Wykrywanie loga banku PKO BP

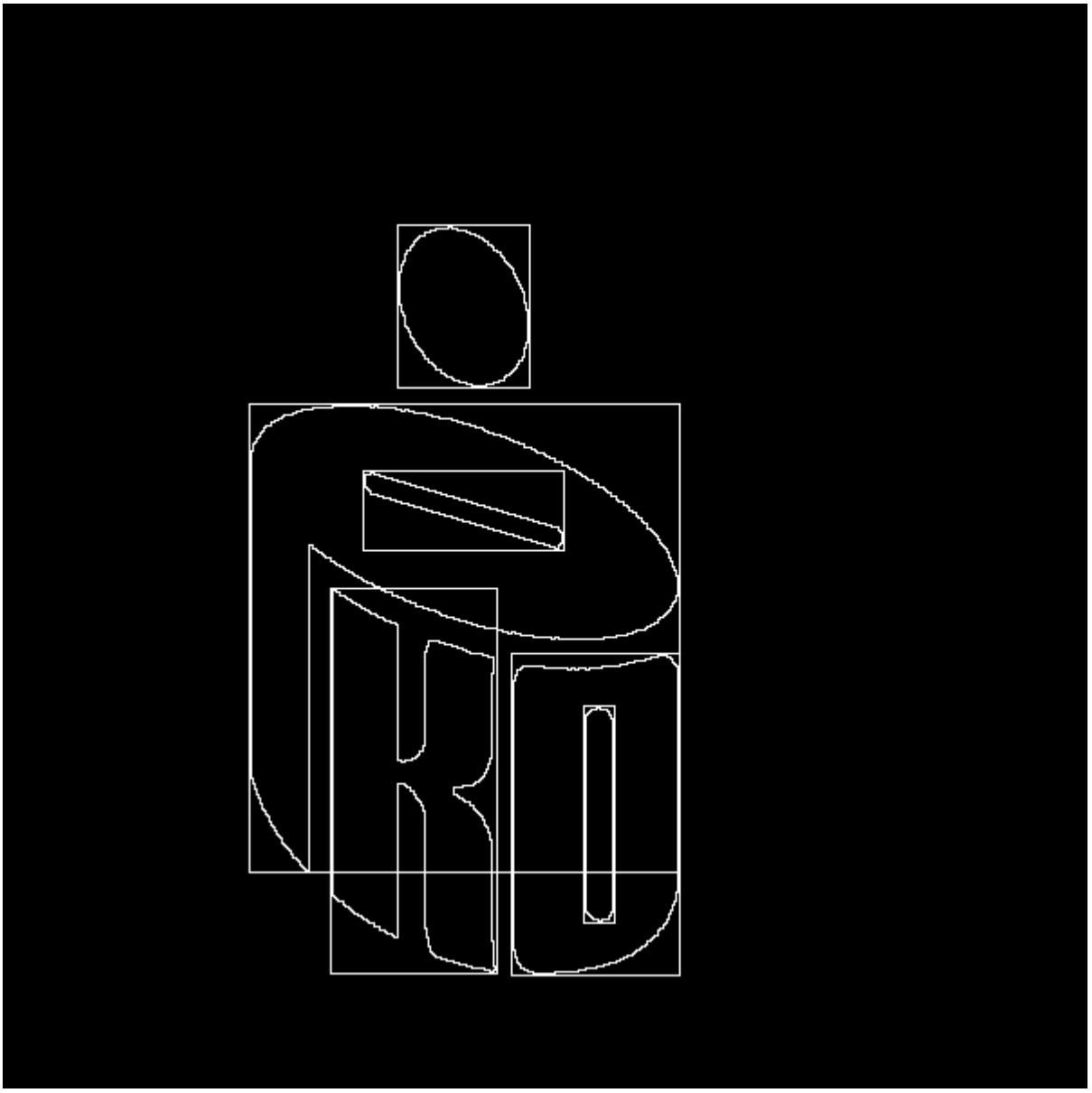
Autor: Jakub Szczepanowski

1. Plan rozwiązania

- konwersja modelu RGB do HSV,
- na podstawie określonych przedziałów wartości modelu dokonanie segmentacji przez progowanie,
- wyodrębnienie odrębnych obszarów do dalszej analizy,
- stwierdzenie przynależności obszarów do klasy loga na podstawie cech,
- klasteryzacja centrów geometrycznych w tym złączenie wykrytych obszarów w jeden reprezentujący logo.

