Sistemas Sensoriais
Traffic sign identification

Amadeusz Szymko 57315
Filip Szymański 57300
Bartosz Bednarski 57297

1. Objective

The goal of the project is to create a software allowing to recognize traffic signs from photos. The software should focus mostly on recognition of a value on a speed limit sign. However it is recommended for the software also to be able to recognize other signs meaning that it should detect if there is a different one in the picture not necessarily knowing what kind of sign that is specifically.

Photos used for testing should be in .png file format and have dimensions of approximately 1000 by 1000 pixels.

2. Approach

To resolve this problem correctly it is crucial to divide it into smaller steps and understand how it should be implemented so that computational time is lowest and the program still works properly.

The workflow of our program can be divided into following steps:

- color format conversion from RGB to HSV
- detection of signs using connected components algorithm
- detection of sign elements (e.g. digits)
- verification of detected components
- classification of signs and recognizing speed limit

3. Implementation

3.1 Color format conversion

All pictures used for testing are in the RGB color format meaning that each pixel in the image has three values for defining its color. Those values are in range from 0 to 255 and each of them represents amount of Red, Green and Blue channel.

Another color format often used for image processing is HSV format. The difference between HSV and RGB is that HSV separates luma from chroma. This means that it separates the image intensity from the color information. In computer vision it is needed to separate color components from intensity for various reasons, such as robustness to lighting changes, or removing shadows. Hence the conversion.

```
public static void BgrToHsv(Image<Bgr, byte> img, Image<Bgr, byte> imgCopy)
        {
            unsafe
            {
                int x, y;
                MIplImage m = img.MIplImage;
                byte* dataPtr = (byte*)m.imageData.ToPointer(); // Pointer to the image
                MIplImage m2 = imgCopy.MIplImage;
                byte* dataPtr2 = (byte*)m2.imageData.ToPointer(); // Pointer to the image
                int width = img.Width;
                int height = img.Height;
                int nChan = m.nChannels; // number of channels - 3
                int padding = m.widthStep - m.nChannels * m.width;

                for (y = 0; y < height; y++)
                {
                    for (x = 0; x < width; x++)
                    {
                        var x0 = x;
                        var y0 = y;

                        int blue = (dataPtr + y * m.widthStep + x * nChan)[0];
                        int green = (dataPtr + y * m.widthStep + x * nChan)[1];
                        int red = (dataPtr + y * m.widthStep + x * nChan)[2];

                        System.Drawing.Color intermediate = System.Drawing.Color.FromArgb(red, green, blue);
                        Hsv hsvPixel = new Hsv(intermediate.GetHue(), intermediate.GetSaturation(), intermediate.GetBrightness());

                        double r = red / 255.0;
                        double g = green / 255.0;
                        double b = blue / 255.0;
                        double hue, sat, val;

                        double cmax = Math.Max(r, b);
                        cmax = Math.Max(cmax, g);
                        double cmin = Math.Min(r, b);
                        cmin = Math.Min(cmin, g);
                        double delta = cmax - cmin;

                        //Hue
                        hue = hsvPixel.Hue;

                        //Saturation
                        if (cmax == 0)
                        {
                            sat = 0;
                        }
                        else
                        {
                            sat = delta / cmax;
                        }

                        //Value
                        val = cmax;

                        (dataPtr2 + y * m.widthStep + x * nChan)[0] = (byte)(Math.Round(val * 255));
                        (dataPtr2 + y * m.widthStep + x * nChan)[1] = (byte)(Math.Round(sat * 255));
                        (dataPtr2 + y * m.widthStep + x * nChan)[2] = (byte)(Math.Round(hue * 255 / 360));

                    }
                }
            }
        }
```

