Graphical Models
-
Car detection in Mario Kart using Graph-Based Algorithms

Chiara Roverato
CentraleSupélec
chiara.roverato@student-cs.fr

Quentin Gopée
CentraleSupélec
quentin.gopee@student-cs.fr

Raphaël Romand-Ferroni
CentraleSupélec
raphael.romandferroni@student-cs.fr

Abstract

As part of the Graphical Models course in CentraleSupélec, our final project intend to explore different graph algorithms tailored for segmenting cars in Mario Kart images and videos. We explore variations such as a tree based method with masking strategies and max-flow/min-cut algorithms for efficient segmentation. We use several optimization and preprocessing techniques to enhance car features for better segmentation. In this report, the two methods used are presented as well as the results obtained.

The code and project files can be found in the GitHub repository at:

https://github.com/LaFerraille/Graph-based-segmentation-for-car-detection-in-Mario-Kart

Feel free to explore the repository for additional details and resources.

1. Introduction

Image segmentation is one of the most crucial task nowadays in image processing and more generally in computer vision. Segment an image means dividing an image into several meaningful and relevant regions based on different properties: color, texture, brightness, size and so on. In the case of Mario Kart, detect the car of the main player in the foreground of image may be a challenging task. Albeit the car stays as the same position during all the race and the color of the car and the character stays the same (green, because we focused on Luigi), it becomes harder when it comes to detect the car when there is ink on the screen, crashes, slides, turn and when the car changes of size with storm.

We propose to overcome these challenges of detect the car in Mario Kart with two graph-based methods for image segmentation: a tree based method with the application of masks and a max-flow/min-cut method. Both these algorithms involve the application of graph theory to construct a representation of an image in the form of a graph.

2. Tree-based method with masking

2.1. Tree-based algorithm

This algorithm has been first introduced by [2]. This algorithm addresses the problem of segmenting an image into regions, it's a graph-based approach to segmentation. The objective is to represent an image into an undirected graph $G = (V, E)$. Here vertices $v_i \in V$ is the set of elements to be segmented, and edges $(v_i, v_j) \in E$ corresponding to pairs of neighboring vertices. We can either consider the 8 neighbors or only the 4 neighbors of each pixel, depending on the computation requirements. Each edge $(v_i, v_j) \in E$ has a corresponding weight $w((v_i, v_j))$, which is a non-negative measure of the dissimilarity between neighboring elements v_i and v_j: here we chose to apply the Euclidean distance between two colored pixel intensities.

When edges and nodes have been processed, we begin the process of segmentation by sorting the edges of the graph in non-decreasing order based on their weights. Initially, each vertex forms its own component. We then iterate through the sorted edges and for each edge, we check if the vertices it connects belong to disjoint components. If so, and if the weight of the edge is smaller than a threshold min_comp_size. If the conditions are not met, the components remain unchanged. As such, we removed the small components iteratively from the forest. We repeat the merging process for all edges in the sorted order until all edges have been processed. At the end, the final segmentation of the vertices into connected components turns out.

Here is quick illustration of this algorithm and how it performs on a simple 8×8 grid of colored pixels.

Here we see different segmentation with different set of parameters. On the bottom left, we perform the segmentation with $\sigma = 1$, neighbor = 4, $K = 0.2$, and min_comp_size $= 12$. And on the bottom right, the segmentation is done with the same Gaussian blur but different neighbor = 8, $K = 1$, and min_comp_size $= 6$. Overall, we clearly see the impact of K (that is part of the threshold process) and min_comp_size in the process of removing small components.

