Analiza i Przetwarzanie Obrazów

Sprawozdanie II

Temat: Operacje morfologiczne i nie tylko na zdjęciach

Autor:
Magda Nowak-Trzos

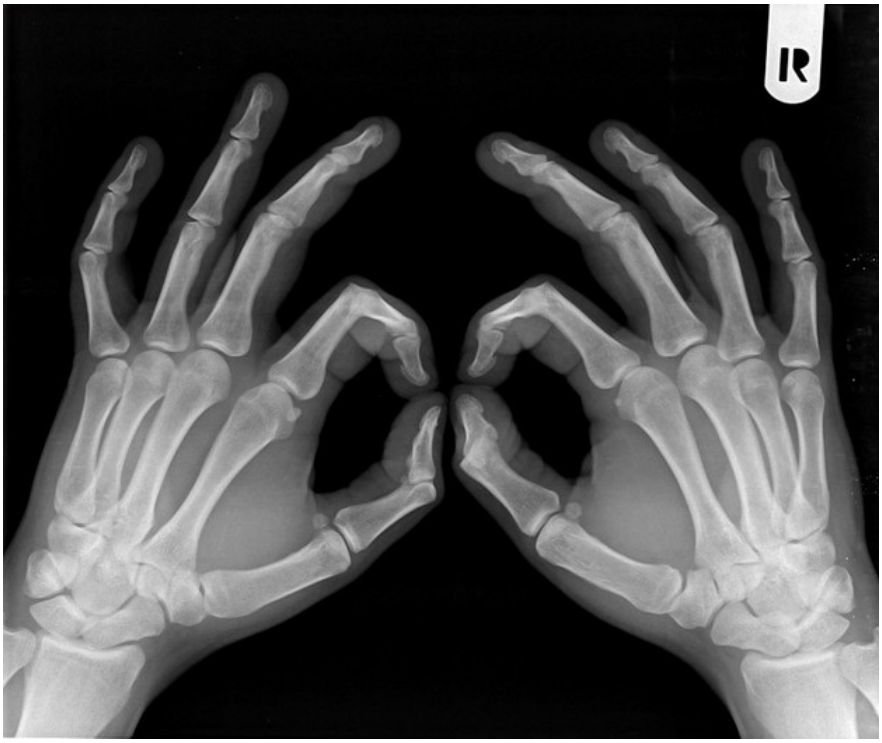
Prowadzący ćwiczenia:
mgr Krzysztof Misztal

Kraków, 17 kwietnia 2014

Spis treści

1 Temat zadania

Celem projektu jest obróbka zdjęcia rentegowskiego dłoni, tak aby pokolorować jak najwięcej kości na różne kolory, celem późniejszego ich policzenia.



Obrazek wczytywany przez aplikację

2 Proponowane rozwiązanie

Rozwiązaniem projektu jest wykonanie kolejno następujących operacji na zdjęciu.
-Na początku wykonuję filtr Canny.
-Następnie na obrazie oryginalnym wykonuję kolejno inwersję, binaryzację otsu i erozję.
-Po wykonaniu operacji logcznej "and" na otrzymanym obrazie i na obrazie po filtrze Canny dostaję krawędzie kości, bez krawędzi skóry.
-Kolejnym etapem jest wykonanie operacji floodfill na obrazie. Przechodząc pętlą po obrazie wykonuję operację flood fill w momencie kiedy następuje zmiana piksela z białego na czarny. Dzięki temu otrzymuję pokolorowane odizolowane części na obrazie.
-Następnie zamieniam wszystkie białe częsci obrazu na czarne i dokonuję kolejnej operacji logicznej 'and' na zdjęciu oryginalnym i zdjęciu z pokolorowanymi kośćmi, co powoduje naniesienie kolorów na zdjęcie oryginalne. Dzięki czemu łatwo można policzyć ile jest kości na zdjęciu.

2.1 Krok 1: Filtr Canny

Operacja filtru Canny, ma na celu detekcję krawędzi na obrazie. W moim programie wykorzystuję własną implementację filtru Canny.
Wykonuję następujące kroki:

- Filtr medianowy do usunięcia szumów - Konwersja do odcieni szarości i rozmycie Gaussa - Wstępna detekcja krawędzi za pomocą filtru Sobela
- Znalezienie wartości i kierunków gradientów opierajac się na wartosciach otrzymanych z filtru Sobela.

- Skasowanie odpowiednich pikseli, niemaksymalnych dla swojego kierunku gradientu + tresholding
- Edge Tracking - przyporządkowanie szarych krawędzi odpowiednio do pikseli czarnych lub białych.

2.1.1 Filtr Sobela

Pierwszym krokiem algorytmu jest konwolucja z dwoma maskami 3x3:

$$G_1 = \begin{pmatrix} -1 & 0 & 0 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

oraz

$$G_2 = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

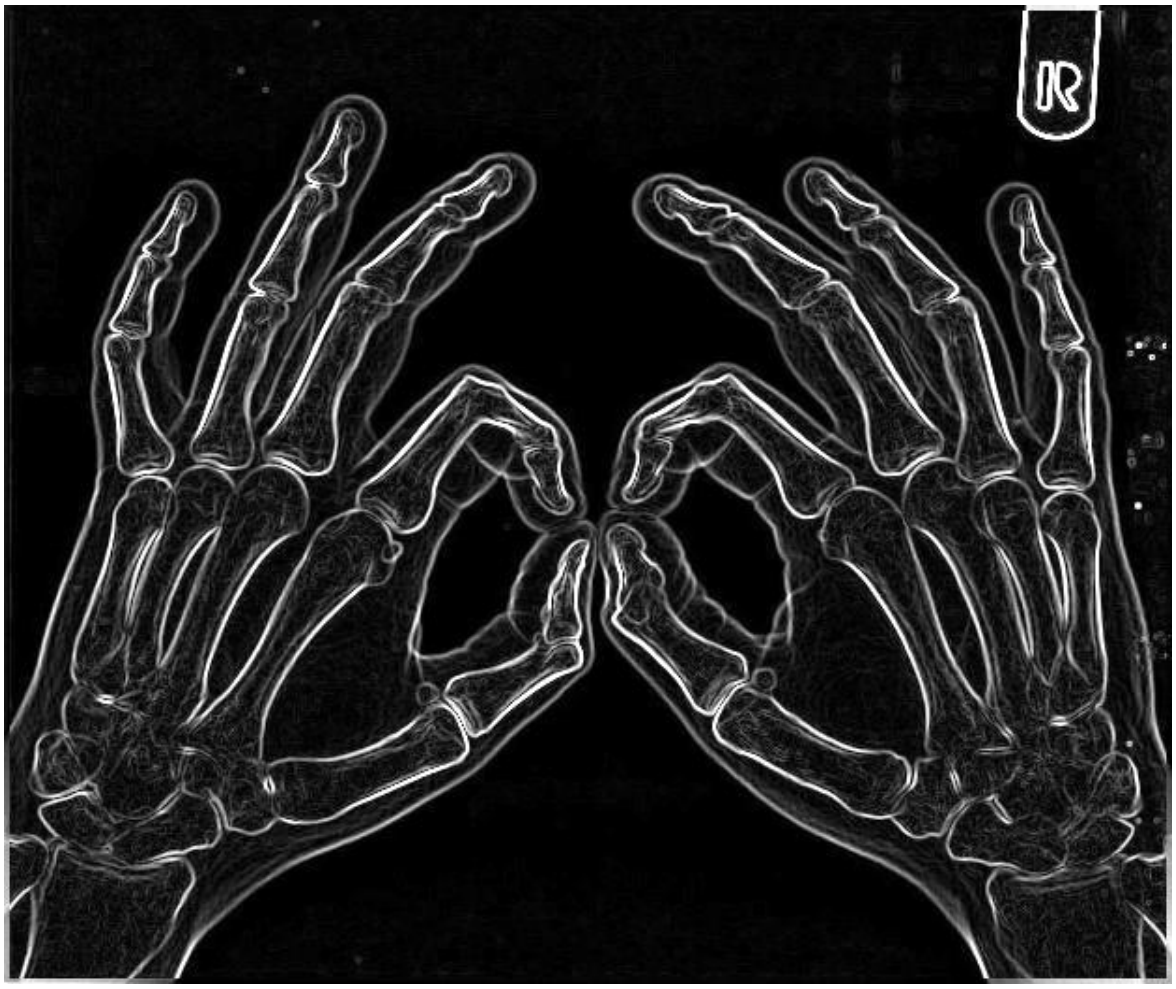
Następnie stosując poniższy wzór wykonujemy zamianę pola wektorowego na skalarne:

