Example-Driven Graphics Recognition

Liu Wenyin

Dept of Computer Science, City University of Hong Kong, Hong Kong SAR, PR China
csliuwy@cityu.edu.hk

Abstract. An example-driven graphics recognition scheme is presented, which is an extension of the generic graphics recognition algorithm we presented years ago. The key idea is that, interactively, the user can specify one or more examples of one type of graphic objects in an engineering drawing image, an the system then learn the constraint rules among the components in this type of graphic objects and recognize similar objects in the same drawing or similar drawings by matching the constraint rules. Preliminary experiments have shown that this is a promising way for interactive graphics recognition.

Keywords: graphics recognition, rule-based approach, case-based reasoning, SSPR.

1 Introduction

As a pattern recognition problem, graphics recognition requires that each graphic pattern be known, analyzed, defined, and represented prior to the recognition (matching with those patterns in the image) process. This is especially true for those approaches (e.g., Neural Network based approaches [1]) that require large sets of pattern samples for pre-training. Similarly, the syntaxes and structures of the patterns should also be pre-defined before recognition in syntactic and structural approaches (e.g., [2]) and the knowledge about the patterns should also be pre-acquired before recognition in knowledge-based approaches (e.g., [2] and [8]). For example, if the task is to recognize lines from images, the attributes or features of the line patterns should be analyzed such that appropriate representations and algorithm can be designed and implemented. Through pattern analysis we know that line patterns in the image space correspond to peaks in the Hough transformed space. Therefore, these peaks are pre-defined features for detecting lines in the Hough Transform based approaches [4]. Usually, the features, syntaxes, and other knowledge about the patterns, e.g., the graphic geometry, are hard-coded in the recognition algorithms. Hence, currently, each graphics recognition algorithm only deals with a limited set of specific, known graphic patterns, e.g., dimension-sets [2], shafts [8]. Once implemented and incorporated in a graphics recognition system, these features, syntaxes, and knowledge cannot be changed. The system can only be used for these pre-defined patterns and cannot be applied to other previously unknown patterns or new patterns. In order to recognize new patterns, the same analysis-design process should be repeated. Hence, these approaches are not flexible to changeable environments.

T. Caelli et al. (Eds.): SSPR&SPR 2002, LNCS 2396, pp. 168-176, 2002.

It is fine to hard-code for those very common graphic primitives, e.g., lines, arcs, characters in the recognition algorithms. However, there are many different classes of graphic symbols/patterns or higher level graphic objects in many different domains of drawings. Even within a single domain, e.g., mechanical drawings or architectural drawings, the number of symbols or commonly re-usable component patterns can be very large. Hence, it is non-realistic to hard-code all of them in the recognition algorithms. A generic method that can automatic learn new or updated patterns for run-time or just-in-time recognition is strongly desired.

In this paper, we propose a new scheme of graphics recognition, which is example-driven. That is, the user provides to the system with a selected set of representative examples for the graphic pattern to be recognized and the system learns the knowledge (attributes/constraints of the components, etc.) about the pattern from these examples and recognizes all graphic patterns similar (in terms of those attributes and constraints) to these examples. In this way, the system does not need to know any pre-defined patterns before the system is built. The knowledge of patterns can be learnt at run-time. The underlying support for this example-driven scheme is the generic graphic recognition algorithm (GGRA) [6, 7] implemented using a rule-based approach. Due to the vector-based nature of GGRA, pre-segmentation of graphic patterns is not required. In this paper, we first briefly explain GGRA and then present the rule-based framework for graphics recognition. Preliminary experiments and concluding remarks are also presented.

2 The Generic Graphics Recognition Algorithm

The Generic Graphics Recognition Algorithm (GGRA) [6, 7] was proposed and constructed based on the observation that all graphic patterns consist of multiple components satisfying (subject to) a set of constraints. For instance, a rectangle comprises a closed sequence of four connected straight lines with four right angles at the four connection points. Even a solid line may consist of several connected and collinear vectorized line fragments.

Most existing graphics recognition algorithms cluster all the potential constituent components at once, while the graphics attributes are determined later. This blind search procedure usually introduces inaccuracies in the grouping of the components, which ultimately account for inaccurate graphics recognition. Moreover, each class of graphic objects requires a particular detection algorithm. In spite of many graphics recognition algorithms reported, no research report has yet proposed to detect all classes of graphics by a generic, unifying algorithm.