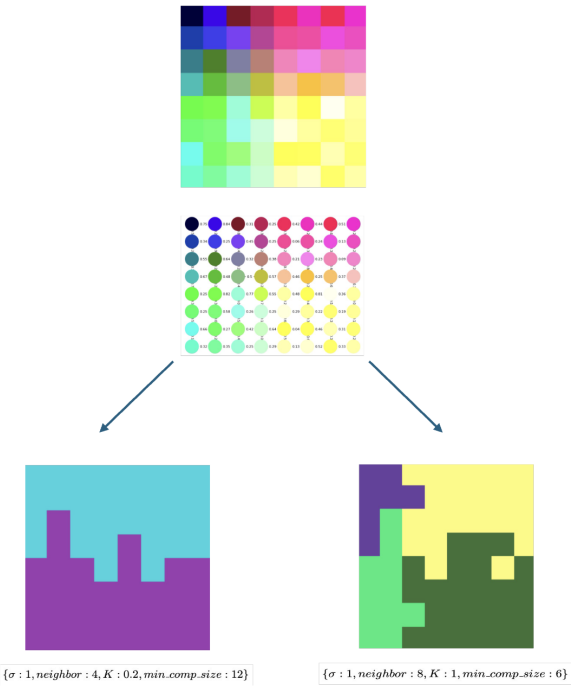


Figure 1. Tree-based method illustrated

2.2. Final pipeline for Mario Kart segmentation

The image processing pipeline begins with the original image, as shown in Figure 2. The image is then resized to reduce computation time, resulting in the resized image shown in Figure 3. Next, a Gaussian blur is applied to smooth the image, as depicted in Figure 4. Finally, the image is segmented using the segmentation algorithm, resulting in the segmented image shown in Figure 5. For this first segmentation, we used a rather big min_comp_size of 800 and we can clearly distinguish three distinct components in the resulting forest.

Figure 2. *Original Image*

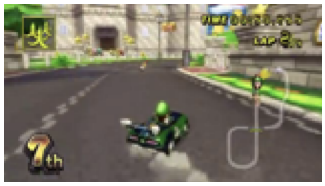


Figure 3. *Resized Image*

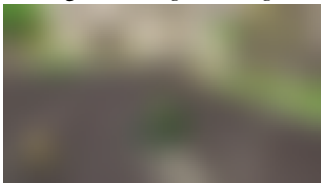


Figure 4. *Blurred Image*

Figure 5. *Segmented Image*

As you can imagine, the algorithm struggles to extract Luigi and its car as one and unique component on the picture. We decided to implement for scratch a masking technique aimed at helping the segmentation in different components and eventually extract the car component which doesn't appear at all in 5. We first decided to apply a black mask to the image to only extract the bigger component of the first forest (here the green area on the right of Figure 5) because the car has to be in there: the road is quite uniform and big enough to form the bigger component here. Figure 6 shows the result after applying a mask to the segmented image, where pixels not belonging to the main component are set to black. Subsequently, a green mask is applied to highlight specific features in green, as depicted in Figure 7. Because Luigi's car and Luigi himself are green, we thought quite relevant to highlight them first in the picture before segment the image. Another segmentation is then performed again with different parameters, setting a lower min_comp_size to 100, resulting in the green-segmented image shown in Figure 8. Finally, the object of interest, in this case, Luigi's car, is detected and highlighted, as shown in Figure 9. In this last step, we filter on the components that are equally similar in size with the car (we chose 1000 as a threshold) and a distance relatively close to the center. That is why in this last step we don't wrongly choose the purple component on the right of 8.

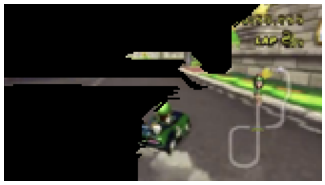


Figure 6. *Masked Image*

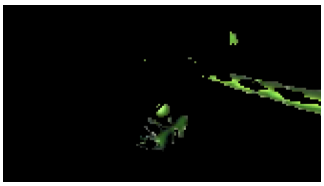


Figure 7. *Green Masked Image*

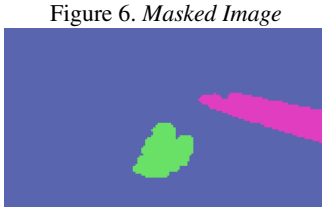


Figure 8. *Re-segmented Image*

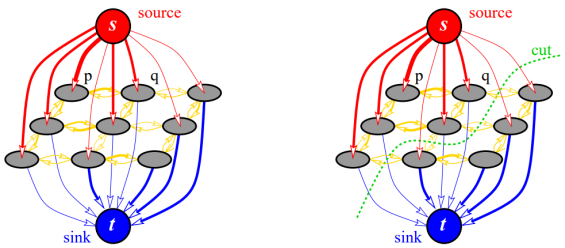
Figure 9. *Detected Image*

We have performed this algorithm to several images in a race to compute a full video of this segmentation process and see the potential wrong segmentation in real time. This Video Processing pipeline follows the same logic as for one image, except the fact that we perform the first segmentation on the full image with smaller components only on the first image of our video. Of course it can be questioned and on can wonder if this works as well at any moments of the race when the car is not as discernable of the picture.

3. Max-Flow/Min-Cut Algorithm

3.1. Boykov-Kolmogorov

Min-cut/max-flow algorithm for energy minimization in image segmentation has first been introduced in [1]. Generally, min-cut/max-flow is a classic algorithm for performing the graph partitioning task. It works by finding the minimum cut in a graph, which means the cut with the lowest weight.



