000 | 39,182 | 27,149 | 13,156 | 19,646 | 15,316 | 0,002 |

Tabela 11: Porównanie wydajności algorytmów *VP-Tree-Index* i *TI-k-Neighborhood-Index.* Tabela zawiera czasy wykonania (w sekundach) poszukiwań k=5 sąsiadów w przykładowych zbiorach danych dla 10% losowo wybranych punktow zbioru danych.



Rysunek 8: Porównanie wydajności algorytmów VP-Tree-Index i TI-k-Neighborhood-Index. Tabela zawiera czasy wykonania (w sekundach) poszukiwań k=5 sąsiadów w przykładowych zbiorach danych dla 10% losowo wybranych punktow zbioru danych

Z rezultatów przeprowadzonych badań wynika, że algorytm *TI-k-Neighborhood-Index* wyszukuje k-sąsiedztwo szybciej niż *VP-Tree-Index*. Wyjątek stanowi zbiór covtype, w którego przypadku oba algorytmy wykonują się w zbliżonym czasie. Z przebiegów wykresów na rysunku 8 dla zbiorów różnych od covtype można wnioskować, że algorytm *TI-k-Neighborhood-Index* będzie wykonywał się tym szybciej niż *VP-Tree-Index* im większy będzie zbiór, w którym wyszukiwanych jest k-sąsiadów.

# *Testy algorytmuDBSCAN*

# Testy algorytmu *TI-DBSCAN*

# Implementacja algorytmu *TI-DBSCAN*

Na przykładowych zbiorach danych przetestowałem dwie wersje algorytmu TIDBSCAN: korzystającą ze struktury indeksu zbudowanego z iteratorów oraz opartą wyłącznie na oryginalnym zbiorze danych. W pierwszym przypadku sortowaniu poddany zostaje indeks, który następnie używany jest w celu dostępu do zbioru danych. W drugim przypadku sortowany jest zbiór danych i żaden indeks nie jest wykorzystywany w dostępie do jego elementów. W tabeli 13 i na rysunku 9 zamieszczono porównanie obu implementacji.

Z rezultatów uruchomień wynika, że w porównaniu do implementacji bez indeksu implementacja z indeksem iteratorów radzi sobie z grupowaniem tym gorzej im więcej rekordów posiada zbiór, który poddawany jest grupowaniu. Największa różnica w czasie wykonania algorytmu wystąpiła przy grupowaniu 30000 punktów zbioru cup98. Dla tego konkretnego przypadku implementacja z bezpośrednim dostępem do danych wykonuje się blisko 4 razy szybciej niż implementacja korzystająca z indeksu. Najmniej znaczące, choć wciąż wskazujące na przewagę implementacji z bezpośrednim dostępem, różnice w czasach wykonania grupowania widoczne są dla zbiorów danych tekstowych karypis\_sport i karypis\_review.

Zgodnie ze szczegółowymi danymi z tabeli 13 w implementacji z indeksem sortowanie wykonuje się do dwóch rzędów wielkości szybciej niż implementacja z bezpośrednim dostępem. Co więcej, koszty utworzenia indeksu są tak niskie, że mogą zostać pominięte w dalszych rozważaniach.

Różnica w czasach sortowania między obiema implementacjami bierze się stąd, że do sortowania obiektów wykorzystywana jest standardowa dla języka C++ funkcja sortowania wektorów, tj. *std::sort*, oparta o algorytm quicksort. Funkcja sortowania biblioteki standardowej korzysta z operacji *swap* zamieniającej miejscami elementy w wektorze, która wywołuje konstruktor kopiujący obiektu będącego elementem sortowanego wektora. W przypadku gdy głęboko kopiowany jest obiekt klasy *Punkt*, to uruchamiane są konstruktory kopiujące wszystkich obiektów będących jego polami, w szczególności kolekcji przechowujących wartości wymiarów. Dlatego, wykonanie konstruktora kopiującego iteratora jest szybsze niż wykonanie konstruktora kopiującego obiektu punktu. Im większy zbiór jest sortowany tym częściej funkcja *swap* jest wywoływana, tym samym częściej wykonywany jest konstruktor kopiujący. W skutek tego, czas sortowania zbioru obiektów punktu jest dłuższy niż czas sortowania zbioru iteratorów.