3.2 Detection of signs

Second step that needs to be done is the selection of a traffic signs. The thing that speed limit signs in European countries have in common is that they have a red circle around it. Now that we do not have to worry about shades of red (HSV color format), basically, the program is told to look for red pixels (a threshold value was chosen to specify what is considered red). Next step is to find a connected component so we know it is not some random pixel. By connected component it is understood that it is a specific combination of pixels that fills some requirements. For instance in this step we are looking for red pixels that have other red pixels in their nearest neighborhood. As the program continues to process the image it finds a pattern of pixels of the same color connected with each other. The following function was used.

```
public static List<List<int[]>> connectedComponents(Image<Bgr, byte> imgHsv, Image<Bgr, byte> img, int hueLimit  =  20, int satLimit
=  50, int valLimit = 30)
        {
            unsafe
            {
                int x, y;
                MIplImage m = imgHsv.MIplImage;
                byte* dataPtr = (byte*)m.imageData.ToPointer(); // Pointer to the image

                int width = imgHsv.Width;
                int height = imgHsv.Height;
                int nChan = m.nChannels; // number of channels - 3
                int padding = m.widthStep - m.nChannels * m.width;

                int hue = 0;
                int value = 0;
                int saturation = 0;

                int[,] indexTable = new int[width, height];
                int currentMax = 0;
                List<int> objects = new List<int>();
                List<(int, int)> aliases = new List<(int, int)>();
                List<int[]> result = new List<int[]>();

                int[,] indexTableBlack = new int[width, height];
                int currentMaxBlack = 0;
                List<int> objectsBlack = new List<int>();
                List<(int, int)> aliasesBlack = new List<(int, int)>();
                List<int[]> resultBlack = new List<int[]>();

                for (y = 1; y < height - 1; y++)
                {
                    for (x = 1; x < width - 1; x++)
                    {
                        value = ((dataPtr + y * m.widthStep + x * nChan)[0] * 100) / 255;
                        saturation = ((dataPtr + y * m.widthStep + x * nChan)[1] * 100) / 255;
                        hue = ((dataPtr + y * m.widthStep + x * nChan)[2] * 360) / 255;

                        if (((hue >= 0 && hue <= hueLimit)||(hue >= 360 - hueLimit && hue <= 360)) && (saturation <= 100 &&
saturation > satLimit) && (value <= 100 && value > valLimit))         //if red
                        {
                            //8 - connectivity
                            if (indexTable[x + 1, y - 1] != 0)
                                indexTable[x, y] = indexTable[x + 1, y - 1];

                            if (indexTable[x, y - 1] != 0 && indexTable[x, y] != indexTable[x, y - 1])
                            {
                                if (indexTable[x, y] == 0)
                                    indexTable[x, y] = indexTable[x, y - 1];
                                else                  //Find lower index,  change  higher  to lower
                                {
                                    if (indexTable[x, y] < indexTable[x, y - 1])
                                        aliases.Add((indexTable[x, y], indexTable[x, y - 1]));

                                    else
                                    {
                                        indexTable[x, y] = indexTable[x, y - 1];
                                        aliases.Add((indexTable[x, y - 1], indexTable[x, y]));
                                    }
                                }
                            }
                            else if (indexTable[x - 1, y - 1] != 0 && indexTable[x, y] != indexTable[x - 1, y - 1])
                            {
                                if (indexTable[x, y] == 0)
                                    indexTable[x, y] = indexTable[x - 1, y - 1];
                                else
                                {
                                    if (indexTable[x, y] < indexTable[x - 1, y - 1])
                                        aliases.Add((indexTable[x, y], indexTable[x - 1, y - 1]));
                                    else
                                    {
                                        indexTable[x, y] = indexTable[x - 1, y - 1];
                                        aliases.Add((indexTable[x - 1, y - 1], indexTable[x, y]));
                                    }
                                }
                            }
                            if (indexTable[x - 1, y] != 0 && indexTable[x, y] != indexTable[x - 1, y])
                            {
                                if (indexTable[x, y] == 0)
                                    indexTable[x, y] = indexTable[x - 1, y];
                                else
                                {
                                    if (indexTable[x, y] < indexTable[x - 1, y])
                                        aliases.Add((indexTable[x, y], indexTable[x - 1, y]));

                                    else
                                    {
                                        indexTable[x, y] = indexTable[x - 1, y];
                                        aliases.Add((indexTable[x - 1, y], indexTable[x, y]));
                                    }
                                }
                            }
                            if (indexTable[x, y] == 0)
                            {
                                currentMax = currentMax + 1;
                                indexTable[x, y] = currentMax;
                                objects.Add(currentMax);
                            }
                        }
                        else
                        {
                            indexTable[x, y] = 0;
                        }
                    }
                }

                IDictionary<int, List<int>> dict = getAliases(aliases);

                for (y = 0; y < height; y++)
                {
                    for (x = 0; x < width; x++)
                    {
                        foreach (int index in dict.Keys)
                        {
                            if (dict[index].Contains(indexTable[x, y]))
                            {
                                indexTable[x, y] = index;
                            }
                        }
                    }
                }

                List<int> toRemove = new List<int>();
                foreach (int obj in objects)
                {
