Politechnika Śląska w Gliwicach

Zaawansowane Biblioteki Programistyczne
15 stycznia 2017

Grupowanie metodą k-średnich

Autor:
Bartłomiej Buchała

Informatyka SSM, semestr II
Rok akademicki 2016/2017
Grupa OS1

Spis treści

1. Wstęp

1.1 Temat projektu

Celem projektu było stworzenie algorytmu wykonującego grupowanie elementów metodą k-średnich w języku C++.

Program miał spełniać następujące warunki:

- Elementy wejściowe zdefiniowane przez zakres.
- Zakres elementu określany jest przez iteratory swobodnego dostępu.
- Na końcu wykonania algorytmu elementy wejściowe zostaną posortowane według grup.
- Funkcje odległości oraz uśredniająca przekazywane do algorytmu.
- Algorytm ma zwrócić wektor zakresów odpowiadających znalezionym grupom (wektor iteratorów wskazujących na początki lub koniec grup).
- Funkcja/Klasa imlementująca algorytm ma być szablonowa i pozwalać na grupowanie danych dowolnego typu.

1.2 Zasada działania algorytmu

Algorytm k-średnich (nazywany również algorytmem centroidów) jest jednym z algorytmów stosowanych w analizie skupień – analizie polegającej na szukaniu i wyodrębnianiu grup obiektów podobnych (skupień). Należy do grupy algorytmów niehierarchicznych (charakteryzuje się koniecznością podania liczby skupień).

Przy pomocy metody k-średnich zostanie utworzonych k różnych możliwie odmiennych skupień. Algorytm ten polega na przenoszeniu obiektów ze skupienia do skupienia tak długo aż zostaną zoptymalizowane zmienności wewnątrz skupień oraz pomiędzy skupieniami (lub alternatywnie zostanie wykonana maksymalna liczba iteracji).

Działanie algorytmu można uprościć w kilku punktach:

1. **Ustalenie liczby grup** – wybór ustalany odgórnie, w zależności od potrzeb. Jedną z metod ustalenia ilości skupień jest umowny jej wybór i ewentualna późniejsza zmiana tej liczby w celu uzyskania lepszych wyników.
2. **Ustalenie początkowych wartości środków skupień (centroidów)** – centroidy można dobrać na kilka sposobów:
   - Losowy wybór k obserwacji.
   - Wybór k pierwszych obserwacji.
   - Wybór taki, aby zmaksymalizować odległości pomiędzy centroidami.
   - Kilkakrotne uruchomienie algorytmu (z losowym wyborem) i wybór najlepszego spośród modeli.

   Jeżeli nie zna się dokładnie typu grupowanych danych, zaleca się aby początkowymi wartościami środków skupień były losowo wybrane pośród elementów wejściowych.
3. **Obliczenie odległości obiektów od środków skupień** – określenie odległości elementów od centroidów przy użyciu określonej metryki (najczęściej stosuje się Euklidesową).
4. **Przypisanie obiektów do skupień** – Dla danej obserwacji porównywane są odległości od wszystkich centroidów i przypisanie elementu do skupienia, do którego środka ma najbliżej.
5. **Ustalenie nowych współrzędnych centroidów** – nowe wartości dla środków skupień wyznaczane zą sa pomocą funkcji uśredniającej (z reguły jest to średnia arytmetyczna).
6. **Wykonywanie kroków 3-5 do czasu, aż warunek zatrzymania zostanie spełniony** – warunkiem stopu może być ilość iteracji zadana na początku lub brak przesunięć obiektów pomiędzy skupieniami w 2 następujących po sobie iteracjach.

2. Analiza problemu

Przed przystąpieniem do pracy nad algorytmem, należało rozwiązać kilka problemów projektowych.

2.1 Problem doboru punktów początkowych

Podstawowym krokiem przy każdym wywołaniu algorytmu jest ustalenie punktów początkowych. Wyniki algorytmu k-średnich silnie zależą od wyboru punktów początkowych, jednak nie ma ściśle określonej reguły, jak je należy wybierać. W zależności od startowej wartości centroidów, wyniki grupowania mogą być inne z każdym uruchomieniem algorytmu (nawet przy identycznych parametrach). Początkowo wydaje się, że wybór losowych współrzędnych jest dobrym rozwiązaniem. Przeszkodą jest jednak założenia generyczności algorytmu – istnieje możliwość wyboru losowego punktu pośród wartości typu int lub klasy reprezentującej punkt na płaszczyźnie dwuwymiarowej, jednak zadanie jest znacznie utrudnione w przypadku np. typu string lub klasy zewnętrznej pochodzącej z innego projektu. W takim przypadku warunkiem uniemożliwiającym wybór losowych współrzędnych jest brak wiedzy na temat budowy typu lub nawet niemożliwość utworzenia obiektu o losowych współrzędnych.