(a) A graph $\mathcal{G}$ (b) A cut on $\mathcal{G}$

Figure 10. Min-cut/Max-flow on a simple image

But before applying this method, we first need to transform

Algorithm 1 Video Processing Pipeline with tree-based clustering segmentation

procedure VIDEOPROCESSINGPIPELINE(*video_path*, $\sigma 1$, $\sigma 2$, neighbor, $K1$, $K2$, min_comp_size1, min_comp_size2)
  $cap \leftarrow$ open video capture with *video_path*
  $fps \leftarrow$ get video frame rate from cap
  $start\_frame \leftarrow \lfloor start\_time \times fps \rfloor$
  $end\_frame \leftarrow \lfloor end\_time \times fps \rfloor$
  set cap position to $start\_frame$
  $previous\_threshold \leftarrow None$
  for $i \leftarrow 0$ **to** num_frames **do**
    $frame \leftarrow$ read frame from cap
    $init\_image \leftarrow$ create and resize PIL image in RGB from $frame$
    $new\_image \leftarrow$ convert $init\_image$ to numpy array
    if $i = 0$ **then**
      $forest, t \leftarrow$ get_segmented_image($params1$, $init\_image$,$previous\_threshold = None$)
      $mask \leftarrow$ get_mask($forest$, pil_image)
      apply $mask$ to new_image to extract larger component
    end if
    $green\_mask \leftarrow$ get_green_mask(new_image)
    apply $green\_mask$ to new_image by setting non-green pixels to black
    $forest, new\_t \leftarrow$ get_segmented_image($params2$, new_image, $previous\_threshold = t$)
    A last mask is then applied to the original $frame$ (the component in the forest that is closest to a predefined point) highlighting the component of interest
    save $frame$ as JPEG file in $output\_folder$ with appropriate naming
    print "End of video."
    break
  end for
  release cap
end procedure

the image into a directed graph $G = (V, E)$, where each edge $(p, q) \in E$ has a corresponding weight $w(p, q)$ and V is the set of nodes.

Firstly, the image is segmented into superpixels OpenCV package from Python. This step groups pixels with similar characteristics into cohesive regions called superpixels, reducing the complexity of the image. For each superpixel p, features such as color and spatial information are extracted. This includes computing the mean color intensity, centroid position, and color histograms in LAB color space. Using these feature, we can compute the weight of the connection between superpixels $(p, q) \in E$ using the similarity:

$$Sim(p,q) = \exp\left(-\frac{(I_p - I_q)^2}{2\sigma^2}\right) \times \frac{1}{dist(centroid_p, centroid_q)}$$

where $dist$ is the L2-norm, I is the LAB representation of the pixel and σ is a hyperparameter set to 5 in our experiments.

Then we manually label some of the superpixels as object (ob) or background (bg), add a source node s and a sink node t that will later define which superpixels correspond to the object and the background, and create a dense graph that connect each nodes according to the strategy described in Table 1:

edge type	weight	condition
(p, q)	$Sim(p, q)$	$(p, q) \in E$
(s, p)	$\lambda Rp_{bg}(p)$	$p \in V, p \notin Ob \cup Bg$
	K	$p \in Oj$
	0	$p \in Bg$
(p, t)	$\lambda Rp_{ob}(p)$	$p \in V, p \notin Ob \cup Bg$
	0	$p \in Oj$
	K	$p \in Bg$

Table 1. Edge Conditions and Weights

where:

$$Rp_{ob}(p) = -\ln(P(I_p|Ob))$$

$$Rp_{bg}(p) = -\ln(P(I_p|Bg))$$

$$K = 1 + \max_{p \in V} \sum_{q,(p,q) \in E} Sim(p, q)$$

and $P(I_p|Ob)$ and $P(I_p|Bg)$ are estimated using histograms constructed with the manually labelled superpixels.

Finally the Boykov-Kolmogorov algorithm is used to cut the graph, repeating these three steps on the search tree until the min-cut is found:

- **Growth**: active nodes recruit new 'free' nodes until all 'active' nodes are depleted. Return an augmenting path $path_{st}$ if one is found, or halt the BK algorithm, indicating the optimal cut was found.
- **Augmentation**: push the maximum amount of flow that $path_{st}$ can handle. Set the saturated nodes and their children as 'orphans'.
- **Adoption**: try to find valid parents for each 'orphans', and set them as 'free' if no valid parent is found.