```

```
    foreach (int index in dict.Keys)
    {
        if (dict[index].Contains(obj) && !index.Equals(obj))
        {
            toRemove.Add(obj);
        }
    }
}
foreach (int key in toRemove)
{
    objects.Remove(key);
}
foreach (int obj in objects)
{
    int count = 0;
    foreach (int num in indexTable)
    {
        if (num.Equals(obj))
            count += 1;
    }

    if (count > 500)
    {
        int code = obj;
        int begX = 10000000;
        int begY = 10000000;
        int endX = -1;
        int endY = -1;

        for (int i = 0; i < height; i++)
        {
            for (int j = 0; j < width; j++)
            {
                if (indexTable[j, i] == code)
                {
                    if (i < begX)
                        begX = i;
                    if (j < begY)
                        begY = j;
                    if (i > endX)
                        endX = i;
                    if (j > endY)
                        endY = j;
                }
            }
        }
        result.Add(new int[] { begY, begX, endY, endX });
    }
}

Image<Bgr, Byte> otsu = img.Copy();

foreach(int[] res in result)
{
    Identify.ConvertToBW_Otsu_coords(otsu, res);
    MIplImage mOtsu = otsu.MIplImage;
    byte* dataPtrOtsu = (byte*)mOtsu.imageData.ToPointer(); // Pointer to the image

    for (y = res[1]; y <= res[3]; y++)
    {
        for (x = res[0]; x <= res[2]; x++)
        {
            //value = (dataPtrOtsu + y * m.widthStep + x * nChan)[0];

            value = ((dataPtr + y * m.widthStep + x * nChan)[0] * 100) / 255;

            if (value<35)         //if black
            {
                //8 - connectivity
                if (indexTableBlack[x + 1, y - 1] != 0)
                    indexTableBlack[x, y] = indexTableBlack[x + 1, y - 1];

                if (indexTableBlack[x, y - 1] != 0 && indexTableBlack[x, y] != indexTableBlack[x, y - 1])
                {
                    if (indexTableBlack[x, y] == 0)
                        indexTableBlack[x, y] = indexTableBlack[x, y - 1];
                    else                  //Find lower index,  change  higher  to lower
                    {
                        if (indexTableBlack[x, y] < indexTableBlack[x, y - 1])
                            aliasesBlack.Add((indexTableBlack[x, y], indexTableBlack[x, y - 1]));

                        else
                        {
                            indexTableBlack[x, y] = indexTableBlack[x, y - 1];
                            aliasesBlack.Add((indexTableBlack[x, y - 1], indexTableBlack[x, y]));
                        }
                    }
                }
                if (indexTableBlack[x - 1, y - 1] != 0 && indexTableBlack[x, y] != indexTableBlack[x - 1, y - 1])
                {
                    if (indexTableBlack[x, y] == 0)
                        indexTableBlack[x, y] = indexTableBlack[x - 1, y - 1];
                    else
                    {
                        if (indexTableBlack[x, y] < indexTableBlack[x - 1, y - 1])
                            aliasesBlack.Add((indexTableBlack[x, y], indexTableBlack[x - 1, y - 1]));
                        else
                        {
                            indexTableBlack[x, y] = indexTableBlack[x - 1, y - 1];
                            aliasesBlack.Add((indexTableBlack[x - 1, y - 1], indexTableBlack[x, y]));
                        }
                    }
                }
                if (indexTableBlack[x - 1, y] != 0 && indexTableBlack[x, y] != indexTableBlack[x - 1, y])
                {
                    if (indexTableBlack[x, y] == 0)
                        indexTableBlack[x, y] = indexTableBlack[x - 1, y];
                    else
                    {
                        if (indexTableBlack[x, y] < indexTableBlack[x - 1, y])
                            aliasesBlack.Add((indexTableBlack[x, y], indexTableBlack[x - 1, y]));

                        else
                        {
                            indexTableBlack[x, y] = indexTableBlack[x - 1, y];
                            aliasesBlack.Add((indexTableBlack[x - 1, y], indexTableBlack[x, y]));
                        }
                    }
                }
                if (indexTableBlack[x, y] == 0)
                {
                    currentMaxBlack = currentMaxBlack + 1;
                    indexTableBlack[x, y] = currentMaxBlack;
                    objectsBlack.Add(currentMaxBlack);
                }
            }
            else
            {
                indexTableBlack[x, y] = 0;
            }
        }
    }
}

dict = getAliases(aliasesBlack);

for (y = 0; y < height; y++)
{
    for (x = 0; x < width; x++)
    {
        foreach (int index in dict.Keys)
        {
            if (dict[index].Contains(indexTableBlack[x, y]))
            {
                indexTableBlack[x, y] = index;
            }
        }
    }
}

toRemove = new List<int>();
foreach (int obj in objectsBlack)
{
    foreach (int index in dict.Keys)
    {
        if (dict[index].Contains(obj) && !index.Equals(obj))
        {
            toRemove.Add(obj);
        }
    }
}

foreach (int key in toRemove)
{
    objectsBlack.Remove(key);
}

foreach (int obj in objectsBlack)
{
    int count = 0;
    foreach (int num in indexTableBlack)
    {
        if (num.Equals(obj))
            count += 1;
    }
    if (count > 400)
    {
        int code = obj;
        int begX = 10000000;
        int begY = 10000000;
        int endX = -1;
        int endY = -1;

        for (int i = 0; i < height; i++)
        {
            for (int j = 0; j < width; j++)
            {
                if (indexTableBlack[j, i] == code)
                {
                    if (i < begX)
                        begX = i;
                    if (j < begY)
                        begY = j;
                    if (i > endX)
                        endX = i;
                    if (j > endY)
                        endY = j;
                }
            }
        }
            resultBlack.Add(new int[] { begY, begX, endY, endX });
    }
}

List<int[]> newBlack = new List<int[]>();
foreach (int[] red in result)
{
    foreach (int[] black in resultBlack)
```

```
                    {
                        if ((black[0] - red[0] > 20) && (black[1] - red[1] > 20) && (red[2] - black[2] > 20) && (red[3] - black[3] > 
20))
                        {
                            newBlack.Add(black);
                        }
                    }
                }

                //Drawing rectangle
                Identify.DrawRectangles(img, result);
                Identify.DrawRectangles(img, resultBlack, 1);
                Identify.DrawRectangles(img, newBlack, 2);

                return new List<List<int[]>> { result, newBlack };
            }
        }
```

