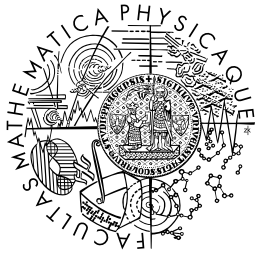


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Václav Ryšlink

Video analysis: an automatic time measurement in the robotic car competition

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Marta Vomlelová Ph.D.
Study programme: Computer Science
Study branch: General Computer Science

Prague 2020

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Rád bych poděkoval Mgr. Martě Vomlelové Ph.D. za vedení této práce, pomoc při uspořádání robotických závodů i připomínky k předkládanému textu. Mé velké díky patří i celé mé rodině, která mě po celou dobu mého studia ohromně podporuje a dodává potřebnou energii vždy, když je to potřeba.

Title: Video analysis: an automatic time measurement in the robotic car competition

Author: Václav Ryšlink

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Marta Vomlelová Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Our main goal was to design an algorithm that would automatically evaluate robotic races from the video of the track's finish section. To solve this problem, we used various image processing methods and consequently proposed two different solutions that differ both in the expected input and the inner logic. The first algorithm can evaluate races of cars of an arbitrary appearance since it recognizes cars based on their reference photos. Although this solution proved to be working in all of our experimental recordings, we are aware of a few situations in which this algorithm could be prone to make mistakes. Therefore, we also came up with another algorithm that works more reliably in exchange for demanding cars to have unique color labels.

Keywords: Image processing, Video analysis, Object tracking, Robotic races

Contents

1. Introduction

Robotic races belong to traditional disciplines of many robotic competitions held all around the world. In the Czech Republic, one of the most significant robotic events is Robotický den, which has its version of robotic races called RoboCarts. These races took place on a circular racetrack where the robots' goal is to complete the stated number of laps in the shortest time possible. Up until now, RoboCarts have been measured manually on a stopwatch by a referee standing next to the track. Because such an approach can be first inaccurate and second complicated for the referees, it was suggested to develop an application that would be able to count and measure times of each driven lap to every car in the race from a recording of the track's finish section.

Therefore, this thesis aims to develop a practical algorithm that would analyze such video recordings and be able to evaluate the races in real-time. The target machine parameters were not specified. So our goal was also to create an algorithm that would execute in real-time without the excessive CPU power and the necessity of the GPU presence. For the implementation itself, it was recommended to use the standard image processing methods and the computer vision library OpenCV.

In the following sections, we will first introduce the rules and course of RoboCarts races (Section 2). Next, we will describe several image processing methods with a focus on moving object tracking, which would constitute an important part of our solution (Section 3). And finally, we will present our two algorithms for automatic race evaluation (see Figure 1.1) and their accuracy on our training data in sections 4 and 5. The last section 6 will then, in particular, discuss our choice of the programming language and the structure and performance of both our algorithms.

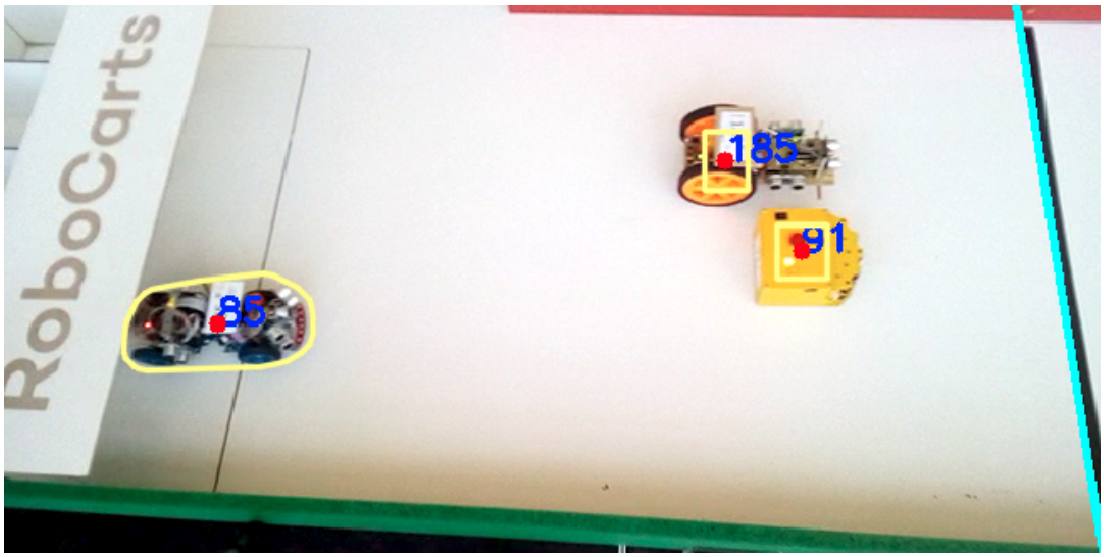


Figure 1.1: Visualization of our algorithm applied on a recorded race from Robotický den 2019.

2. Robotic races

Robotický den [1] is the biggest event of amateur robots in the Czech Republic to promote and present robotics to the public. Within this event, participants can compete with their prepared robots in various disciplines challenging robots' ability to orient and interact with the environment. One of its traditional contests is a competition called RoboCarts [2], in which autonomous robots compete in a simulated race.

2.1 RoboCarts rules and specifications

In each RoboCarts race, there can participate up to five cars that aim to complete several rounds as fast as possible (referees announce the precise number of target laps before the races). After the race ends, robots obtain points based on the number of completed rounds and the order in the finish. Robots with the most points qualify for the final race, where the absolute winner is determined.

As for robot construction, participants are restricted by the maximum size of $20 \times 10 \times 10$ cm. Every car should also have a 5×5 cm space for the sticker marking on the top. Other aspects of construction, such as the number of wheels, car color, and shape, are not specified.

Races take place on a white square racetrack (280×280 cm) bounded by green outer barriers. The circuit is formed by three red inner walls in the shape of the letter 'H' in the central part and two green inner walls perpendicularly connected to the middle of the bottom and upper outer barriers. All barriers are firmly fixed and at least 10 cm high. The finish line is black, approx. 1.5 – 2 cm wide and located in the right part of the track (see Figure 2.1).

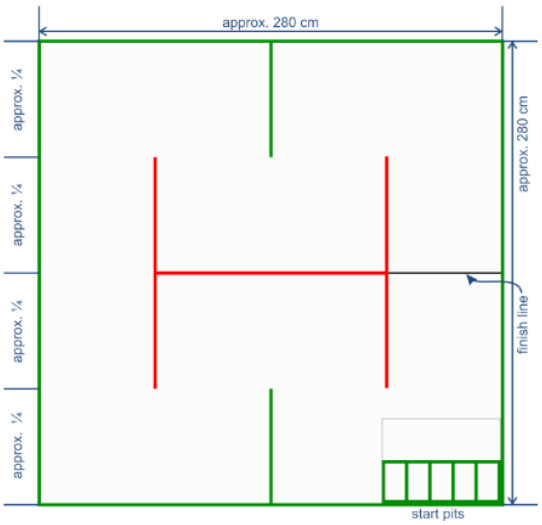


Figure 2.1: The official format of the RoboCarts' track and the starting box.

2.2 Course of the race

At the beginning of each race, the referee places a starting box on the track with its front side facing the finish line. Participants then put their cars one after another into separated pits of the starting box.

When all cars are ready to go, a referee manually opens the front side of the box and starts the race. The time for each car is being measured after its first finish line crossing. When all cars leave the pits, the starting box is manually removed. To complete a lap, cars need to drive through the whole circuit and cross the finish line, not just touch it. During the race, it is not allowed to remove non-moving cars or otherwise help to non-working robots.

The race ends after all cars finish the specified number of rounds or when the time limit of the race is reached. In the end, the referee announces the results and writes down the times.

3. Image processing methods

To make the later description of our algorithms more understandable, we will first describe some standard image processing terms and methods and introduce the problem of moving object tracking in a video.

3.1 Color spaces

In computer graphics, images can be displayed using various color representations. For humans, the most natural one is the RGB representation as it copies the way how the colors are processed in our eyes – like a mixture of red, green, and blue color. However, for some applications, specifying a color as a mixture of three other colors may be inconvenient.

HSV (hue, saturation, value) and HSL (hue, saturation, lightness) are on the other hand color formats in which the color is described only by the first channel while the other ones determine the tones and shades. Therefore, expressing a particular color and its similar variants is much more intuitive and better described compared to the RGB format. For that reason, HSV and HSL color schemes are preferred for specifying colors in various image segmentation tasks.

Regardless of the application, the vast majority of image processing algorithms at some stage converts images to the *grayscale* representation, where the pixels have only one dimension of 256 values. Thus, it is favored in situations when the computations with multiple channels would be expensive. A special type of grayscale color scheme is the *binary* representation, where all pixels are either white or black.

3.2 Image noise filtering

An image never captures the objects' original appearance because the *image noise* always corrupts it. The image noise is a term for random variations of pixel values in an image mostly caused by the heat, electricity, and illumination in the camera sensors. Because these random pixels may represent a problem for the image processing algorithms, it is common to reduce the image noise with blurring techniques during the preprocessing stage.

Generally, the blurring techniques [3] are based on the linear convolution of an image with an $n \times n$ convolutional kernel[1]. The kernel is then consecutively slid over every image pixel to transform the pixel value according to a convolution function applied on the adjacent pixels underneath the kernel.

Probably the most often used technique for blurring is the *Gaussian blur*, named after the *Gaussian function*

$$g(x, y) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{d^2}{2\sigma^2}} \tag{3.1}$$

where $d = \sqrt{(x - x_c)^2 + (y - y_c)^2}$ is the distance from the center pixel (x_c, y_c).

[1] n is usually odd and significantly smaller than the dimensions of an image that is blurred.

Gaussian blur reduces the image noise by smoothing the image but in a way that most of the original edges are still preserved [4].

3.3 Thresholding

Thresholding [5] is a basic image segmentation technique based on comparing values of image pixels to a threshold value T. Pixels with an intensity surpassing T are converted to white pixels (value one), and the rest of the image is turned to black (value zero). Traditionally, the thresholding is applied to grayscale images, but it is possible to use images in RGB or HSV formats as well. The threshold value just needs to be specified for every dimension separately. For specific image segmentation tasks, it is also very common to threshold images with two threshold values – the lower and the upper value, and masking out all the pixels that do not fit into the range.

3.4 Morphological image processing

Morphological image processing [6] is a set of non-linear image operations that are used to alter the shape of objects in images. They have proven to be particularly useful for noise removal and image enhancement during image segmentation stages. As an input, morphological operations require a binary image[2], and a binary template called the *structuring element*. The structuring element is then consecutively positioned over the input image, at every possible location it fits, to transform the underlying central pixel according to its neighborhood.

Depending on the type of operation, the central pixel is transformed based on whether the structuring element fits into the objects or whether it at least hits them. In other words, the central pixel turns white if all, or at least some of its adjacent pixels under the template, are also white. As such, we recognize two fundamental morphological operations: *erosion* and *dilation*.

