Available online at www.sciencedirect.com
ScienceDirect
Journal of Visual Languages and Computing
18 (2007) 504–512

Journal of Visual Languages & Computing

www.elsevier.com/locate/jvlc

Application of a transparent interface methodology to image processing

Luigi Cinque[a,*], Sergio Sellers Cañizares[a], Steven Tanimoto[b]

[a]*University of Rome, "La Sapienza", Italy*
[b]*University of Washington, Seattle, USA*

This paper is dedicated to Professor Stefano Levialdi.[1]

Abstract

Many software engineering projects involve a significant design component in which an algorithm must be formulated as a sequence of processing steps that meets a solution criterion. As the problems tackled become more complex, it becomes increasingly important to create and use tools that help designers understand and manage the design process. We demonstrate the use of design tool called T-STAR in the domain of image processing, in which a toolkit called the TRAnsparent Image Problem Solving Environment (TRAIPSE) is extended to solve face-recognition problems. Key features of TRAIPSE are its visual interface to the space of partial image processing algorithms and its support for automatic assistance in exploring the space. The specific application we present is the analysis of human face images.

Keywords: Face detection; Image processing; Tree structure; State-space search; Graphical interface; Exploratory data processing; Image operator

*Corresponding author. Tel.: +39 06 499 18508; fax: +39 06 8419188.
E-mail address: cinque@di.uniromal.it (L. Cinque).

[1]From his early work in image processing languages (PIXAL) to his later work on user interfaces, he has provided inspiration to us to design highly usable technology for image processing. His work in the area of shrinking operators is closely related to the morphological operators used in this paper. His involvement with pyramid image processing during the 1980s helped promote the multiresolution image processing methodology that informs one part of the TRAIPSE system presented here.

1045-926X/$ - see front matter
doi:10.1016/j.jvlc.2007.08.008

1. Introduction

The complexity of many software systems has been increasing over the years, as a result of increasing expectations, feature creep, and the availability of new software techniques. Along with this growth, the challenge of designing effective systems has also intensified. The need for powerful tools that help designers understand the systems they are designing has never been greater.

One approach to improving human understanding of systems being designed is wrapped up in the term "transparent interfaces." A transparent interface is one that reveals some of the inner workings of the system behind it. A transparent interface fosters understanding of the system by helping designers and users better see the processes that are manifested by the system. Whereas a system with a "standard" interface allows the inputs and outputs of the system to be monitored, the transparent version provides additional visualizations of states and internal dataflows or relationships in such a way that the human can more easily verify the correctness and appropriateness of the system's computations.

There are many approaches to design. In the study we report here, one particular approach used: problem solving with state-space search. A rationalization for this method has been given by late Simon [1]. Our goal has been to investigate the use of transparency using this approach and not (so far) to create design software for professionals.

In the rest of the paper, we describe some of the challenges and opportunities associated with applying the methodology in the field of image-processing.

2. Designing image-processing algorithms

Image-processing applications typically involve several distinct steps of processing: preprocessing of the pixels; segmentation of the image into regions or edges and edge segments; the extraction of features such as shape, color, texture, or local geometry; construction of a structural description; and classification.

In each of these steps, there are many alternative operations and/or parameterizations of the operations to be applied. The choices depend upon the application.

The low-pass filtering that might be appropriate for video-frame analysis might not be appropriate for localizing a machine part on a conveyor belt. If segmentation is performed with thresholding, then the threshold for cancer-cell isolation is likely to be different from that for finding and counting coins or detecting animals.

One of the challenges of designing an image-processing application is the management and evaluation of alternative sequences of operations. While this is not unique to image-processing, it does mean that software methodologies tend to be somewhat different in this domain [2]. We describe, in following sections, a toolkit called TRAnsparent Image Problem Solving Environment (TRAIPSE) that assists designers in formulating, testing, and keeping track of alternative sequences.

Image-processing has a particularly good aspect in relation to transparent interfaces: images are visual objects that can easily be displayed, and so they do not require complex interpretive mappings, as might, for example, networks that would have to be laid out according to some possibly arbitrary scheme. Our TRAIPSE system takes advantage of the ready displayability of images, using processed images as visual proxies for the operator sequences that produce them.

3. The T-STAR software framework

The TRAIPSE image-processing toolkit that we will describe in more detail later is built on top of a software platform called T-STAR. This platform, the Transparent STate-space search ARchitecture, provides an application-independent framework in which to build interactive problem-solving programs. It includes a method for representing trees of states derived from starting states, and offers both a tree-rendering algorithm and methods for handling user interaction with the trees. The implementation of T-STAR uses the Python language and the Tkinter graphical user interface toolkit [3].