To connect different areas of one object a helper function had to be added.

```
public static IDictionary<int, List<int>> getAliases(List<(int, int)> lista)
        {
            lista.Sort((first, second) => first.Item1.CompareTo(second.Item1));
            IDictionary<int, List<int>> dict = new Dictionary<int, List<int>>();

            foreach ((int, int) x in lista)
            {
                if (dict.ContainsKey(x.Item1))
                    dict[x.Item1].Add(x.Item2);
                else
                    dict.Add(new KeyValuePair<int, List<int>>(x.Item1, new List<int>() { x.Item2 }));
            }
             dict.OrderByDescending(pair => pair.Key);

            List<int> toRemove = new List<int>();
            foreach (int key in dict.Keys.OrderByDescending(y => y))
            {
                foreach (int key2 in dict.Keys.OrderByDescending(x=> x))
                {
                    if(!key.Equals(key2) && !toRemove.Contains(key))
                    {
                        if(dict[key2].Contains(key))
                        {
                            dict[key2].AddRange(dict[key]);
                            toRemove.Add(key);
                        }
                    }
                }
            }
            foreach(int key in toRemove)
            {
                dict.Remove(key);
            }

            foreach(int key in dict.Keys.ToList())
            {
                dict[key] = dict[key].Distinct().ToList();
            }

            return dict;
        }
```