Jednakże, bezpośrednie sortowanie zbioru danych nie jest na tyle wolniejsze od sortowania indeksu niż różnica czasu wykonania grupowania wynikająca z dostępu do danych. Dostęp przez posortowany indeks jest wolniejszy niż bezpośredni dostęp do posortowanego zbioru ponieważ kolejne przeglądanie elementów powoduje „skakanie” po pamięci. W kolejnych rozważaniach będę się odnosił do algorytmu *TI-DBSCAN* jako do implementacji korzystającej bez indeksu iteratorów.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| zbioór | **liczba punktów** | ***TI-DBSCAN* INDEX EPS=0,05** | | | | | ***TI-DBSCAN* NO INDEX EPS=0,05** | | | | |
| **wykonanie algorytmu** | **grupowanie** | **budowa indeksu** | **obliczanie odległości** | **sortowanie** | **wykonanie algorytmu** | **grupowanie** | **budowa indeksu** | **obliczanie odległości** | **sortowanie** |
|
| karypis\_sport | 1000 | 0,843 | 0,839 | 0,000 | 0,003 | 0,000 | 0,856 | 0,836 | N/A | 0,003 | 0,017 |
| 2000 | 1,625 | 1,621 | 0,000 | 0,003 | 0,001 | 1,639 | 1,592 | N/A | 0,004 | 0,044 |
| 4000 | 7,964 | 7,955 | 0,000 | 0,008 | 0,001 | 7,650 | 7,519 | N/A | 0,009 | 0,114 |
| 6000 | 20,876 | 20,855 | 0,000 | 0,011 | 0,002 | 19,957 | 19,581 | N/A | 0,011 | 0,369 |
| 8000 | 61,519 | 61,497 | 0,000 | 0,019 | 0,003 | 54,700 | 54,193 | N/A | 0,019 | 0,466 |
| karypis\_review | 500 | 0,105 | 0,104 | 0,000 | 0,001 | 0,000 | 0,122 | 0,104 | N/A | 0,002 | 0,016 |
| 1000 | 0,691 | 0,687 | 0,000 | 0,003 | 0,000 | 0,728 | 0,679 | N/A | 0,003 | 0,045 |
| 2000 | 3,847 | 3,840 | 0,000 | 0,005 | 0,000 | 3,877 | 3,802 | N/A | 0,005 | 0,070 |
| 3000 | 8,759 | 8,750 | 0,000 | 0,008 | 0,001 | 8,656 | 8,492 | N/A | 0,007 | 0,156 |
| 4000 | 16,494 | 16,482 | 0,000 | 0,011 | 0,001 | 16,395 | 16,093 | N/A | 0,010 | 0,223 |
| covtype | 1000 | 0,024 | 0,019 | 0,000 | 0,005 | 0,000 | 0,234 | 0,016 | N/A | 0,011 | 0,203 |
| 5000 | 0,914 | 0,837 | 0,000 | 0,078 | 0,001 | 0,920 | 0,780 | N/A | 0,010 | 0,125 |
| 10000 | 3,140 | 2,695 | 0,000 | 0,442 | 0,004 | 2,574 | 2,298 | N/A | 0,011 | 0,270 |
| 30000 | 19,732 | 19,700 | 0,000 | 0,023 | 0,009 | 17,321 | 15,985 | N/A | 0,021 | 1,362 |
| 60000 | 63,537 | 63,473 | 0,001 | 0,044 | 0,020 | 49,141 | 46,744 | N/A | 0,047 | 2,465 |
| cup98 | 1000 | 2,711 | 2,696 | 0,000 | 0,014 | 0,001 | 1,665 | 1,475 | N/A | 0,003 | 0,175 |
| 5000 | 70,497 | 70,384 | 0,000 | 0,111 | 0,002 | 70,747 | 60,992 | N/A | 0,115 | 9,629 |
| 10000 | 960,283 | 959,879 | 0,000 | 0,397 | 0,006 | 435,661 | 430,565 | N/A | 0,419 | 7,006 |
| 30000 | 10601,600 | 10601,500 | 0,000 | 0,022 | 0,012 | 4696,180 | 4694,960 | N/A | 0,023 | 1,288 |
| 60000 | - | - | - | - | - | - | - | - | - | - |

Tabela 12: Porównanie wydajności implementacji algorytmu TI-DBSCAN na przykładowych zbiorach danych. Tabela zawiera czasy wykonania (w sekundach) grupowań z parametrami MinPts=5 oraz Eps=0,05\*przekątna danego zbioru. Notacja: ‘-‘ rezultat nie został osiągnięty w czasie mneijszym niż 3 godziny lub z powodu wyczerpania pamięci operacyjnej.