Rozwiązaniem problemu jest wybór losowych elementów z zakresu podanego do grupowania. W pierwszej iteracji środki skupień znajdują się dokładnie w tym samym miejscu, w którym istnieją niektóre z danych wejściowych.

2.2 Miara odległości

Algorytm ma zapewniać generyczność – dlatego dla typu danych na którym chcemy wykonać grupowanie musi istnieć miara odległości (metryka), pozwalająca określić odległość pomiędzy dwoma dowolnymi elementami tego typu (w najczęstszym scenariuszu - pomiędzy daną wejściową, a centroidem). Zakłada się, że odległość ewaluowana jest jako wartość typu `double`. W celu zapewnienia poprawności działania algorytmu, konieczne jest przekazanie do funkcji grupującej obiektu funkcyjnego (z przeciążonym operatorem `()`) zwracającą wartość zmiennoprzecinkową, będącą miarą odległości pomiędzy 2 elementami. Przykład implementacji dla typu `int` znajduje się poniżej.

```
/// <summary>
/// Obiekt funkcyjny, bedacy miara odleglosci pomiedzy dwoma wartosciami typu int.
/// </summary>
struct Int_distance
{
        inline double operator()(int p1, int p2)
        {
                return (double)abs(p1 - p2);
        }
};
```

Sposób w jaki implementowana jest metryka leży w pełni w kwestii osoby korzystającej z algorytmu. Dla punktów w przestrzeni dwuwymiarowej może to być metryka euklidesowa lub miejska (Manhattan), dla typu char odległość dzieląca znaki w tablicy ASCII, itp.

2.3 Funkcja uśredniająca

Podobnie jak w przypadku metryki, z powodu generyczności funkcji konieczne jest zadeklarowanie obiektu funkcyjnego określającego, w jaki sposób obliczany będzie nowy centroid spośród n elementów należących do skupienia. Funktor powinien pozwalać na 2 operacje:

- Kumulację obiektów – dodanie nowego obiektu (przekazanego jako referencję) do skumulowanej wartości oraz inkrementacja licznika dodanych punktów. Realizowane przez przeciążony operator +=.
- Uśrednienie – wyliczenie nowych współrzędnych centroidu na podstawie skumulowanej wartości oraz liczby skumulowanych elementów. Realizowane przez przeciążony operator `()`. Jako parametr należy przekazać obiekt obliczanego typu – jest to poprzednia wartość środku skupienia. Jeżeli liczba obiektów należących do danej grupy jest większa niż 0, przekazany obiekt jest aktualizowany do wartości nowego centroidu. Jeżeli liczba elementów w grupie jest równa 0, centroid nie jest nadpisywany.

Przykład implementacji uśredniającego obiektu funkcyjnego dla typu `int` znajduje się poniżej.

```
struct Int_average
{
        /// Wartosc domyslna nowego centroidu.
        int newCentroid = 0;

        /// Licznik kumulowanych wartosci.
        int count = 0;

        /// <summary>
        /// Przeciazony operator (), wyliczajacy nowa wartosc centroidu usredniona z n
            elementow.
        /// </summary>
        /// <param name="oldCentroid">Aktualna wartosc centroidu</param>
        inline void operator()(int &oldCentroid)
        {
                if (count != 0)
                        newCentroid = (int)(newCentroid / count);
                else
                        newCentroid = oldCentroid;

                oldCentroid = newCentroid;

                newCentroid = 0;
                count = 0;
        }

        /// <summary>
        /// Przeciazony operator +=, kumulujacy kolejna wartosc.
        /// </summary>
        /// <param name="value">Kolejna kumulowana wartosc.</param>
        inline void operator +=(const int & value)
        {
                newCentroid += value;

                count++;
        }
};
```

Warunkiem koniecznym do poprawnego działania algorytmu jest zadeklarowanie obiektu funkcyjnego z przeciążonymi operatorami () oraz += przyjmującymi po jednym parametrze typu tożsamego do grupowanych danych. Reszta szczegółów dotyczących implementacji leży w kwestii użytkownika.

2.4 Warunki stopu

Zaimplementowana funkcja pozwala na wybór jednego z dwóch (lub obu jednocześnie) warunku zatrzymania wykonywania algorytmu.

- Maksymalna liczba iteracji – program kończy grupowanie elementów po wykonaniu określonej liczby iteracji zadanej jako parametr wejściowy.
- Osiągnięcie stanu stabilnego – jako stan stailny określamy sytuację, w której w dwóch kolejnych iteracjach nie nastąpi żadna zmiana przynależności do skupienia.

Podanie warunku stopu jako parametr odbywa się przez typ wyliczeniowy.