After it recognizes a red object it draws a rectangle around it. The method is rather self explanatory and is made purely for visualization purposes. Following snippet presents that.

```
public static void DrawRectangles(Image<Bgr, byte> img, List<int[]> sign_coords, int type = 0)
        {
            unsafe
            {
                int x, y;
                MIplImage m = img.MIplImage;
                byte* dataPtr = (byte*)m.imageData.ToPointer(); // Pointer to the image

                int width = img.Width;
                int height = img.Height;
                int nChan = m.nChannels; // number of channels - 3
                int padding = m.widthStep - m.nChannels * m.width;


                byte[] colors = new byte[3];
                if (type.Equals(0))
                    colors = new byte[]{0 ,0 ,255 };
                else if (type.Equals(1))
                    colors = new byte[] { 255, 0, 0 };
                else if (type.Equals(2))
                    colors = new byte[] { 0, 255, 0 };

                for (y = 0; y < height; y++)
                {
                    for (x = 0; x < width; x++)
                    {
                        foreach(int[] sign in  sign_coords)
                        {
```

```
                    //Drawing rectangle
                    if (
                        x > sign[0] && x < sign[2] && (y == sign[1] || y == sign[3]) || //vertical lines
                        y > sign[1] && y < sign[3] && (x == sign[0] || x == sign[2])    //horizontal lines
                        )
                    {
                        // get pixel address
                        (dataPtr + y * m.widthStep + x * nChan)[0] = colors[0];
                        (dataPtr + y * m.widthStep + x * nChan)[1] = colors[1];
                        (dataPtr + y * m.widthStep + x * nChan)[2] = colors[2];
                    }
                }
            }
        }
    }
}
```

After a part of an image was recognized as an object it is remembered by saving the coordinates of opposite rectangle corners. This variables are later used for specifying what part of picture should be used in order to find sign elements. Using this allows to significantly decrease size of a loop and saves time as most part of input picture does not need to be processed.

3.3 Detection of sign elements

After a specific area is recognized next step is to recognize symbols in it. Again, the connected components method is used but this time with a different threshold parameter because a black object (single digit or other symbol) has to be recognized.

3.4 Verification of digit components

After objects are found they are put through OTSU Thresholding algorithm. The algorithm's purpose is to binarize picture data meaning that the objects should be converted to a black and white image (no grayscale). This makes digits recognition much more efficient.

```
public static void ConvertToBW_coords(Image<Bgr, byte> img, int threshold, int[] sign_coords)
        {
            unsafe
            {
                int x, y;
                MIplImage m = img.MIplImage;
                byte* dataPtr = (byte*)m.imageData.ToPointer(); // Pointer to the image

                int nChan = m.nChannels; // number of channels - 3

                //MAIN MEAN
                for (y = sign_coords[1]; y <= sign_coords[3]; y++)
                {
                    for (x = sign_coords[0]; x < sign_coords[2]; x++)
                    {
                        int blue = (dataPtr + y * m.widthStep + x * nChan)[0];
                        int green = (dataPtr + y * m.widthStep + x * nChan)[1];
                        int red = (dataPtr + y * m.widthStep + x * nChan)[2];
                        int ave = (byte)Math.Round((blue + green + red) / 3.0);

                        if (ave <= threshold)
                        {
                            for (int i = 0; i < 3; i++)
                                (dataPtr + y * m.widthStep + x * nChan)[i] = 0;
                        }
                        else
                        {
                            for (int i = 0; i < 3; i++)
                                (dataPtr + y * m.widthStep + x * nChan)[i] = 255;
                        }
                    }
                }
            }
        }

        public static void ConvertToBW_Otsu_coords(Image<Bgr, byte> img, int[] sign_coords)
        {
            unsafe
            {
                int x, y;
                MIplImage m = img.MIplImage;
                byte* dataPtr = (byte*)m.imageData.ToPointer(); // Pointer to the image

                int width = sign_coords[2] - sign_coords[0];
                int height = sign_coords[3] - sign_coords[1];
                int nChan = m.nChannels; // number of channels - 3
                int[] histogram = new int[256];
                int threshold = 0;

                double weight_bg = 0.0;
                double mean_bg = 0.0;
                double variance_bg = 0.0;
                double weight_fg = 0.0;
                double mean_fg = 0.0;
                double variance_fg = 0.0;
                int hist_elements = 0;
                double class_variance = double.MaxValue;
                for (y = sign_coords[1]; y < sign_coords[3]; y++)
                {
                    for (x = sign_coords[0]; x < sign_coords[2]; x++)
                    {
                        int blue = (dataPtr + y * m.widthStep + x * nChan)[0];
                        int green = (dataPtr + y * m.widthStep + x * nChan)[1];
                        int red = (dataPtr + y * m.widthStep + x * nChan)[2];
                        int ave = (byte)Math.Round((blue + green + red) / 3.0);
                        histogram[ave] += 1;
                    }
                }

                for (int current_threshold = 1; current_threshold < 255; current_threshold++)
                {
                    hist_elements = 0;
                    weight_bg = 0.0;
                    mean_bg = 0.0;
                    variance_bg = 0.0;
                    weight_fg = 0.0;
                    mean_fg = 0.0;
                    variance_fg = 0.0;
                    double current_class_variance = 0.0;

                    //background
                    for (int i = 0; i < current_threshold; i++)
                    {
                        weight_bg += histogram[i];
                        mean_bg += i * histogram[i];
                        hist_elements += histogram[i];
                    }
                    weight_bg /= width * height;
                    mean_bg /= hist_elements;
                    for (int i = 0; i < current_threshold; i++)
                    {
                        variance_bg += Math.Pow((i - mean_bg), 2) * histogram[i];
                    }
                    variance_bg /= hist_elements;
                    hist_elements = 0;

                    //foreground
                    for (int i = current_threshold + 1; i < 256; i++)
                    {
                        weight_fg += histogram[i];
                        mean_fg += i * histogram[i];
                        hist_elements += histogram[i];
                    }
                    weight_fg /= width * height;
                    mean_fg /= hist_elements;
                    for (int i = current_threshold + 1; i < 256; i++)
                    {
                        variance_fg += Math.Pow((i - mean_fg), 2) * histogram[i];
                    }
                    variance_fg /= hist_elements;

                    //result and compare
                    current_class_variance = weight_bg * variance_bg + weight_fg * variance_fg;
                    if (current_class_variance < class_variance)
                    {
                        class_variance = current_class_variance;
                        threshold = current_threshold;
                    }
                }
                ConvertToBW_coords(img, threshold, sign_coords);
            }
        }
```