Rysunek 9: Porównanie wydajności implementacji algorytmu TI-DBSCAN na przykładowych zbiorach danych. Tabela zawiera czasy wykonania (w sekundach) grupowań z parametrami MinPts=5 oraz Eps=0,05\*przekątna danego zbioru.

# Implementacja struktury punktu

Podobnie jak w przypadku algorytmu *TI-k-Neighborhood-Index* i *Vp-Tree-Index* przetestowałem wydajność *TI-DBSCAN* w zależności od implementacji punktu. W tabeli 14 oraz na rysunku 10 zamieszczono wyniki uruchomień algorytmu *TI-DBSCAN* z użyciem zarówno gęstej jak i rzadkiej implementacji punktu.

Wnioski płynące z wykonanych badań są takie same jak w przypadku algorytmu *TI-k-Neighborhood-Index* w rozdziale 6.2.1.2. Warto zwrócić uwagę na anomalię zbioru covtype w przypadku którego sprawniej wykonywały się uruchomienia wykorzystujące rzadką reprezentację punku. Jednakże różnica w czasie wykonania algorytmów korzystających implementacji punktu jest na tyle mała, że w świetle pozostałych wyników wnioski z uruchomień nie odbiegają od wniosków z rozdziału 6.2.1.2.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| zbioór | **liczba punktów** | ***TI-DBSCAN* DENSE EPS=0,05** | | | | ***TI-DBSCAN* SPARSE EPS=0,05** | | | |
| **wykonanie algorytmu** | **grupowanie** | **obliczanie odległości** | **sortowanie** | **wykonanie algorytmu** | **grupowanie** | **obliczanie odległości** | **sortowanie** |
|
| karypis\_sport | 1000 | 98,565 | 95,095 | 0,235 | 3,146 | 0,856 | 0,836 | 0,003 | 0,017 |
| 2000 | - | - | - | - | 1,639 | 1,592 | 0,004 | 0,044 |
| 4000 | - | - | - | - | 7,650 | 7,519 | 0,009 | 0,114 |
| 6000 | - | - | - | - | 19,957 | 19,581 | 0,011 | 0,369 |
| 8000 | - | - | - | - | 54,700 | 54,193 | 0,019 | 0,466 |
| karypis\_review | 500 | 19,938 | 18,494 | 0,115 | 1,329 | 0,122 | 0,104 | 0,002 | 0,016 |
| 1000 | 110,317 | 106,999 | 0,234 | 3,060 | 0,728 | 0,679 | 0,003 | 0,045 |
| 2000 | - | - | - | - | 3,877 | 3,802 | 0,005 | 0,070 |
| 3000 | - | - | - | - | 8,656 | 8,492 | 0,007 | 0,156 |
| 4000 | - | - | - | - | 16,395 | 16,093 | 0,010 | 0,223 |
| covtype | 1000 | 0,234 | 0,016 | 0,011 | 0,203 | 0,093 | 0,031 | 0,000 | 0,062 |
| 5000 | 0,920 | 0,780 | 0,010 | 0,125 | 1,438 | 1,033 | 0,119 | 0,193 |
| 10000 | 2,574 | 2,298 | 0,011 | 0,270 | 3,681 | 3,447 | 0,073 | 0,125 |
| 30000 | 17,321 | 15,985 | 0,021 | 1,362 | 22,937 | 22,490 | 0,031 | 0,422 |
| 60000 | 49,141 | 46,744 | 0,047 | 2,465 | 70,564 | 69,493 | 0,052 | 0,931 |
| cup98 | 1000 | 1,665 | 1,475 | 0,003 | 0,175 | 0,372 | 0,344 | 0,001 | 0,023 |
| 5000 | 70,747 | 60,992 | 0,115 | 9,629 | 8,310 | 8,231 | 0,007 | 0,072 |
| 10000 | 435,661 | 430,565 | 0,419 | 7,006 | 33,074 | 32,904 | 0,014 | 0,156 |
| 30000 | 4696,180 | 4694,960 | 0,023 | 1,288 | 319,353 | 318,786 | 0,047 | 0,530 |
| 60000 | - | - | - | - | 1345,570 | 1344,310 | 0,085 | 1,177 |

Tabela 13: Porównanie wydajności algorytmu *TI-DBSCAN* w zależności od implementacji punktu na przykładowych zbiorach danych. Tabela zawiera czasy wykonania (w sekundach) grupowań z parametrami MinPts=5 oraz Eps=0,05\*przekątna danego zbioru. Notacja: ‘-‘ rezultat nie został osiągnięty w czasie mneijszym niż 3 godziny lub z powodu wyczerpania pamięci operacyjnej.