3. Specyfikacja zewnętrzna

3.1 Wykorzystanie algorytmu

Program został przygotowany zgodnie z myślą wykorzystania podobną do algorytmu sortowania znajdującego się w standardowej bibliotece wzorców (`std::sort()`). Wykorzystanie algorytmu odbywa się w kilku krokach:

1. Dołączenie pliku nagłówkowego do projektu poprzez zastosowanie komendy:

```
#include "K_means.h"
```

2. Utworzenie elementu klasy szablonowej o konkretnym typie:

```
K_means<Typ_Danych> k_means;
```

Gdzie <Typ_Danych> oznacza dowolny, dostarczony przez użytkownika typ danych.

3. Wykonanie algorytmu grupowania k-średnich

```
result = k_means.Group(first, last, distanceMeasure, groupAverage, k, MaxIterations,
    StopCondition, PrintOutput);
```

Elementem zwróconym będzie wektor iteratorów wskazujących na pierwsze elementy grup posortowanych elementów.

4. Opcjonalnym krokiem jest wywołanie wyświetlenia elementów kolekcji za pomocą funkcji DisplayCollection. Aby funkcja działała poprawnie, elementy w zakresie muszą implementować operator «.

```
DisplayCollection(Iterator first, Iterator last)
```

3.2 Opis parametrów wywołania funkcji Group

first – jest iteratorem swobodnego dostępu (używanym w takich strukturach jak na przykład vector czy deque ze standardowej biblioteki wzorców) wskazującym na pierwszy element z grupowanego zakresu.

last – jest iteratorem swobodnego dostępu (używanym w takich strukturach jak na przykład vector czy deque ze standardowej biblioteki wzorców) wskazującym na ostatni element z grupowanego zakresu.

distanceMeasure – obiekt funkcyjny opisujący metrykę pomiędzy elementami danego typu. Więcej informacji można znaleźć w podrozdziale 2.2.

groupAverage – obiekt funkcyjny opisujący funkcję uśredniającą elementy danego typu. Więcej informacji można znaleźć w podrozdziale 2.3.

k – określa maksymalną liczbę iteracji algorytmu. W przypadku, gdy parametr `StopCondition` ustawiony jest na `StableState`, wartość tego parametru jest nieistotna.

StopCondition – określa warunek stopu (2.4). Jest to argument typu wyliczeniowego i przyjmuje jedną z 3 wartości:

- MaxIterations – warunkiem stopu jest wykonanie odpowiedniej liczby iteracji (podanej w parametrze `k`).
- StableState – program wykonuje się tak długo, aż w kolejnych iteracjach nie nastąpi zmiana przynależności do skupienia żadnego pośród wszystkich grupowanych elementów.
- Both – połączenie powyższych. Program wykonuje się identycznie jak w przypadku `StableState`, jednak może się zakończyć wcześniej jeżeli została wykonana określona liczba iteracji.

PrintOutput – określa, czy informacje o współrzędnych centroidów i przynależności punktów do skupień mają być wypisywane na strumień wyjściowy. Uwaga: może znacznie zwiększyć czas wykonywania algorytmu.

Przykładowe wywołanie funkcji dla wektora o elementach typu całkowitego:

```
result = k_means.Group(vec.begin(), vec.end(), Point2D_distance(), Point2D_average(), 4, 3,
    StableState, false);
```

Gdzie `vec` jest wektorem ze standardowej biblioteki wzorców zawierającym wartości typu `int`, a `Point2D_distance()` i `Point2D_average()` są obiektami funkcyjnymi stworzonymi zgodnie z zasadami opisanymi w podrozdziałach 2.2 i 2.3.

3.3 Opis parametrów wywołania funkcji DisplayCollection

first – jest iteratorem swobodnego dostępu (używanym w takich strukturach jak na przykład vector czy deque ze standardowej biblioteki wzorców) wskazującym na pierwszy element z grupowanego zakresu.

last – jest iteratorem swobodnego dostępu (używanym w takich strukturach jak na przykład vector czy deque ze standardowej biblioteki wzorców) wskazującym na ostatni element z grupowanego zakresu.

Przykładowe wywołanie funkcji dla wektora o elementach typu całkowitego:

```
k_means.DisplayCollection(vec.begin(), vec.end());
```

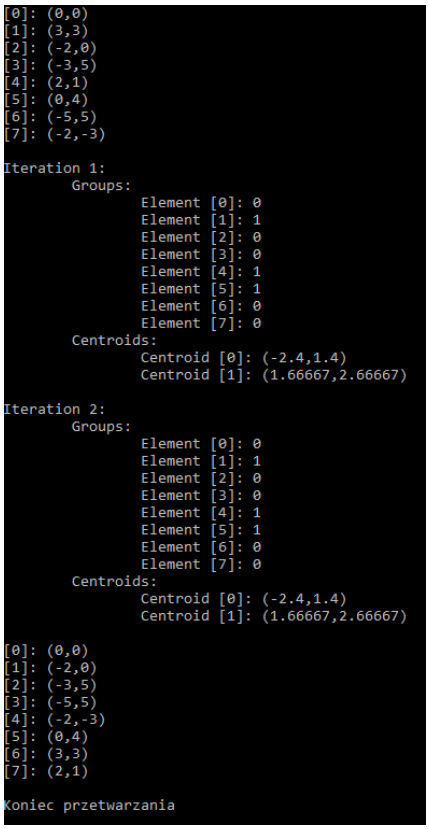
Gdzie `vec` jest wektorem ze standardowej biblioteki wzorców zawierającym wartości typu `int`.