$$I(x,y) = \sqrt{G_1^2 + G_2^2} \tag{1}$$

Metoda realizująca filtr Sobela. Metoda ta przyjmuje tablice obiektów klasy Grad. Klasa ta zawiera dwa pola, jedno odpowiadające za wartość gradientu a drugie za jego kierunek. Metoda Sobel wpisuje do tablicy elementow Grad wartości i kierunki gradientow dla każdego piksela. Będzie to później wykorzystywane przy usuwaniu odpowiednich pikseli z obrazu.

```
public void Sobel(MarvinImage imageIn, MarvinImage imageOut, Grad[][] tablica)
    {
        MarvinImage image = imageIn.clone();
         for (int x = 3; x < imageIn.getWidth()-3; x++) {
             for (int y = 3; y < imageIn.getHeight()-3; y++) {

                 int p0=image.getIntComponent0(x-1, y-1);
                 int p1=image.getIntComponent0(x, y-1);
                 int p2=image.getIntComponent0(x+1, y-1);
                 int p3=image.getIntComponent0(x+1, y);
                 int p4=image.getIntComponent0(x+1, y+1);
                 int p5=image.getIntComponent0(x, y+1);
                 int p6=image.getIntComponent0(x-1, y+1);
                 int p7=image.getIntComponent0(x-1, y);
                 int xxg = ((p2+2*p3+p4)-(p0+2*p7+p6));
                 int yyg = ((p6+2*p5+p4)-(p0+2*p1+p2));
                 int g = (int) Math.hypot(xxg,yyg);

                 if(g > 255) g = 255;
                 else if(g<0) g = 0;
                 imageOut.setIntColor(x, y, g, g, g);

                  if(xxg==0)
                 {
                    tablica[x][y] = new Grad(Math.toRadians(90),Math.hypot(xxg,yyg));
                 }
                 else
                 {
                    tablica[x][y] = new Grad(Math.atan2(yyg,xxg),Math.hypot(xxg,yyg));
                 }
             }
         }
}
```

2.1.2 Rezultat



Obrazek po filtrze Sobela

2.1.3 Znalezienie wartości i kierunkow gradientow po filtrze Sobela

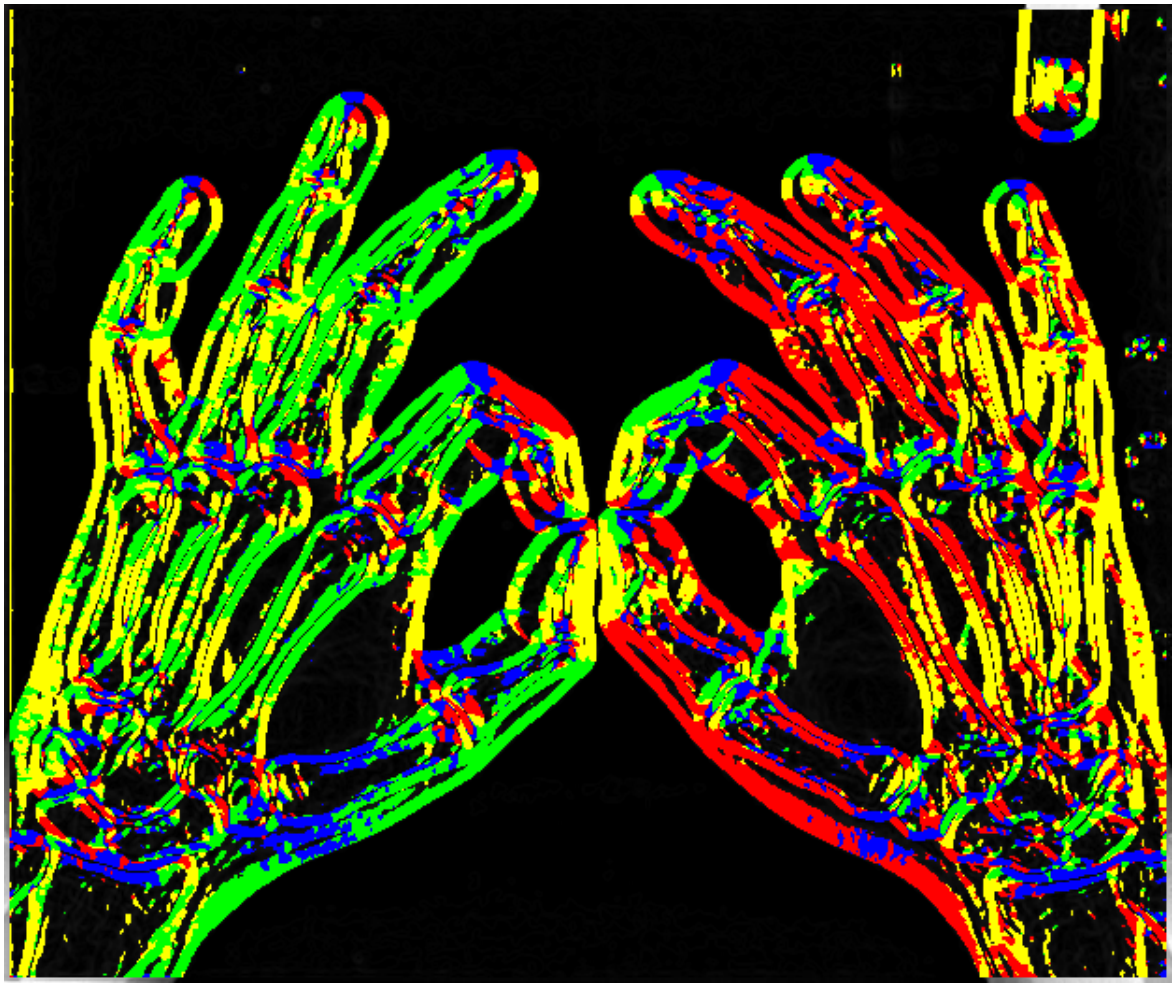
Na tym etapie utworzyłam metodę zaokrąglającą kierunki gradientów do określonych wartosci (0, 45, 90 i 135). Piksel o danym kierunku gradientu polorowałam na inny kolor odpowiednio na żółty zielony niebieski i czerwony.