Rysunek 10: Porównanie wydajności algorytmu TI-DBSCAN w zależności od implementacji punktu na przykładowych zbiorach danych. Tabela zawiera czasy wykonania (w sekundach) grupowań z parametrami MinPts=5 oraz Eps=0,05\*przekątna danego zbioru.

# Badanie wydajności *TI-DBSCAN* w zależności od wartości Eps

W tabeli 15 i na rysunku 11 przedstawiono czasy wykonania uruchomień algorytmu *TI-DBSCAN* dla wartości parametrów MinPts=5 i Eps równe 0, 1/20 przekątnej dziedziny i przekątnej dziedziny. Dla wartości parametru Eps równego przekątnej dziedziny algorytm TI-DBSCAN wykonuje się najwolniej, ponieważ wszystkie punkty dziedziny należą do otoczenia epsilonowego dowolnego punktu. W takim przypadku nierówność trójkąta nie przyspiesza procesu wyznaczania otoczenia epsilonowego, które musi zostać wyznaczone dla każdego punktu. Najbardziej selektywny przypadek następuje gdy Eps jest równy 0. Rezultatem grupowania są klastry takich samych punktów o odpowiedniej liczności. Z uwagi na najwyższą możliwą selektywność nierówność trójkąta wyklucza z przestrzeni potencjalnych sąsiadów dużą liczbę punktów znakomicie przyczyniając się do zmniejszenia czasu wyznaczania otoczenia danego punktu. Dla danego MinPts i dowolnego Eps większego od Eps\_0= 0 i mniejszego od przekątnej dziedziny Eps\_max czas wykonania grupowania danego zbioru będzie nie wolniejszy niż czas grupowania dla Eps\_max i nie szybszy niż czas wykonania grupowania dla Eps\_0. W zależności od charakterystyki zbioru czas ten może być bliższy rezultatom osiąganym dla Eps\_max lub Eps\_0. Przykładami takich sytuacji są eksperymenty wykonane dla Eps równego 1/20 przekątnej dziedziny. Dla większości eksperymentów czas wykonania algorytmu był mniejszy lecz bliższy temu dala Eps\_max. Wyjątek stanowi zbiór covtype, dla którego czasy wykonania eksperymentów są znacznie szybsze niż dla Eps\_max.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| zbioór | **liczba punktów** | **TI-DBSCAN EPS=MAX** | | | | **TI-DBSCAN EPS=1/20MAX** | | | | **TI-DBSCAN EPS=0** | | | |
| **wykonanie algorytmu** | **grupowanie** | **obliczanie odległości** | **sortowanie** | **wykonanie algorytmu** | **grupowanie** | **obliczanie odległości** | **sortowanie** | **wykonanie algorytmu** | **grupowanie** | **obliczanie odległości** | **sortowanie** |
|
| karypis\_sport | 1000 | 1,669 | 1,643 | 0,006 | 0,021 | 0,856 | 0,836 | 0,003 | 0,017 | 0,025 | 0,004 | 0,004 | 0,018 |
| 2000 | 6,494 | 6,447 | 0,003 | 0,044 | 1,639 | 1,592 | 0,004 | 0,044 | 0,058 | 0,009 | 0,003 | 0,046 |
| 4000 | 27,239 | 27,067 | 0,008 | 0,115 | 7,650 | 7,519 | 0,009 | 0,114 | 0,148 | 0,024 | 0,008 | 0,116 |
| 6000 | 62,661 | 62,286 | 0,011 | 0,369 | 19,957 | 19,581 | 0,011 | 0,369 | 0,439 | 0,045 | 0,011 | 0,383 |
| 8000 | 113,865 | 113,117 | 0,019 | 0,466 | 54,700 | 54,193 | 0,019 | 0,466 | 0,564 | 0,072 | 0,019 | 0,473 |
| karypis\_review | 500 | 0,532 | 0,515 | 0,001 | 0,016 | 0,122 | 0,104 | 0,002 | 0,016 | 0,020 | 0,002 | 0,002 | 0,017 |
| 1000 | 2,117 | 2,071 | 0,003 | 0,043 | 0,728 | 0,679 | 0,003 | 0,045 | 0,051 | 0,004 | 0,002 | 0,045 |
| 2000 | 8,967 | 8,881 | 0,005 | 0,071 | 3,877 | 3,802 | 0,005 | 0,070 | 0,085 | 0,009 | 0,006 | 0,070 |
| 3000 | 20,270 | 20,105 | 0,007 | 0,154 | 8,656 | 8,492 | 0,007 | 0,156 | 0,178 | 0,017 | 0,008 | 0,154 |
| 4000 | 36,300 | 36,063 | 0,010 | 0,224 | 16,395 | 16,093 | 0,010 | 0,223 | 0,268 | 0,027 | 0,010 | 0,229 |
| covtype | 1000 | 7,439 | 6,662 | 0,025 | 0,568 | 0,234 | 0,016 | 0,011 | 0,203 | 0,218 | 0,010 | 0,010 | 0,203 |
| 5000 | 748,364 | 748,234 | 0,004 | 0,160 | 0,920 | 0,780 | 0,010 | 0,125 | 0,266 | 0,125 | 0,005 | 0,135 |
| 10000 | 4611,050 | 4610,710 | 0,007 | 0,360 | 2,574 | 2,298 | 0,011 | 0,270 | 0,728 | 0,458 | 0,010 | 0,260 |
| 30000 | - | - | - | - | 17,321 | 15,985 | 0,021 | 1,362 | 3,072 | 1,695 | 0,027 | 1,355 |
| 60000 | - | - | - | - | 49,141 | 46,744 | 0,047 | 2,465 | 5,819 | 3,344 | 0,042 | 2,433 |
| cup98 | 1000 | 7,601 | 6,826 | 0,026 | 0,581 | 1,665 | 1,475 | 0,003 | 0,175 | 0,250 | 0,016 | 0,012 | 0,223 |
| 5000 | 733,807 | 733,654 | 0,004 | 0,158 | 70,747 | 60,992 | 0,115 | 9,629 | 0,220 | 0,129 | 0,008 | 0,088 |
| 10000 | 4378,300 | 4378,110 | 0,007 | 0,204 | 435,661 | 430,565 | 0,419 | 7,006 | 0,842 | 0,530 | 0,000 | 0,312 |
| 30000 | - | - | - | - | 4696,180 | 4694,960 | 0,023 | 1,288 | 3,068 | 2,044 | 0,016 | 1,009 |
| 60000 | - | - | - | - | - | - | - | - | 9,209 | 6,453 | 0,046 | 2,715 |