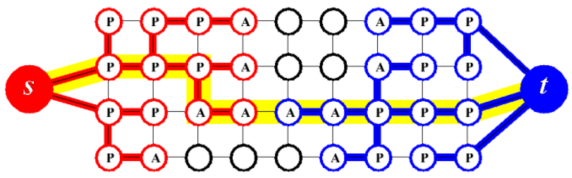


Figure 11. Example of a search tree at the end of the growth stage

3.2. Final pipeline for Mario Kart segmentation

The object segmentation and highlighting process begin with the manual labelling of object and background pixels on the input image, resulting in the creation of Figure 12. These labelled pixels serve as seeds for the subsequent steps of the algorithm. The image is then divided into superpixels, as illustrated in Figure 13. Each superpixel's contour is outlined to show the segmentation. Next, the mean LAB color value of each superpixel is computed, resulting in the filled superpixel image, depicted in Figure 14, where each superpixel is filled with its corresponding

color. Finally, the car is highlighted on the image using a masking technique that removes small components around the object. This final result is shown in Figure 15, where the car stands out prominently against the background.

Figure 12. *Seeds*

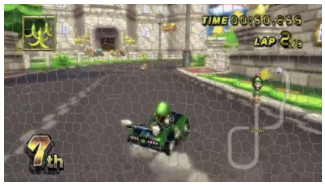


Figure 13. *Superpixels*

Figure 14. *Filled Superpixels*

Figure 15. *Detected Object*

Algorithm 2 Video Processing Pipeline with Max-Flow Algorithm

procedure VIDEOPROCESSINGPIPELINE($video\_path$, $threshold$)
  $cap \leftarrow$ open video capture with *video_path*
  $fps \leftarrow$ get video frame rate from cap
  $start\_frame \leftarrow \lfloor start\_time \times fps \rfloor$
  $end\_frame \leftarrow \lfloor end\_time \times fps \rfloor$
  set cap position to $start\_frame$
  for $i \leftarrow 0$ **to** num_frames **do**
    $object\_pixels, background\_pixels \leftarrow$ manuel_seeds()
    $SuperPixels \leftarrow$ gen_sp($frame$, $object\_pixels, background\_pixels$)
    $G, s, t \leftarrow$ gen_graph($frame$, $SuperPixels$)
    $G\_residual \leftarrow$ Max-Flow(G, s, t)
    $mask \leftarrow$ get_mask($frame$, G, $G\_residual$)
    $mask \leftarrow$ remove_small_components($mask$, threshold)
    $output\_image \leftarrow$ apply $mask$ to $frame$
    return $output\_image$
    save $frame$ as JPEG file in $output\_folder$ with appropriate naming
    print "End of video."
    break
  end for
end procedure

4. Results

We evaluated the methods on a set of 15 annotated images extracted from a 1 minute video containing various situations (normal, storm, tiny, bananas, ink...) using the Mean Intersection over Union (MIoU) metric:

$$\text{MIoU}(P, GT) = \frac{\sum_{i=1}^{N} \frac{|P_i \cap GT_i|}{|P_i \cup GT_i|}}{N}$$

where P are the predicted masks and GT are the ground truths.

As shown in Table 2, the two methods give similar results, with a slightly better performance for the Boykov-Kolmogorov method (higher MIoU and smaller variance).

Method	mIoU
Tree Based (TB)	0.58 ± 0.24
Boykov-Kolmogorov (BK)	0.59 ± 0.21

Table 2. Mean Intersection over Union (MIoU) on 15 annotated images

This can be explained by taking a look at the qualitative results in Figure 16:

- **Row 1**: In normal conditions, both methods give good results
- **Row 2**: when Luigi gets hit buy a storm and becomes tiny, the BK method struggles to detect the change in size while the TB method adapts the mask to the new conditions
- **Row 3**: In presence of a lot of green around Luigi, the green mask used in the preprocessing of TB fails to identify the right region and sometimes locates the kart in the grass
- **Row 4**: when some elements are close to Luigi, such as bananas or flames when he's drifting, and occludes a part of the car, the TB method struggles to make the distinction between all these elements and label everything as Luigi, while the BK method manages to make the difference.

5. Conclusion

In this project, we explored graph-based algorithms for car detection in Mario Kart images and videos. We implemented two distinct approaches: a tree-based method with masking strategies and the Boykov-Kolmogorov algorithm. Both methods leverage graph theory principles to segment images efficiently.

Through our experiments, we observed that the Boykov-Kolmogorov algorithm slightly outperformed the tree-based method in terms of Mean Intersection over Union (MIoU), with higher average MIoU and smaller variance. This indicates that the Boykov-Kolmogorov algorithm exhibits more robust segmentation performance across various scenarios encountered in Mario Kart gameplay.

However, both methods have their strengths and weaknesses. The tree-based method with masking strategies proved effective in adapting to changes in size, while the Boykov-Kolmogorov algorithm demonstrated better resilience to challenging scenarios involving complex backgrounds and occlusions.

References

[1] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max- flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, 2004. 2

[2] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient graph-based image segmentation. *International Journal of Computer Vision*, 59(2):167–181, 2004. 1

Figure 16. Qualitative Results