Kod metody

```
1   void Gradienty(Grad[][] tablica, MarvinImage imageIn, MarvinImage imageOut)
2   {
3       for (int x = 3; x < imageIn.getWidth()-3; x++) {
4           for (int y = 3; y < imageIn.getHeight()-3; y++) {
5             double a = (Math.PI)/8;
6            if(tablica[x][y].wart>20){
7               if((tablica[x][y].kier>=-8*a && tablica[x][y].kier<-7*a ) // kolor żółty
8               ||(tablica[x][y].kier>=-a && tablica[x][y].kier<a ||
9               (tablica[x][y].kier>7*a && tablica[x][y].kier<=8*a )))
10              {
11                 imageOut.setIntColor(x,y,255, 255, 0);
12                 tablica[x][y].kier=0;
13              }
14              else if((tablica[x][y].kier>=-7*a && tablica[x][y].kier<-5*a )|| // kolor
                    zielony
15              (tablica[x][y].kier>=a && tablica[x][y].kier<3*a))
16              {
```

```
17                      imageOut.setIntColor(x,y,0, 255, 0);
18                      tablica[x][y].kier=45;
19                  }
20                  else if((tablica[x][y].kier>=-5*a && tablica[x][y].kier<-3*a )|| // kolor
                        niebieski
21                  (tablica[x][y].kier>=3*a && tablica[x][y].kier<5*a))
22                  {
23                      imageOut.setIntColor(x,y,0, 0, 255);
24                      tablica[x][y].kier=90;
25                  }
26                  else if((tablica[x][y].kier>=-3*a && tablica[x][y].kier<-a )|| // kolor
                        czerowny
27                  (tablica[x][y].kier>=5*a && tablica[x][y].kier<7*a))
28                  {
29                      imageOut.setIntColor(x,y,255, 0, 0);
30                      tablica[x][y].kier=135;
31                  }
32              }
33          }
34      }
35  }
```

2.1.4 Rezultat



Obrazek po pokolorowaniu pikseli w zależności od kierunku ich gradientów

2.1.5 Ścienianie krawedzi - Usunięcie odpowiednich pikseli

Kolejny krok to ścienianie krawędzi na podstawie wartości ich gradientów. W zależności od koloru piksela jest on porównywany z odpowiednimi sąsiadami i jezeli wartość jego gradientu jest maksymalna zostaje on na obrazie, a jesli nie, jest ustawiany na piksel czarny. Jednocześnie w tym fragmencie programu wykonałam częściowy tresholding. Piksele o wartości gradientu większej niż 80 zostały oznaczone na białe, a mniejsze od 40 zostały usunięte.

Kod realizujący powyższe operacje:

```
void kasowanie(MarvinImage imageIn, MarvinImage imageOut, Grad[][] tablica, DoubleMatrix
    kasowanie){
       for (int x = 3; x < imageIn.getWidth()-3; x++) {
            for (int y = 3; y < imageIn.getHeight()-3; y++) {
             if(tablica[x][y].kier==0) // kolor zolty - 0 stopni
                {
                if(tablica[x][y].wart!=Math.max(tablica[x][y].wart,( Math.max(tablica[x-1][y
                    ].wart,tablica[x+1][y].wart))))
                    {
                        kasowanie.put(x,y,1);
                    }
                    else
                    {
                        kasowanie.put(x,y,0);
                    }
                }
                if(tablica[x][y].kier==135)  // kolor zielony 135 stopni
                {
                  if(tablica[x][y].wart!=Math.max(tablica[x][y].wart,(Math.max(tablica[x+1][
                      y-1].wart, tablica[x-1][y+1].wart))))
                    {
                        kasowanie.put(x,y,1);
                    }
                    else
                    {
                        kasowanie.put(x,y,0);
                    }
                }
              if(tablica[x][y].kier==90)// kolor niebieski 90 stopni
                {
                    if(tablica[x][y].wart!=Math.max(tablica[x][y].wart,(Math.max(tablica[x][
                        y-1].wart,tablica[x][y+1].wart))))
                    {
                        kasowanie.put(x,y,1);
                    }

                    else
                    {
                        kasowanie.put(x,y,0);
                    }
                }
                if(tablica[x][y].kier==45) // kolor czerwony 45 stopni
                {
                    if(tablica[x][y].wart!=Math.max(tablica[x][y].wart,(Math.max(tablica[x
                        -1][y-1].wart,tablica[x+1][y+1].wart))))
                    {
                        kasowanie.put(x,y,1);
                    }
                    else
                    {
                        kasowanie.put(x,y,0);
                    }
                }
            }
        }
    }
```

2.1.6 Rezultat

Obrazek po ścienianiu krawędzi

2.1.7 EdgeTracking

Kolejnym krokiem było ustawienie pozostałych pikseli kolorowych jako białych lub czarnych w zależnoci od ich sąsiadów. Jeśli piksel kolorowy ma w swoim otoczeniu najbliższym piksel biały, jego wartośc jest ustawiana na biały. Jesli nie ma przeszukiwane jest otoczeie 5x5. Jeżeli piksel biały zostanie znaleziony w tym otoczeniu to analizowany piksel jest ustawiany na biały, jesli jednak tak się nie stanie jest on ustawiany na czarny.

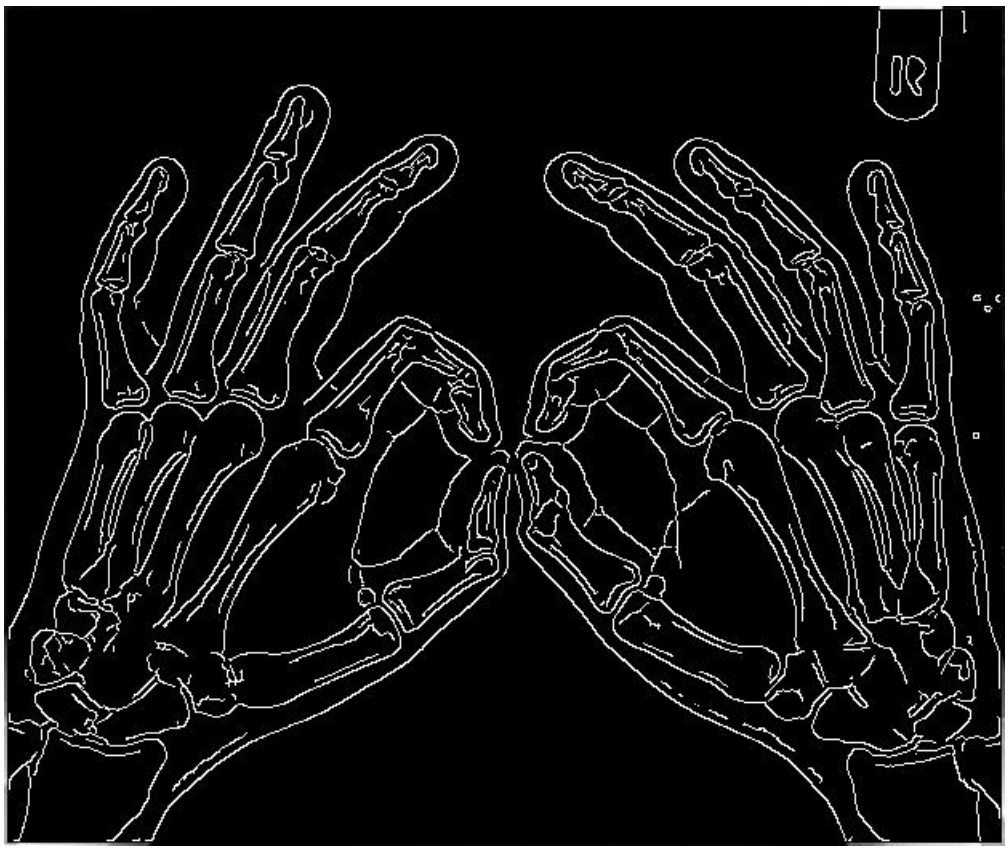
Kod metody

```
1     public void EdgeTracking(MarvinImage imageIn, MarvinImage imageOut)
2     {       for (int x = 2; x < imageIn.getWidth()-2; x++) {
3             for (int y = 2; y < imageIn.getHeight()-2; y++) {
4                     if(imageIn.getIntComponent0(x, y)!=0 && // rozne niz czarny
5                        imageIn.getIntComponent0(x, y)!=255) // rozne niz bialy
6                     {
7                         check1:
8                         for( int i = x-1, k = 0; k <3;k++,i++)
9                         {
10                            for ( int j = y-1,l=0; l <3; l++,j++)
11                            {
```

```
12                          if(imageIn.getIntComponent0(i, j)==255)
13                              {
14                                  imageOut.setIntColor(x, y, 255,255,255);
15                                  break check1;
16                              }
17                              else
18                              {
19                                  for( int ii = x-2,m=0; m <5;m++,ii++)
20                                   {
21                                      for ( int jj = y-2,n=0; n <5; n++,jj++)
22                                      {
23                                          if(imageIn.getIntComponent0(i, j)==255)// czy
                                                znajdzie bialy?
24
25                                           {
26                                              imageOut.setIntColor(x, y, 255,255,255);
27                                              break check1;
28                                           }
29                                           else
30                                      {
31                                          imageOut.setIntColor(x,y,0,0,0);
32                                      }
33                                      }}}}}}}}}
```