The Generic Graphics Recognition Algorithm (GGRA) [6, 7] we previously proposed is a more flexible and adaptive scheme that constantly checks the graphic object's syntax rules and updates the object's parameters while grouping its components. This generic graphics recognition methodology takes vectors as input. These vectors can be produced by any vectorization algorithm, in particular our sparse pixel vectorization algorithm (SPV) [5]. As shown in **Fig. 1**, which is the C++ code illustration of the framework, GGRA (in **runWith**(…)) consists of two main phases

based on the hypothesis-and-test paradigm. The first step is the hypothesis generation, in which the existence of a graphic object of the class being detected is assumed by finding its first key component from the graphics database (by calling prm = gobj->**find FirstComponentFrom**(gdb)). The second step is the hypothesis test, in which the presence of such graphic object is proved by successfully constructing it from its first key component and serially extending it to its other components. In the second step, an empty graphic object is first filled with the first key component found in the first step (by calling gobj->**fillWith**(prm)). The graphic object is further extended as far as possible in all possible directions (d<=gobj->**numOfExtensionDirections**()) in the extension process—a stepwise recovery of its other components (**extend**(d, gdb)). After the current graphic object is extended to all extension directions, a final credibility test (gobj->**isCredible**()) prevents the inclusion of false positives due to accumulative error. If the extended graphic object passes the test, it is recognized successfully and added to the graphics database (gdb), otherwise all found components are rejected as being parts of the anticipated object which should be deleted. Regardless of whether the test is successful or not, the recognition process proceeds to find the next key component, which is used to start a new hypothesis test.

```
template <class AGraphicClass>
class DetectorOf
{
        DetectorOf() {}
        void runWith(GraphicDataBase& gdb) {
          while (1) {
            AGraphicClass* gobj = new AGraphicClass();
            Primitive* prm = gobj->findFirstComponentFrom(gdb);
            if (prm == null) return;
            if (!gobj->fillWith(prm)) continue;
            for (int d=0;d<=gobj->numOfExtensionDirections(); d++)
              while (gobj->extend(d, gdb));
            if (!gobj->isCredible()) delete gobj;
            else gobj->addTo(gdb)
          }
        }
        boolean extend(int direction, GraphicDataBase& gdb) {
          Area area = extensionArea(direction);
          PrimitiveArray& candidates = gdb.search(area);
          for (int i=0; i < candidates.getSize(); i++) {
            if (!extensible(candidates[i])) continue;
            updateWith(candidates[i]);
            break;
          }
          if (i < candidates.getSize()) return true;
          return false;
        }
};
```

Fig. 1. Outline of the C++ implementation of GGRA

In the extension procedure (**extend**(…)), an extension area is first defined at the current extension direction according to the object's current state, e.g., the most recently found component (by calling area = **extensionArea**(direction)). All

candidates of possible components that are found in this area and pass the candidacy test are then inserted into the candidate list, sorted by their nearest distance to the current graphic object being extended (by calling candidates = gdb.**search**(area)). The nearest candidate undergoes the extendibility test (**extensible**(candidates[i])). If it passes the test, the current graphic object is extended to include it (by calling **updateWith**(candidates[i])). Otherwise, the next nearest candidate is taken for the extendibility test, until some candidate passes the test. If no candidate passes the test, the extension process stops. If the graphic object is successfully extended to a new component, the extension process is iterated with the object's updated state.

Since in the first phase we find the first key component of the object to be recognized, making the correct hypothesis is crucial, and should be properly constrained. If it is over-constrained, only few objects will be found, while under-constraining it would lead to too many false alarms. If no key component can be found, no more objects of the type being sought can be detected and the recognition process (**runWith**(…)) stops.

The generic object recognition algorithm can be instantiated for the recognition process of a variety of objects. Especially, GGRA has been successfully applied to detection of various types of lines [9], text, arrowheads, leaders, and dimension-sets, hatched areas. However, in these applications, the rules (defining the graphic classes) are hard-coded in the overridden member functions of their classes. In this paper, GGRA is further generalized and applied to detection of user-defined types of graphic objects by implementing the abstract functions (in bold fonts in **Fig. 1**) in GGRA using the rule-based approach.