3.4.1 Erosion

When eroding a binary image B with a structuring element s, the operation is denoted as $B' = B \ominus s$. The pixel $B'(x, y) = 1$ only if the structuring element centered on the pixel $B(x, y)$ completely fits the white pixels in the input image. Thus, erosion removes random noise and strips away objects' boundary pixels that do not fit the structuring element (see the middle picture in Figure 3.1).

3.4.2 Dilation

Similarly to the previous case, we denote the dilation of a binary image B with a structuring element s as $B' = B \oplus s$. In the new binary image B', the pixel $B'(x, y)$ equals to one, only if there is at least one white pixel under the structuring element s positioned over the pixel $B(x, y)$. All the foreground objects of B' are therefore expanded and their inner holes filled (see the right picture in Figure 3.1).

[2]With some modifications even grayscale images can be used.

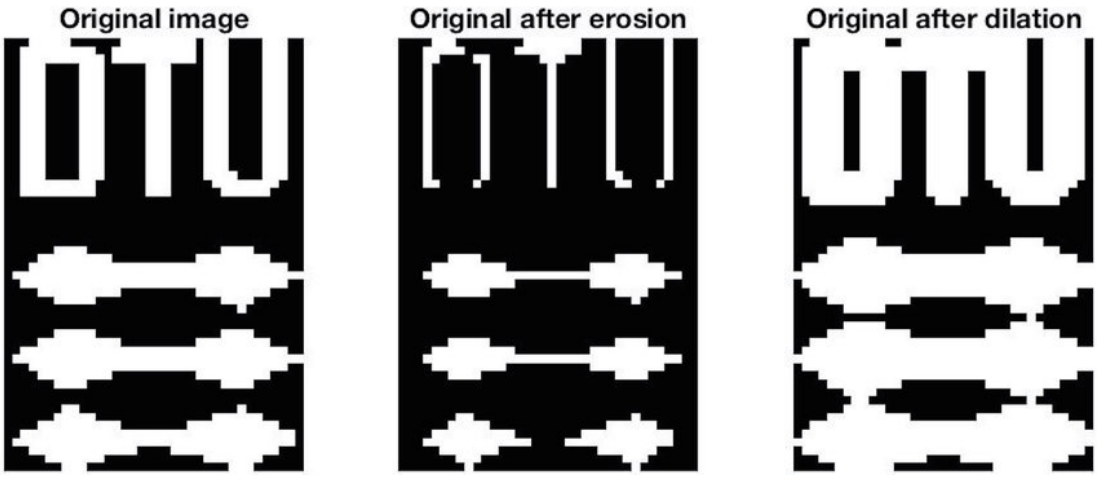


Figure 3.1: The effect of morphological erosion and dilation on a binary image [7].

3.4.3 Compound morphological operations

Opening and *closing* are compound morphological operations created by applying dilation after erosion and erosion after dilation with the identical structuring element. Compared to the fundamental operations, opening and closing are less destructive to the original objects' appearance.

The opening operation is similar to erosion because it removes smaller foreground objects from the image. But while the erosion also shrinks the objects, the opening only alters objects' boundaries in a way that the structuring element can be fitted inside them.

Closing is used to fill the inner holes of objects without extending the original size. It is because the later erosion removes all the new pixels added by the first dilation. Since the last operation of closing is erosion, the background's boundaries are shaped according to the structuring element.

3.5 Edge detection

Edges are important image features because they provide us with information about the outline and structure of captured objects. From our point of view, edges represent significant local changes in the pixels' intensity. But since an image can be viewed as an array of a sampled continuous function, these changes in pixel intensity can be more suitably defined as the image *gradient*. For an image I, we can define the gradient as a vector of partial derivatives representing gradients in the vertical and horizontal directions.

$$\nabla I = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{bmatrix} \tag{3.2}$$

Once the gradients in both directions are known, it is possible to compute the gradients magnitude G as

$$G = \sqrt{G_x^2 + G_y^2} \tag{3.3}$$

and the gradient's direction Θ as

$$\Theta = \arctan\left(\frac{G_y}{G_x}\right) \tag{3.4}$$

In practice, the gradients in both directions are approximated using various operators. One of the most frequently used operators is the *Sobel operator*, which computes the gradients of an image I with a 3×3 convolutions in the following manner [3].

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \times I \qquad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \times I \tag{3.5}$$

3.5.1 Canny edge detection algorithm

Despite being published in 1986, the Canny edge detection algorithm [8] is regarded as the optimal edge detector. It is built on three main principles: keep a low error rate, precisely localize edge points and give only one response to a single edge. The algorithm itself consists of four following stages.

Firstly, an input image is smoothed using Gaussian filters (Section 3.2) to eliminate the image noise.

Secondly, gradients in horizontal (G_x) and vertical (G_y) directions are computed using a convolution with the Sobel operator (3.5). The overall magnitude of an edge and its direction is then computed, as it was shown in equations 3.3 and 3.4. This representation already gives us a good notion of image edges. However, the edges can appear to be quite blurry.

Thirdly, to thin the blurry edges, a *non-maximum suppression* is used. In this step, every pixel is scanned and checked whether it is the local maximum in the gradient direction – the direction perpendicular to the edge. Pixels in an edge that are local maximums are then transformed into white pixels, and the rest of the pixels are turned to black pixels.

And lastly, the Canny edge detection algorithm ends with a process of *hysteresis thresholding*. While in the previous stage, the edges were made sharper and thinner, there is still a possibility that some of them originated from the image noise. To remove those edges, two threshold values – minimal and maximal, are used. Then, all pixels with gradient magnitude larger than the maximal threshold are kept as confirmed edges, and all the pixels with gradient magnitude smaller than the minimum threshold are discarded. Those pixels with gradient intensity lying in between the two thresholds are preserved only if they are connected to a confirmed edge. The output of this stage and the whole algorithm is displayed in Figure 3.2.

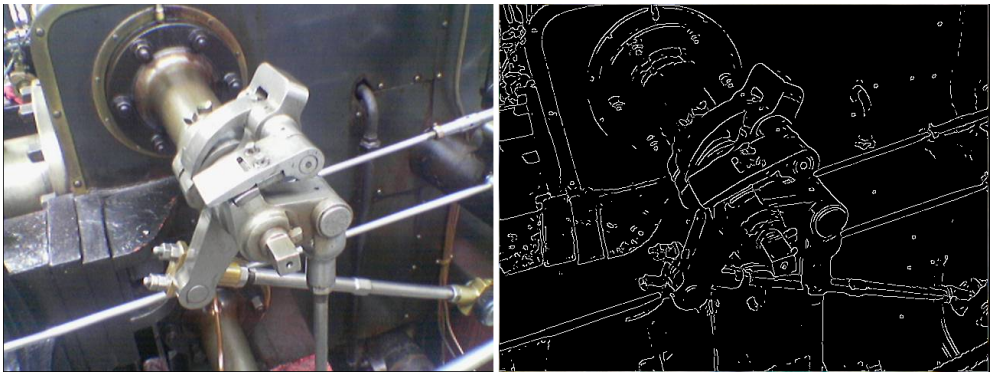


Figure 3.2: Edges detected by Canny Edge algorithm [9].

3.5.2 Hough line transform

Hough line transform [10] is a technique for detecting straight lines in images. Generally, all lines in a plane can be represented in the form

$$y = mx + c \tag{3.6}$$

where m corresponds to the line slope and c to the y-intercept. Nonetheless, this representation is not convenient for vertical lines as the value of the slope would reach infinity.

Thus, for the line detection in images, it is preferable to express the lines in the polar coordinate system as

$$y = (-\frac{\cos\theta}{\sin\theta})x + (\frac{r}{\sin\theta}) \tag{3.7}$$

with r denoting the arc length and θ denoting the angle of a line with the x-axis. By arranging the previous formula into the form

$$r_\theta = x_0 \cos\theta + y_0 \sin\theta \tag{3.8}$$

it is possible to define the family of all lines[3] (r_θ, θ) going through a point (x_0, y_0). Let us consider only those lines where $r > 0$ and $0 < \theta < 2\pi$. If we then plot all the lines passing through a point (x_0, y_0) in the Hough space, the corresponding graph of (r_θ, θ) pairs will be a sinusoid. Furthermore, by plotting the sinusoids of more points into the Hough space, we can search for their common intersection and find out which line goes through all of them (see Figure 3.3).

The Hough Line Transform algorithm, in a similar manner, iterates through all the edges in the image, plots all the lines passing through them into the Hough space, and searches for the intersections. If a point (r_θ, θ) is intersected by more than some threshold number of minimal sinusoids, the corresponding straight line is declared to be present in the image [12].

[3]The two dimensional space of (r_θ, θ) pairs is called the *Hough space*.

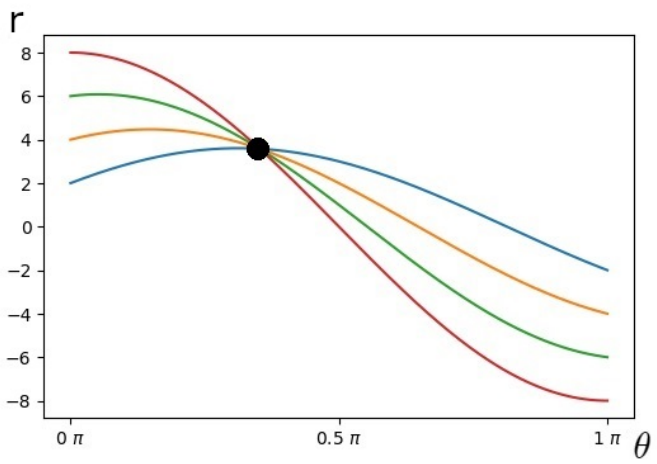


Figure 3.3: Sinusoids of four different (x, y) pairs intersecting in a single point [11].

3.6 Tracking of moving objects

Tracking of moving objects [13] is a standard computer vision discipline typically used for video surveillance [14] or vehicle counting systems [15]. Traditionally, it comprises two main stages: detecting moving objects in individual frames and tracking those detections across the consecutive video frames. For our purposes, it will be convenient to include here also a third stage – the identification of detected objects.

To not confuse the whole process of tracking with its second stage, we will refer to the tracking stage as an assignment stage[4]. It should also be mentioned that for all the following techniques, we will assume that the analyzed video was captured by a static camera.

3.6.1 Detecting objects by movement

When detecting a movement, we aim to find the coherent regions of pixels that correspond to the moving objects and separate them from the background of each frame.

In this section, we will focus on detection techniques based on the principle of image subtraction. These techniques are based on a simple observation that the movement in video frames corresponds to changes in pixel color and intensity. Despite the simplicity, these techniques have universal usage and work very efficiently.

Image subtraction

As the name suggests, the image subtraction is carried out by subtracting the values of corresponding pixels in the two input images. It is possible to subtract images in arbitrary color formats, but the most convenient format to work with

[4]Because its purpose is to assign detections from the current frame to the ones in the previous frame.

is the grayscale representation. Because many color schemes do not support negative values, the absolute value of the difference usually needs to be taken.

