Fault-tolerant search of container codes.*

JUAN ROSELL
GABRIELA ANDREU
ALBERTO PÉREZ
Universidad Politécnica de Valencia
DISCA
Camino de Vera s/n,Valencia 46022
SPAIN

Abstract: This paper describes a method to locate and recognize container code characters. This method is aimed to make more fault-tolerant a system based on tophat transformation, segmentation algorithms, filters and classifiers. Our aim is to be able to find characters which could be shadowed or missclassified. The system has to deal with outdoor images. Our aim is to obtain a list of characters which contains all, or as much as possible, characters of the container's registration number in order to recognize them. This work is part of a higher order project whose aim is the automation of the entrance gate of a port.

Key–Words: Computer vision, segmentation, character recognition.

1 Introduction

Currently in most trading ports, gates are controlled by human inspection and manual registration. This process can be automated by means of computer vision and pattern recognition techniques. Such a process should be built by developing different techniques, such as image preprocessing, image segmentation, feature extraction and pattern classification. The process is complex because it has to deal with outdoor scenes, days with different climatology (sunny, cloudy...), changes in light conditions (day, night) and dirty or damaged containers. Digits and characters may be clear and/or dark and some may appear framed to distinguish them from the rest of the code. Interdigit distances are not either fixed. A sample image may be seen in figure 1.

A first approach to the process of code detection is presented in a previous work ([1]) and the overall process is discussed also in [2]. In these works, authors use a morphological operator called tophat ([3],[4]) to segment the images of containers. Though this method had good results, we tried to improve its performance.

Finding the suitable method to correctly process images is a difficult task, due to the lack of general methods that can be applied independently of the type

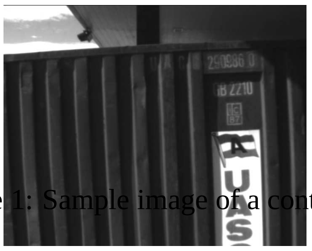
Figure 1: Sample image of a container.

of problem to be solved. As methods in computer vision always result to be ad-hoc implementations for each problem, other steps can be applied besides those traditional in image processing. In this article, we propose a method to take together the results of applying segmentation and classification to several images representing the same container, in order to improve the overall process of identifying the characters of the container code.

In our case, as mentioned before, climatology and light conditions can affect significantly our search process. The difficulties that our system has to face are: poorly contrasted simbols, shadows produced by roofs, changing shadows at different times of the day, errors produced by the process itself in any step (segmentation, classification....).

We developed an algorithm to identify characters belonging to the container code in one image. The

*Acknowledgments:This work has been partially supported by grant FEDER- CICYT DPI2003-09173-C02-01.

input data for this algorithm are grayscale images I, being $f(x, y)$ the gray level of pixel located in coordinates (x, y). The aim of this algorithm is identifying the characters of the container's code in the image.

Tests have shown, however, that taking a single image to find the code of the container is a risk because of the difficulties mentioned, thus, we try to find an algorithm that makes our system more fault-tolerant. We extend the first algorithm to work with sequences of images representing the same truck container. This way, we diminish the influence of different illumination in images and have a bigger chance of finding the complete code.

For this new algorithm, we take a sequence of grayscale images I, and apply the previous algorithm for each image in the sequence. Then try to find out from these results which the container's code is.

If we represent as $\Gamma(I)$ the set of objects found for image I, our method finds the elements from these sets with a high probability of belonging to the code of the container. Those elements appearing in most of the images with a high confidence in each image, will be more likely to be part of the container's code.

We define a sequence as a set of images representing a truck container. If we call Ψ the set of images of a sequence, $\Psi = \{I_0, I_1...I_n\}$; in order to find objects in set $\Gamma(I_0)$ that correspond to objects in images from I_1 to I_n we have to find a geometrical transformation T_{0i} from coordinates of image I_i into coordinates of image I_0. This is done by superposing objects from I_i on objects of I_0 and trying to find the location in which, the distances of the centres of objects in both images are minimum. To find this transformation T_{0i} we take advantage of the fact that the truck is moving on a flat surface, so the code will be in all images, more or less at the same height. The final set, corresponding to what we could call the segmentation of the sequence will be denoted as $SS(\Psi)$.

Authors of [5], present an investigation which is currently under development. The aim of the authors of this paper is to use the optical flow in order to shrink the area where the container code could be found and speed up the segmentation process; also, this would help us to remove noisy elements and gain in accuracy. However, this method has proved to be time consuming, and currently efforts are done in order to optimize it. In a future, we expect both investigations to merge.

