A Low-Complexity Constructive Learning Automaton Approach to Handwritten Character Recognition

Aleksei Ustimov[1], M. Borahan Tümer[2], and Tunga Güngör[1]

[1] Department of Computer Engineering, Boğaziçi University, İstanbul, Türkiye
[2] Department of Computer Science Engineering, Marmara University, İstanbul, 90210, Türkiye

Abstract. The task of syntactic pattern recognition has aroused the interest of researchers for several decades. The power of the syntactic approach comes from its capability in exploiting the sequential characteristics of the data. In this work, we propose a new method for syntactic recognition of handwritten characters. The main strengths of our method are its low run-time and space complexity. In the lexical analysis phase, the lines of the presented sample are segmented into simple strokes, which are matched to the primitives of the alphabet. The reconstructed sample is passed to the syntactic analysis component in the form of a graph where the edges are the primitives and the vertices are the connection points of the original strokes. In the syntactic analysis phase, the interconnections of the primitives extracted from the graph are used as a source for the construction of a learning automaton. We reached recognition rates of 72% for the best match and 94% for the top five matches.

Key words: Syntactic Pattern Recognition, Handwritten character recognition, 2D Space, Parsing, Automaton, Graph, Data sequence, Expected Value, Matching

1 Introduction

Pattern recognition receives increasing attention since longer than three decades. While considerable progress have been made some challenging problems remain unsolved [1, 5].

Syntactic Pattern Recognition (SPR), one of the four a subcategories of Pattern Recognition consists of two parts: lexical analysis (segmentation) and syntactic analysis (parsing).

In lexical analysis a pattern is reconstructed in terms of primitives[7], that are obtained using deterministic [2] or adaptive [19] way. Syntactic analysis is responsible for grammatical inference in the training session or grammar checking in the recognition session.

In our method we use structural representation technique. Presented data samples are segmented into simple strokes (primitives) with three properties: *length*, *slope* and *shape (curvature).* We use a graph to hold of a segmented character. Edges and vertices of the graph represent the strokes and the connection points of these strokes, respectively.

For training and recognition we use the syntactic approach. Segmented and digitally represented shape is converted to an automaton. Conversion process is performed using an intermediate stage where the graph is converted to a sequence of elements, each of which holds the smallest unit of the structure. In the training session, the obtained automaton contributes to the grammar of the specific class and, in the recognition session, is matched to the grammar of each class to detect the best match.

Research for recognition based on curve representation has been reported in the literature. In [12] (topological representation) the primitives of the presented curve are encoded with the slope property and consecutive primitives having the same slope form a part of the curve. The parts are converted into the string representation of the curve (coding). The authors in [2] (coding representation with syntactic recognition) used similar to our's representation except the curvature property. Primitives of a 2D ECG signal were adaptively created using ART2 ANN in [19] (coding representation with syntactic recognition). A series of fuzzy automata, each with a different level of detail, participate in the classification process.

A shape analysis model was reported in [17] (graph representation and template matching recognition). The curves in skeletonized characters are approximated to the sequence of straight line segments and converted into a graph representation. Graphs and trees representation with a mixture of syntactic and template matching recognition was used in [10] to recognize various elements on architectural plans. Straight lines formed the primitive alphabet. Small patterns (windows, doors, stairs, etc.) were represented using graphs and recognition was performed by graph matching. The author in [13] (graphs and trees representation and natural recognition) used SPR to classify objects in medical images. The method learns the structure of the wrist bones with all variations and recognizes defects caused by bone joints or dislocations.

A new method for cursive word recognition was proposed in [18] (coding representation with template matching recognition). Single characters are segmented into straight lines, arcs, ascenders, descenders and loops. Primitives are coded into 1D sequences and the system searches for common rules. Recognition of the traffic signs, reported in [8] (coding representation and neural networks recognition). The author used two types of primitives: lines with the slope property and circles with the breaking point orientation property.

In most of the cases, authors used a small number of properties for segmented curves with complex algorithms for training and recognition. Increasing the complexity of the algorithm will result in an increase in the recognition rates to some extent. Our aim was to develop an approach to handwritten character recognition the main strength of which will be the simplicity and high speed on the expense of slightly less accurate recognition rates. All main parts of our method are simple and can be implemented with $O(n)$ running time. This method is useful in cases where the computational power of the hardware is not very high and the data to be recognized is not too distorted. As an example we may take pocket computers with handwritten word recognition software plus cooperative user with good handwriting.