Now it's needed to recognize a number in a component passed from previous step. Before that it should be mentioned that a set of pictures of digits (0 to 9) was delivered with project requirements. The way objects were verified is that firstly template objects were scaled to be the same size as found object.

Next step is to compare tested objects with template digits. As mentioned before OTSU Thresholding algorithm was used so images are black and white and also they are of the same size (pixel dimensions wise). The way it is done is that each value of pixel in tested image is compared to a corresponding value of pixel in template image. If the values match that template digit's comparison factor is incremented. After the processing is finished a template digit with highest comparison factor is chosen as the proper digit.

```
public static int DetectDigit(Image<Bgr, byte> img, List<Image<Bgr, Byte>> digits, int[] sign_coords, double tolerance = 0.8)
        {
            unsafe
            {
                int x, y;
                MIplImage m = img.MIplImage;
                byte* dataPtr = (byte*)m.imageData.ToPointer(); // Pointer to the image

                int width = sign_coords[2] - sign_coords[0];
                int height = sign_coords[3] - sign_coords[1];
                int area = width * height;
                int nChan = m.nChannels; // number of channels - 3
                int padding = m.widthStep - m.nChannels * m.width;
                int[] digits_similarity = new int[] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
                int maximum_similarity = 0;
                int index_max_similarity = -1;
                double similarity_factor;
                ConvertToBW_Otsu_coords(img, sign_coords);

                for (int i = 0; i < 10; i++)
                {
                    MIplImage m2 = digits[i].MIplImage;
                    byte* dataPtr2 = (byte*)m2.imageData.ToPointer(); // Pointer to the image

                    for (y = 0; y < height; y++)
                    {
                        for (x = 0; x < width; x++)
                        {
                            if ((dataPtr + (y + sign_coords[1]) * m.widthStep + (x + sign_coords[0]) * nChan)[0] == (dataPtr2 + y *
                                        m2.widthStep + x * nChan)[0])
                            {
                                digits_similarity[i]++;
                            }
                        }
                    }
                    if (digits_similarity[i] > maximum_similarity)
                    {
                        index_max_similarity = i;
                        maximum_similarity = digits_similarity[i];
                    }
                }
                //Console.Out.WriteLine(index_max_similarity);
                //Console.Out.WriteLine(digits_similarity);
                similarity_factor = (double)maximum_similarity / area;
                if (similarity_factor > tolerance)
                    return index_max_similarity;
                else
                    return -1;
            }
        }
```

3.5 Classification of signs and recognizing speed limit

To classify a speed limit sign it was implied that a speed limit sign can only contain digit objects. If this is fulfilled a sign will be classified as a speed limit and its value will be assigned by merging two or more digit values together. For example 30 km/h limit is a 3 and a 0 objects in one sign object.