3 The Rule-Based Graphics Recognition Framework

Due to GGRA's generalized and stepwise nature, it is a good candidate to serve as the basis for the recognition framework for the graphic classes that are previously unknown but specified or defined at run-time. In this paper, we extend GGRA to such recognition framework by implementing the abstract functions (in bold fonts in **Fig. 1**) in GGRA using the rule-based approach. That is, the rule-based algorithms (and the code) in these functions are the same for all graphics classes. Each graphics class is specified using a set of rules (attributes and constraints), which are stored in the knowledge database. In the recognition process for a particular class, its rules are taken for testing and execution in the same algorithms. The knowledge base is managed separately from the main algorithms, which are fixed for all graphic classes. Hence, to make the work for a new graphics class, the only thing we need to do is to add the rules, which specify the components and their attributes/constraints, to the knowledge base. The rules are also updated when new positive/negative examples are provided for existing graphic classes.

In this section, we present how the rules for a particular graphic class are represented, learnt, and used in the recognition process.

Knowledge Representation Scheme for Graphics Classes

In order to specify a graphics class, we design the representation scheme for a graphics class as follows. Each object of such graphic class should have the following attributes or features.

1. The ID for this class, which can be specified by the user or an automatic program.
2. The components (in sequence) of the class, which can be any previously known graphic classes. Currently, we use lines, arrowheads, and textboxes as primitive types, whose attributes are known. Once new graphics classes, which can either be manually specified by the user or be automatically learnt from examples, are added, they can also be used as the types of components of future graphics classes.
3. The attributes of each individual component, which can be used to filter out those graphic objects that cannot be candidates for the component. The graphic type for this component is the most important attribute for the component. The attributes for each type can be different. For examples, the attributes for a line segment can include its shape (which can be straight, circular, free formed, etc.) and style (which can be one of the pre-defined styles: continuous, dashed, dash-dotted, dash-dot-dotted, etc.), line width, length, angle, etc. An attribute can be specified with tolerances. For example, a line width can be 5±1 pixels and an angle can be 45±5º. An attribute, e.g., the graphics type, can also be fixed. Most often, if a textbox is required, a line is usually not allowed. Sometimes, line shape and style are also not flexible attributes for a component.
4. The constraints between each individual component and the entire object or other components that are in previous position in the component sequence. For example, the relative location (or angle) of the component in the entire object is a constraint between the component and the entire object. A constraint between two components can be intersection/connection/perpendicularity/parallelism (width a tolerated distance) between two straight lines, or concentricity/tangency between two arcs, or positional (above/under, left/right, or inside/outside) between two rectangles, and so on. Tolerances are also necessary due to many reasons including drawing digitization and vectorization.

The types of attributes and constraints can also be expanded to include new ones while a few primitive types of attributes and constraints are defined initially. For examples, the connection of two lines (of any shape and any style) is defined as that the minimum distance of an endpoint of one line to an endpoint of the other line is less than a tolerance (e.g., half of the line width).

Knowledge Acquisition for a Particular Graphics Class from Examples

Knowledge acquisition is the process in which the rules (mainly, the attributes and constraints) to represent particular graphics classes are obtained. Admittedly, a user can write all the rules manually. However, to enable example-driven graphic recognition, automatic acquisition of the rules is indispensable. Hence, we implement

an automatic learning process for a particular graphics class from the examples provided by the user.

In the automatic learning process, we need to determine the ID of the class, its components and sequence, especially, the first key components, which is critical in starting the recognition process (as shown in **Fig. 1**). More importantly, we need to determine the attributes of individual components and constraints among components. While the ID can be obtained quite easily (as we discussed in the last sub-section), determination of other things, however, is not-trivial.

First of all, the first key component and the sequence of the remaining components should be determined. Although there are multiple choices for the sequence, a good sequence can greatly reduce the complexity of the constraints and speedup the searching process for component candidates. We define the following heuristic rules for determination of the component sequence.

1. The components within an example are first sorted according to the priorities of their graphic types. The priority of a particular graphic type is determined as inversely proportional to the occurrence frequency of this type of objects in all graphic drawings, which can be statistically obtained. The lower the frequency, the higher the priority. The reason is that the graphic objects of those common types can be quickly filtered out during the candidacy test to save much time in later constraints checking, in which this sequence can filter out those non-promising combination of components more quickly than other possible sequences. Hence, the priority list can be sorted in the decreasing order of the occurrence frequencies of the graphic types. For example, solid lines are the most dominant graphic type in engineering drawing and hence this type is of the lowest priority.
2. If two components are of the same type, other attributes, e.g., length, size, can be used to sort their priorities.
3. When the first key component is determined, the sequence of the remaining components can be done similarly according to the type priorities. Or alternatively, the nearest principle can be used to find the next components. If multiple components are the nearest, positional sequences, e.g., from left to right, from top-down, can be used.
4. If multiple examples have been provided, the alignment (or correspondence) between the components of each example should be done. The most conformable sequence is chosen as the final sequence of components for this graphic class.
5. Optionally, the user's interaction can also be used as a method to specify the sequence. For example, we can ask experienced users to pick the key components in his examples first when the examples are provided.