Once a proper image difference is obtained, it is then thresholded (Section 3.3) with a threshold value usually set to 40 % of the intensity range [16]. In the resulting binary image, white pixel regions are regarded as moving objects and the black pixels as the static background [13].

Background subtraction

To apply background subtraction (BGS), it is first necessary to capture the scene without any objects. This reference image will then represent the background that will be consecutively subtracted from all the frames in the video. With this approach, any deviation from the original background is recorded, and so both static and moving objects are detected.

Frame differencing

Compared to the BGS technique, frame differencing (FD) uses the previous frame as the reference background. Because FD regularly updates the reference image, it does not suffer from sudden background changes and camera movements, which can negatively influence the BGS method.

On the other hand, FD can detect only those objects whose positions did change in successive frames. Moreover, it may have a problem detecting objects with smooth and monochromatic surfaces as their movement is locally not well observable.

3.6.2 Detecting object by color

Apart from the movement, particular objects can also be recognized for other properties, such as shape, texture, or color. Nevertheless, for fast-moving objects that can even easily overlap, the most promising feature to detect is the color.

In relatively constant light conditions, color is a very reliable identification feature. For color detection, we will use a technique called *color filtering*. To filter a particular color, we need first to specify its lower and upper boundary. The most convenient color formats to work with are HSV or HSL formats (Section 3.1) since only their first component determines the color. After the color boundaries are set, we need to ensure that the source image has the same color format. And lastly, we threshold (Section 3.3) the source image with the two color boundaries and obtain a binary mask, where all pixels outside the color range are masked out.

3.6.3 Assignment of detections

This stage aims to find the correspondence between the detections found in the previous frame and the detections from the current frame. For that purpose, we will utilize the *Hungarian algorithm*.

Hungarian algorithm

The Hungarian algorithm [17] is a combinatorial optimization algorithm used for solving assignment problems. Let us define two sets F and D of the same cardinality[5] n, and an assignment function $X : i \to j,\ i \in F,\ j \in D$. Let also $c_{ij} \in \mathbb{R}$ be a cost for assigning an element $j \in D$ to $i \in F$. The goal of an assignment problem is to find an assignment function that minimizes the following cost function

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \tag{3.9}$$

subject to constraints that each element must be paired with exactly one another element

$$\sum_{i=1}^{n} x_{ij} = 1\ \forall j = 1, 2, ..., n \tag{3.10}$$

$$\sum_{j=1}^{n} x_{ij} = 1\ \forall i = 1, 2, ..., n \tag{3.11}$$

where

$$x_{ij} = \begin{cases} 1, & \text{if } j \in D \text{ is assigned to } i \in F \\ 0, & \text{otherwise} \end{cases} \tag{3.12}$$

The Hungarian algorithm takes as an input the cost matrix C and returns the assignment of the minimal total cost in the polynomial time. For the maximal assignment, it is sufficient to negate all values in C. The precise descriptions of all Hungarian algorithm's stages can be found in [17].

3.6.4 Identification of detections

Generally, two images can be compared in many different ways and using various similarity metrics. The choice of the technique should, therefore, mainly depend on the type of images and expected quality of the comparison – some of the techniques may work better in exchange for longer execution time. For our application, the two following techniques achieved great results while still being able to run in real-time.

Histogram similarity

An image histogram is a type of histogram that expresses the tonal distribution of colors in an image. For a grayscale image, the histogram will be defined as a bar chart with 256 bins representing every possible intensity value. Similarly, a histogram for a color image (e.g., in RGB format) will be depicted by the combination of histograms from each channel.

[5] If one of the set has bigger cardinality, it would be necessary to adjust the conditions to allow some elements from the larger set to be unassigned.

There are several formulas for computing the histogram's similarity [18]. One of the simple methods to calculate the similarity of two histograms H_1, H_2 is to compute their *intersection* as

$$d(H_1, H_2) = \sum_b \min(H_1(b), H_2(b)) \tag{3.13}$$

where the summation goes over all bins/intensities b of the particular representation. The intersection methods is very fast to compute but because of the mechanics it does not tell much about concrete distributional differences. A more complex methods, such as the histograms' *correlation* (3.14), attempt to improve that by increasing the computational complexity of the comparison [19].

$$d(H_1, H_2) = \frac{\sum_I (H_1(b) - \bar{H_1})(H_2(b) - \bar{H_2})}{\sqrt{\sum_I (H_1(b) - \bar{H_1})^2 \sum_I (H_2(b) - \bar{H_2})^2}} \tag{3.14}$$

$$\bar{H_k} = \frac{1}{b} \sum_J H_k(b) \tag{3.15}$$

Template matching

Template matching [20] is a technique for finding the best matching location of a template T in a larger source image I. It consecutively slides the template over the source image and compares the similarity with the source image according to a particular metric. The comparison score is then saved in the result matrix R in a way that value at $R(x, y)$ corresponds to the similarity score of a template placed with its top-left corner on the pixel $I(x, y)$. The overall best matching area is finally found by locating the minimum (or for some metrics the maximum) value in R.

Now, we will list four commonly used comparison metrics. The most basic one is the *Sum of absolute differences* (SAD) (3.16). It only takes the absolute difference of pixels of the template and the underlying source image area and finds the region with the smallest sum of differences. From the vector point of view, it can be regarded as the L1 norm.

$$R(x, y) = \sum_{x', y' \in T} |T(x', y') - I(x + x', y + y')| \tag{3.16}$$

It is well known that in some applications the L1 norm suffers from its inability to penalize bigger differences of individual components. The same applies for template matching and it can be fixed by using a method analogical to L2 norm – the *Sum of square differences* (SSD) (3.17).

$$R(x, y) = \sum_{x', y' \in T} (T(x', y') - I(x + x', y + y'))^2 \tag{3.17}$$

While the previous two methods search for the position with the minimal score, the best matching found by the *Cross-correlation* (CC) (3.18) has the biggest score. In CC, the score equals to the sum of the corresponding pixel values product. This has, however, an important disadvantage because, for an

arbitrary template, the highest score will correspond to the brightest area in the source image.

$$R(x,y) = \sum_{x',y' \in T} (T(x',y') \cdot I(x+x', y+y')) \tag{3.18}$$

For that reason, instead of vanilla CC, it is common to use *Mean shifted cross-correlation* (MSCC) (3.19). Compared to CC, it solves the problem of bright patches by subtracting an average pixel value from pixels in the template and the source image

$$R(x,y) = \sum_{x',y' \in T} (T'(x',y') \cdot I'(x+x', y+y')) \tag{3.19}$$

$$T'(x',y') = T(x',y') - \tfrac{1}{w \cdot h} \cdot \sum_{x'',y'' \in T} T(x'',y'') \tag{3.20}$$

$$I'(x+x', y+y') = I(x+x', y+y') - \tfrac{1}{w \cdot h} \cdot \sum_{x'',y'' \in T} I(x+x'', y+y'')] \tag{3.21}$$

where w and h correspond to template's (3.20) and source image's (3.21) width and height.

Normalization of template matching metrics

All discussed metrics can be further improved by the normalization which is especially useful when we compare the match scores from several different templates. The problem is that the scores can be skewed because of the difference in templates' sizes and intensity variances. Therefore, it is a common practice to normalize scores in the following way

$$R_{norm}(x,y) = \frac{R(x,y)}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x', y+y')^2}} \tag{3.22}$$

Or in case of MSCC

$$R_{norm}(x,y) = \frac{R(x,y)}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x', y+y')^2}} \tag{3.23}$$

When it comes to choosing the metric, there is not a definitive guideline to follow. The most frequently used are normed or standard versions of SSD and MSCC. Trucco and Verri [21], in their book, argue that the SSD metric should generally work better. However, other people say that SSD works better only if the templates match precisely and for the general real-world setting recommend MSCC [22].

4. Algorithms for race evaluation

In this section, we will describe our solution for automatic evaluation of robotic races which can be divided into three separate parts – Finish line location (Section 4.1), Tracking algorithms (Sections 4.2 and 4.3) and Finish logic (Section 4.4).

Initially, we planned to present only one tracking algorithm, referred to as the General tracking algorithm (GTA). However, after we collected more training recordings and carried out several experiments, it turned out that there are few situations in which the GTA does not tend to behave entirely predictably (Section 5.3). To get around this drawback, we developed another tracking algorithm, the Color tracking algorithm (CTA), that works precisely even in those problematic situations at the cost of demanding cars to have unique color labels.

4.1 Finish line location

Because the camera is fixed, we can assume that the finish line's position does not change during the race. Therefore, it is sufficient to locate the finish line only in the first frame of the video and then use the same location for all the consecutive frames.

We know that the finish line is always black, but to find a particular line based only on its color would be reasonably challenging. Nevertheless, we have an advantage that we know precisely the environment we are going to be searching in. Since the camera is placed next to the track, the finish line is the rightmost vertical line in each frame. The finish line is vertical because it is perpendicular to the track border under the camera. And it is rightmost because next to the finish line there is only the plain white track without any structure. If the camera were placed high enough, it would probably be possible to cover the outer wall right to the finish line as well. This setting would be, however, firstly hard to achieve and secondly would make the cars smaller, hence less recognizable.

With this observation in mind, the location of the finish line is straightforward. First, the frame needs to be converted to the grayscale format. In the grayscale image, the Canny edge algorithm (Section 3.5.1) detects all edges, in which the Hough Line Transform algorithm (Section 3.5.2) finds all lines. Found lines are automatically prolonged across the whole frame.

To determine whether a line is vertical, we use a simple heuristic. Because we have all lines represented by their endpoints, we can consider a line to be vertical if its endpoints differ more in their y coordinates than in their x coordinates. More precisely, a line l represented by two points (x_1, y_1) and (x_2, y_2) is considered vertical if $|x_2 - x_1| < |y_2 - y_1|$. This approach is not an optimal one, but for our application, it serves sufficiently.

Finally, to identify the rightmost line, we iterate through the endpoints of all vertical lines and find the one with the largest x-coordinate. A line corresponding to this endpoint is proclaimed as the finish line, and its endpoints are saved. The step-by-step process of finish line detection is depicted in Figure 4.1.