3.4 Przykład scenariusza użycia

Dla poniższego kodu:

```
K_means<Point_2D> k_means;
vector<Point_2D> vec = vector<Point_2D>();

//Fill vec with data

k_means.DisplayCollection(vec.begin(), vec.end());

auto result = k_means.Group(vec.begin(), vec.end(), Point2D_distance(),
    Point2D_average(), 3, 2, StableState, true);

k_means.DisplayCollection(vec.begin(), vec.end());
```

Wynik działania będzie następujący:



Rysunek 1: Przykładowe wywołanie algorytmu

4. Specyfikacja wewnętrzna

4.1 Struktura projektu

Solucja aplikacji zawiera 6 plików. Większość deklaracji funkcji znajduje się w plikach nagłówkowych – ze względu na generyczność klasy niemożliwa jest deklaracja fragmentów kodu w plikach źródłowych. Kompilator tworzy klasy szablonowe na etapie kompilacji.

main.cpp jest plikiem poglądowym i zawiera scenariusze testowe wykorzystania algorytmu. Nie stanowi integralnej części reszty projektu i służy jedynie pokazaniu przykładowych użyć algorytmu.

bitmap_image.hpp – plik biblioteki zewnętrznej pochodzący ze strony `http://www.partow.net/programming/bitmap/index.html`. Dzięki funkcjom zadeklarowanym w tym pliku możliwe jest wczytywanie bitmap do aplikacji napisanej w języku C++ i ich modyfikacja/zapis. Fragmenty tej biblioteki wykorzystywane są w pliku `main.cpp` do testowania algorytmu na bitmapach.

IntTest.h – plik zawierający obiekty funkcyjne metryki i funkcji uśredniającej dla liczb całkowitych.

Point_2D.h – zawiera deklaracje następujących elementów:

- Klasa **Point_2D** – klasa reprezentująca punkt na płaszczyźnie euklidesowej. Posiada 2 pola zmiennoprzecinkowe (x i y), odpowiadająca odpowiednio współrzędnym na osiach x i y.
- Przeciążony **operator «** dla typu Point_2D – pozwala na wypisanie wyżej wymienionej klasy na strumień wyjściowy.
- Obiekt funkcyjny **Point2D_distance** – funktor określający w jaki sposób ma zostać obliczona odległość pomiędzy dwoma obiektami klasy Point_2D.
- Obiekt funkcyjny **Point2D_average** – funktor określający w jaki sposób kumulować oraz uśrednić wartość z wielu obiektów typu Point_2D.

K_means.h – plik zawierający główną klasę K_means szerzej opisaną w punkcie 4.2.

4.2 Klasa K_means

4.2.1 Typy

returnIterator – jest typem zwracanym prze funkcję `Group`. W rzeczywistości jest to iterator wektora, którego elementy wskazują na wartości typu zdefiniowanego przez użytkownika (elementy wejściowe).

4.2.2 Pola

Centroids – współrzędne centroidów (środków grup). Zmieniają się co iterację.

currentGroupId – tablica wartości całkowitych przechowująca informację o aktualnej przynależności danych do grup. Wartość i-tej komórki odpowiada numerowi grupy do której należy i-ty element.

nextGroupId – tablica wartości całkowitych przechowująca informację o przynależności danych do grup w następnej iteracji. Wartość i-tej komórki odpowiada numerowi grupy do której będzie należeć i-ty element.

distancesMatrix – macierz wartości zmiennoprzecinkowych określająca odległość danych od środków grup. Indeks kolumny odpowiada numerowi grupy, a indeks wiersza numerowi elementu.

returnValues – wektor iteratorów typu `returnIterator` wskazujących na elementy będące kolejnymi początkami grup.

iterationCounter – licznik iteracji algorytmu, służy do sprawdzania warunku stopu.

groupNumber – liczba grup (skupień) podanych przez użytkownika.

elementCount – liczba elementów znajdujących się w zakresie określonym iteratorami początku i końca. Wyliczana na podstawie argumentów przekazywanych przez użytkownika.

stopConditionFulfilled – flaga określająca, czy został spełniony warunek stopu. Sprawdzana na końcu każdej iteracji.

groupMinimum – zmienna pomocnicza przechowująca informację o najmniejszej odległości punktu pośród wszystkich odległości pomiędzy punktem a centroidami.

groupStartIndex – zmienna pomocnicza używana w trakcie obliczania indeksu elementu wskazywanego przez iterator początku grupy.