After the sequence is determined, the attributes of each component is also determined. If a single example is provided, the values of the attributes can be directly calculated from the example. For example, the relative position/angle, length, angel, etc., can be calculated for line types. If permitted, a tolerance can also be added for each attributes. If multiple examples are provided, the values of the attribute for the same component all examples are used to determined the range of values that the attribute can take.

Then the constraints between the current components and each of the previous components in the sequence are determined. For each pair of component, each

possible constraint in a candidate constraint list (as we discussed in the previous sub-section) is tested. If the constraint passed the test, then this constraint is valid for this pair of components. Otherwise, the final constraint list for this graphic class does not include this constraint. If only a single example is provided, the tolerance of the constraint can be set strictly. If multiple examples are provided, the tolerance should be set to include all possibilities in the examples. If one among the many examples given for this graphic pattern violates the constraint, then this constraint is not a mandatory for this pattern and should not be included in the final constraint list. Even more, if more examples, especially for those negative examples (e.g., false alarms removed by the user), are provided later, the tolerance should be updated or even the entire constraints should become invalid.

Matching for Recognition of a Particular Graphics Class

Once the rules for a particular graphic class are known, its recognition process mainly consists of searching the current graphics database for its components with the rules using GGRA (as shown in **Fig. 1**). In this sub-section, we only discuss the main functions that should be implemented. Implementations of others are intuitive. We start the process by finding its first key component, whose attributes should conform to those of the first one in the component sequence for this graphics class. Starting with the first key component found by the **findFirstComponentFrom**(…) function, in which all attribute requirements for the first component are met, we find the other components for this graphic object one by one using the **extend**(direction, …) function. The **numOfExtensionDirections**() function returns the number of components for this graphics class. The "direction" parameter in **extend**(direction, …) function specifies which component the current extension procedure is searching for. The **search**(…) function returns those candidates, which pass all attribute requirements for the current component. Each candidate undergoes further tests in **extensible**(candidates[i]) function, in which all constraints between this components and others are checked. The first candidate that passes the tests is used as the current component. If such a component can be found the graphic object is successfully extended to this component and the extension to the next component in the sequence will begin until all components are successfully found and the entire graphic object is successfully recognized. Otherwise, it means failure of recognizing the graphic object.

4 Experiments

We have implemented the rule-based graphics recognition algorithm for simple graphic patterns that consists of only various types of lines and use it to implement our strategy of example-driven graphics recognition. An example of the graphic pattern that we want to recognize is specified by clicking all of its components. For example, as shown in **Fig. 2**, the user can click the solid circles and the concentric dashed circles as the example of the pattern we want to recognize. The system automatically

determines the dashed circle as the first key component and concentricity is the main constraint. The system then automatically finds other similar graphic patterns which contain the same kinds of components and are constrained similarly in the drawing. Including the example, four objects of this class have been recognized due to that we used a larger tolerance to the central angle of the arcs in the implementation. If we had selected the top-left or bottom-right pattern as the example, only three objects could have been recognized. This is due to that each such example contains a partial arc that cannot be successfully matched during recognition. Anyway, the experiment has proved that the current implementation has already been able to do example-driven graphics recognition.