The rest of the paper is organized as follows. The lexical analysis and syntactic analysis are explained in detail in Sections 2 and 3, respectively. In Section 4 we explain the practical application of the method proposed. Section 5 concludes the paper.

2 Lexical Analysis

In the lexical analysis phase we prepare the presented data for the syntactic analysis. In this section, we first explain the alphabet and then the segmentation.

2.1 Alphabet

In SPR the alphabet of primitives is obtained either adaptively or is known *a priori*. In our method we decided to use a predefined alphabet of primitives. When the type and format of the data are known, a human supervisor may provide a sufficient alphabet. To represent the strokes of handwritten characters we introduce the term *arc*. An arc is a small segment of a stroke for which the set of property values can be calculated, and it may be completely represented by *length*, *slope* and *shape*, each with a small set of discrete values.

The length of an arc is the number of pixels that form the arc. The assignment of length values may depend on the expected height of the character's body (character without ascenders and descenders).

The slope of an arc is the second property that contains direction information. To determine the slope of an arc, we use a line between the arc ends, called the *chord*. The chord of a sample arc is shown in Fig. 1.

The shape property indicates the direction of the arc's curvature. In calculation of this property we again use the cord. The value of this property may change according to the number, distance and position of each pixel in an arc according to the chord.

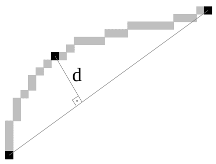

Fig. 1. The farthest point from the chord on the arc.

2.2 Segmentation

Segmentation module is responsible for breaking a given figure into a set of arcs in such a way that all strokes in the figure are encoded by the arcs. In our method, an entire figure is expressed in terms of arcs. In order not to lose the syntactic properties in the figure, arcs have to be connected to each other in the same manner as in the original figure. A good way to represent such a structure is to base the representation on a graph. The edges of the graph are the arcs with the properties converted to numerical values. The vertices of the graph are the common points of the arcs on the original figure.

3 Syntactic Analysis

In the training session, the task of the syntactic component is to generate a grammar for each presented class of data. A grammar is a set of rules that help us distinguish patterns. In the recognition session, the grammar is used as part of input along with the reconstructed signal. Here the task is to determine a *degree of match* with which a new signal matches the grammar presented.

3.1 Conversion from Graph to Sequence of Code Pairs

The graph produced by lexical analysis holds a character in terms of arcs and their interconnections. In the training and recognition sessions we use automata. The structure of the pattern that we use in automata differs from the one produced by the lexical analysis. In order to convert the graph representation of the signal to an automaton we use an intermediate step. In this step, to preserve the 2D topology, a sequence of connected edge pairs, called *code pairs*, is extracted from the initial graph. Each element of the sequence contains two edges of a graph with one common vertex. The length of the sequence obtained from a graph will depend on the number of edges of each vertex. A vertex that connects n edges will produce $C_2^n = \frac{n!}{(n-2)!2!}$ different pairs.

3.2 Grammatical Inference

In the training session, sequences obtained from characters belonging to the same class have to be merged into one learning structure. Knowing that a sequence consists only of primitives and the primitives are unique, we employ automata. An automaton is an appropriate structure for storing syntactic data, assumes that all states are unique, has an ability to learn, and may be trained by infinitely long input sequences.

During the insertion of each code pair to automaton, the state and transition probabilities are updated. In order to always maintain a normalized automaton, we use *linear reward-inaction* (L_{R-I}) probability update method used in learning automata [11]. After the insertion of an arc pair into an automaton, both states (corresponding to codes) are rewarded. To reward more than one state at once we changed the reward and punishment formulae as follows:

$$\begin{array}{l} P_i^{t+1} = P_i^t + (1 - P_i^t)\lambda\frac{1-P_I}{n-P_I} \\ P_k^{t+1} = P_k^t(1 - \lambda) \end{array} \tag{1}$$

where n, $0 < \lambda < 1$, P_i, P_k, P_I and t denote the number of the states to be rewarded, learning rate, probability of the rewarded state, the probability of the punished state, the probability sum of the rewarded states, and time, respectively. The last (added) factor in the reward formula distributes total reward according to the prior probabilities of the rewarded states: the higher the probability, the lesser the reward. Transition probabilities are updated using the same formulae where $n = 1$, which neutralizes the last factor.