However, if the sign contains different objects (e.g dangerous turn or a perpendicular crossroad) it will be classified as a different alert or restriction sign. The object inside will be compared with template objects created for the purpose of this assignment. This way it could be possible not only to differ between a restriction and a prohibition but also determine specifically what sign that is which could hugely enlarge usages of the solution.

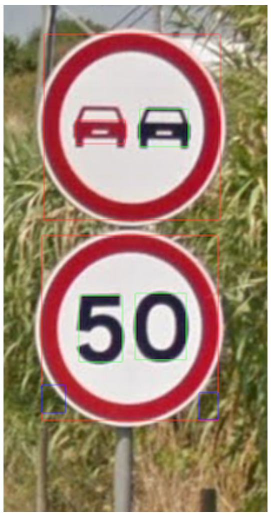
Another functionality that was added was that a program could recognize a triangle. A method for that was implemented.

```
public static int DetectTriangle(Image<Bgr, byte> img, List<Image<Bgr, Byte>> digits, int[] sign_coords, double tolerance = 0.8)
{
    unsafe
    {
        int x, y;

        Image<Bgr, Byte> imgCopy = img.Copy(); // undo backup image - UNDO
        BgrToHsv(img, imgCopy);
        MIplImage m = imgCopy.MIplImage;
        byte* dataPtr = (byte*)m.imageData.ToPointer(); // Pointer to the image

        int width = sign_coords[2] - sign_coords[0];
        int height = sign_coords[3] - sign_coords[1];
        int area = width * height;
```

```
                int nChan = m.nChannels; // number of channels - 3
                int padding = m.widthStep - m.nChannels * m.width;
                int[] digits_similarity = new int[digits.Count()];
                int maximum_similarity = 0;
                int index_max_similarity = -1;
                double similarity_factor;


                int hueLimit = 60;
                int satLimit = 50;
                int valLimit = 30;

                int hue = 0;
                int value = 0;
                int saturation = 0;

                for (int i = 0; i < digits.Count(); i++)
                {
                    MIplImage m2 = digits[i].MIplImage;
                    byte* dataPtr2 = (byte*)m2.imageData.ToPointer(); // Pointer to the image

                    for (y = 0; y < height; y++)
                    {
                        for (x = 0; x < width; x++)
                        {
                            value = ((dataPtr2 + y * m.widthStep + x * nChan)[0] * 100) / 255;
                            saturation = ((dataPtr2 + y * m.widthStep + x * nChan)[1] * 100) / 255;
                            hue = ((dataPtr2 + y * m.widthStep + x * nChan)[2] * 360) / 255;

                            if (((hue >= 0 && hue <= hueLimit) || (hue >= 360 - hueLimit && hue <= 360)) && (saturation <= 100 && saturation > satLimit) && (value <= 100 && value > valLimit))
                            {
                                value = ((dataPtr + y * m.widthStep + x * nChan)[0] * 100) / 255;
                                saturation = ((dataPtr + y * m.widthStep + x * nChan)[1] * 100) / 255;
                                hue = ((dataPtr + y * m.widthStep + x * nChan)[2] * 360) / 255;

                                        if (((hue >= 0 && hue <= hueLimit) || (hue >= 360 - hueLimit && hue <= 360)) && (saturation <= 100 && saturation > satLimit) && (value <= 100 && value >
valLimit))
                                {
                                    digits_similarity[i] += 1;
                                }
                            }
                        }
                    }
                    if (digits_similarity[i] > maximum_similarity)
                    {
                        index_max_similarity = i;
                        maximum_similarity = digits_similarity[i];
                    }
                }
                //Console.Out.WriteLine(index_max_similarity);
                //Console.Out.WriteLine(digits_similarity);
                similarity_factor = (double)maximum_similarity / area;
                if (similarity_factor > tolerance)
                    return index_max_similarity;
                else
                    return -1;
            }
        }


        public static List<string[]> CreateFinalList(List<int> classification, List<int[]> signsObjects, List<int[]> numberObjects)
        {
            unsafe
            {
                List<string[]> signs = new List<string[]>();
                if (!signsObjects.Count().Equals(0))
                {
                    List<int[]> foundedDigits = new List<int[]>();
                    string sign_value = "";
                    int index = 0;

                    //Sorting digits founded inside sign
                    foreach (int classifier in classification)
                    {
                        if (classifier >= 0)
                        {
                            foundedDigits.Add(new int[] { classifier, numberObjects[index][0], numberObjects[index][1], numberObjects[index][2], numberObjects[index][3] });
                        }
                        index += 1;
                    }

                    //Merging sorted digits as one number
                    foundedDigits = foundedDigits.OrderBy(x => x[1]).ToList();
                    foreach (int[] number in foundedDigits)
                    {
                        sign_value += number[0].ToString();
                    }

                    //Adding founded signs to list
                    foreach (int[] sign in signsObjects)
                    {
                        string[] dummy_vector = new string[5];

                        //Checking digits position in order to sign position - if digit is inside sign then sign is speed limit type
                                if ((!foundedDigits.Count().Equals(0)) && (sign[0] - foundedDigits[0][1] < 0 && sign[1] - foundedDigits[0][2] < 0 && sign[2] - foundedDigits[0][3] > 0 && sign[3] -
foundedDigits[0][4] > 0))
                            dummy_vector[0] = sign_value;   // Speed limit
                        else
                            dummy_vector[0] = "-1";     // Another sign


                        dummy_vector[1] = sign[0].ToString(); // Left-x
                        dummy_vector[2] = sign[1].ToString();  // Top-y
                        dummy_vector[3] = sign[2].ToString(); // Right-x
                        dummy_vector[4] = sign[3].ToString();  // Bottom-y
                        signs.Add(dummy_vector);
                    }

                }
                return signs;

            }

        }

    }
```