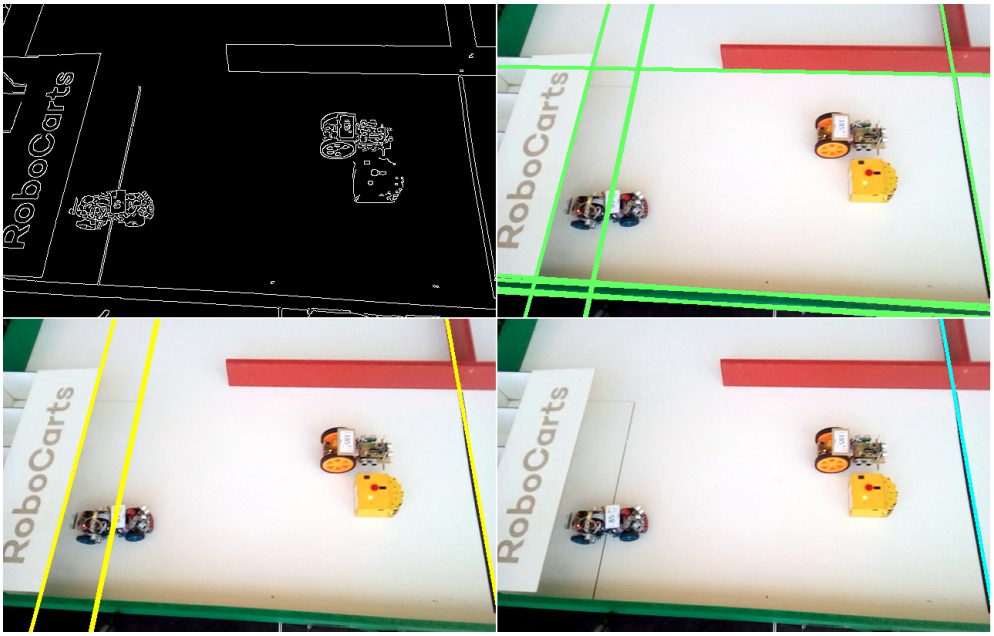


Figure 4.1: Steps of the finish line location: edge detection (top-left), straight line detection (top-right), vertical line detection (bottom-left) and the rightmost line detection (bottom-right).

4.2 General tracking algorithm

We gave this algorithm attribute 'general' because it does not require cars to have any specific appearance. It is because this algorithm identifies cars based on the reference photos that we need to provide beforehand. Further usage specifications are included in the User documentation (Section A.1.3).

4.2.1 Detection of moving objects

The very first step in the tracking algorithm's pipeline is to detect moving objects. Because both background subtraction (BGS) (Section 3.6.1) and frame differencing (FD) (Section 3.6.1) have their advantages and disadvantages, we decided to keep both approaches available and let the user choose what algorithm should be used. But, no matter the method, we always need a reference frame for every current frame – the background image for BGS and the previous frame for FD. We then transfer both frames to their grayscale format and subtract them (Section 3.6.1).

The pixel intensities in the grayscale difference image now correspond to how much the two original photos differed – white pixels mean an absolute difference, and black pixels mean no difference at all. Based on that, we perform binary thresholding (Section 3.3) and proclaim all pixels with intensity less or equal to the threshold value (40% of the range) as static and set their value to zero. The rest of the pixels with intensity surpassing the threshold value are set to one and are considered as regions where a movement could occur (Section 3.6.1).

4.2.2 Morphological operations and convex hulls

With the binary masks obtained from the previous stage, we could already proceed to the assignment stage. But because up to this point, we have been approximating the movement as a change in pixels' color and intensity, some of the detections may be visibly distorted – they can have irregular contours, contain holes or, in some cases, be even discontinuous.

To solve this problem, we use morphological opening and closing operations (Section 3.4), which smoothen the found objects and remove the present noise. Because cars tend to have an oval shape, all morphological operations use an ellipsoidal kernel. The size of kernels should always be set according to the camera and video settings. In our experiments, the $(27, 27)$ kernels achieved good results while not slowing down the algorithm. To smooth out the objects more, we transform the contours into convex hulls. The final effect of both operations can be seen in the following Figure 4.2.

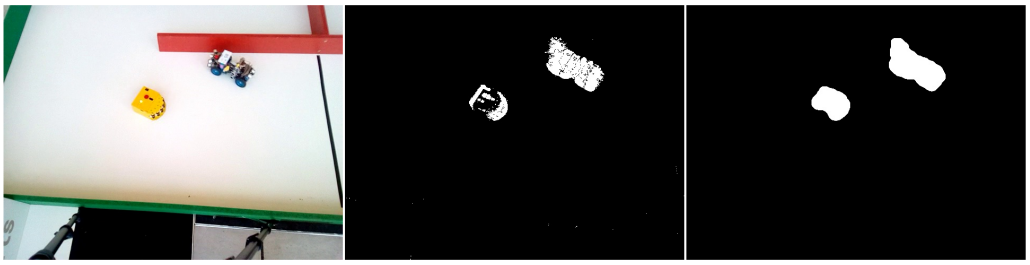


Figure 4.2: Effect of morphological operations and convex hulls on detected objects. Original figure (left), frame difference of two consecutive frames (middle), morphological opening and closing applied on the frame difference (right).

4.2.3 Further filtering

At this moment, we ought to have accurate detections of actual moving objects. However, we still can not be sure that all the detections correspond to actual cars. Many other things could appear on the video – an arm reaching for a car (see Figure 4.3), a shadow, or objects put in the proximity to the track. Another source of unwanted detections is also the random camera movement that may make certain regions of the scene seemingly move.

To speed up the algorithm, we want to filter out these false detections, so they are not further processed. But by the same logic, we also do not want to spend much time filtering them. Therefore, we decided to filter the detections based on their width and height proportion – keeping only those detections with the aspect ratio[1] in the range from $\frac{1}{3}$ to 3. We can perform this kind of filtering only because the race rules restrict the car proportions (Section 2.1). With this filtering, we are certainly not able to filter out all false detections, but we should be able to eliminate most of them. The exact value of the aspect ratio range was found by experimenting.

[1] Aspect ratio of an image is a ratio of its width to its height.

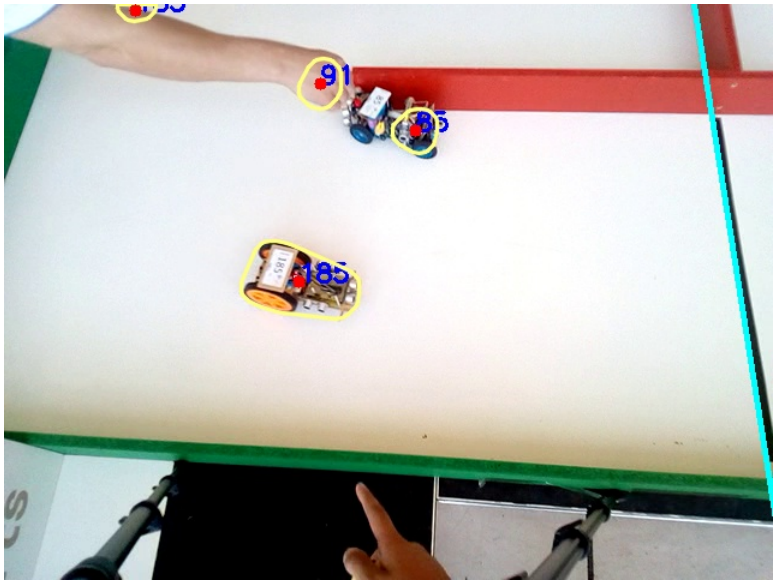


Figure 4.3: A hand reaching to the track identified as one of the cars. The other hand at the bottom of the frame successfully filtered out and not registered.

4.2.4 Detection assignment

Now, we are in a situation when we have two sets of detections – the ones from the current frame and the ones from the previous frame, and we intend to pair them up. Because we can safely assume that the corresponding detections will have a similar location in both frames (see Figure 4.4), it is possible to measure the detections correspondence using the *Intersection over Union* (IoU) metric – the ratio of overlap and union area of two objects.

So for each detection from the current frame, we compute the IoU with every detection from the previous frame. Then we create a cost matrix C in a way that a $C(i, j)$ element will represent a negative value[2] of IoU of the i-the detection from the current frame with the j-th detection from the previous frame. In this form, we pass the matrix to the Hungarian algorithm (Section 3.6.3) to get an optimal assignment – an assignment with the maximal sum of IoUs. To ensure we do not pair up not corresponding detections, only pairs with IoU greater than 0.01 are allowed.

However, we still need to remember that some of the current detections represent objects that were not present in the previous frame and similarly that some of the moving objects from the previous frame could disappear from the scene. If there are more detections in the current frame than in the previous one, the not paired up detections are regarded as new moving objects in the scene. On the other hand, if some previous detections were not paired up, they are not discarded immediately, but only after they are not paired up in five consecutive frames. This measure helps especially when detecting moving objects with the frame differencing technique (Section 3.6.1) where the cars can get lost on some frames more quickly.

[2]We need to enter the negative values because the Hungarian algorithm searches for the minimal sum assignment and we want the maximal assignment.

Figure 4.4: Example of contours detected in several consecutive frames.

4.2.5 Identification of detections

Before the race started, we took a photo of every participating car. At this time, we will use these photos to identify the detections. This task can also be described as a classification problem because we classify each detection d with the most similar car c. We will measure the similarity score S as a manually optimized weighted sum of two normalized scores obtained from the histogram comparison (HC) (Section 3.6.4) and template matching (TM) (Section 3.6.4) techniques in the following way.

$$S(d,c) = 0.85 \times \mathrm{TM}_{norm}(d,c) + 0.15 \times \mathrm{HC}_{norm}(d,c) \tag{4.1}$$

Since various scores of similarity metrics can be also negative, we used for the normalization the *softmax* function transforming a vector $z = (z_1, ..., z_K) \in \mathbb{R}^K$ to a vector $y = (y_1, ..., y_K)$ where

$$y_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{4.2}$$

The only thing we need for HG is to cut out the bounding rectangle of detection from the scene, compute its histogram, and compare it to histograms of all reference images (Section 3.6.4). More specifically, we chose the intersection comparison method because the more complex comparing methods did not bring significant improvements. Since histograms of reference cars are used repeatedly during the race, it is also convenient to precompute and save them beforehand.

In TM, our goal is to find an area in the source image that accurately matches the template (see Figure 4.5). After cutting out a detection's bounding box, we first compare its size to the size of the reference photo and use the smaller one as a template and the bigger one as a source image. If each of the images is larger in one dimension, we add a corresponding number of black pixel rows or columns to the cut-out region and use it as a source image. It is only now possible to carry out the TM and let the algorithm output the best matching source image area and its similarity score for all reference photos. The similarity metric of our

choice was the normed mean shifted cross-correlation since it gave us the best results.

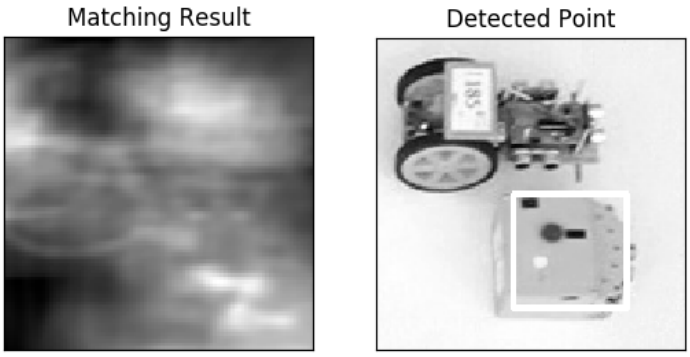


Figure 4.5: An example of the template matching algorithm matching a car to a cluster.

Identification frequency

Having everything set up, there are now basically two options for how often to perform the identification. A simpler and faster approach would be to identify each detection only once – for example, when it first appears on the frame or when it crosses the finish line. This method was, however, shown to be not very reliable. The problem was that the cars could be differently oriented at the moment of identification or detected with an excessive amount of neighborhood, which distorts the accuracy of the identification.