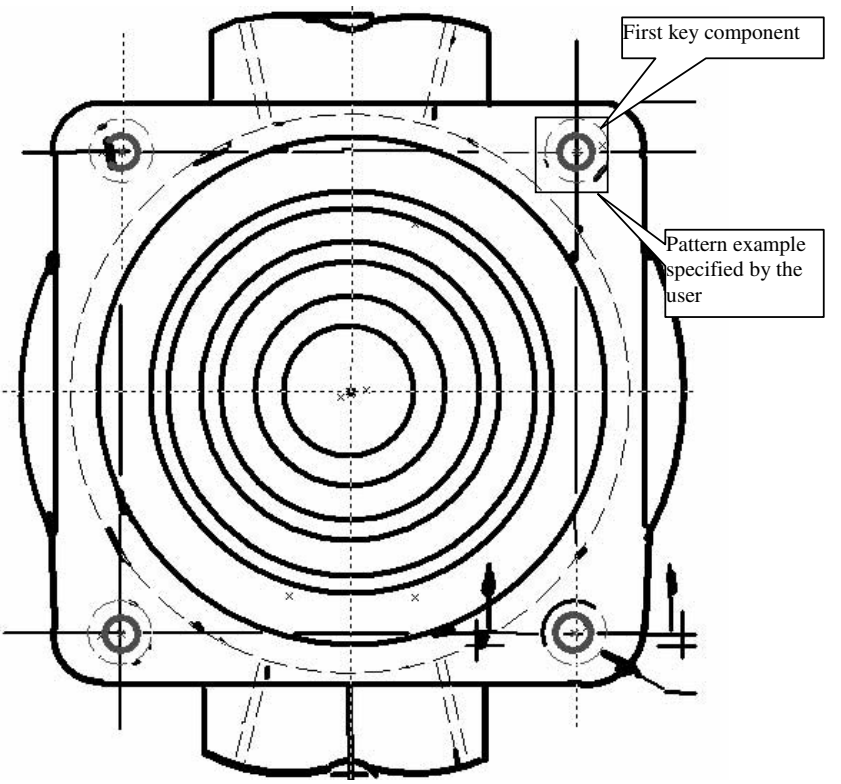

Fig. 2. Results of example-driven graphics recognition

Current, only single example can be used in our experiments. Recognition based on multiple examples, especially negative examples (from user's manual correction of false alarms), will be soon implemented. We will also test the algorithm on more complex graphic patterns, e.g., including arrowheads and textboxes in the future.

5 Summary and Future Work

We have presented a rule-based graphics recognition framework, which is based on the generic graphics recognition algorithm [6]. We have also applied it to build the example-driven graphics recognition scheme and obtained preliminary but promising results.

The scheme features a manual user interface for providing examples for particular graphic patterns, a rule-based representation of graphic patterns, and an automatic learning process for the constraint rules. This scheme provides a flexible approach, which are suitable for recognition of those graphic patterns that are unknown previously before the recognition system has been built. Such interactive graphic recognition scheme is especially useful in the current stage when automatic recognition cannot always produce reliable results. This scheme can also be used as efficient way for automatic knowledge acquisition for graphic patterns.

Although we currently only use single examples for learning graphic patterns, we believe that the scheme can also fit the cases of multiple examples from both positive and negative perspectives. Especially, the user's feedback, e.g., manual correction to those misrecognitions can be good resource to correctly learn the graphic patterns.

6 References

1. Cheng, T., Khan, J., Liu, H., Yun, D.Y.Y.: A symbol recognition system. In: Proc. ICDAR93 (1993).
2. den Hartog J.E., ten Kate T.K., and Gerbrands J.J.: Knowledge-Based Interpretation of Utility Maps. Computer Vision and Image Understanding 63(1) (1996) 105-117.
3. Dori D.: A syntactic/geometric approach to recognition of dimensions in engineering machine drawings. Computer Vision, Graphics, and Image Processing 47(3) (1989) 271-291
4. Dori D.: Orthogonal Zig-Zag: an Algorithm for Vectorizing Engineering Drawings Compared with Hough Transform. Advances in Engineering Software 28(1) (1997) 11-24
5. Dori D. and Liu W.: Sparse Pixel Vectorization: An Algorithm and Its Performance Evaluation. IEEE Trans. on Pattern Analysis and Machine Intelligence 21(3) (1999) 202-215.
6. Liu W. and Dori D.: Genericity in Graphics Recognition Algorithms. In: Graphics Recognition: Algorithms and Systems, eds. K. Tombre and A. Chhabra, Lecture Notes in Computer Science, Vol. 1389, pp. 9-21, Springer (1998).
7. Liu W. and Dori D.: A Generic Integrated Line Detection Algorithm and Its Object-Process Specification. Computer Vision and Image Understanding 70(3) (1998) 420-437
8. Vaxiviere P. and Tombre K.: Celestin: CAD Conversion of Mechanical Drawings. IEEE Computer Magazine 25(7) (1992) 46-54