The acronym T-STAR (T*) makes an analogy with the A-Star (A*) algorithm framework for intelligent problem solving. Both methods add power to state-space search: A* provides a framework in which to incorporate heuristic evaluation functions, while T* offers transparency and human interaction to the search.

To illustrate the T-STAR framework independent of our image-processing application, we use a traditional blocks-world planning example [4]. In the initial state of the puzzle, there are three blocks, labeled A, B, and C. Block C is on top of B, and B is on top of A, which is directly on the table. The objective is to find a sequence of moves that results in A on top of B and B on top of C. A move involves picking up one block and putting it down either on top of another block or directly on the table. Fig. 1 presents a screen shot of T-STAR configured for this problem. It shows a tree of states that includes both the initial state (at the root) and the goal state. The T-STAR software displays the tree, and it allows

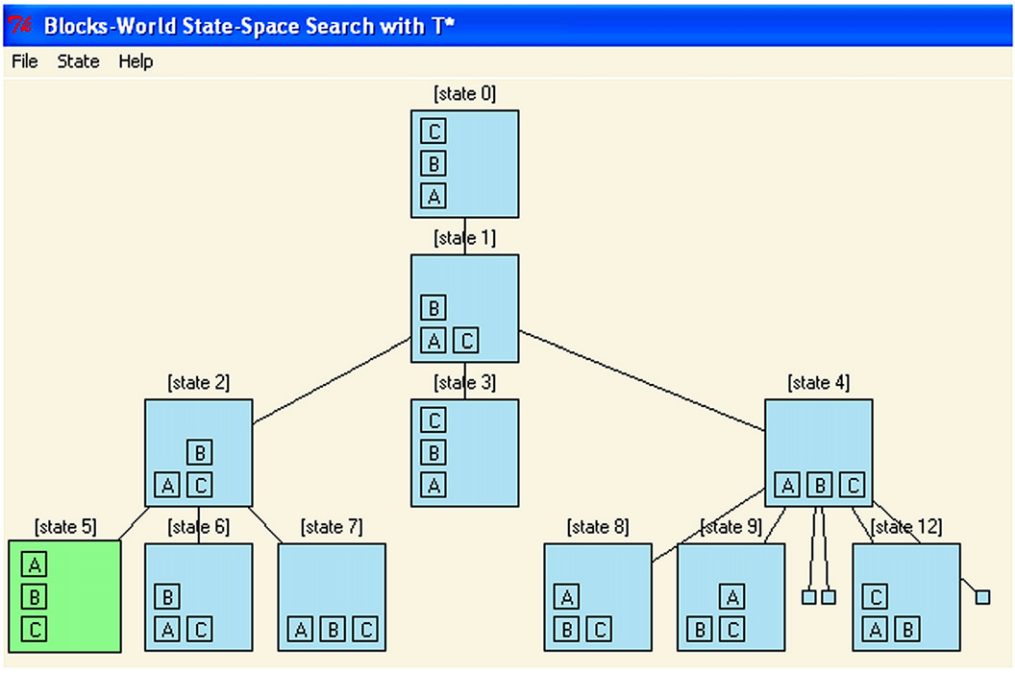

Fig. 1. A display of the T-STAR interface for the blocks-world planning problem. The initial state is at the root of the tree and (in this case) the leftmost leaf state satisfies the goal criterion.

the user to apply operators and grow the tree and to control the way the tree is displayed. Additional details about T-STAR were reported elsewhere [5].

4. Adaptation to image-processing

In this section, we introduce a toolkit that adapts the T-STAR framework for the domain of image-processing. The toolkit is called TRAIPSE. It provides a specialization of T-STARs state class, as well as specific operators for transforming images.

T-STAR does not pose any constraints on what a state can be or how it is represented. Similarly, TRAIPSE does not limit what can be a state. However, the current implementation of TRAIPSE provides display methods only for states represented by single images, and its operators consist primarily of single-argument functions that map an image to an image. These display methods and operators could be supplemented with methods that work with more varied and complex data objects. Fig. 2 shows how states correspond to images in TRAIPSE.

One feature of TRAIPSE that is influenced by the nature of digital images is that it automatically reduces the resolution of the source image for use in creating the display of the tree of states visited. This helps manage screen real estate, and it can reduce the time required to explore a given set of states. However, the user is provided with a method to demand a full-resolution display of any state at will. This potentially large image is immediately computed using the same sequence of operators used to create the small version, and it is displayed in a separate window. This general principle of details on demand is applicable to other domains, but it may require more complex rendering mechanisms than in the case of images, in which it is so natural.