The second and more reliable approach is to take advantage of cars being tracked and perform the identification every n-th frame. More specifically, on every n-th frame, we obtain the similarity scores for a detection and find the car corresponding to the highest similarity score. If this score is strictly higher than the detection's current highest similarity score, we replace it with the new one and update the car associated with the detection accordingly. Using these regular updates, we should be able to identify all detections when they are captured in the cleanest form and when they have the same orientation as cars in the reference images. Such noiseless detections should ideally result in the highest similarity scores and be preserved until the car leaves the scene.

Identification in car clusters

Up until now, we have been assuming that one detection corresponds to only one car. Nevertheless, if several moving cars drive near next to each other in a cluster, they are detected as one detection (see Figure 4.6). And such detections need to be treated differently.

We will use the same TM technique as we have already used in the previous case, but we will make it more robust. Instead of trying to match only one car, we will try to match the detection with all possible combinations of cars and choose the most probable one.

So for a particular combination, we iterate through all the cars it contains. For each car, we find the most corresponding area in the cluster and save its

Figure 4.6: Two cars driving next to each other detected as a single detection.

matching score. To choose the most probable combination of cars, we consider the similarity score (computed as an average of individual matching scores) and the extent to which predicted car positions in the cluster intersect. We need to consider the intersection factor because the template matching algorithm process one car at a time, and so it can match all cars from a combination to one place. Therefore, we need to choose such a combination in which car positions do not intersect and have the highest similarity. If all combinations intersect, we choose the one with the smallest overlap. After that, we can finally discard the cluster's detection and replace it with detections of the individual cars that were found inside it.

To avoid matching all possible combinations on every detection, it is necessary to provide the algorithm with an area of the biggest car in the race[3]. By having this threshold value, we can easily recognize detections that can potentially contain more cars and detections with one car only.

4.3 Color tracking algorithm

As we outlined at the beginning of this section, the Color tracking algorithm (CTA) demands that the cars are distinguished by unique colors. Similarly, as we needed to supply the GTA (Section 4.2) with reference photos of participating cars, now we have to specify the unique colors of individual cars. Because the color detection is quite sensitive, and because the room's lighting conditions can change, we define those colors as ranges rather than single values. More details about the process of color picking can be found in the Attachment (Section A.1.4).

[3]Such and area can be computed using one of the pits from the starting box.

4.3.1 Object detection, tracking and identification

Once the color spans are defined, we can advance to the algorithm itself. Compared to the GTA, the most significant difference is that now we can identify all cars on the racetrack interchangeably. Meaning, all three stages of tracking can be solved with only one technique – the color filtering (Section 3.6.2).

More specifically, we will consequently filter each frame with all defined color ranges and obtain for each range a binary mask, where the white pixels depict regions fitting into a particular color range. Because of the lighting conditions, even the colors might not always be detected correctly. And as a result, the detected regions may be distorted. To fix that, we use the morphological operations (Section 3.4) again and replace contours with their convex hulls in every binary mask obtained from the color filtering (see Figure 4.7).

Figure 4.7: Detection of two colors with the color filtering technique – color contours (left), union of color filter masks (right).

From the color uniqueness, each binary mask should now contain maximally one coherent color region. But if for some reason – image noise or poorly chosen color spans, there are more detected regions, we preserve only the one with the largest area.

Because we have already detected and unambiguously identified all moving objects in each frame, we no longer need to search for corresponding detections in consecutive frames. And therefore, with noticeably less effort, we achieved the same or even better results than the General tracking algorithm.

4.4 Finish logic

Finally, now is everything prepared for the measuring part. We can locate the finish line, detect and identify moving objects in each frame, and track them across the video. What is left is to determine when a car completes a lap and correctly save the lap time.

4.4.1 Finish crossing

The most natural way to detect whether a detection crosses the finish line is to identify both the finish line and detections with some points and check whether the detections' x coordinates pass the finish line.

In section 4.1, we saved the finish line as two endpoints of a line going through the finish. But due to the camera angle, the finish line may appear to be skewed, and as a result, the x coordinates of its endpoints may slightly differ. So rather than identifying the position of the whole finish line with one of its endpoints, we represent the finish line's x coordinate with the x coordinate of the finish line's midpoint.

As for a detection, we identified its position with its rightmost point. There are also other points that could be chosen – the leftmost point or the centroid. However, none of them achieve as good stability as the rightmost point. The problem was that the detection may deform itself while crossing the black finish line and also change its shape when leaving the scene.

4.4.2 Completing a lap

Now, we can determine when a detection crosses the finish line in every single frame. However, to determine whether the car associated with the detection truly completed a full lap, we need to check two things additionally.

Firstly, we have to ensure that each detection during a single pass finishes only once. So after a detection finishes for the first time, we ignore all its next potential finishes.

And secondly, it can easily happen that an autonomous car gets lost on the track and start driving in the opposite direction. To make sure that we register only cars finishing in the right direction, we allow a car to finish only if its current centroid has a larger x coordinate than its centroid on the previous frame. A more in-depth analysis of this topic for GTA and CTA is discussed in Section 5.3.3.

When everything is checked, and a car is found to be completing a lap, the algorithm updates the car's statistics by increasing the number of completed laps and saving the time it took to finish it. More specifically, we measure the time as a product of the reciprocal video's FPS and the number of the frame when a car finished: $time = \frac{1}{FPS} \times frame\_number$.

5. Evaluation

At the beginning of the project, we managed to record only one race from Robotický den 2019. Since the event is held only once a year in June, it was not possible to get another real race recordings or test the algorithm in the actual competition. To gather more data, we simulated the race of autonomous cars with cars on remote controllers. Compared to autonomous robots, our RC cars were slightly bigger and heavier but, more importantly, noticeably faster, making the car tracking more difficult. The training races took place on the same track that is being used in the official event. As for the camera, we used a mobile phone camera fixed on a tripod. Altogether we organized two tournaments from which eleven races were recorded.

Both of our algorithms showed great performance on our videos and proved to be ready for the official usage. The General tracking algorithm (GTA) correctly evaluated the race from the official event and all eleven training races. When testing the Color tracking algorithm (CTA), we were not able to use all of the training data because the cars during our experimental races were generally not equipped with unique color labels. As a result, we could test the CTA on four of our training races, where the cars had unique colors naturally or were provided with unique color labels. All four videos were evaluated correctly[1]. By the correct evaluation, we mean that all completed laps were counted and measured for the right car.

In this section, we will, therefore, discuss situations which we noticed were for the algorithms more problematic, propose the ways how to prevent them, and describe the algorithms' output, which enables us to correctly evaluate even the races, where the algorithms were mistaken. But first of all, we would like to mention what other methods we also tried to implement into our solution but turned out to be inconvenient for our application.

5.1 Other tested approaches

Apart from the presented algorithms, we tried several other approaches that could potentially solve some of the stages in the problem of race evaluation but failed for various reasons. We especially experimented with alternatives for car detection and identification.

We tried to recognize cars based on the shape of objects on their labels (square, rectangle, circle, triangle) [13] or utilize existing optical character recognition (OCR) algorithms [23] for reading the numbers on the labels. However, because the cars drove relatively fast and the videos were recorded only on a mobile phone, it was not possible to clearly recognize nor the shapes nor the numbers. Additionally, for OCR detectors, there were also troubles with identifying numbers that are not always orientated the same way in real-time.

When we asked *Doc. Ing. Filip Šroubek, Ph.D. DSc.* (Institute of Information Theory and Automation AV ČR) for advice, he recommended us to try labeling

[1] Those four videos were among the nine training videos that were also correctly assessed by the GTA.

the cars with ArUco markers [24], the traditional marking system used for the pose estimation in computer vision. But this approach was unsuccessful as well because, for the available ArUco markers' detectors, the car labels were too small to detect.

In the GTA itself, we attempted to incorporate to the identification stage also different feature matching techniques such as ORB, SIFT, or SURF [25]. Still, this approach did not bring any improvement for our data and even resulted in worse accuracy than the presented combination of template matching and histogram comparison.

5.2 Program output

The program terminates when there is no video frame left or when a user stops the program deliberately. In both cases, the race is regarded as completed, and the results are prepared. The output with results is printed out in a text file, which has two main sections: *Race evaluation* and *Final order*. For the GTA, we also incorporated a third section called *General times* containing times of each finish with similarity scores associated with the finishing vehicle (see Figure 5.1).

```
        ========================
             Race evaluation
        ========================

Orange: 2 completed laps
        1. lap: 26.7 s
        2. lap: 27.2 s

Yellow: 2 completed laps
        1. lap: 21.6 s
        2. lap: 18.6 s

Blue:   0 completed laps

        ========================
               Final order
        ========================

1. Yellow: 2 rounds, total time: 40.2 s
2. Orange: 2 rounds, total time: 53.9 s
3. Blue:   0 rounds

        ========================
              General times
        ========================

(1) Time: 4.367  s, Scores: {'Orange': 0.247, 'Yellow': 0.482, 'Blue': 0.271}
(2) Time: 4.467  s, Scores: {'Orange': 0.563, 'Yellow': 0.176, 'Blue': 0.26}
(3) Time: 5.333  s, Scores: {'Orange': 0.303, 'Yellow': 0.193, 'Blue': 0.504}
(4) Time: 25.967 s, Scores: {'Orange': 0.323, 'Yellow': 0.412, 'Blue': 0.265}
(5) Time: 31.167 s, Scores: {'Orange': 0.484, 'Yellow': 0.231, 'Blue': 0.285}
(6) Time: 44.567 s, Scores: {'Orange': 0.325, 'Yellow': 0.412, 'Blue': 0.264}
(7) Time: 58.367 s, Scores: {'Orange': 0.478, 'Yellow': 0.257, 'Blue': 0.264}
```

Figure 5.1: Race report of the General tracking algorithm.

In the *Final order* section, cars are primarily ordered by the number of completed rounds and secondary by the total sum of times they needed to complete them. Times of individual laps are then broken down in the *Race Evaluation* part.

As we will discuss in the following sections, for both algorithms, there are situations where they can make a mistake. To ensure that the right order and lap times can still be obtained even when the algorithms make some unexpected mistakes, all finish photos are being saved, and the time of every finish passing is being kept. These times are then written on the finish photos with additional

information – finish number, frame number, name, and similarity score of the associated car in case of GTA (see Figure 5.2).