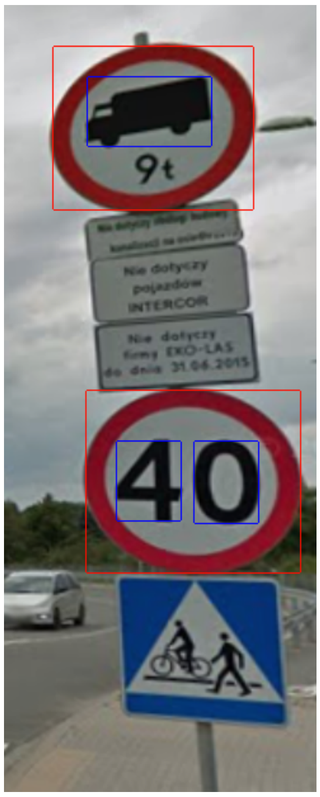
4. Results

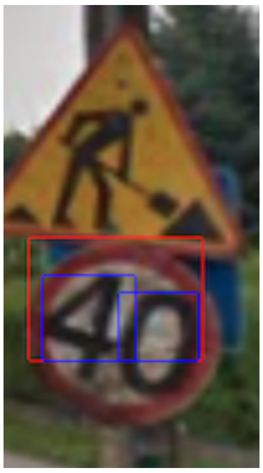
The program managed to recognize a sign on all supplied images. Speed limit was defined properly on most of them and a value of restriction was returned. If a sign was not a speed limit it was classified as a different sign without differing whether it was a hazard or restriction. An example from the original database below.



As it can be seen on the example above the two signs were detected. On the upper one a car symbol is detected so it proves that the connected component method works. On the lower one both values were verified and the program returned a proper speed limit.

Also it should be mentioned that delivered database was widened to check how the program works in harder conditions. Some images were put there in order just to interfere with the program. Different situations were used. Some of them were big angle of view, poor quality of picture or even poor quality of signs. Examples below.





The surprisingly big amount of extra photos from second positively passed the test and confirmed that the implementation of OTSU was a good idea.

A different approach was discussed along with the results. If there was a database of not only digits but also other symbols (such as a heavy vehicle or a t letter in the first picture), the program could expend its field of use and detect specific signs. Some steps were made to check this solution but a no valuable results were obtained. The reason for that was probably the quality of samples used a template models.