EFEKT KOŃCOWY

2.1.8 Rezultat



Obrazek po zastosowaniu Filtru Canny

2.2 Krok 2: Usunięcie obwódki wokół kości

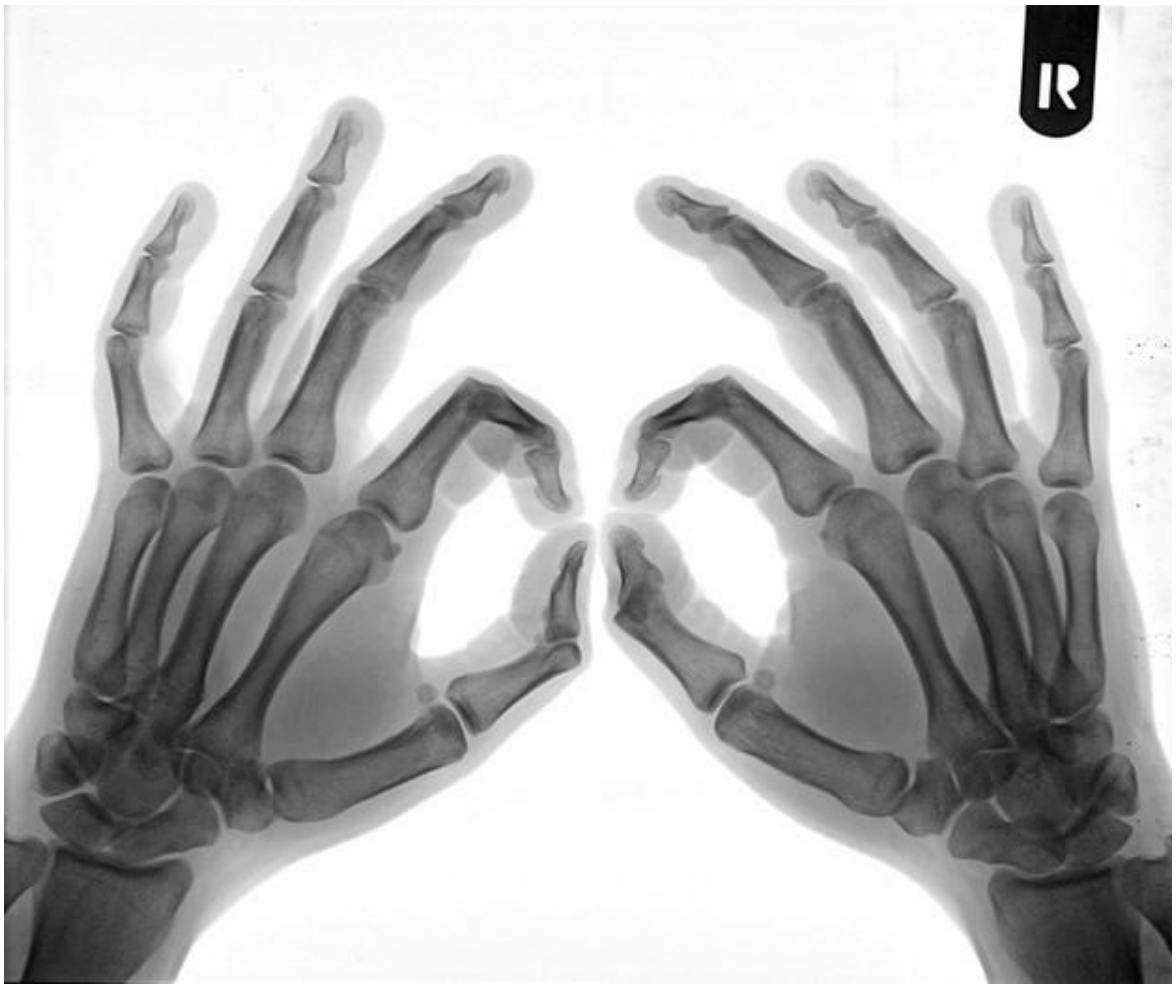
Kolejnym krokiem było usunięcie obwódki - obramowania skóry - wokół kości. Do uzyskania tego celu należało nałożyc na siebie dwa obrazy, jeden to obraz po filtrze Canny a drugi to obraz po następujących przekształceniach:
- Inwersja obrazu - Negatyw
- Binaryzacja Otsu - czyli znalezienie odpowiedniego progu do thresholdingu na podstawie analizy histogramu obrazu.
- Kilkukrotna erozja otrzymanego obrazu celem zakrycia obwódki kości.

2.2.1 Inwersja Obrazu

Operacja konwersji na negatyw polega na odwróceniu wartościu każdego kanału modelu RGB dla każdego piksela obrazu. Metoda przyjmuje jako argument obiekt klasy Bitmap zawierający obraz. Na początku tworzony jest nowy obiekt klasy Bitmap na podstawie przyjmowanego argumentu, dwie pętle przechodzą przez obraz oryginalny i odwracają wartość każdego kanału po czym nowy piksel wstawiany jest do nowego obrazka, który na końcu jest zwracany.

```
1  public void process(
2              MarvinImage imageIn,
3              MarvinImage imageOut,
4              MarvinAttributes attributesOut,
5              MarvinImageMask mask,
6              boolean previewMode) {
7          int r, g, b;
8          for (int x = 0; x < imageIn.getWidth(); x++) {
9              for (int y = 0; y < imageIn.getHeight(); y++) {
10                 r = (255 - (int) imageIn.getIntComponent0(x, y));
11                 g = (255 - (int) imageIn.getIntComponent1(x, y));
12                 b = (255 - (int) imageIn.getIntComponent2(x, y));
13
14                 imageOut.setIntColor(x, y, r, g, b);
15             }
16         }
17     }
```