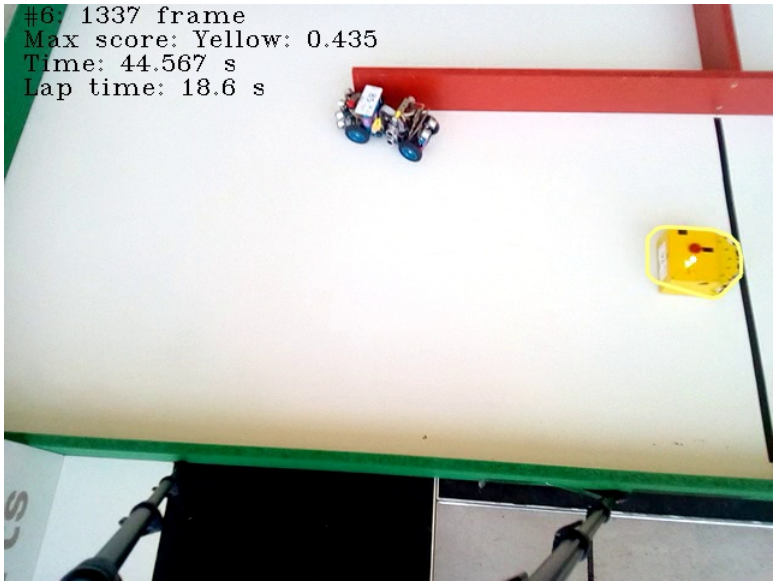


Figure 5.2: One of the finish photos from the official race measured by the General tracking algorithm.

5.3 Problematic situations

Our algorithms can potentially make different kinds of mistakes. They can identify an unfamiliar object as one of the cars, temporarily lose track of a car, misidentify a car for a short time, or detect a single car as multiple moving regions. However, all these mistakes are not serious as long as they do not occur nearby the finish line since otherwise, they do not affect the output of the program.

Very generally, we can divide those serious mistakes into two categories: misidentification and misdetection. The misidentification occurs when a car is correctly detected but wrongly identified when finishing. It means that the time is measured for the wrong car. Under the misdetection category, we count mistakes that occur when a car is not detected in the finish or detected multiple times.

Even though most of the causes were briefly mentioned in the previous section 4, we would like to recapitulate them now and also include their further description with possible ways how to avoid them.

5.3.1 Similar cars

It is not surprising that the GTA can have a problem with the identification of similar cars (see Figure 5.3) because neither the histogram comparison (Section 3.6.4) nor the template matching (Section 3.6.4) was designed to be entirely accurate. But for our application, they present an ideal solution for the real-time execution requirement.

As for the histogram comparison, the only thing it takes into account is the image color aspect. That means it completely ignores the structural information, and thus if two or more cars have the same dominant colors, they appear not much different from the histogram perspective. So, when a detection is compared to two reference photos with similar color distribution, the difference between the two similarity scores will be very small. The comparison can be further negatively affected by inaccurate detections that capture only part of the car or excessive nearby surroundings.

On the contrary, the template matching should tolerate such imperfect detections because it can find the matching part everywhere in the source image. Additionally, it incorporates the objects' structure to the computation, so it should compare images more reliably. But it also has a significant drawback. During template matching, the template is positioned only in one direction, but cars on the track can be rotated differently. And because of that, the template matching can be forced to make incorrect decisions.

To ensure that the cars are correctly classified most of the time, we need to provide the GTA with such reference photos that are as distinct as possible. For our experiments, picking only the unique part of cars as the reference photos yielded better results than picking the whole cars. This practice manifests itself during the identification of cars in clusters where the algorithm understandably regards the matched part as an entire car. However, this should affect the time measuring only minimally.

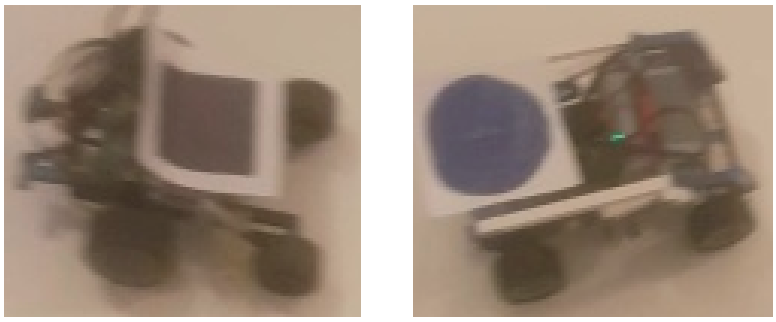


Figure 5.3: Two photos of cars that could be in some situations misplaced by the GTA.

5.3.2 Clusters of cars

Especially at the beginning of the race, cars are very likely to cross the finish line in a cluster (see Figure 4.6). And so in case of the GTA, the cars will be registered as a single detection. As was previously described, the problem of clusters is solved by iterating through all possible combinations of cars (Section 4.2.5). In our experiments, the identification of cars in clusters was always correct. But we are aware that in some situations, cars in a cluster can appear to be visually connected and make the identification for the algorithm more difficult.

5.3.3 Finishing in the opposite direction

Because autonomous cars can sometimes behave unexpectedly, they can accidentally start driving in the opposite direction. Again, as long as this does not happen near the finish line, we do not have to be concerned. But if a car drives through the finish line in the opposite direction, it must be noticed. Because not only, that we can not count that drive-through as a completed lap, but we also have to remember that the following finish pass in the correct direction should be for the same detection invalid as well.

Our algorithms solve this situation by not allowing the same detection to finish more than once and ignoring the finish passes of all detections that emerged on the right to the finish line. The only problem is when such a detection crosses the finish line in the opposite direction, and we lose track of it (the detection drives off from the frame or stops moving). Because when the same detection/car appears again and finishes in the right direction, it will be counted as a completed lap. We experienced it in one of our recordings when the GTA was able to correctly evaluate it only after we increased the number of frames on which the abandoned detections are preserved. However, having this number increased by default could potentially make some evaluation less accurate as the abandoned detections could be associated with other cars driving nearby them.

In the CTA, we prevent this by saving the information about the finish back driving not only to the detection but also to an associated car. In this way, we can count the car's finish back drives and check it every time the car crosses the finish line in the correct direction. If the number of back drives is zero, then it completed a lap. If the back drive number is positive, we decrease it by one and ignore the finish passing. We did not implement this logic into the GTA because the identification at the time of finish crossing does not have to be correct, and the number of back drives could be potentially increased for the wrong car.

5.3.4 Multiple detections of a single car

As we discussed in section 3.6.1, cars with large smooth monochromatic surfaces are more difficult to detect. But as it turned out, such cars are in the GTA also more prone to be detected as several moving regions. Thus, when such a detection finishes, the time is measured to all cars associated with the moving regions. When this happened in our experimental races, we managed to solve it by increasing the size of the structuring element of morphological operations (Section 3.4). However, we are aware that in the next races, this may not be an option as using a larger structuring element is more computationally demanding.

5.3.5 Track displacement and camera movement

Initially, in the GTA, we intended to detect movement by the background subtraction (BGS) (Section 3.6.1). Nevertheless, after experimenting with the videos, we added the option to detect movement with frame differencing technique (FD) (Section 3.6.1). That is, because the RC cars were bigger and heavier, it was much easier for them to move with the walls of the track. As a result, the displaced track appeared to be different from the background reference and, therefore, in the following frames detected as a movement. But because the reference image

is updated regularly in FD, it can quickly adapt at any track adjustments. The very same applies to the random camera movement, which is very hard to prevent entirely.

Although BGS appears in this aspect as a less practical solution, it also has its advantage. Because compared to FD, it can also detect non-moving cars. And that is very convenient because the cars can stop, and the algorithm does not lose track of it, and so the history of identification is not lost.

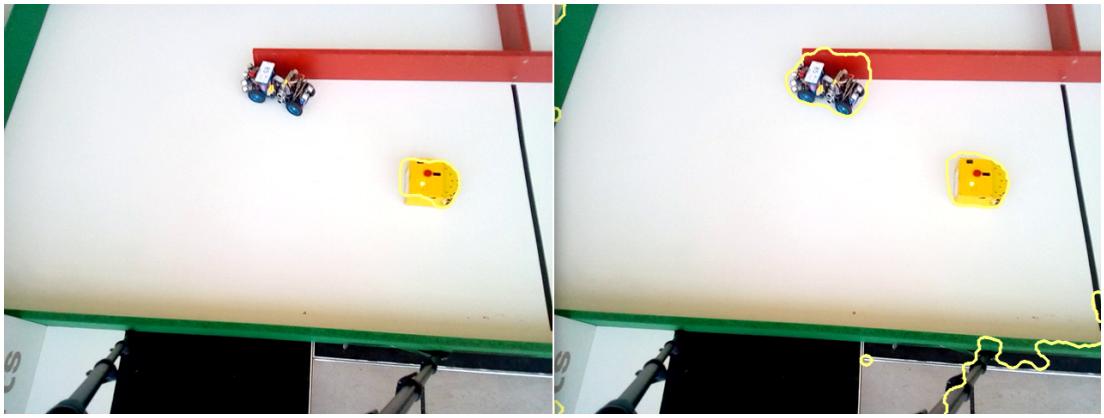


Figure 5.4: Comparison of frame differencing (left) and background subtraction (right). While frame differencing detects only the moving car, background subtraction detects also the non-moving car and other frame regions that differ from the reference image.

5.3.6 Right color choice

The most important thing for the CTA is the right choice of colors and the adjustment of the span of accepted shades. When using the natural colors of cars instead of the labels, it should not matter if the color regions are coherent. Because the CTA by default allows cars to finish another lap only after five seconds, a car with disjoint color regions should be registered during a finish pass only once[2]. This applies as long as the regions have the same color, otherwise it is a problem of poorly chosen colors.

We also strongly advise against choosing black or white as one of the unique colors since those are colors of the track and the finish line.

5.4 Final recommendation

Our experiments proved that both of our algorithms could be used in the official races. However, from our experience, the more reliable and convenient usage is offered by the CTA. Because compared to the GTA, with carefully picked color labels, there is far less chance that the detected cars will be misidentified or otherwise missed during the finishing. Furthermore, since there can start up to

[2]When the first region crosses the finish and disappears from the scene, the other region can be still in front of the finish line.

five robotic cars in every race, it is sufficient to prepare five different color labels and use them repeatedly for every race. And so, it is unnecessary to take a photo of each car and reconfigure the initial algorithm setting before every race but only once at the beginning of the race day. An example of a unique color palette is shown in Figure 5.5 below.

Figure 5.5: Recommended choice of 5 unique colors (HSV codes: (10, 70, 70), (80, 70, 70), (150, 70, 70), (210, 70, 70), (260, 70, 70)).

6. Implementation

In this section, we will first specify what programming language and computer vision library we used and describe the basic structure of both our algorithms. Then, we will discuss the options for the computer-camera connection and the algorithms' execution time.

6.1 Programming language and library

At the beginning of the work, we stood before the decision of what programming language and what libraries we would use. After reading several reviews and several implementations of computer vision algorithms, we decided to choose the recommended OpenCV library [26] as it seemed to be the most used and referred library. OpenCV is an open-source computer vision library written in C and C++ with a strong focus on real-time application. Apart from C and C++, it also provides interfaces to several other languages such as Python, Java, or Matlab, making it easy to use both for industry and research applications.

From the programming languages supported by OpenCV, we eventually picked Python. We chose this programming language because Python enables developers to experiment and change the code very efficiently. Our only concern with Python was the speed of execution since, as an interpreted language, it is known to be slow. But it turned out that with the direct bindings to C++ API of OpenCV, the execution of image processing techniques in Python is quick enough even for our real-time requirements.

6.2 Program structure

Because of the differences in the General and Color tracking algorithms, we implemented each version in a separate program. We will refer to them as the General algorithm (GA) and the Color algorithm (CA).