4.2.3 Metody

void DisplayCollection(Iterator first, Iterator last) – metoda nie integrujaca się z głównym algorytmem. Jej zadaniem jest wyświetlenie wszystkich elementów kolejkcji począwszy od elmentu wskazywanego przez iterator `first`, a kończąc na elemencie wskazywanym przez iterator `last`. Typ danych musi posiadać przeciążony operator «.

- `Iterator first` – iterator wskazujący na element typu podanego przez użytkownika będącym początkiem zakresu.
- `Iterator last` – iterator wskazujący na element typu podanego przez użytkownika będącym końcem zakresu.

returnIterator* Group(Iterator first, Iterator last, DistancePredicate &distanceMeasure, AveragePredicate &groupAverage, int maxIteration, int k, StopConditions stopCondition, bool printOutput = false) – dyspozytor iteratora swobodnego dostępu. Stanowi zabezpiecznie przed użyciem funkcji dla innych typów iteratorów. Parametry wywołania są tożsame z funkcją właściwą (za wyjątkiem znacznika iteratora) i opisane są w punkcie 3.2.

returnIterator* Group(Iterator first, Iterator last, std::random_access_iterator_tag, DistancePredicate &distanceMeasure, AveragePredicate &groupAverage, int maxIteration, int k, StopConditions stopCondition, bool printOutput) – główna funkcja implementująca algorytm grupowania metodą k-średnich. Znaczenie parametrów wywołujących opisane jest w punkcie 3.2. `std::random_access_iterator_tag` jest parametrem dodawanym sztucznie przez funkcję dyspozytora – ponieważ publicznie dostępnie jest jedynie dyspozytor, uniemożliwia to podanie argumentów iteratora nie będącymi iteratorami swobodnego dostępu. Jest to sposób wykorzystywany w standardowej bibliotece wzorców przy funkcji `std::distance()`. W ciele funkcji wywoływana jest funkcja zgodnie z założeniami algorytmu (1.2).

void Initialize() – funkcja wywoływana na samym początku funkcji `Group`. Zajmuje się sprawdzeniem poprawności argumentów (ilość grup, zakres elementów), zaalokowaniem pamięci na struktury pomocnicze (`Centroids`, `returnValues`, `currentGroupId`, `nextGroupId` oraz `distancesMatrix`) w zależności od ilości skupień oraz elementów do pogrupowania. Ustawia również pola na odpowiednie wartości w celu przygotowania do nowego wykonania algorytmu (ustawienie `iterationCounter` oraz `groupStartIndex` na 0, `stopConditionFulfilled` na `false` oraz `groupMinimum` na `DBL_MAX` odpowiadającej maksymalnej wartości zmiennoprzecinkowej).

void AssignStartingPoints(Iterator first) – funkcja wywoływana w funkcji `Group` zaraz po funkcji `Initialize`. Spośród wszystkich elementów wejściowych wybiera `groupNumber` początkowych wartości centroidów. Jako argument funkcja `Group` przekazuje do niej iterator wskazujący na pierwszy element.

void DisplayCurrentIterationState() – metoda wyświetlająca informacje o aktualnej przynależności danych do grup oraz współrzędnych centroidów. Jest ona wywoływana przy każdej iteracji algorytmu, o ile parametr `printOutput` ustawiony został na wartość `true`.

void Finish() – metoda wywoływana pod koniec metody `Group` dealokująca pamięć struktur tymczasowych inicjalizowanych w metodzie `Initialize` – jedynym wyjątkiem jest struktura `returnValues`, która jest zwracana poza klasę.

5. Testowanie aplikacji

W celu sprawdzenia poprawności działania aplikacji wykonano 2 rodzaje testów:

- Testy funcjonalne
- Wykorzystanie algorytmu do grupowania pikseli na bitmapach

Pierwszy z nich został wykonany w celu sprawdzenia poprawności zaimplementowanego rozwiązania. Drugi rodzaj testów posłużył do zaprezentowania przykładowych wyników, jakie można uzyskać posługując się algorytmem k-średnich.