Tabela 14: Porównanie wydajności algorytmu TI-DBSCAN w zależności od wartości parametru Eps. Tabela zawiera czasy wykonania (w sekundach) grupowań z parametrami MinPts=5 oraz Eps=[1; 0,05; 0]\*przekątna danego zbioru. Notacja: ‘-‘ rezultat nie został osiągnięty w czasie mneijszym niż 3 godziny lub z powodu wyczerpania pamięci operacyjnej.

Rysunek 11: Porównanie wydajności algorytmu TI-DBSCAN w zależności od wartości parametru Eps. Tabela zawiera czasy wykonania (w sekundach) grupowań z parametrami MinPts=5 oraz Eps=[1; 0,05; 0]\*przekątna danego zbioru

# Bibliografia

1. Kryszkiewicz, M., Lasek, P.: TI-DBSCAN: Clustering with DBSCAN by Means of the Triangle Inequality, ICS Research Report, Warsaw University of Technology, April (2010)
2. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A Density-Based Algorithm of Discovering Clusters in Large Spatial Database with Noise. In: Proc. Of KDD’96, Portland (1996) 226-231
3. Kryszkiewicz, M.: Efficient Determination of Neighborhoods Defined in Terms of Cosine Similarity Measure, ICS Research Report, Warsaw University of Technology, April (2011)
4. Kryszkiewicz, M., Lasek, P.:A Neighborhood-Based Clustering by Means of the Triangle Inequality and Reference Points, ICS Research Report, Warsaw University of Technology
5. Yianilos, P. N.: Data Structures and Algorithms for Nearest Neighbor Search in General Metrics Spaces, The NEC Research Institute
6. Bozkaya, T., Ozsoyoglu M.: Distance-based indexing for high-dimensional metric spaces, Case Western Reserve University
7. Blackard J.A (1999, lipiec) The Forest CovType dataset. [Online ]. <http://ftp.ics.uci.edu/pub/machine-learning-databases/covtype/covtype.info>
8. Parsa I. (1999, luty) The UCI KDD Archive: KDD Cup 1988 Data. [Online]. <http://kdd.ics.uci.edu/databases/kddcup98/kddcup98.html>
9. Karypis G. (2002, sierpień) The various datasets used in evaluating the performance of CLUTO's clustering algorithms. [Online]. <http://glaros.dtc.umn.edu/gkhome/cluto/cluto/download>