We have organized the paper as follows, in section 2 we explain briefly the algorithm used to identify characters in one single image, in section 3 we describe the algorithm used to process the sequences of images, in section 4 we will describe the data we used, in section 5 we describe the experiments done; in section 6 we show the results we obtained in the experiments and in section 7 we discuss our conclusions.

2 Image processing

In this section we explain how we process images on their own. This processing is aimed to create the set $\Gamma(I)$ for each individual image I, by extracting characters of the code of the container represented in the image. We propose a process to extract and identify characters in images representing truck containers that can be divided in four phases (see figure 2).

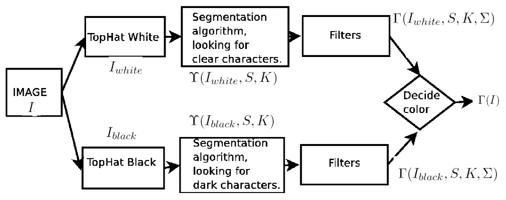
IMAGE
I
I_{white}
TopHat White
Segmentation algorithm, looking for clear characters.
$\Upsilon(I_{white}, S, K)$
Filters
$\Gamma(I_{white}, S, K, \Sigma)$
I_{black}
TopHat Black
$\Upsilon(I_{black}, S, K)$
Segmentation algorithm, looking for dark characters.
Filters
$\Gamma(I_{black}, S, K, \Sigma)$
Decide color
$\Gamma(I)$

Figure 2: The recognition process step by step.

Each image is first preprocessed by using tophat, as a result, we will have two different images, one I_{white} the result of applying white tophat and I_{black} the result of applying black tophat.

After preprocessing, images are segmented a number of times depending on the segmentation algorithm used, this way we can be sure we will have a more robust process. The result of the segmentation process is a set of objects (denoted by $\Upsilon(I_{black}, S, K)$ and $\Upsilon(I_{white}, S, K)$, where S represents a segmentation algorithm and K is a set of parameters k_i, where each k_i corresponds to an instantation of S.). These two sets, however, contain objects which are irrelevant for our search of characters, and we need to define a way to clean up these objects. The way is to implement filters. Applying a filter to $\Upsilon(I, S, K)$ means substracting a determined set of objects Δ from $\Upsilon(I, S, K)$, such that $\Delta \subseteq \Upsilon(I, S, K)$ where $\Delta = \{p \in \Upsilon(I, S, K) : \sigma(p) = 0\}$. We call the function $\sigma(p)$ the filtering function, which will be a map $\Upsilon(I, S, K) \to \{0, 1\}$. The filters are aimed to remove objects which do not fit in several constraints, for instance, shape, contrast, represent a character... If we have a family of filters Σ, we can define the final set, result of the segmentation and the filtering as:

$$\Gamma(I_{white}, S, K, \Sigma) = \{\Upsilon(I_{white}, S, K) - \bigcup \Delta_{\sigma_i \in \Sigma}\}$$

$$\Gamma(I_{black}, S, K, \Sigma) = \{\Upsilon(I_{black}, S, K) - \bigcup \Delta_{\sigma_i \in \Sigma}\}$$

We decide which is the correct color of the image by keeping the set which contains more objects with a confidence over 80%.

Unfortunately, characters can be missed in this process due to shadows in the image, due to damages in their location on the container or errors in the process. Also, brights on the container, or other elements may appear classified as objects, and this is an undesired effect. We need a method to improve the resistance of the system to such errors.

3 Sequence processing

In this section we explain how we process a sequence of images to extract the container's code from it. We first try to group objects (which are the result of applying the previous algorithm to each image in the sequence) in each image into clusters of close objects. This way, we can use properties of clusters such as the centre of mass of the cluster to faster overlap objects of a reference image with the objects in the following images.

3.1 Clustering objects

Code simbols in the container appear following a defined structure. Though we don't know prior to take the picture, what kind of lay-out the characters will have and either the amount of valid objects in the image, it is true that, after the processing, we can apply some knoweledge on the resulting objects in order to get rid of those which are not likely to be in the code, but have gone trough up to this stage of the processing. For instance, simbols belonging to the code appear close one to each other; we can use this feature to group objects in clusters and use only the valid cluster, i.e. the cluster with the code, to perform further calculations.