2.2.2 Rezultat



Obrazek po zastosowaniu negatywu

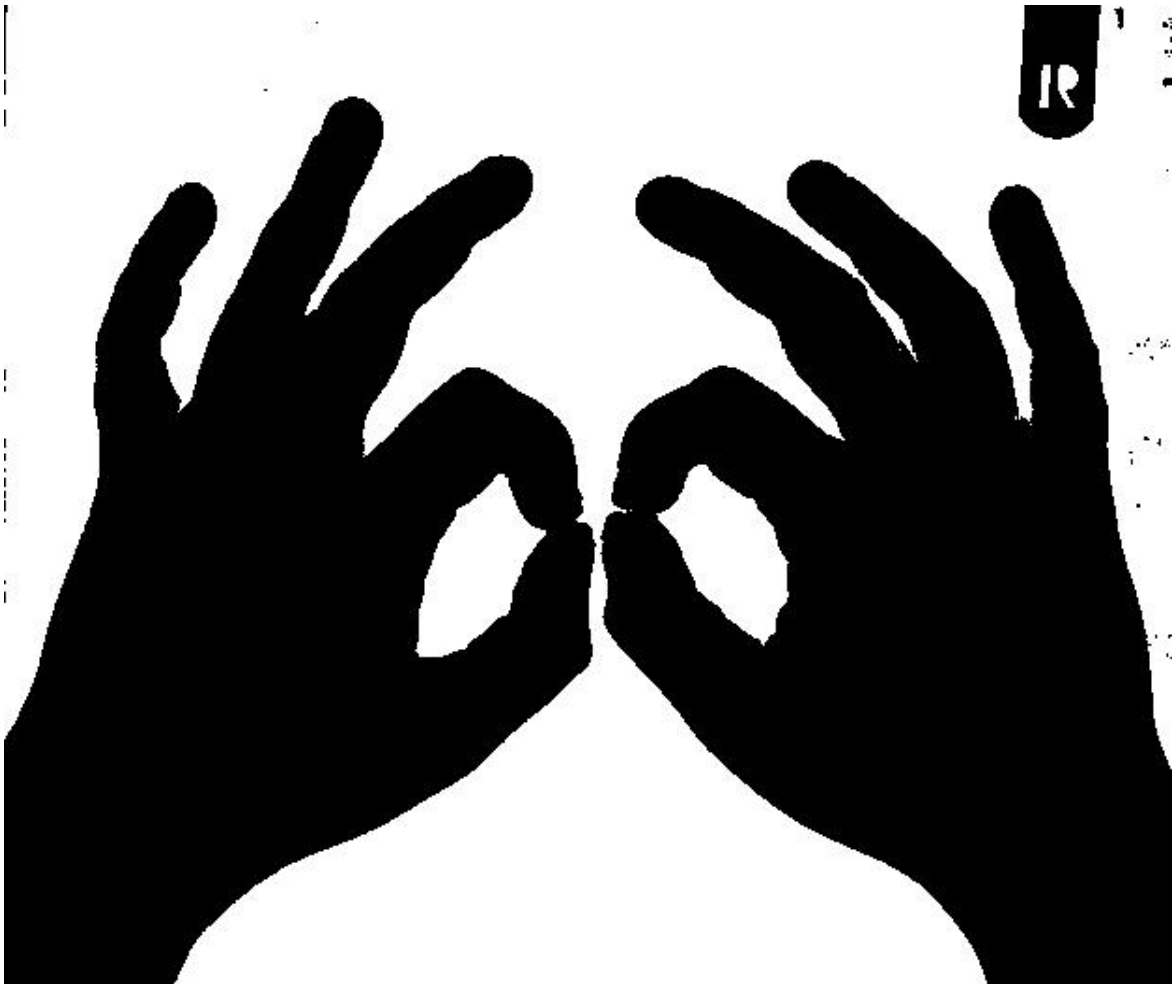
2.2.3 Binaryzacja Otsu

Kolejny krok to binaryzacja metodą Otsu. Polega ona na thresholdingu obrazu z odpowiednio dobranym progiem. Próg dobierany jest na podstawie histogramu obrazu.

```
int[] H = new int [256];
        for(int x = 0; x < imageIn.getWidth();++x)
            for(int y = 0; y < imageIn.getHeight(); ++y)
                    {
                        H[imageIn.getIntComponent0(x,y)]++;
                    }
        int LP = imageIn.getWidth()*imageIn.getHeight();
        int SU[] = new int [255];
        for ( int i = 0; i < 255; i++)
        {
            SU[i] = H[i]*i;
        }
        int W=0;
        int MAX=0;
        int SUP=0;
        double R;
        int T1=0;
        int T2=0;
        for ( int i =0; i <255; i++)
        {
```

```
21              W = W+H[i];
22              if(W==0) continue;
23              int WP = LP-W;
24              if(WP==0) break;
25                      SUP = i*H[i];
26                      int SG = SUP/W;
27                      int SD = (SU[i]-SUP)/WP;
28                      R = Math.pow(W*WP*(SG-SD),2);
29
30                  if(R>=MAX)
31                  {
32                      T1 = i;
33                  }
34                  if(R>MAX)
35                  {
36                      T2 = i;
37                      MAX=(int)R;
38                  }
39              }
40
41              int prog = (int)((T1+T2)/2);
42
43              for (int x = 0; x < imageIn.getWidth(); x++)
44              {
45                  for (int y = 0; y < imageIn.getHeight(); y++) {
46                      int k = (int) imageIn.getIntComponent0(x, y);
47                      if(k > prog) k = 255;
48                      else k = 0;
49                      imageOut.setIntColor(x, y, k, k, k);
50                      }
51              }
52      }
```