3.3 Pattern Matching

In the training session we obtain a trained automaton for each class presented. In the recognition session we follow the algorithm explained above and stop after the generation of the sequence of code pairs is completed. Then, instead of generating an automaton, we match the sequence to all trained automata and choose the best match.

When the training session completes, the system ends up with the normalized automata. For the matching operation we still need a few more values.

Matching Parameters During matching, when a sequence of code pairs is presented to a trained automaton, a *score* (τ) is calculated that denotes the presence of the specific arcs and their connections to each other. To calculate the score we search the trained automaton for the states and transitions that represent each pair in the matched sequence:

$$\tau = \sum_{i,j \le n} P(s_i)P(t_{ij}) + P(s_j)P(t_{ji}) \tag{2}$$

where n is the number of states, $s_i, s_j \in S$ are the present states and $t_{ij}, t_{ji} \in T$ are the transitions between s_i and s_j. In other words, the score shows the extent to which the sequence is present in the automaton.

The automaton of each class produces a score for the sequence of the test character. The scores produced cannot be directly compared. To be able to compare those scores, they have to be converted into a common scale. For this purpose, we use the expected value of the scores that each trained automaton produces for the characters it represents. To determine this value, for each automaton, we use a second pass over the training data to calculate the scores of training characters and the expected values from the collection of obtained scores.

$$E(\tau) = \tfrac{1}{n}\sum_{i=1}^{n} \tau_i \tag{3}$$

We use this value as a reference point to show that the closer a character's score to the expected value, the higher is the degree of match. The degree of match is calculated by the formula:

$$D_m = 1 - \frac{|E - \tau|}{E^2}\alpha \qquad (4)$$

where D_m is the degree of match, E is the expected value (or the average) of the scores, E^2 is a second central moment (or variance), and $\alpha \in (0; 1)$ is a constant. To quantify the asymmetry of the distribution of the scores we use the third central moment [15, 16].

Matching In the recognition session, an unknown character passes the lexical analysis phase and is converted to a sequence of code pairs. Next, the sequence obtained is matched against all automata and degrees of match are computed. Finally, the character is assumed to belong to the class whose automaton produced the highest degree of match.

4 Practical Application

In our application we used the Turkish alphabet which contains 29 letters and is similar to the English alphabet. The Turkish alphabet does not use the three letters *q*, *w*, and *x*; instead it includes six new letters *ç*, *ğ*, *ı*, *ö*, *ş*, and *ü*.

4.1 Preprocessing

Before presenting the handwritten character images to the lexical analysis component they have to be preprocessed. The lexical analysis component works with one-pixel thin foreground strokes, so our preprocessing consists of two operations: *binarization* [4] and *skeletonization* [9, 14, 6].

4.2 Segmentation

The first step in segmentation locates the special pixels, called *reference points*, on the skeletonized character. There are two types of such pixels: cross points and ends of the lines.

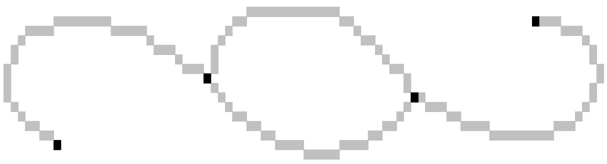

Fig. 2. Cross and end points located on a shape.

A cross point denotes a pixel that has more than two foreground pixels in its 8-pixel neighborhood. An end point is a pixel with only one foreground pixel in its 8-pixel neighborhood. In Fig. 2, the cross points and ends of the lines are determined.

Initially, a stroke between each pair of reference points is assumed to be an arc. The next step is to calculate the values of the properties for each arc. In our application we used a predefined number of values for each property.

The length property may take three values: *short* (s), *medium* (m) and *long* (l).

The slope may take four values: *horizontal* (h), *vertical* (v), *positive* (p) and *negative* (n). If the chord encloses an angle of less than a threshold value (about 15^o) with the x-axis or with the y-axis, it is assumed to be horizontal or vertical, respectively. A positive value is assigned in cases when the chord lies in the first and third (positive) quarters of the circle drawn around. The chord locating in the second and fourth (negative) quarters is assigned a negative slope value. For example, the chord of the arc in Fig. 1 is not close to any of the axes and lies in the positive quarters of the circle, so the slope property value of that arc is positive.