Given an image I, we obtain the set $\Gamma(I) = \{r_0, r_1..r_n\}$, the set of objects found in the segmentation of image I that do meet some constraints (as mentioned in the previous section). We calculate recursively the clusters of $\Gamma(I)$, denoted by κ_k, as sets containing objects $r_i \in \Gamma(I)$ whose distance must be less or equal to a given ϵ. The algorithm we use to create this clusters is as follows:

- step 1. Initialize the clusters. $\forall\, r_i \in \Gamma(I) : \kappa_i = \{r_i\}$.
- step 2. $\forall\, \kappa_i$.
- step 3. $\forall\, r_m \in \kappa_i$.
- step 4. if $\exists\, r_p : r_p \in \kappa_j \wedge distance(r_m, r_p) < \epsilon \wedge p <> m \wedge j <> m \rightarrow \kappa_i = \kappa_j \cup \kappa_i$.
- step 5. go to step 2 until there are no more changes in the sets κ_i.
- step 6. Choose the κ_i that contains the biggest amount of objects.

After running this algorithm, the cluster with the biggest amount of objects will contain the truck container code. We calculate the centre of mass of this cluster. This centre of mass will be useful in the following algorithm, when we try to match objects in one image with objects in the other.

3.2 Sequence processing

Image sequence processing is a post-process method that we apply to be able to improve performance. One of the drawbacks of our system is that it can segment characters but then fail when classifying them, or a character can be missed because of a shadow; losing accuracy. In an effort to avoid this problem, we take advantage of the fact that we can take more than one picture per container. Some examples of sequences of images can be seen in figure 3.

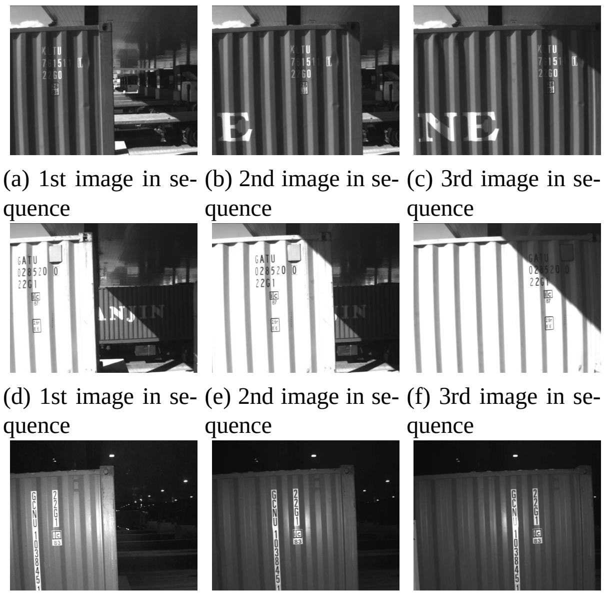
(a) 1st image in sequence (b) 2nd image in sequence (c) 3rd image in sequence
(d) 1st image in sequence (e) 2nd image in sequence (f) 3rd image in sequence
(g) 1st image in sequence (h) 2nd image in sequence (i) 3rd image in sequence

Figure 3: Samples of sequences of a container, taken at different hours of the day.

We develop an algorithm to segment and classify characters in each image and then try to extract from the complete set the characters of the container. We have the advantage that container codes will always, more or less, appear at the same height in pictures, but moved some pixels left or right, depending how the lorry was moving. This gave us the idea of trying to find a transformation that could move objects appearing in successive images to the location of objects in the first one of a sequence. Giving us the possibility of correcting errors such as:

1 Loss of characters because of shadows.

2 Irrelevant objects that appear classified as characters. They will be called from now on noisy characters.

3 Missclasifed characters.

For the first problem, shadows, we trust that, by taking several pictures of the container, we will be able to find the characters of the code in most of the cases. If, eventually, any character is shadowed in any image, it is likely we have found it in previous pictures, so, by checking all them together, we can have it in the final solution. The second issue, is easier to achieve, irrelevant objects classified as characters will not appear in more than one or two images; thus, if we set the number of images to 3 or more we can get rid of all of them in the final solution. The last problem has to do with characters which are missclasified due to, for instance, a bad segmentation of the image. As in the first case, if such a problem appears, we can still know which is the correct one by keeping only characters which appeared in more images.

The algorithm takes a set of images representing the same container and processes them as explained later. For each image we calculate the centre of mass as explained in the previous point. Each object p in the solution set $SS(\Psi)$ has an associated counter to know how many times it appears in different images (accessed as $p.counter$ in the algorithm). Also, it has a confidence value calculated as the average of its confidence in all pictures where it has appeared (accessed as $p.confidence$). An auxiliar function $overlap(p, q)$ is used, which returns $TRUE$ in case objects p and q overlap one on each other. Function $simbol(q)$ returns the simbol associated to this object by the classifier.