5.1 Testy funkcjonalne

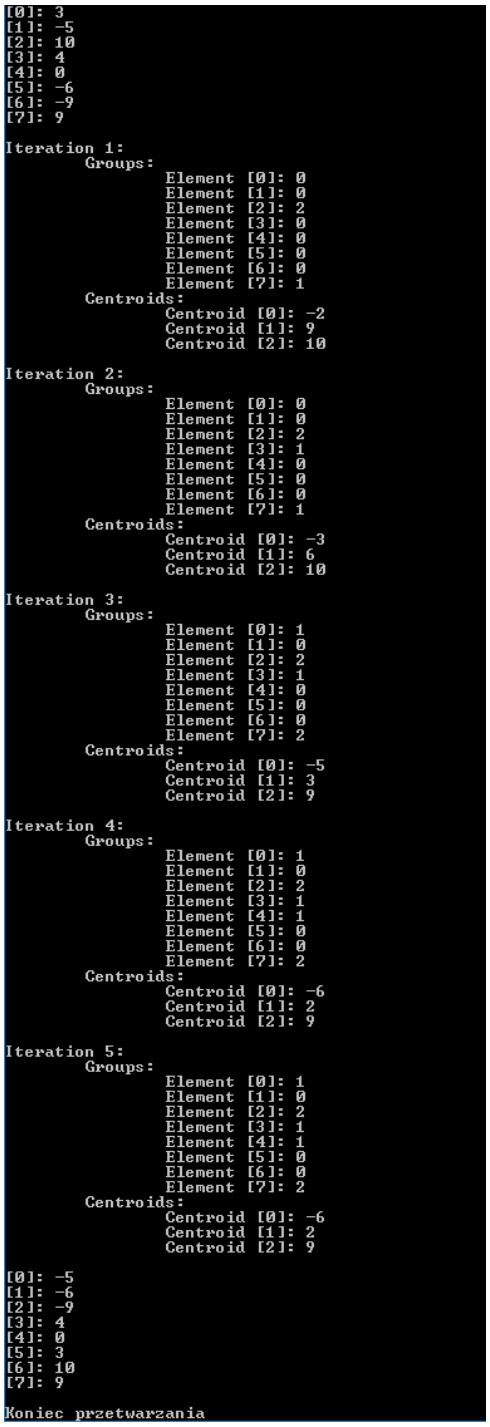
Testy funkcjonalne wykoanano na dwóch typach danych: reprezentującym liczby całkowite i wbudowany w język C++ typ `int` oraz własnej klasie `Point_2D` reprezentującej punkt w przestrzeni euklidesowej. Na czas testów flaga `printOutput` ustawiona została na wartość `true`.

Na rysunku 2 zaprezentowano i omówiono kolejne kroki wywołania dla typu całkowitego. Wykonano podział na 3 grupy, a warunkiem stopu był stan stabilny.

1. Przed pierwszą iteracją w sposób losowy wybrano wartości centroidów pośród elementów początkowych. Wybór padł na wartości 0, 9 oraz 10.
2. W pierwszej iteracji wpierw przypisano dane do grup na podstawie odległości od centroidów. Miarą odległości była wartość dzieląca obie liczby na osi X. Liczby 0, 9 oraz 10 jako środki grup zostały automatycznie przypisane do grup kolejno 0, 1 oraz 2. Pozostałe elementy najbliżej położone były wartości 0, więc przypisano je do grupy numer 0. Następnie wyliczono nowe wartości centroidów – grupy 1 oraz 2 zawierały tylko po jednym elemencie, więc ich wartość nie uległa zmianie. Nowa wartość centroidu grupy zerowej obliczona została za pomocą średniej arytmetycznej z 6 należących do niej elementów i jest równa -2.
3. W iteracji drugiej iteracji można zaobserwować, że element o indeksie 3 (i wartości 4) jest bliżej położony środka grupy pierwszej (odległość wynosi 5) niż zerowej (odległość równa 6), więc w tej iteracji zmienia skupienie na numer 1. W ten sposób zmianie ulegają skupienia 0 i 1, dlatego następuje przesunięcie współrzędnych ich środków (na kolejno -3 i 6).
4. W iteracji trzeciej można zaobserwować migrację elementu o indeksie 0 do grupy pierwszej oraz elementu o indeksie 7 do grupy 2. Ponieważ wartości centroidów są tego samego typu co elementy wejściowe, ich wartość po ponownym obliczeniu jest zaokrąglana w dół.
5. Iteracja czwarta – element o indeksie 4 (wartość 0) zmienia grupę na pierwszą, gdyż bliżej mu do wartości 3 niż -5. Na nowo obliczane są środki skupień grup 0 i 1.
6. W iteracji piątej nie ma przejść elementów między grupami – osiągnięto stan stabilny, dalsze iteracje nie są potrzebne.
7. Na końcu następuje grupowanie oraz wyznaczenie elementów zwracanych – na początek zakresu przesuwane są elementy należące do grupy zerowej: -5, -6 i -9. W następnej kolejności znajdują się dane należące do pierwszego skupienia, czyli 4, 0 i 3. Na końcu znajdują się elementy grupy drugiej – 10 oraz 9. Wektor zwracany zawiera iteratory wskazujące na na indeksy 0, 3 oraz 6 (początki kolejnych grup).