The shape property may take three values: *straight* (s), *convex* (x) and *concave* (v). To determine the shape property value we draw a chord and calculate the average distance d_{avg} of the pixels in the arc to the chord and the number of pixels, p_a and p_b, vertically above and below the chord, respectively. The average distance will show us the degree of the arc's concavity and the number of points will help us distinguish between convex and concave values. The arc is assumed to be straight if the degree of concavity is less than a specific threshold value. If the arc is not straight, we check the number of pixels above and below the chord. If $p_a > p_b$ the shape property of the arc is set to convex, and in the case of $p_a < p_b$, to concave.

There are $3 \times 4 \times 3 = 36$ different value combinations for an arc. Each combination is called the *code*. Codes are generated as follows: each property value is given a number that is unique for each property (length: $s = 1$, $m = 2$, $l = 3$; slope: $h = 1$, $v = 2$, $p = 3$, $n = 4$; shape: $s = 1$, $x = 2$, $v = 3$). The code is a number that contains all properties in the same order: $code = 100 \times length + 10 \times slope + 1 \times shape$. For example, an arc with property values as medium, positive and convex is encoded to $100 \times 2 + 10 \times 3 + 1 \times 2 = 232$.

The properties of these arcs may fail to be calculated: an arc may be too long or short, or may be a too complex curve to be clearly defined as convex or concave. Such arcs are broken up into several simpler arcs by locating additional reference points. The leftmost and the rightmost arcs in Fig. 2 are examples of complex arcs. To locate additional reference points we use the *bounding box operation*, where an arc is surrounded by a tight (bounding) box. The pixels that touch the sides of the box become new reference points. In Fig. 3 two points for the leftmost and the rightmost arcs were located using this operation.

In the case that the bounding box operation does not produce any new reference point, we apply the *farthest point operation*, where the pixel that has the highest distance from the chord, as shown in Fig. 1, is the additional reference point. In Fig. 3 one point for the middle arcs were located using the farthest point operation.

The segmentation process continues until all arcs are assigned valid values for their three properties.

The representation of the segmented character is based on the reconstruction of the original figure in terms of primitives (arc codes) instead of arcs and presenting it as a graph. For instance, the graph corresponding to the symbol in Fig. 3 consists of 10 vertices and edges, as shown in Fig. 4. We assume that the code of the arc contains sufficient information about the arc and we do not have to keep the pixel details.

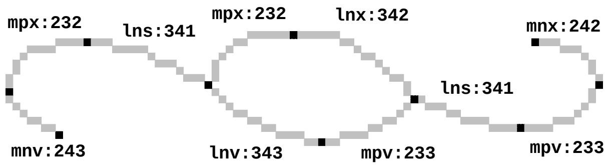

Fig. 3. Property values' abbreviations and codes of the all arcs in the shape.

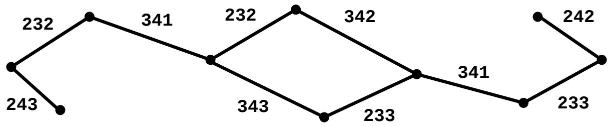

Fig. 4. Graph representation of the shape in Fig. 3.

4.3 Code Pairs

To convert graph obtained from the lexical analysis to the sequence of code pairs we extract all possible combinations of connected edges from each vertex. As states earlier the number of pairs from a vertex that connects n edges will be $C_2^n = \frac{n!}{(n-2)!2!}$. The only exception is the vertex that contains one edge, which produces the pair with 0 on one side. For instance, the graph in Fig. 4 produces the following sequence: 0-243, 243-232, 232-341, 341-232, 341-343, 232-343, 232-342, 343-233, 342-233, 342-341, 233-341, 341-233, 233-242 and 242-0. The order of the pairs is irrelevant since each pair is a part of the graph topology and the order of the codes in a pair is also irrelevant since the edges in a graph are not directed.

4.4 Automaton

Presenting one element of a sequence, a code pair, to an automaton consists of a few simple and straightforward operations. First, each code in a code pair is assumed to represent a state of an automaton. New states are created if the automaton does not contain a newly presented code. Second, a connection between the codes in the pair represents a transition between states. We create two transitions with opposite directions for each code pair [20], because transitions in an automaton are directed. The automaton corresponding to the graph in Fig. 4 is shown in Fig. 5.

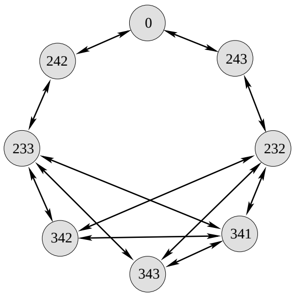

Fig. 5. Automaton representation of the graph in Fig. 4.