2. Analiza problemu

Analizy loga dokonano na podstawie obrazu referencyjnego nie uwiecznionego na zdjęciu, tak, aby nie uzależniać reprezentacji od warunków środowiskowych. Polegała ona na początku na analizie kolorystycznej HSV w celu wyodrębnienia obszarów. Następnie na dokonaniu transformacji afinicznych loga oraz obliczania cech (niezmienników momentowych) dla poszczególnych elementów loga. Dokonano również transformacji rotacji zwiększając odporność cech elementu P na to przekształcenie zawierającego w swoich granicach inne elementy (a przynajmniej ich części), lecz zakres wartości dla większości przypadków powodował większą liczbę nieprawidłowo wykrytych przypadków z czego ostatecznie zrezygnowano. Potencjalnym rozwiązaniem tego problemu byłoby analizowanie elementów z wyczyszczeniem tła, a nie bezpośrednio na obrazie posegmentowanym. Wartości cech zawarto jako średnia arytmetyczna oraz odchylenie standardowe z otrzymanej próby. Początkowo liczba cech wynosiła 2 i była to cecha M1 i M7 ze względu na ich inwariantność. Okazały się one niewystarczające przez zaliczanie obszarów ewidentnie nie spełniających kryteriów. Zatem do tego zestawu dołączyła jeszcze cecha M3 uwzględniająca inne aspekty geometryczne przez użycie do jej obliczenia momentów 3 rzędu.



3. Szczegóły techniczne

Zadanie projektowe wykonano w całości w języku C++. Rozwiązanie podzielono na kilka plików kodu źródłowego wraz z plikami nagłówkowymi:

- Projekt.cpp - główny plik programu,
- converter.cpp - plik zawierający funkcję do konwersji modelu kolorystycznego,
- processing.cpp - zawiera funkcje do: analizy obrazów (funkcja pomocnicza), segmentacji, erozji, dylacji, otwarcia, zamknięcia, znajdowania pojedynczego obiektu, znajdowania centrum masy, znajdowania konturów, określania czy piksel jest konturowy, znajdowania zamkniętych obszarów,
- features.cpp/hpp - zawiera cechy referencyjne oraz funkcje do obliczania pola, obwodu, cech, średniej, odchylenia standardowego, detekcji loga, znajdowania centrum geometrycznego, odległości euklidesowej, algorytm DBSCAN i k najbliższych sąsiadów,
- utils.cpp/hpp - zawiera struktury HSV, BoundingBox, Point, klasę Image.

4. Opis poszczególnych etapów rozwiązania

a) konwersja do HSV: był to najprostszy etap całego rozwiązania ze względu na dostępność prostych do implementacji wzorów na konwersję modelu. Konwersja ta nie jest jednak taka sama w każdym przypadku, a przynajmniej ostateczne przedziały jej wartości różnią się między sobą. HSV jest modelem odwróconego stożka, więc intuicyjnym jest określenie odcieni koloru w zakresie od 0 do 360 stopni, w wielu jednak implementacjach można się spotkać z przedziałem np. do 255, w przypadku nasycenia czy też wartości z przedziałem od 0 do 1, a nie od 0 do 100. Ograniczyłem się jednak do zastosowania podejścia intuicyjnego. Do przechowywania między innymi zakresu odcienia wykraczającego poza zakres jednego bajta utworzono oddzielną strukturę, a do przechowywania całego obrazu klasę zawierającą wskaźnik do tablicy dwuwymiarowej matrycy danych.

```
for (i = 0; i < rows; i++)
     for (j = 0; j < cols; j++) {
          int blue = _image(i, j)[0];
          int green = _image(i, j)[1];
          int red = _image(i, j)[2];
          HSV hsv;
          int hue, saturation, value;

          temp = std::min(std::min(blue, green), red);
          value = std::max(std::max(blue, green), red);

          if (temp == value)
               hue = 0;
          else {
               if (red == value)
                    hue = ((green - blue) * 60 / (value -
temp));
               if (green == value)
                    hue = 120 + ((green - blue) * 60 / (value -
temp));
               if (blue == value)
                    hue = 240 + ((green - blue) * 60 / (value -
temp));
          }
          if (hue < 0)
               hue = hue + 360;

          if (value == 0)
               saturation = 0;
```

```
                else
                        saturation = (value - temp) * 100 / value;

            value = (100 * value) / 255;

            hsv.hue = hue;
            hsv.saturation = saturation;
            hsv.value = value;
            hsv_image[i][j] = hsv;
        }
```

b) segmentacja: z użyciem warunku segmentacji, którym okazały się wartości (składowa value) większe lub równe 85 liczby zaliczone do tego przedziału określano jako segment, poniżej jako tło. Wartość ta jest jednak wyśrodkowania, dla zdjęć lepiej oświetlonych wartość ta była równa np. 90 co dawało jeszcze lepsze wyniki.