A previous step before applying the algorithm is not taking into account all images for which the colour was wrongly detected. This step just takes an average of pictures voting characters are clear and those voting characters are dark; those which lose the voting are discarded.

Formally, the algorithm is defined as follows:

1 Given a set of images $\Psi = \{I_0, I_1, I_2, ...I_n\}$, we take a reference image I_0, that will contain the entire code.

2 $SS(\Psi) = \Gamma(I_0)$.

3 Calculate the centre of mass of $\Gamma(I_0)$.

4 $\forall\, I_i \in \Psi, i > 0$.

4.1 Calculate the centre of mass of $\Gamma(I_i)$.

4.2 Find a geometrical translation T, from $\Gamma(I_i)$ into $\Gamma(I_0)$ such that, by applying T we are sure that the most of the objects of $\Gamma(I_i)$ are overlapping objects of $\Gamma(I_0)$.

4.3 $\forall\, p \in SS(\Psi), p.counter = p.counter + 1 \iff \exists\, q \in \Gamma(I_i) : overlap(p, q) = 1 \wedge simbol(p) = simbol(q)$

4.4 $SS(\Psi) = SS(\Psi) \bigcup \{q \in \Gamma(I_i) : \nexists\, p \in SS(\Psi) : overlap(p, q)$

5 Go to step 3.

6 Calculate the average confidence as: $\forall\, p \in SS(\Psi)$: $p.confidence = \frac{\sum(q_i.confidence)}{p.counter}, \forall q_i \in I_j, j <> 0 \wedge overlap(p, q)$

It must be remarked than, in the case any $I_i \in \Psi$ is empty or does not contain the suitable objects, it is removed from the set and not used in the algorithm. For instance, an image could contain no object.

We implemented the set $SS(\Psi)$ with a hash table; in order to store information about which objects appear, which overlap, which are new and the classifier confidence they have. With these structure we have objects ordered and easily reachable. Our hash table stores characters from 'A' to 'Z' and from '0' to '9'.

4 DATA

We used 51 real images to perform our experiments, corresponding to 17 containers. These images represent truck containers and have a size of 720×574 pixels in gray levels. They were acquired under real conditions in the admission gate of the port of Valencia; in several days under different light conditions. Digits and characters can be clear or dark and they appear in both plain and non-plain surfaces. We selected randomly a set from a large amount of pictures and assured all variability was represented in this set of pictures (sunny or cloudy days, daytime or nighttime, damaged containers...).

We used pictures which shew the complete code. This is not an important constraint, if we consider that we can take a sequence of pictures as big as we want; and that, we can set the camera wherever it is more useful. But it makes it easy for the algorithm to detect when most objects are fitted.

5 EXPERIMENTS

We applied tophat with 6 repetitions either for clear as for dark characters. We used LAT ([6]) in our experiments, in [7] we show a comparison of other algorithms' performance. It was applied several times to each image, varying its parameters in order to cover all possible ilumination situations. Also, it had to be executed twice, once seeking for clear characters and again, seeking for dark characters. This was done this way, because segmentation algorithms were not provided with concrete information about each image (illumination, number of characters...).

A k-Nearest Neighbours classifier was trained using 654 real images of trucks what means a total of 9810 characters. We gathered 288 features representing gray levels from each normalized character (12×24 pixels). We applied PCA ([8]) to reduce the dimesionality of data, and at the end, only 60 features were used. When classifying objects with k-NN neighbours, we used $k = 3$. The contrast value was calculated experimentally and we tried to set it low enough as to not lose shady characters but also high enough to filter as much noise as possible.

We check whether objects have been correctly found in a similar way as proporse in [7]. The difference is that now, we are not only considering the inclusion boxes of the objects but also if the label assigned to them is correct or not. Any object that meets both conditions is considered a hit. We compare the result of applying the algorithm to a sequence of images with the result of each image on their own.

In the experiments we took two different results, on one hand, we have results for each individual image. We counted how many valid objects were found and how many irrelevant objects had gone successfuly through filters. On the other side, the other result we obtained was the number of valid and irrevelant objects found by taking the entire sequence of images and analyzing it with the algorithm.