5.2 Grupowanie pikseli

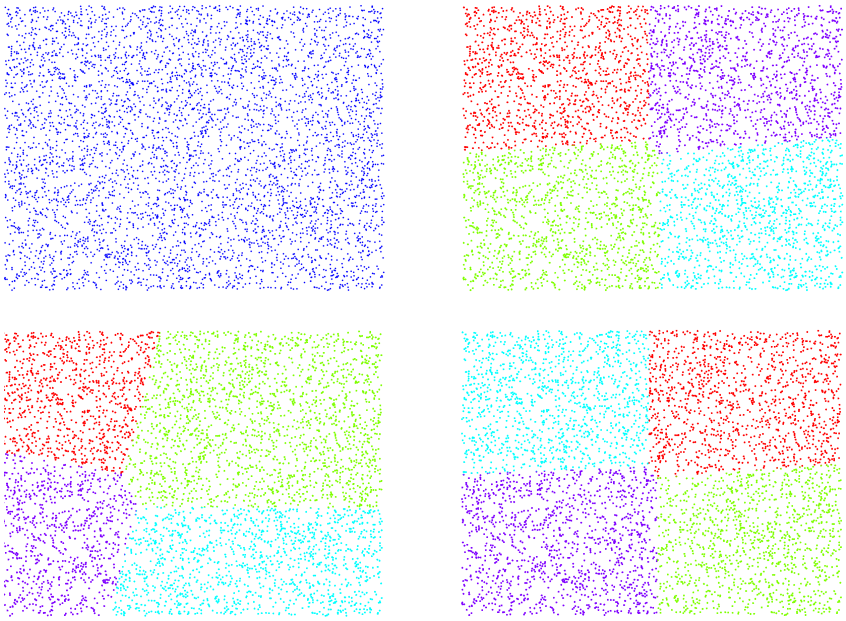
Testy praktyczne wykonane zostały na bitmapach przygotowanych własnoręcznie lub pobranych z Internetu. Testy te pozwalają prześledzić zachowanie algorytmu. W tym celu skorzystanego z biblioteki zewnętrznej dostępnej pod adresem `http://partow.net/programming/bitmap/index.html`. W trakcie wczytywania danych (współrzędnych pikseli) pomijano piksele białe (o wartości RGB równej FFFFFF).



Rysunek 2: Przykładowe wywołanie algorytmu dla zmiennych typu całkowitego

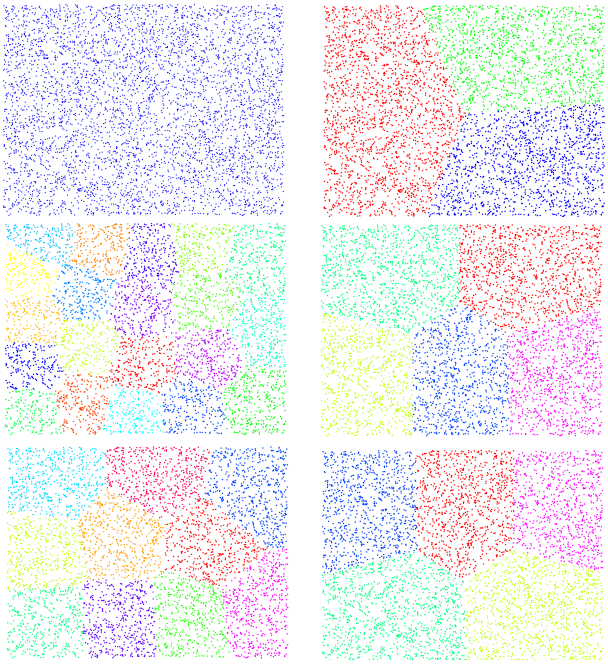
5.2.1 Zależność od warunku stopu

Na rysunku nr 3 przedstawiono wyniki wykonania algorytmu dla różnych warunków stopu. Liczbę grup ustawiono na 4. Można zaobserwować fakt, że dla małej liczby iteracji algorytm nie zdążył jeszcze ustabilizować wyników i dla w miarę równomiernego rozkładu danych na płaszczyźnie grupy są bardzo nierówne. Dla większej liczby iteracji (10) wynik jest zbliżony do tego otrzymanego poprzez wybranie stanu stabilnego.



Rysunek 3: Od górnego lewego rogu zgodnie ze wskazówkami zegara: obraz oryginalny, stan stabilny, liczba iteracji = 10, liczba iteracji = 2