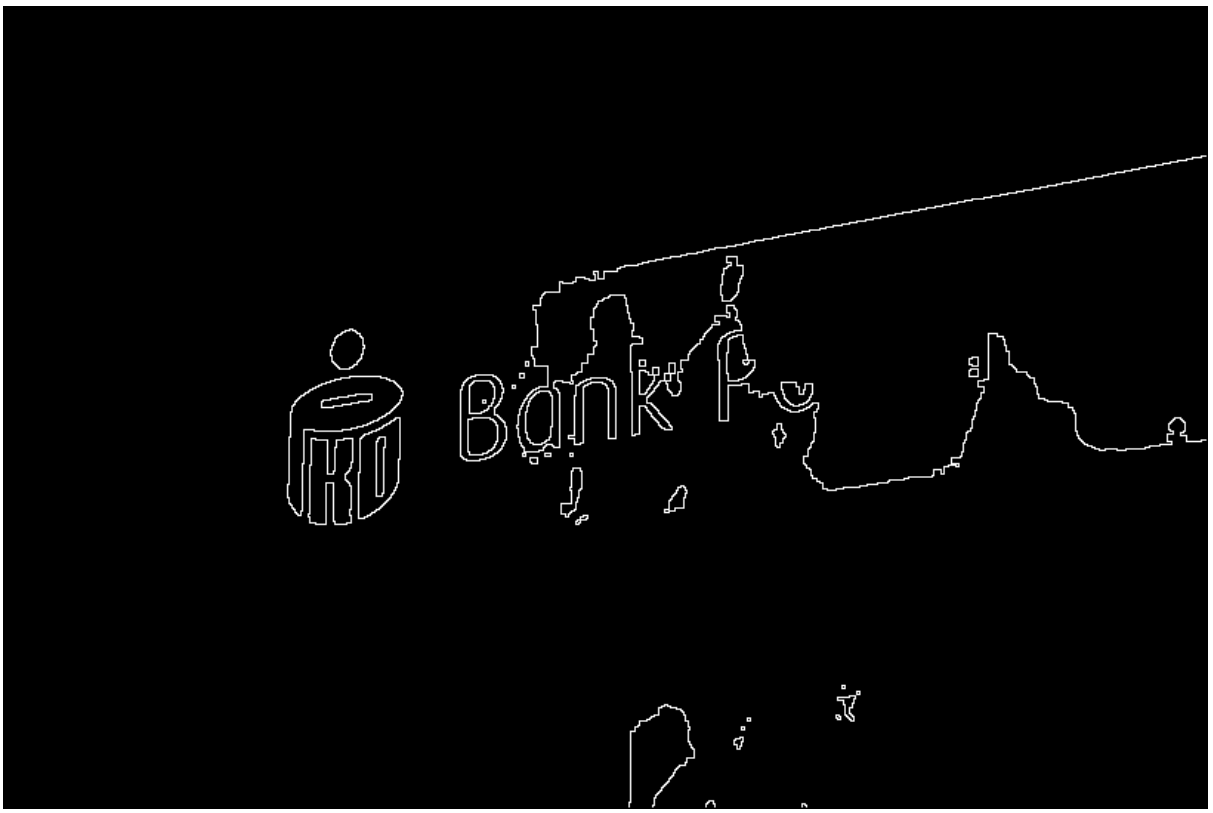
```
for (int i = 0; i < rows; i++)
      for (int j = 0; j < cols; j++) {

      if (predicate(hsv[i][j]))
            bin_image.at<uchar>(i, j) = 255;
      else
            bin_image.at<uchar>(i, j) = 0;
}
```