Processing the video, our algorithms' backbone is a loop iterating through the frames coming from the video stream. The only frame that is treated differently is the first frame in which the finish line is detected. All the following frames are processed in the same pipeline of subroutines called as static functions from individual modules of our framework.

Once the contours of moving objects are extracted, they are immediately turned into instances of appropriate classes – *Blob* in the GA and *ColorContour* in the CA. These classes serve as containers for additional information associated with contours (centroid location, bounding rectangle, etc.) and for methods needed for finish crossing detection and, in case of *Blob* class, for object identification as well.

Because in the CA, it is possible to identify every detection unambiguously, we can define one *ColorContour* instance for each unique color before the race starts. And then, every time a particular color appears in the frame, we associate it with the correct *ColorContour* instance.

With the GA, the situation is more complicated. Because the detections are firstly detected based on the movement, we can not be sure whether the identification is correct during the whole course of the algorithm. Therefore, we introduced another class called *Car* whose instances correspond to the reference photos. The identification itself is then based on assigning a particular instance of *Car* to a *Blob* instance. Thus if the algorithm evaluates that more detections in a frame correspond to a single car, it is possible to do so.

The summarized structure of both algorithms can be found in the following pseudo-codes.

Algorithm 1: General algorithm

```
   Input: video_stream, reference_images
   Parameters: n, min_cluster_area
 1 tracked_detections = [ ]
 2 cars = Car(reference_images)
 3 first_frame = video_stream.next()
 4 finish_endpoints = detect_finish(first_frame)
 5 frame_num = 1
 6 while video_stream.has_next() do
 7     frame_num += 1
 8     frame = video_stream.next()
 9     frame_detections = [ ]
10     frame_contours = detect_contours(frame)
11     for contour in frame_contours do
12         detection = Blob(contour)
13         if detection.area > min_cluster_area then
14             cluster_detections = process_cluster(detection)
15             frame_detection.append(cluster_detections)
16         else
17             frame_detection.append(detection)
18     iou_m = compute_ious(tracked_detections, frame_detections)
19     optimal_assignment = hungarian_alg(iou_m)
20     tracked_detections = update_positions(optimal_assignment)
21     if frame_num % n == 0 then
22         update_identifiactions(tracked_detections)
23     check_finish_crossing(tracked_detections)
```

Algorithm 2: Color algorithm

```
Input: video_stream, unique_colors
colors = ColorContour(unique_colors)
first_frame = video_stream.next()
finish_endpoints = detect_finish(first_frame)
frame_num = 1
while video_stream.has_next() do
    frame_num += 1
    frame = video_stream.next();
    for color in colors do
        binary_mask = threshold(frame, color)
        biggest_contour = max(detect_contours(binary_mask))
        update_position(color, biggest_contour)
        check_finish_crossing(color)
```

6.3 Camera connection

Since the objective of this thesis was to develop a working algorithm for the race evaluation, we did not focus on optimizing the computer-camera connection.

The algorithms were tested on prerecorded videos saved on the local storage. Other than that, we tested the connection to a laptop's web camera and a phone's camera. The web camera can be connected directly through the OpenCV library. Connecting to the phone's camera is possible through a USB cable or wirelessly via WiFi. But for both approaches, it is necessary to install a particular mobile application (we tested the Android version of IP Webcam [27]).

However, regardless of the connection, it should always be checked that the video quality is appropriate and that the frame orientation is correct. Since the low quality may negatively influence the algorithms' performance, especially the identification process (Section 4.2.5) in the GA, and the wrong orientation prevents the algorithm from finding the finish line (Section 4.1).

6.4 Performance

To measure the execution time of both algorithms, we used the 480p video (640×480 px) from the official race with three participating cars and ran it in a single thread on 2.40GHz i5-6300U CPU and 8 GB of RAM. With the GA, the processing of a single video frame without any visualization took on average 0.013 s (min 0.010 s, max 0.055 s), which was fast enough to process it in the original 30 FPS. The CA also executed fast enough but needed for a single frame of the same video on average 0.033 s (min 0.031 s, max 0.049 s). As we can see, the CA was on average more than two times slower, which was caused by the need to convert each frame into an HSV format before applying the color filters.

7. Conclusion

Our main goal was to design and implement an algorithm that would help referees at robotic contests to automatically evaluate robotic races from a video recording. This particular initiative came from organizers of a Czech event called Robotický den, which has its race discipline called RoboCarts.

In this thesis, we first described the rules of robotic races and inspected the racetrack's appearance in section 2. Before presenting our algorithms, we covered in section 3 several image processing methods that we had used to process and analyze the video. Special attention was given to the problem of moving object tracking as it constitutes an essential part of our solution.

In section 4, we presented two different algorithms for automatic robotic race evaluation – the General and Color algorithm. We came up with two algorithms because we found out that if we provided cars with labels of unique colors, the whole process of car tracking would get noticeably smoother and more reliable. So while in the first algorithm, we do not expect cars to have any particular appearance, for the second algorithm, it is necessary to stick on each car a unique color label. After the race ends, both algorithms output the final order and times of every completed lap. To retrospectively check the correctness, all finish photos are saved and annotated with information about the time and identified car.

Initially, we had only one recording of an official race. So for testing and evaluation purposes, we organized and recorded two training tournaments for remotely controlled cars. As we discussed in section 5, the General algorithm successfully measured the official race and all eleven training races. With the Color algorithm, we correctly evaluated all four training races in which the cars have unique colors.

And finally, in section 6, we discussed our choice of programming language, the camera settings, and the processing times of both our algorithms. An extra focus was also given to the algorithms' structure and their classes.

In the beginning, we did not have any particular resources for this specific task. Therefore, the main contribution of this work is experimenting with several different approaches and finding such methods that could be used in standard conditions. Since Robotický den was canceled this year, we did not manage to try our algorithms in official races. However, even without this testing, we recommend using the Color algorithm since it should generally provide us with more reliable results and with reusable color labels also more convenient usage.

If our algorithms proved to be useful, the next possible step would be implementing a mobile application, which could make the usage even more convenient. As for further algorithm improvements, it would be very interesting to gather more robotic car images and train a deep learning model for the detection stage. Another practical improvement would also be a tool for the automatic extraction of car reference images during the race, which would improve the potential usage of the General tracking algorithm.

Because of the solution's generality, there is also a possibility to try using an adjusted version of our algorithms for tracking completely different objects or even slightly different purposes, such as finding out the most frequent color of cars or peoples' clothes from street-view cameras.

Bibliography

[1] Robotický den. Robotický den. http://robotickyden.cz. Accessed: 2020-06-01.

[2] Robotický den. RoboCarts. http://robotickyden.cz/2020/rules/2020-RoboCarts-ENv1.pdf. Accessed: 2020-06-01.

[3] E. Šikudová, Z. Černeková, V. Benešová, Z. Haladová, and J. Kučerová. *Počítačové videnie. Detekcia a rozpoznávanie objektov*, pages 38–49. Wikina, Praha, 2013.

[4] L. G. Shapiro and G. C. Stockman. *Computer vision*, pages 153–154. Prentice Hall, New Jersey, 2001.

[5] OpenCV. Image thresholding. https://docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html. Accessed: 2020-04-26.

[6] N. Efford. *Digital image processing: a practical introduction using Java (with CD-ROM)*, pages 271–297. Addison-Wesley Longman Publishing Co., Inc., Boston, 2000.

[7] A. Cereser. *Time-of-flight 3D Neutron Diffraction for Multigrain Crystallography*. PhD thesis, 2016.

[8] J. Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986.

[9] SimsContPics. The canny edge detector applied to a color photograph of a steam engine. https://en.wikipedia.org/wiki/Canny_edge_detector. Accessed: 2020-04-26.

[10] P. V. C. Hough. Method and means for recognizing complex patterns, December 18 1962. US Patent 3,069,654.

[11] T. Kacmajor. Hough lines transform explained. https://tomaszkacmajor.pl/index.php/2017/06/05/hough-lines-transform-explained/. Accessed: 2020-07-11.

[12] OpenCV. Hough line transform. https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html. Accessed: 2020-04-26.

[13] P. H. L. Shilpa and M. R. Sunitha. A survey on moving object detection and tracking techniques. *International Journal Of Engineering And Computer Science*, 5(5), 2016.

[14] K. A. Joshi and D. G. Thakore. A survey on moving object detection and tracking in video surveillance system. *International Journal of Soft Computing and Engineering*, 2(3):44–48, 2012.

[15] A. P. Shukla and M. Saini. Moving object tracking of vehicle detection: A concise review. *International Journal of Signal Processing, Image Processing and Pattern Recognition*, 8(3):169–176, 2015.

[16] M. Kaliczyńska R. Szewczyk, C. Zieliński. *Automation 2018: Advances in Automation, Robotics and Measurement Techniques*, pages 292–294. Springer International Publishing, New York City, 1959.

[17] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[18] OpenCV. Histogram comparison. `https://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/histogram_comparison/histogram_comparison.html`. Accessed: 2020-04-26.

[19] W. Jia, H. Zhang, X. He, and Q. Wu. A comparison on histogram based image matching methods. 2006.

[20] OpenCV. Template matching. `https://docs.opencv.org/master/d4/dc6/tutorial_py_template_matching.html`. Accessed: 2020-04-26.

[21] E. Trucco and A. Verri. *Introductory techniques for 3-D computer vision*, volume 201, pages 145–149. Prentice Hall Englewood Cliffs, New Jersey, 1998.

[22] A. Reynolds. Understanding and evaluating template matching methods. `https://stackoverflow.com/a/58160295`. Accessed: 2020-06-01.

[23] OpenCV. OCR of hand-written data using kNN. `https://docs.opencv.org/master/d8/d4b/tutorial_py_knn_opencv.html`. Accessed: 2020-06-02.

[24] OpenCV. Detection of ArUco markers. `https://docs.opencv.org/trunk/d5/dae/tutorial_aruco_detection.html`. Accessed: 2020-06-02.

[25] E. Karami, S. Prasad, and M. Shehata. Image matching using SIFT, SURF, BRIEF and ORB: performance comparison for distorted images. *arXiv*, 2017.

[26] OpenCV. *The OpenCV Reference Manual*, June 2020.

[27] P. Khlebovich. IP Webcam. `https://play.google.com/store/apps/details?id=com.pas.webcam&hl=cs`. Accessed: 2020-06-01.