The current implementation of TRAIPSE provides only limited facilities for working with parameterized transformations such as an image thresholding function that takes a scalar threshold value as an argument as well as the input image. The simplest way to use such functions now is to create a set of sample instances with a range of threshold values, and to consider each instance to be a separate operator. Another approach is to build a parameter-selection mechanism into the operator so that it first computes a suitable value (e.g., threshold) by examining the image, and then it applies the function. Future work will consider more advanced approaches.

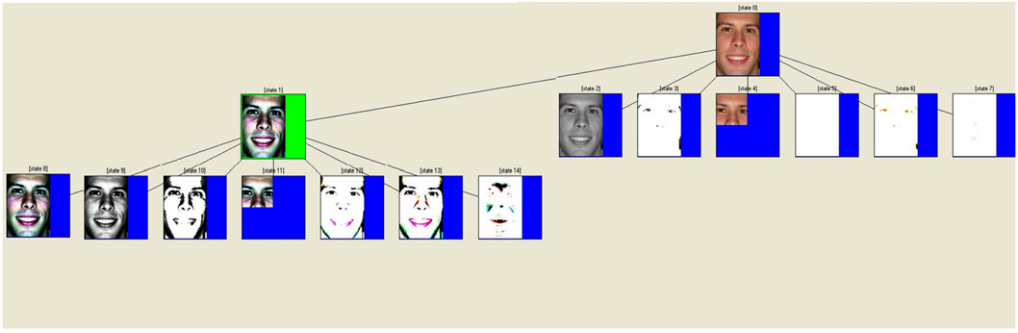

Fig. 2. An illustration of a three-level tree of states constructed using face-image-processing operators in TRAIPSE.

5. Application to face recognition

In this section, we describe a detailed example in which we used TRAIPSE to implement and present the design of a human-face image analysis system. The face recognition system has the goal to determine in a picture where the eyes, nose, and mouth are. It allows the user to introduce a picture and apply any sequence of operators on it. The T* methodology allows the user to easily compare the effect of each operator, so the user instead of having to choose one possibility, can explore many of them at the same time and easily compare what can be obtained from each one of them. Then, the tree can be applied to other images to check the effectiveness of that particular solution as a general solution. Thanks to this functionality, it was possible to quickly establish a common path to extract each one of the features valid for all the studied images. Still, there were some challenges that could not be foreseen, and so more experimentation was performed to settle the best combinations of operators, as will be seen later in this section.

5.1. Operators

The following operators are provided in the package TRAIPSE_FD, which is the version of TRAIPSE we created for face processing. Note that there is a mix of general and application-specific functions. We include the complete set so that the reader can appreciate characteristics of the mix.

1. Equalize: equalizes the image.
2. Gray: converts the image into gray-scale.
3. Low-intensity regions: gets the low-intensity regions by computing HSV and thresholding on the V value to get a mask. Then the mask and a gray-valued image are ANDed and the result is returned.
4. Eyes region: using knowledge of where the eyes are on a face, it extracts the upper half of the picture and discards 15% on each side.
5. threshold-20: makes a threshold, keeping the values lower than 20.
6. threshold-40: makes a threshold, keeping the values lower than 40.
7. erosion_8_con: erodes the image using an 8-connected structuring element.
8. erosion_4_con: erodes the image using a 4-connected structuring element.
9. dilation_8_con: dilates of the image using an 8-connected structuring element.
10. dilation_4_con: dilates of the image using a 4-connected structuring element.
11. getEyes: applies the eye detection algorithm explained below.
12. paintEyes: paints on the original image the eyes' coordinates found by the operator getEyes.
13. noseMouthRegion: extracts the nose and mouth regions. Knowing where the eyes are, the nose and mouth are on the rectangle that is taken from these two points to the bottom of the image.
14. getBrightRegions: looks at each pixel and if its value is greater than or equal to 245 on a 0–255 scale, it is set to 0, and if it is less than 245 it is set to 255. So this way we get an image that has black regions where there were bright regions and the rest is white.
15. noseRegion: gets the upper half from the noseMouth regions.
16. getNose: applies a nose detection algorithm similar to the eye detection algorithm but considering just one point, so all the checks done for the eyes are not needed.

17. paintNose: paints on the original image the nose coordinates found by the operator getNose.
18. mouthRegion: gets the lower half from the noseMouth regions.