6 RESULTS

In table 1, results are shown for some sequences. Under the column *objects*, we show the number of valid objects in the container, that is, the number of characters in the container's code. Under columns *image0* to *image3*, we show results for each image on its own, first number stands for the number of valid objects found in this image, and the second number stands for the number of noisy objects that could not be removed; the addition of both numbers gives the amount of objects found in the picture. In case a sequence had less than 4 images, we filled the corresponding cells with dashes. The last column, gives the same information after applying the sequence algorithm.

In each row we show results for each image of a sequence and the result for the sequence by applying the sequence algorithm. For instance, first row corresponds to first sequence. For first image of this sequence, we obtain 21 objects, 15 of them correspond to the code and the others are noisy objects, that is, 30% of the objects are noisy objects; for the second image we obtain again 15 correct objects and 3 noisy objects; on the other side, by applying the sequence algorithm we obtain 15 objects that correspond exactly to the container's code. In sequence 8 however, the sequence algorithm loses 3 characters of the solution, and in any of the images of the sequence on their own we would have more hits, but, on the other side, the result of the algorithm has no noisy object and it is difficult to choose which picture in any sequence would have the best results.

Sequence	*Objects*	Img0		Img1		Img2		Process	
		H	*N*	*H*	*N*	*H*	*N*	*H*	*N*
1	*15*	15	6	15	3	11	0	15	0
2	*17*	14	5	12	3	11	6	15	1
3	*15*	12	22	12	10	11	5	12	0
4	*15*	14	2	14	2	4	0	11	1
5	*15*	15	5	15	10	9	6	13	3
6	*15*	10	10	13	5	14	4	11	3
7	*15*	13	16	10	9	0	5	5	0
8	*15*	13	3	15	0	14	1	12	0
9	*15*	14	6	14	6	11	4	13	2
10	*15*	15	6	15	4	11	3	15	1
11	*16*	13	4	15	4	12	2	11	2
12	*15*	10	13	11	3	11	0	11	0
13	*17*	12	10	14	10	11	7	12	4
14	*15*	15	16	15	4	-	-	15	0
15	*15*	14	13	14	12	13	9	14	2
16	*17*	16	1	16	9	-	-	15	1
17	*15*	15	18	15	13	-	-	15	2

Table 1: Comparison of results searching in one picture of the sequence or in the complete sequence as a unit.

Though using the sequence algorithm may be seen as adding a time penalty to the execution, it may bee seen that, it is impossible to find all objects just with one image. In addtion, the process itself can be implemented in such a way, that time penalty does not influence the final result. It could, for instance, be divided into two different processes, one taking pictures, segmenting them and locating objects on them; and the other one, taking these objects and comparing them with the previous following the sequence algorithm presented now. These two processes follow the producer-consumer paradigm and that can be easily parallelized.

7 CONCLUSIONS

We present a method for detecting container's registration number by identifying simbols in different images in a sequence of the same container passing in front of a camera. This process improves significantly the amount of container's codes that we can identify.

We used images selected randomly from a large set of real images. Our effort was driven by the fact that we wanted to adjust the process, in such a way, that we did not miss any character (or the lowest possible amount) in the registration number. Our evaluation of the different solutions then penalties the lose of characters.

Further efforts will focus on improving execution times by parallelizing the execution of the algorithm to take advantage that it can be divided into two independent execution flows, and being able to improve the accuracy on detecting characters with low contrast, by better characterizing noise in the classifier.

References:

[1] Salvador, I., Andreu, G., Pérez, A.: Detection of identifier codes in containers. Proc. SNRFAI-2001. Castellón, Spain. May de 2001. **1** (2001) 119–124

[2] Salvador, I., Andreu, G., Pérez, A.: Preprocessing and recognition of characters in container codes. ICPR2002, Canada, 2002 (2002)

[3] Woods, R.G.R.: Threshold selection using a minimal histogram entropy difference. Addison-Wesley (1993)

[4] Soille, P.: Morphological image analysis: Principles and applications. Springer Verlag (1999)

[5] Atienza, V., Rodas, A., Andreu, G., Pérez, A.: Optical flow-based segmentation of containers for automatic code recognition. Lecture Notes in Computer Science **3686** (2005) 636–645

[6] Kirby, R.L., Rosenfeld, A.: A note on the use of (gray level, local average gray level) space as an aid in threshold selection. IEEE Transactions on Systems, Man and Cybernetics SMC-9 (1979) 860–864

[7] Rosell, J., Pérez, A., Andreu, G.: Segmentation algorithms for extraction of identifier codes in containers. Int. Conf. (VISAPP-2006), Portugal (2006) 375–380

[8] Fukunaga, K.: Statistical Pattern Recognition. Second edition edn. Academic Press (1990)