2.2.4 Rezultat



Obrazek po zastosowaniu binaryzacji Otsu

2.2.5 Erozja obrazu

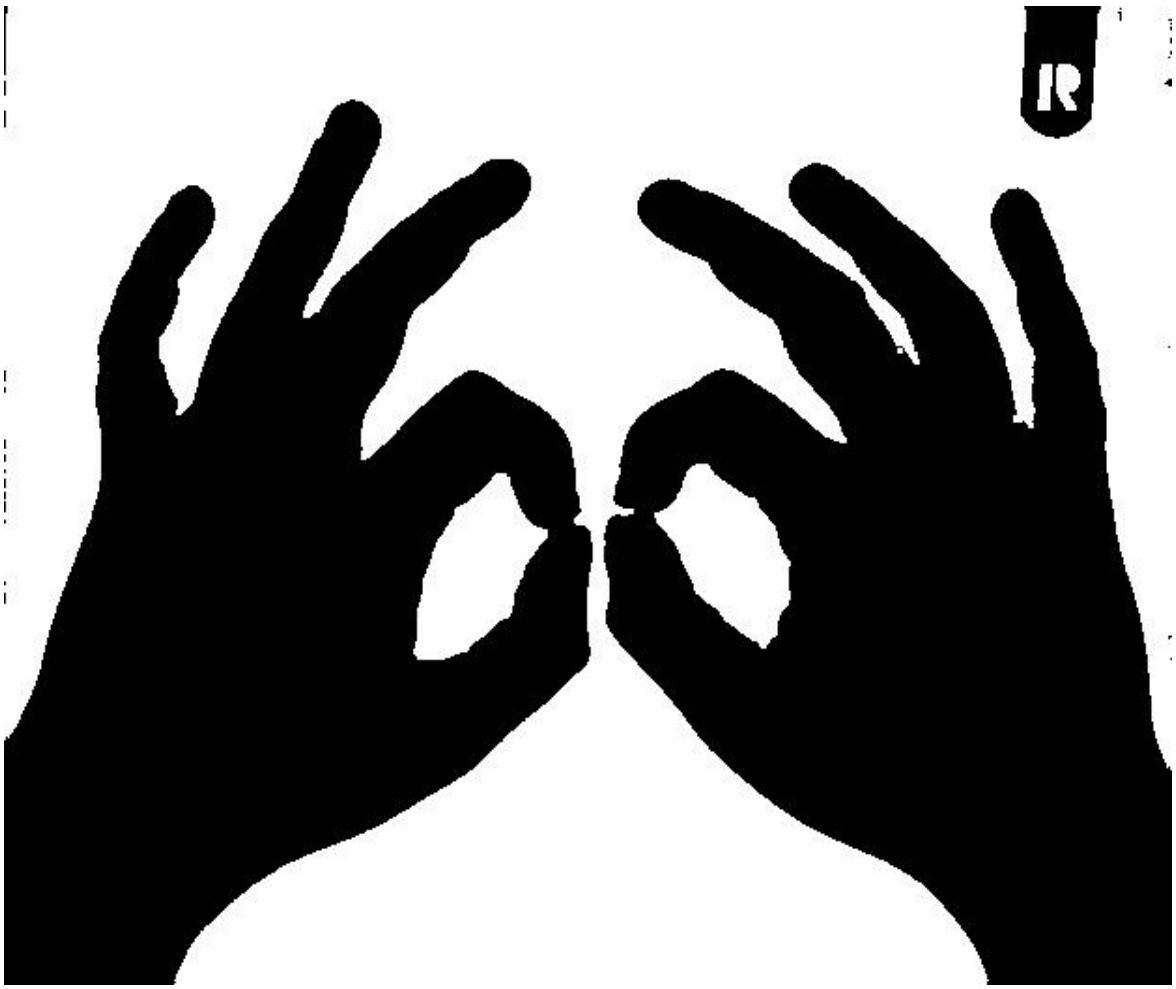
Kolejny etap do wykonanie erozji na otrzymanym obrazie, aby później po nałożenia obrazu końcowego na obraz po filrze Canny obwódki kości zostały usunięte.

Erozja to operacja morfologiczna polegająca na zastosowaniu sumy Minkowskiego na obrazie i elemencie strukturalnym SE. W praktyce polega to na przyłożeniu do każdego piksela obrazu elementu strukturalnego (SE) w jego punkcie centralnym i jeśli choć jeden piksel sąsiedztwa przykryty przez SE ma wartość '0', piksel bieżący też przyjmuje wartość '0' (tła).

```
int [][] SE = {{0,0,0},{0,0,0},{0,0,0}};

        for (int x = 2; x < imageIn.getWidth()-2; x++) {
            for (int y = 2; y < imageIn.getHeight()-2; y++) {

                for (int i = x-1, k =0; k<3; i++,k++)
                {
                    for ( int j =y-1, l =0; l <3; j++,l++ )
                    {
                        if(!(SE[k][l]==(int)imageIn.getIntComponent0(i, j)))
                        {
                            imageOut.setIntColor(x, y, 255,255,255);
                            break;
                        }
```

```
                    }
                }
            }
        }
```

2.2.6 Rezultat



Obrazek po zastosowaniu erozji

2.3 Krok 3: Nałożenie na siebie obrazów z kroku 2 i 3

W tym kroku tworzę tablice odpowiadające kazdemu z obrazów a następnie wykonuję ich sumę logiczną i konwertuję na obraz otrzymując obraz kości bez obwódki.