c) wyodrębnienie oddzielnych obszarów: był to zdecydowanie najtrudniejszy etap rozwiązania. Podstawową koncepcją do stworzenia przeze mnie algorytmu do znajdowania zamkniętych obszarów było wykorzystanie faktu, że w dowolnym punkcie krawędziowym segmentu zamkniętego przechodząc iteracyjne po kolejnych punktach wracamy do punktu startowego w skrócie, że tego typu obszary zawierają sprzężenie zwrotne. Pierwszym etapem ku jego stworzeniu było więc wykrywanie krawędzi, którego dokonano przy użyciu operatora logicznego zawierającego gradient w każdym kierunku rozważanego punktu.

```
for (int i = 1; i < I.rows - 1; i++)
      for (int j = 1; j < I.cols - 1; j++) {
            if (_I(i, j) == 255 && (_I(i, j) != _I(i - 1, j - 1) ||
                  _I(i, j) != _I(i - 1, j) ||
                  _I(i, j) != _I(i - 1, j + 1) ||
                  _I(i, j) != _I(i, j + 1) ||
```

```
                _I(i, j) != _I(i - 1, j) ||
                _I(i, j) != _I(i + 1, j + 1) ||
                _I(i, j) != _I(i + 1, j) ||
                _I(i, j) != _I(i + 1, j - 1)))
                contours.push_back(Point(j, i));
    }
```

Następnie, biorąc pierwszy dostępny w kolekcji punkt konturu przechodzono do kolejnych punktów z użyciem pola 3x3. Pierwsze w kolejności do wybrania były punkty najbliższe czyli położone na krzyż, jeżeli nie znaleziono takowego wybierano punkt diagonalny. Jeżeli nie uda się znaleźć następnego punktu ścieżka jest uznawana za otwartą. Jeżeli natomiast w n'tym kroku algorytm dojdzie do punktu startowego jest to obszar zamknięty. Bounding box okalający ten kontur jest uznawany jako kandydat do dalszej analizy. Rozważane już punkty są dodawane do zbioru, tak aby nie nastąpiło ich ponowne wybranie, a kontury są usuwane z kolekcji. Stworzony przeze mnie algorytm ma jednak mankament - w pewnych przypadkach zapętla się z niezrozumiałych przeze mnie przyczyn. Jest to z pewnością w jakiś sposób związane z łączeniem się przyległych do siebie obszarów. Wymaga więc on separowalności obszarów. Problem ten rozwiązuje jednak użycie operacji morfologicznych otwarcia oraz zamknięcia. Otwarcie wygładza krawędzie i rozdziela obszary, a zamknięcie znacznie upraszcza reprezentację łącząc wiele rozdrobnionych obszarów w jeden.