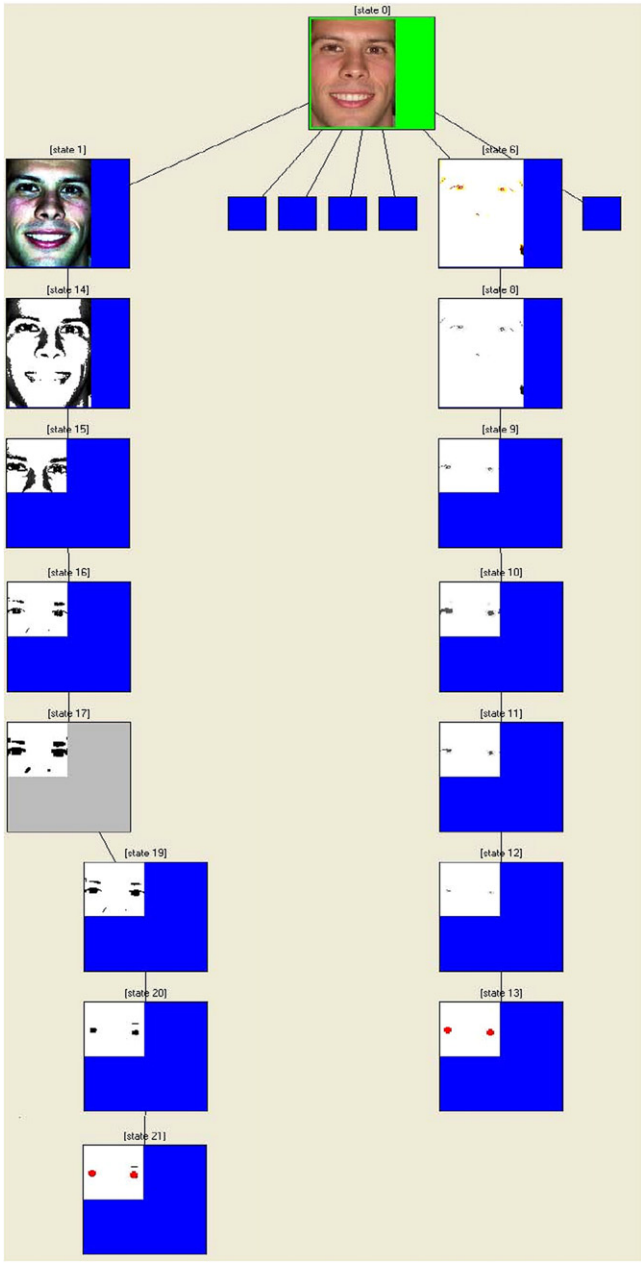

Fig. 3. Illustration of eye localization. Two alternative sequences can be easily compared using the T* framework.

19. getMouth: applies a mouth detection algorithm similar to the eye detection algorithm but considering just one point, so all the checks done for the eyes are not needed.
20. paintMouth: paints on the original image the mouth coordinates found by the operator getMouth.

Detection of the eyes, nose, and mouth involve similar processing sequences, with variations in the details. As an illustration of the methodology, we describe the analysis for the eyes. Fig. 3 shows two alternative sequences in the development of the algorithm for eye localization. The sequence on the left involves equalization, grayscale conversion, thresholding, morphological filtering, and an eyes-specific localization operator. On the other hand, the sequence on the right, which was effective in this example, but less effective with additional examples, begins with thresholding.

Once we have the two rectangles around the eyes, we calculate the middle point of each one of them. These points give the coordinates of each of the eyes. Then these coordinates are translated into the whole original image. Fig. 4 shows the result of applying the algorithm in the cropped image we had obtained on the previous stage.

In Fig. 5, we see the eyes once the coordinates have been translated to the original image.

Finally, Fig. 6 illustrates the result once all the features have been extracted.

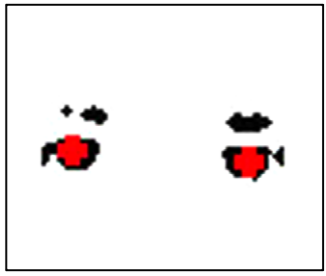

Fig. 4. Eye indicators positioned on subimage.

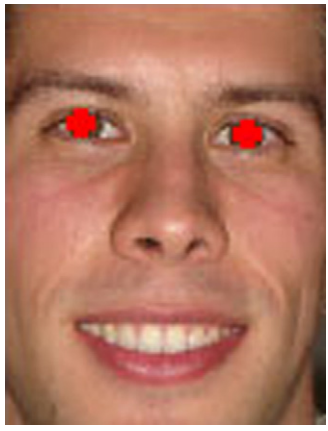

Fig. 5. Eye indicators positioned on original image.

Fig. 6. Eye, nose, and mouth indicators positioned on original image.