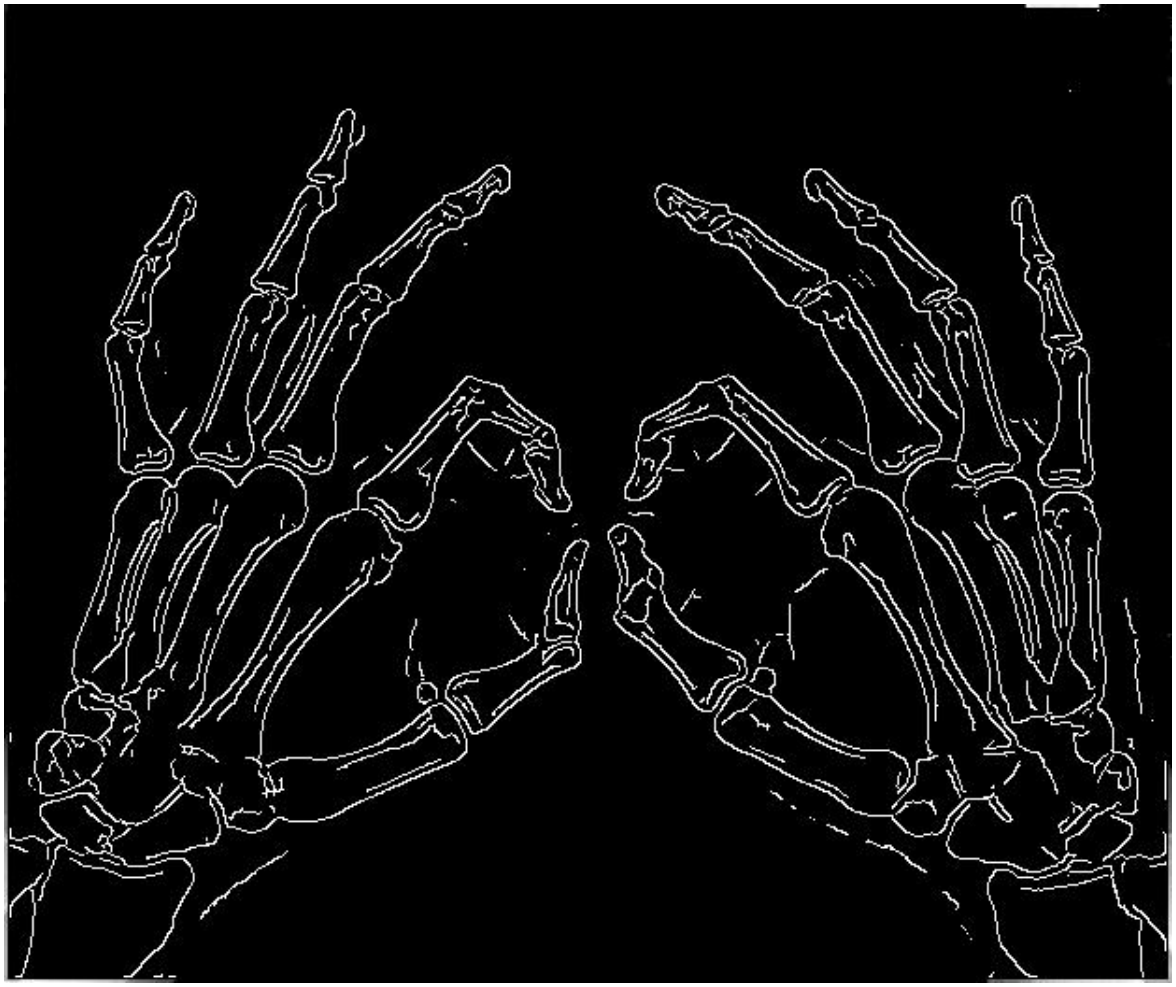
2.3.1 Czynności

Kod

```
public void DodawanieDwochZdjec(DoubleMatrix[] tablica1, DoubleMatrix[] tablica2,
    MarvinImage imageOut )
   {
    int r,g,b;
        for (int x =0 ; x < imageOut.getWidth(); x++)
```

```
5           {
6               for ( int y = 0; y<imageOut.getHeight(); y++)
7               {
8
9                   int r1 = Math.abs((int)tablica1[0].get(x,y))&Math.abs((int)tablica2[0].get(x
                        ,y));
10                  int g1 = Math.abs((int)tablica1[1].get(x,y))&Math.abs((int)tablica2[1].get(x
                        ,y));
11                  int b1 = Math.abs((int)tablica1[2].get(x,y))&Math.abs((int)tablica2[2].get(x
                        ,y));
12
13                  r =Math.abs((int)tablica1[0].get(x,y))-r1;
14                  g =Math.abs((int)tablica1[1].get(x,y))-g1;
15                  b =Math.abs((int)tablica1[2].get(x,y))-b1;
16
17
18                  imageOut.setIntColor(x, y, r, g, b);
19              }
20          }
21      }
```

2.3.2 Rezultat



Obrazek po nałożeniu obrazu z filtru Canny i po binaryzacji i erozji

2.4 Krok 3: Domknięcie kości

W kolejnym etapie dokonałam dylajci otrzymanego obrazu celem " domknięcia kości " tak aby tworzyły figury zamknięte. Operacja ta była potrzebna do pokolorowania każdej koci z osobna na inny kolor.

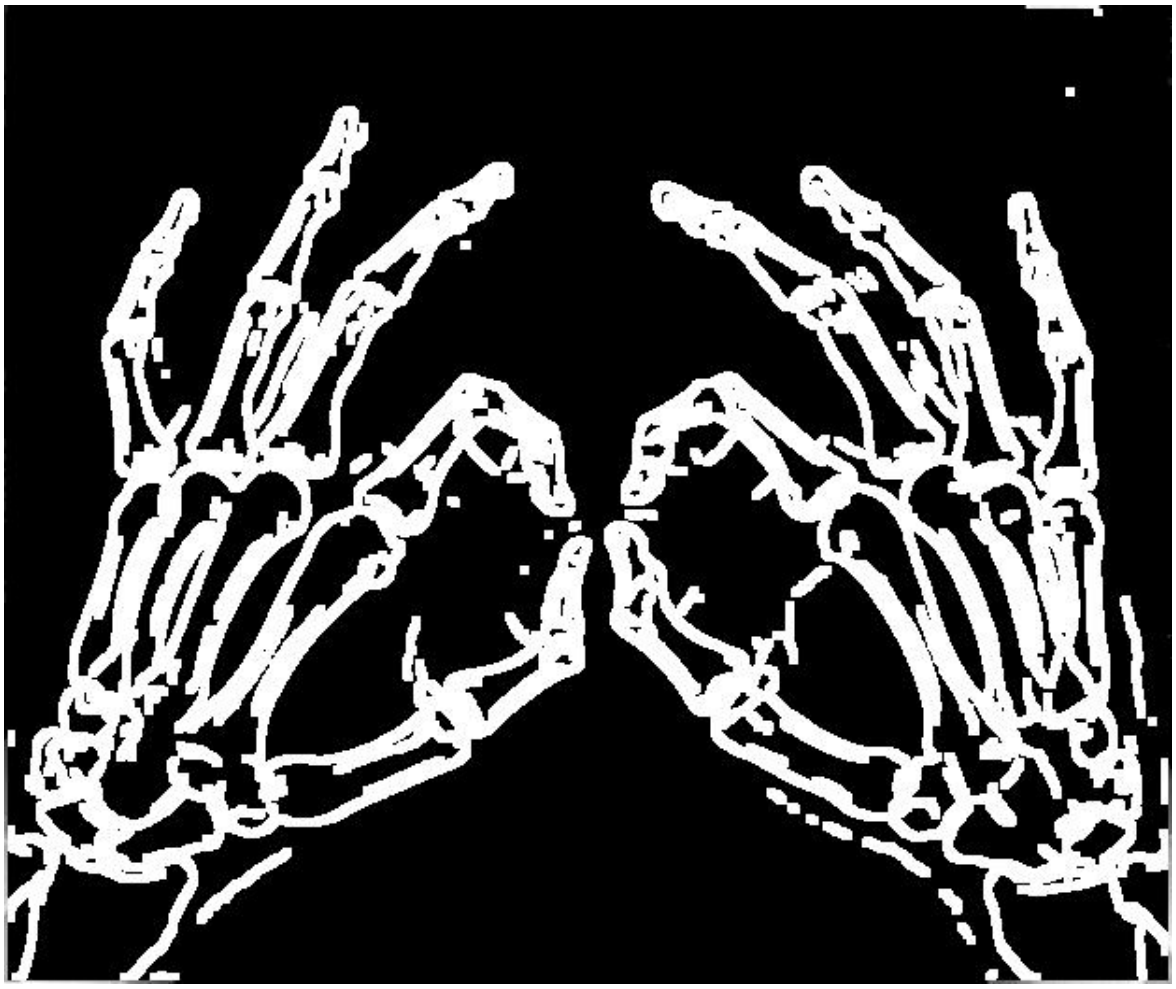
Opracja Dylacji to operacja odwrotna do erozji.

2.4.1 Czynności

Kod

```
int [][] SE =
        {{255,255,255},
         {255,255,255},
         {255,255,255}};

    DoubleMatrix tablica = new DoubleMatrix(imageIn.getWidth(),imageIn.getHeight());

      for (int x = 2; x < imageIn.getWidth()-2; x++) {
          for (int y = 2; y < imageIn.getHeight()-2; y++) {

              for (int i = x-1, k =0; k<3; i++,k++)
              {
                  for ( int j =y-1, l =0; l <3; j++,l++ )
                  {
                      if(SE[k][l]==(int)imageIn.getIntComponent0(i, j))
                      {
                         // imageOut.setIntColor(x, y, 255,255,255);
                          tablica.put(x,y,1);
                          break;
                      }
                  }
              }
          }
      }

          for (int x = 2; x < imageIn.getWidth()-2; x++) {
              for (int y = 2; y < imageIn.getHeight()-2; y++) {

                  if(tablica.get(x,y)==1)
                  {
                      imageOut.setIntColor(x, y, 255,255,255);

                  }
               }
          }
      }
```

2.4.2 Rezultat



Obrazek po zastosowaniu dylacji na obrazie

2.5 Krok 4: Kolorowanie kości

Kolejnym krokiem było pokolorowanie kości na kolory. Do tego celu zastosowałam algorytm Flood-Fill. Jest on wywoływany przy każdej napotkanej zmianie piksela z białego na czarny, dzięki temu każda pełna figura zostaje pokolorowana. Co więcej użyłam tutaj rólnież funkcji losującej kolory, tak aby każda kośc była pokolorowana na inny kolor. Ponadto użyłam w tym kroku metody zamieniającej wszytskie piksele białe na czarne, żeby później łatwo można było nałożyc otrzymany obraz na obraz oryginalny.