List of Figures

List of Abbreviations

BGS Background subtraction

CA Color algorithm

CC Cross-correlation

CTA Color tracking algorithm

FD Frame differencing

GA General algorithm

GTA General tracking algorithm

HS Histogram similarity

MSCC Mean shifted cross-correlation

SAD Sum of absolute differences

SSD Sum of square differences

TM Template matching

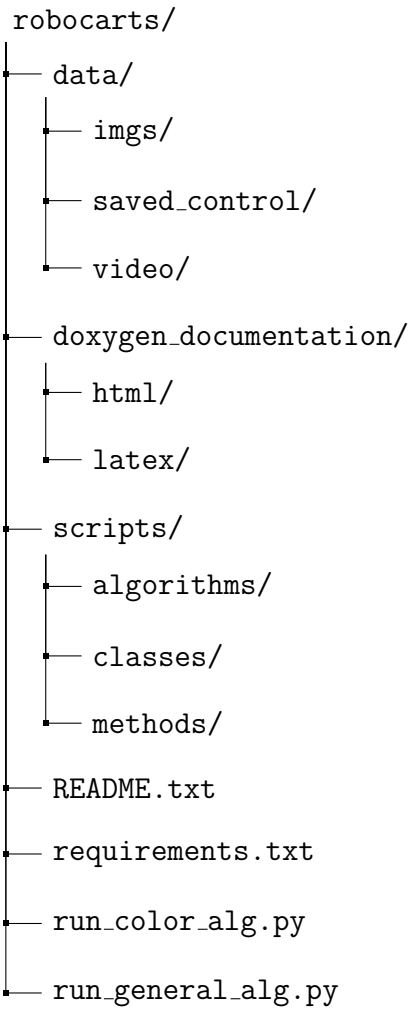
A. Attachments

A.1 User documentation

In this section, we will describe how the attached zip file's structure looks like and how to run and test our applications properly.

A.1.1 Folder structure

After downloading the attached zip file, you can unzip it to an arbitrary folder in your PC or laptop[1]. The unzipped *robocarts* folder should have the following structure.



There are three main directories: `data` contains testing data, finish reports and photos from two recordings, `doxygen_documentation` contains a *Doxygen* auto-generated documentation and `scripts` contains the Python scripts of our algorithms.

Then there are two main application scripts (`run_general_alg.py` and `run_color_alg.py`), the `README.txt` with instructions and the `requirements.txt` listing all Python packages necessary for the program execution.

[1] All the development and testing was conducted on the Ubuntu operating system.

A.1.2 Requirements

Our applications need to be run with Python 3.5+ and several compulsory packages listed in the `requirements.txt`. The easiest way how to install them all at once is to use the *pip* Python package manager with the following command

```
pip install -r requirements.txt
```

Since our scripts contain relative paths, both of our main applications must be executed right from the `robocarts` directory.

A.1.3 General algorithm program description

We will first describe the usage of the General algorithm which is run with the `run_general_alg.py` script.

Compulsory parameters

There are two compulsory parameters for this program. The source of a video (`-video`) and the location of reference photos (`-imgs`). The video source can be easily set up by either entering a path to a saved video file or "*webcam*" (connecting the web camera) in the program's parameters. Other options, such as wireless or wired phone connection, should work similarly – by specifying URL or another source identifier in the video path parameter, but may potentially require additional adjusting in the script.

Video size and orientation

Extra attention should be given to the orientation and size of the video frames. If the video for some reasons came up flipped, differently rotated, or was too big for real-time processing, it is necessary to specify it in the program's parameters as well (Section A.1.5).

Reference photo extraction

For extracting the reference photos, we do not provide any particular subroutine. The reference photos used in our experiments were manually cut out from race photos. As we discussed in section 5, the best results were achieved when we cut out only the most distinct parts of individual cars. It also helps if those rectangular parts were of different sizes. We suppose that those photos will always be saved locally and so the specification of their path should not present any problems. The picture files should have unique names because the filenames are used as unique IDs.

Finish line detection

In a standard way, the finish line is automatically detected in the first frame of a video, as we described in section 4.1. But to not limit ourselves only to one type of a racetrack, we added an option to manually select the finish line by the parameter `--manual_finish` in the first frame of a video.

To select the finish line manually, you need to choose two points in a frame that will determine the finish line. After the selection, you can either confirm your choice by left-clicking or deny it by right-clicking on a new window with the highlighted finish line (see Figure A.1). After the confirmation, terminate the subroutine by pressing 'q', and the video starts being processed. The finish line will be automatically prolonged across the whole frame.

Figure A.1: Manual selection of the finish line in the first frame of the official race recording.

Control

When the application is run, and everything is set correctly, a window with a particular video is invoked, and the tracking is visualized by highlighting the contours of the moving objects (see Figure 1.1). Whenever a detection finishes, it is announced in the terminal. We can pause/resume the video by pressing "p" or terminates it by pressing "q". After the video ends, the race results and finish photos are prepared in the folder specified by the parameters `-save_folder` and `-race_name`.

There is also a possibility to terminate the race but keep the video and the algorithm going by pressing "n". This way, the program will print the current results, delete the statistics from its working memory, and start from the beginning. Nevertheless, this option is more useful for the Color algorithm where we can reuse the same color labels for multiple times.

A.1.4 Color algorithm program description

In many aspects, Color algorithm program `run_color_alg.py` is similar to the General algorithm program. Therefore, we will mainly focus on things that are different.

Compulsory parameters

Naturally, `run_color_alg.py` also requires us to specify the video source, for which applies the same recommendations as we previously discussed for the General algorithm.

The second compulsory input is unique colors, which can be entered in two different ways. Firstly, we can specify color ranges as a series of numbers with the parameter `-boundaries`. Every HSV range must be represented by exactly six integers – first three numbers specify the color lower boundary, the later three numbers the upper boundary (e.g., we would specify two color ranges as `-boundaries` 100 80 80 120 90 90 60 50 50 70 60 60). Or secondly, we can use a special color subroutine in which we manually choose the colors from a particular reference photo (`-img_pick`).

Color picking subroutine

When the program is run with the reference photo, the color picking subroutine is invoked, and the reference photo in the RGB and HSV format is displayed. By left-clicking on a particular pixel of the HSV photo, the program automatically generates a binary mask where we can see what regions of the photo will fit into the pixel's color range (the pixel's HSV value ± the span). The color range is then saved to the program by left-clicking on the displayed binary mask and can be deleted from the program's memory by right-clicking on the mask. Example photos from the color picking subroutine can be seen in the following Figure A.2.

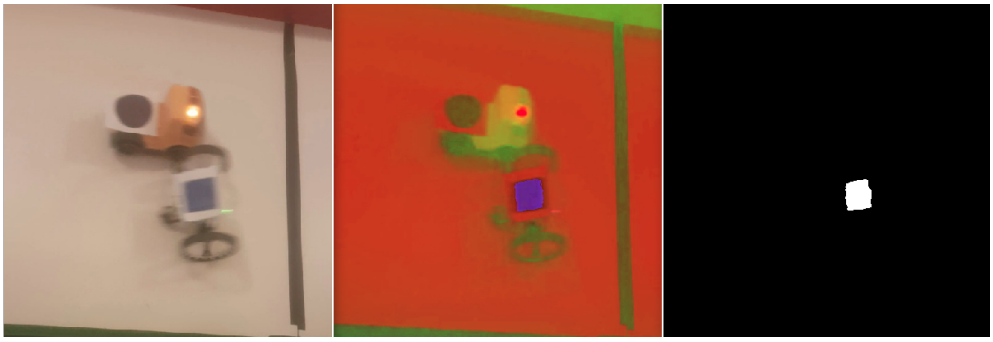


Figure A.2: The original photo in RGB and HSV format and the binary mask produced by the color picking subroutine after clicking on the blue square.

After pressing the 'q' key, the subroutine is terminated, and the video starts being processed in the same way as we described in section A.1.3.

Control

While the video is being processed, the Color algorithm is controlled exactly the same as the General algorithm. And there is the option for the manual finish line selection as well.

A.1.5 Other parameters

In the previous sections, we described the compulsory parameters for each program and a few others. In the following table, we summarize all possible param-

eters for both programs in the format 'parameter name (default value)'.

Both algorithms

−**video** (None)

Path to the video source.

−**start_frame** (1)

Starting frame number for saved recordings.

−**testing** (None)

Test of an algorithm on a particular video specified by a filename.

−**m_kernel** (27)

Size of the $n \times n$ kernel used in morphological operations.

−**fps** (30)

FPS of the video.

−**save_folder** ("./data/saved/")

Path where the race report and finish photos will be saved in the 'race_name' subfolder.

−**race_name** (None)

Name of the race, after which the save subfolder will be named. If not specified, it will be resolved from the video filename.

−**rotate** (0)

Specifies how many times each frame should be rotated for 90° to right.

−**resize** (1)

Scale of how every frame should be resized.

−−**flip** (-)

Specifies if each frame should be flipped alongside the horizontal axes.

−−**manual_finish** (-)

Switch flag for manual finish line selection.

General algorithm

−**imgs** (None)

Path to reference car images.

−**n_delete** (5):

Number of consecutive frames in which a detection needs to be absent to be deleted.

−id_freq (5)

Specifies how often the detections in a video are reidentified.

−n_finish (5)

Minimal number of frames on which a detection needs to be present to finish.

−min_iou (0.01)

Minimal value of IoU for assigning two detections from the consecutive frames.

−cluster_area (7000)

Minimal area (in pixels) of a detection to be regarded as a cluster.

−bg_img (None)

Path to a background image. If specified, the background subtraction technique is used instead of the frame differencing.

Color algorithm

−img_pick (None)

Path to an image for the color picking subroutine.

−boundaries (None)

Ranges of unique colors.

−span (10 20 40)

Span of shades for selected colors.

A.1.6 Program output

The program output (finish photos and the race report) from a single video is by default saved into a `data/saved/race_name` folder. Depending on the algorithm, it is placed either to `general` or `color` subfolder which is further divided into numbered subfolders corresponding to individual races (created by pressing "n" during the processing).

A.1.7 Testing

To test our programs, we included in the `data/video` folder two race recordings (`official_race.mp4`, `train_02.mp4`). All necessary reference images to both of the videos are placed in the corresponding folders in `data/imgs`, named after the video files. We also put all finish photos and race reports from our evaluation in `data/saved_control`.

For the testing purposes, please use the `-testing` parameter with the specific filename. This way, all the compulsory parameters will be solved automatically. These are examples, how to test both programs on two different videos.

```
python3 run_general_alg.py -testing official_race.mp4
python3 run_color_alg.py -testing train_02.mp4
```

If you want to test our algorithms on all available recordings, download them from the link specified in the attached `README` file. To preserve the project's structure, please replace the original `data` folder with the downloaded one. In the `README` file, there are also links to YouTube videos in which we demonstrate our algorithms on all recordings.

The names of all training videos are list in the following table. Please note that because the colors are selected manually, and the color labels are not perfect, your testing of the CA may behave differently.

Filename	**General alg.**	**Color alg.**	**Used colors**
official_race.mp4	✓	X	
train_01.mp4	✓	X	
train_02.mp4	✓	✓	blue, yellow
train_03.mp4	✓	✓	red
train_04.mp4	✓	X	
train_05.mp4	✓	X	
train_06.mp4	✓	X	
train_07.mp4	✓	✓	red, yellow
train_08.mp4	✓	X	
train_09.mp4	✓	X	
train_10.mp4	✓	✓	red, yellow
train_11.mp4	✓[2]	X	

✓: correct evaluation.

X: evaluation was not possible.

[2]The misidentification at the end of the race can be corrected by increasing the number of frames, after which a detection is deleted from the memory, in the program's parameters (`-n_delete 10`).