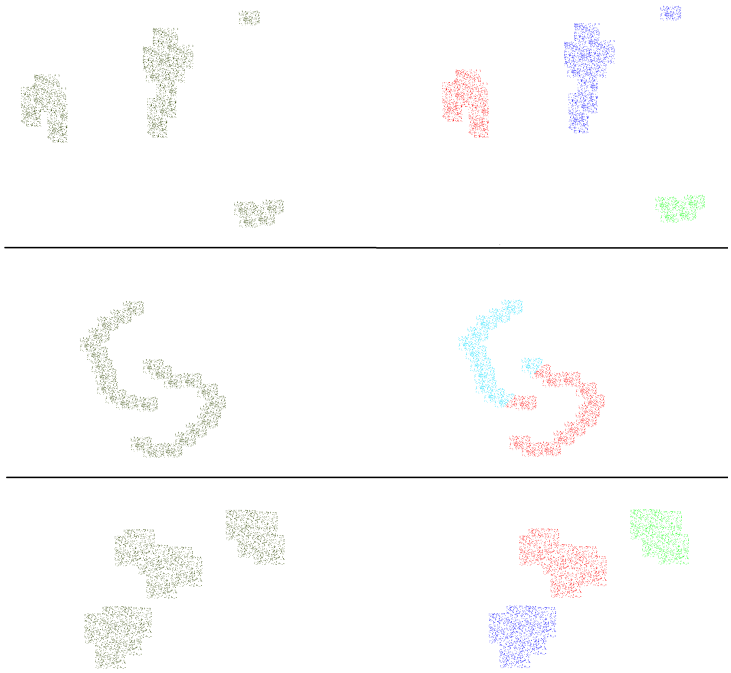
5.2.2 Zależność od liczby grup



Rysunek 4: Od górnego lewego rogu zgodnie ze wskazówkami zegara: obraz oryginalny, k = 3, k = 5, k = 5, k = 10, k = 20

Na rysunku nr 4 przedstawiono wyniki wykonania algorytmu dla różnych liczby grup. Jako warunek stopu ustawiono stan stabilny. Algorytm zachowuje się poprawnie każdej liczby grup, jednak należy zwrócić uwagę, że dla identycznych parametrów wykonania (w tym przypadku k = 5) algorytm może zwrócić różne wyniki – jest to efekt losowego doboru centroidów.

5.2.3 Zależność od danych wejściowych



Rysunek 5: Po lewej: obrazy przed grupowaniem, po prawej po grupowaniu

Na rysunku nr 5 przedstawiono wyniki wykonania algorytmu dla różnych kształtów bitmap. Jako warunek stopu ustawiono stan stabilny. Dla przykładów pierwszego i trzeciego algorytm sprawił się bardzo dobrze, grupując według wyraźnie oddzielonych grup (dla pierwszegho przykładu jeszcze lepsze wyniki możnaby otrzymać zwiększając liczbę grup do 4). Dla przykładu drugiego rozwiązanie nie jest do końca optymalne – w tym przypadku lepsze mogłoby się okazać wykorzystanie innego algorytmu grupowania.

6. Podsumowanie

W trakcie realizacji tego projektu, nauczyłem się bardzo przydatne może być wykorzystanie iteratorów języka C++ jako argumentów. Do tej pory wykonując operacje na tablicach/kontenerach lub ich fragmentach posługiwałem się najczęściej wskaźnikami. Iteratory posiadają kilka zalet stawiających ich nad wskaźniki:

- Metody udostępniane przez standardową bibliotekę wzorców ułatwiającą operacje na iteratorach (np. `std::sort`, `std::distance` czy `std::swap`).
- Zwiększona elastyczność pozwalająca na iterację po elementach nawet w bardziej skomplikowanych strukturach danych (takich jak graf czy drzewo).
- Zwiększone bezpieczeństwo typów.
- Ułatwione tworzenie generycznego kodu.

Zachowują się przy tym jak wskaźniki (posiadając między innymi operatory dereferencji oraz inkrementacji). Jedyną niedogodnością jaką napotkałem, to brak zależności dziedziczenia pomiędzy poszczególnymi typami iteratorów. Przykładowo: każdy iterator swobodnego dostępu jest iteratorem dwukierunkowym, jednak nie łączy ich relacja dziedziczenia. Rozpoznanie typu iteratora musi odbywać się przez dodatkową strukturę `std::iterator_traits` oraz struktury typu `_iterator_tag`.

Metoda centroidów jest ciekawym algorytmem grupowania używanym na przykład do kwantyzacji wektorów. Pozwala ona na utworzenie grup w dość krótkim czasie (złożoność czasowa jest z reguły ograniczona przez liczbę iteracji, natomiast pamięciowa to iloczyn liczby grup i elementów do pogrupowania). Najlepiej radzi sobie zdecydowanie w przypadkach, kiedy dane wejściowe są bardziej skupione (w określonych miejscach natężenie danych jest większe). Nie zawsze otrzymywane dane są optymalne (jak w drugim przykładzie na rysunku 5) – dla takich danych lepszym rozwiązaniem moze okazać się np. metoda hierarchiczna. Na względzie trzebe mieć również, że osiągnięcie stanu stabilnego może trwać bardzo długo, ze względu na minimalne przeskoki danych między grupami w trakcie nastepujących po sobie iteracji, co wymusza kolejne. Ograniczenie maksymalną liczbą iteracji może dać wyniki niemal identyczne co stan stabilny w dużo krótszym czasie.