2.5.1 Czynności

Kod - Algortm Flood - Fill

```
1  public void floodFill(MarvinImage image, Point node, Color targetColor, Color
       replacementColor) {
2      int width = image.getWidth();
3      int height = image.getHeight();
4      int[] target = targetColor.getRGB();
5      int[] replacement = replacementColor.getRGB();
6      if (target != replacement) {
7        Deque<Point> queue = new LinkedList<Point>();
8        do {
9          int x = node.x;
10         int y = node.y;
```

```
        while (x > 5 && image.getIntComponent0(x - 1, y) == target[0]
                     &&image.getIntComponent1(x - 1, y) == target[1]
                     && image.getIntComponent2(x - 1, y) == target[2]) {
          x--;
        }
        boolean spanUp = false;
        boolean spanDown = false;
        while (x < width-5 && image.getIntComponent0(x, y) == target[0]
                           &&image.getIntComponent1(x , y) == target[1]
                           && image.getIntComponent2(x, y) == target[2]) {
          image.setIntColor(x, y, replacement[0],replacement[1],replacement[2]);
          if (!spanUp && y > 5 && image.getIntComponent0(x, y - 1) == target[0]
                               &&image.getIntComponent1(x, y-1) == target[1]
                               &&image.getIntComponent2(x , y-1) == target[2])

                    {
            queue.add(new Point(x, y - 1));
            spanUp = true;
          } else if (spanUp && y > 5 && image.getIntComponent0(x, y - 1) != target[0]
                                     && image.getIntComponent1(x, y-1)!= target[1]
                                     && image.getIntComponent2(x, y-1) != target[2]) {
            spanUp = false;
          }
          if (!spanDown && y < height - 5 && image.getIntComponent0(x, y + 1) == target[0]
                                          &&image.getIntComponent1(x , y+1) == target[1]
                                          && image.getIntComponent2(x, y+1) == target[2]){
            queue.add(new Point(x, y + 1));
            spanDown = true;
          } else if (spanDown && y < height - 5 && image.getIntComponent0(x, y + 1) !=
              target[0]
                                                &&image.getIntComponent1(x , y+1) != target
                                                    [1]
                                                && image.getIntComponent2(x, y+1) != target
                                                    [2]){
            spanDown = false;
          }
          x++;
        }
      } while ((node = queue.pollFirst()) != null);
    }
  }
```

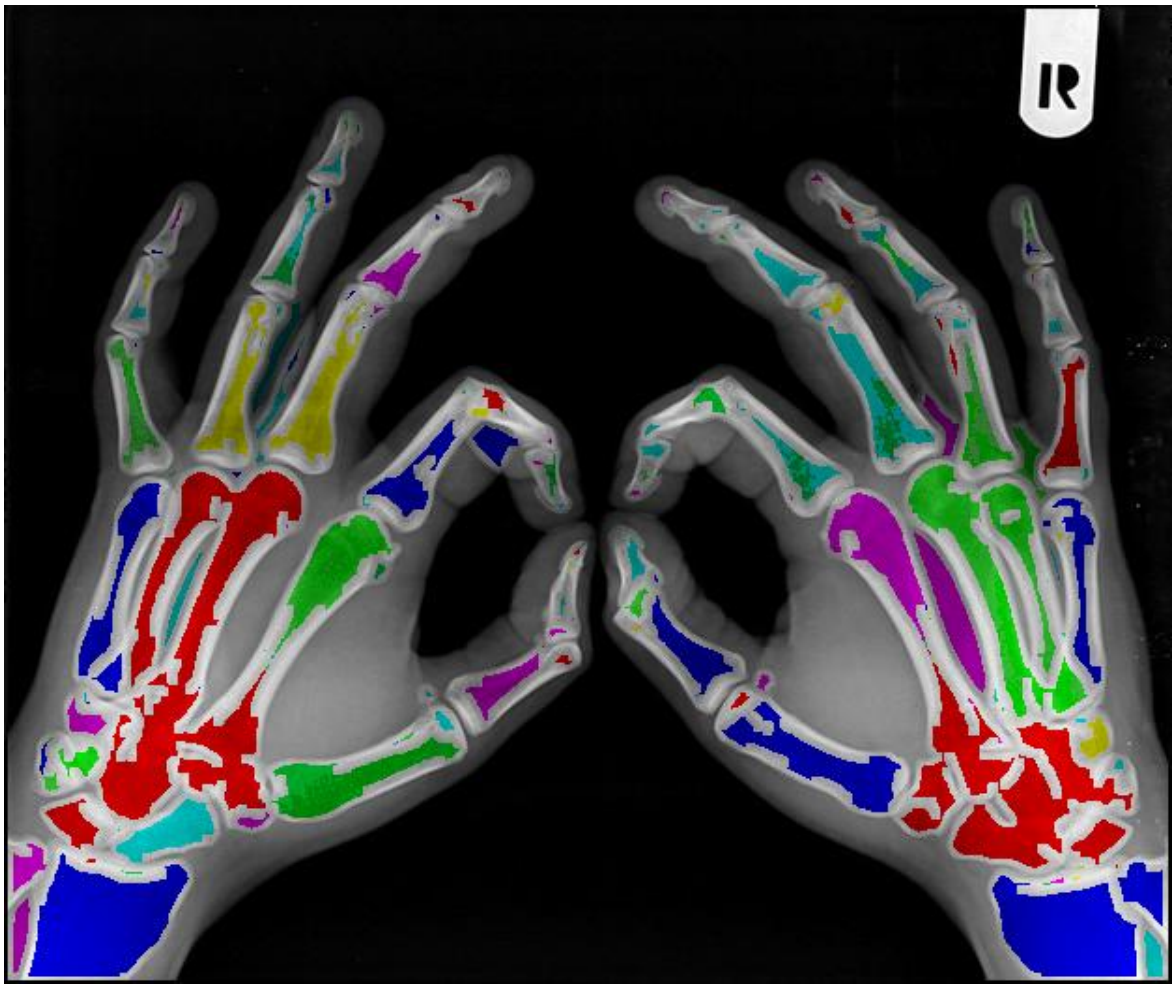
2.5.2 Rezultat

Obrazek po pokolorowaniu kości

2.6 Krok 5: Efekt końcowy

Ostatnim etapem było nałożenie na siebie obrazu oryginalnego i obrazu z pokolorowanymi kośćmi. Do tego celu w poprzednim kroku zamienilam wszystkie białe piksele na czarne, tak aby można było z powodzeniem wykonac operację logiczną "and" na tych dwóch obrazach. Do mojego programu dołączyłam funkcję licząca ilosc kolorów na obrazie, przy założeniu że jedna kośc ma wiecej niż 50 pikseli.

2.6.1 Rezultat



Obrazek oryginalny po nałożeniu kolorów.

3 Wnioski

Podumowując program kolorował kości niedokładnie. Trudno byłoby wyodrębnić wszystkie kości ze względu na brak możliwości odpowiedniego ich " domknięcia". Ponadto Filtr Canny napisany przeze mnie nie działa idealnie i wprowadza zakłócenia do efektu końcowego.
Liczba Kości na moim obrazie wynosi 55. Jest to liczba zbliżona do prawdziwej liczby kości w ludzkich dłoniach (54). Program jednak wykryl kilka obszarów które kośćmi nie są, i nie wykrył kilku takich które nimi są, więc przy innym zdjęciu rentgenowskim algorytm mógłby nie zadziałać poprawnie. Jednak do policzenia kości na tym konretnym zdjęciu nadał się dobrze.

Literatura

[1] Strona laboratorium misztal.edu.pl

[2] http://rosettacode.org/wiki/Bitmap/Flood_fill