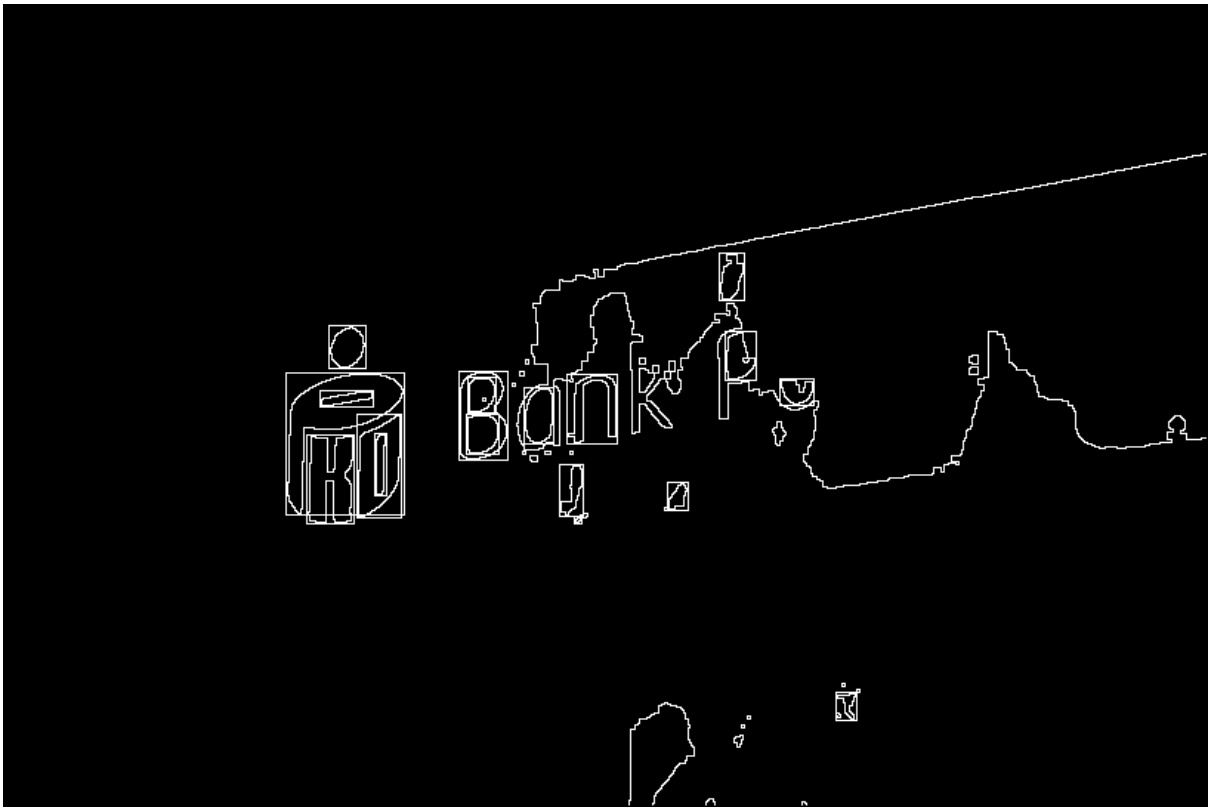


```
     while (searchForStartingPoint) {
           diagonal_point = Point();
           break_loop = false;

           for (char v = -1; v <= 1; v++) {
                for (char h = -1; h <= 1; h++) {

                     if (v != 0 || h != 0) {
                          int i = current_point.y + v;
                          int j = current_point.x + h;

                          if (j < 1 || j >= I.cols - 1 || i < 1
|| i >= I.rows - 1) {
                             prev_points.insert(current_point);
                               prev_points.insert(Point(j, i));
                               contours.remove(Point(j, i));
                               contours.remove(current_point);
                               break;
                          }

                          if (stringLength > 50 &&
(starting_point.x >= current_point.x - 1 &&
```

```
starting_point.x <= current_point.x + 1 &&
starting_point.y >= current_point.y - 1 &&
starting_point.y <= current_point.y + 1)) {

                                searchForStartingPoint = false;
                                break_loop = true;
                                break;
                           }

                           if (i != current_point.y &&
j != current_point.x && prev_points.find(Point(j, i)) ==
prev_points.end() && isContour(I, i, j)) {
                                diagonal_point = Point(j, i);
                           }
                           else if (prev_points.find(Point(j, i))
== prev_points.end() && isContour(I, i, j)) {

                               prev_points.insert(current_point);
                                current_point = Point(j, i);
                                break_loop = true;
                                box.x0 = std::min(box.x0, j);
                                box.y0 = std::min(box.y0, i);
                                box.x1 = std::max(box.x1, j);
                                box.y1 = std::max(box.y1, i);
                                contours.remove(current_point);
                                stringLength++;
                                break;
                           }
                      }
                 }
                 if (break_loop)
                      break;
            }
            if (!break_loop && diagonal_point != Point()) {
                 current_point = diagonal_point;
                 prev_points.insert(starting_point);
                 box.x0 = std::min(box.x0, diagonal_point.x);
                 box.y0 = std::min(box.y0, diagonal_point.y);
                 box.x1 = std::max(box.x1, diagonal_point.x);
                 box.y1 = std::max(box.y1, diagonal_point.y);
                 contours.remove(current_point);
                 stringLength++;
```

```
            }
            else if (!break_loop) {
                  if (stringLength > 50 && starting_point.x >=
current_point.x - 2 && starting_point.x <= current_point.x + 2 &&
starting_point.y >= current_point.y - 2 && starting_point.y <=
current_point.y + 2)
                        searchForStartingPoint = false;
                  break;
            }
      }

      prev_points.insert(starting_point);
      contours.remove(starting_point);
```

Przy znalezieniu kolejnego punktu bounding box jest aktualizowany. Do tego dodano ograniczenie określające minimalną długość konturu, między innymi po to, aby nie rozważać zbyt małych obszarów.