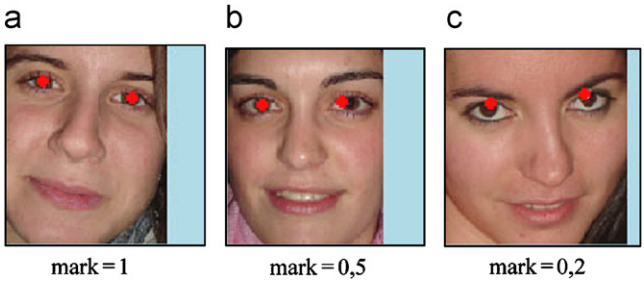

Fig. 7. Scoring scheme for eye extraction. (a) Correct positioning: mark = 1; (b) one eye slightly off: mark = 0.5; and (c) both eyes slightly off: mark = 0.2.

5.2. Experiments

For the extraction of each one of the features mentioned above there was a clear common path; the doubt was around the correct number of erosions and dilations needed to prepare for the feature detection operators getEyes, getNose, and getMouth. So the experimentation has been focused on finding the combination of erosions and dilations that set the best scenario to apply the three developed feature extraction operators for the majority of the pictures.

According to how accurate the solution was, a mark was given. It was 1 if the solution was just the exact point it should be, 0 if it was out of the feature it had to detect, or a value between 0 and 1 if it was still inside the desired feature but not in the exact point it was intended to be. With the eyes, if one eye was well-detected and the other one was still OK but not accurate, it was given a mark between 0.5 and 0.9. If both of them were OK but not accurate, it was given a mark between 0.1 and 0.4. If neither of them was at least OK, the mark was immediately set to 0. Fig. 7 gives some examples of the marks.

6. Discussion

We have seen how T-STAR and TRAIPSE allow one to easily find solutions to a problem such as face analysis. One begins with a specific example and then validates the

tree paths as general solutions by applying the same sequences on other examples. The paths typically need to be fine-tuned as the set of examples is enlarged.

These experiments would have been difficult had they been approached with another methodology than the transparent TRAIPSE one. Initially, the path to the solution was unclear, as all possibilities seemed equally correct, and the number of possible combinations made it impossible to take a brute-force approach to the problem. In addition to this, we would not so quickly have found a clear common section (the steps previous to the experimentations) if we were not working with the transparent framework, because it allowed us to easily compare the effects on alternative branches. And even though sometimes we did not clearly foresee the effects of some of the operators, when we saw their effects or the effects of their combinations with other operators, it was a useful stimulus in developing new ideas, trying new paths or even developing new operators.

As this process required human validation, if we had tried to approach a solution with "traditional programming", we would have had to generate 3360 pictures just to evaluate the possibilities for the localization of eyes. And there would be only the original picture and the final picture; we would not have had the possibility of watching what was happening in the middle. Knowing the intermediate states, it is possible to think of new possibilities. The process is like debugging. As it is possible to extend a whole tree, if you get the expected result, the problem is over and you can go on to the next image. However, you do not have to watch the intermediate states if you do not want to. But if you have not got the expected result, you can watch the nodes of the tree that made the attempt go wrong.

The framework offers a new way to address state-space based problems and a way to easily compare the possible combinations of operators. It creates a new concept of algorithm, which is dynamically created by the user with the possibility to be adapted for each particular image.

It is notable that although these experiments have focused on face feature detection, the system used for it (T-STAR and TRAIPSE) allows the application of the mentioned operators for any kind of image, as long as they make sense. So a guideline would be trying always to build the operators as generically as possible, limiting as much as possible the use of domain specific operators, so when facing a new problem you can rely on that library of operators to make the first steps. Then, while watching the effects of existing operators, the developer can get ideas for possible new combinations and for new operators.

Acknowledgments

We would like to thank P. Bottoni, A. Malizia and W. Winn for their constructive comments about the T-STAR software. Preparation of this paper was funded in part by NSF Grant CNS-0613550.

References

[1] H. Simon, The Sciences of the Artificial, third ed., Cambridge MA, MIT Press, 1996.
[2] S. Tanimoto, Advances in software engineering and their relations to pattern recognition and image processing, Pattern Recognition 15 (3) (1982) 113–120.
[3] F. Lundh, Introduction to Tkinter, 1999, 〈http://python.net/crew/fredrik/tkintro/Introduction.htm〉.
[4] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, second ed., Prentice-Hall, Eaglewood Cliffs, NJ, 2003.
[5] S. Tanimoto, S. Levialdi, A transparent interface to state-space search programs, in: Proceedings of ACM SoftVis 2006, Brighton, UK, 2006.