d) klasyfikacja obszarów do klas elementów logo: przynależności do klas w przestrzeni cech dokonywano dzięki różnicy pomiędzy otrzymaną wartością a ustaloną średnią tak, aby była mniejsza lub równa wielokrotności odchylenia standardowego w zależności od potrzeb. W przypadku detekcji całego loga jako jednego elementu próbowano dokonać również porównania z cechami reprezentującymi całe logo z tym, że wartości te były bardzo zbliżone do cech elementu P przez co często P było

uznawane jako całe logo. Problem ten ponownie nie istniałby (prawdopodobnie) gdyby poszczególne elementy były wycinane bez otoczenia.

```
std::vector<BoundingBox> logo_elems;

      for (BoundingBox box : close_areas) {

            cv::Mat clip = bin_image(cv::Rect(cv::Point(box.x0,
box.y0), cv::Point(box.x1, box.y1)));
            double m1 = M1(clip);
            double m7 = M7(clip);
            double m3 = M3(clip);

            if (((abs(m1 - DOT_M1) <= DOT_M1_STD * 6 && abs(m7 -
DOT_M7) <= DOT_M7_STD * 6 && abs(m3 - DOT_M3) <= DOT_M3_STD * 2)
||
                  (abs(m1 - P_M1) <= P_M1_STD * 2 && abs(m7 - P_M7)
<= P_M7_STD * 2 && abs(m3 - P_M3) <= P_M3_STD * 2) ||
                  (abs(m1 - K_M1) <= K_M1_STD * 2 && abs(m7 - K_M7)
<= K_M7_STD * 2 && abs(m3 - K_M3) <= K_M3_STD * 2) ||
                  (abs(m1 - O_M1) <= O_M1_STD * 2 && abs(m7 - O_M7)
<= O_M7_STD * 2 && abs(m3 - O_M3) <= O_M3_STD * 2)))
                        logo_elems.push_back(box);
      }
```

e) klasteryzacja: analizy skupień mającej na celu wyłonienie instancji loga z segmentów zaliczonych jako elementy loga dokonano poprzez implementację algorytmu DBSCAN oraz centrów geometrycznych bounding boxów. Punkty zaliczone jako szum były odrzucane w finalnych wynikach, te zaś którym przypisana została etykieta łączyły się w jeden box okalający całe logo. Rozwiązanie to zostało wzbogacone o dynamiczne ustalanie odległości epsilonowej z wykorzystaniem średniej odległości euklidesowej k-tego najbliższego sąsiada każdego z punktów. Wykorzystując informację o tym, że logo zawiera 4 elementy k ustalono na 3, czyli na odległość od najdalszego elementu logo. W ten sposób nie było mowy o złym dobraniu epsilona, który przy zmianie skali nie obejmował wszystkich elementów loga. Dodano również maksymalną liczbę elementów w klastrze, która broniła przed dobieraniem sobie do loga obszarów niepoprawnie sklasyfikowanych jako część loga. Są jednak przypadki, gdzie wykrywane są np. 3 elementy loga. Należałoby wtedy zmodyfikować ten hiperparametr wykorzystując informację o klasach elementów (wiedza o tym, że nie należy dodawać drugiego elementu o klasie np. K skoro najbliższy element o tej klasie został już dodany do klastra). Pewne więc aspekty rozwiązania można by poprawić, mimo to daje on zadowalające rezultaty działania.

```
	std::vector<double> knn_distances = calculate_knn(centers,
3);

	double eps = average(knn_distances);

	if (eps < I.cols / 2 && eps < I.rows / 2) {
		std::vector<int> labels = DBSCAN(centers, eps, 2, 4);

		std::map<int, BoundingBox> logos;

		for (size_t i = 0; i < labels.size(); ++i) {
			if (labels[i] != -1) {

				BoundingBox logo_box(I.cols, I.rows);

				if (logos.find(labels[i]) != logos.end()) {
					logo_box = logos[labels[i]];
				}

				logo_box.x0 = std::min(logo_elems[i].x0,
logo_box.x0);
				logo_box.y0 = std::min(logo_elems[i].y0,
logo_box.y0);
				logo_box.x1 = std::max(logo_elems[i].x1,
logo_box.x1);
				logo_box.y1 = std::max(logo_elems[i].y1,
logo_box.y1);

				logos[labels[i]] = logo_box;
			}
		}

		for (auto& elem : logos) {
			final_logos.push_back(elem.second);
		}
	}
	else {
		final_logos = logo_elems;
	}
```

4. Podsumowanie

Cel jakim jest wykrywanie loga banku PKO BP udało się zrealizować. W powyższym sprawozdaniu wskazano jednak kilka mankamentów czy też koncepcji rozwoju rozwiązania. Przedstawiono w nim także przykład działania programu, który przetestowano na kilku zdjęciach prezentujących logo w różnych akwizycjach oraz w ilości równej dwa loga na jedno zdjęcie. Te przykładowe różne zdjęcia będą zaprezentowane na obronie.