4.5 Results And Discussion

The main strength of our method is its simplicity and time complexity. All main parts of our method are simple and can be implemented with $O(n)$ running time. In the training session, the algorithm requires two passes over the training data. The runtime of the second pass can be made negligible by using only a small portion of the training data. For example, if an automaton was built using 10000 samples, then the second pass may be completed with only 100 samples chosen randomly. Segmentation is performed by a single pass over all pixels of a sample image and all arcs extracted are used once in an automaton generation. All structures used in our method are simple and require limited amount of memory. Each trained automaton consists of 36 states and 36 × 36 transitions at most.

Since there is no database available for Turkish handwritten character recognition, we compiled a new database. The data set of 12000 handwritten characters was collected from 10 writers. After testing our system on the Turkish handwritten characters, we obtained promising results in the area of character recognition. Each character in the test set was presented to all automata and degrees of match provided by each automaton were sorted in descending order. In 71.94% of the cases the score of the correct automaton was the highest (positioned on the top of the order) and in 93.79% of the cases was among the top five results. Distribution of the recognition rates from the top result to the top five results is shown in Fig. 6.

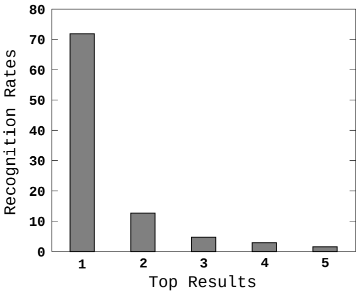

Fig. 6. Average positions of the correct automata in the recognition order.

The major part of the correct recognitions is concentrated on a first position. The positions not shown in figure have the average values less than one percent.

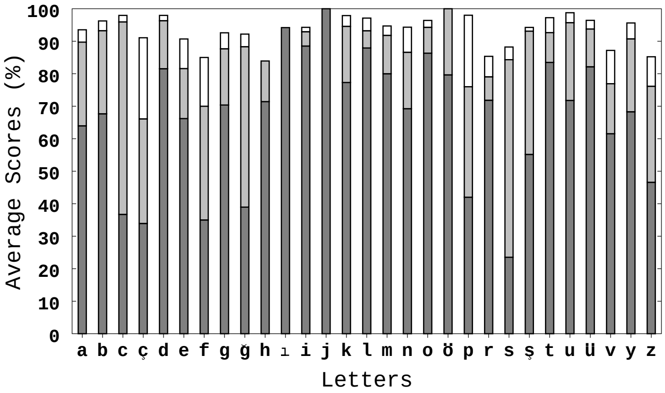

Fig. 7. Distribution of the top five results for each letter.

In our system, the simpler the topology of a character, the higher the recognition performance. The distribution of the recognition results for each character is shown in Fig. 7. The top result is marked with the darkest portion of the bars. The top second and third results are combined and marked with the gray color. The fourth with fifth results are also combined and shown with the white colored portions of the bars. As we expected, the simplest letters such as *ı*, *i*, *j*, *l*, *o*, *ö* and *t* have top recognition rates of more than 80%, while the lowest rates were calculated for the letters *c*, *ç*, *f*, *g̃* and *s* with many curved portions.

We also studied the recognition errors for some of the most frequent letters in Turkish: *a*, *e*, *i*, *l*, *n* and *r* (Table 1). Most frequently *a* is erroneously recognized

Table 1. Misclassifications of some of the most frequent letters in Turkish. The tables show the erroneous choices and their percentages among all misclassifications of the letter in the header of a relevant table.

a	**e**	**i**	**l**	**n**	**r**
o 35.96%	v 23,95%	l 26,53%	i 57,14%	i 21,85%	i 39,80%
ü 15.79%	r 20,96%	r 22,45%	r 19,05%	ş 17,65%	ş 18,37%
e 15.35%	o 11,38%	ü 12,24%	v 7,14%	r 16,81%	l 13,27%
c 7.02%	i 10,78%	ş 10,20%	ç 4,76%	e 12,61%	c 5,10%
u 5.70%	c 10,18%	v 8,16%	ş 4,76%	ü 11,76%	e 5,10%

as *o*, *ü*, *e*, *c* and *u*. The potential reason is that those letters have a similar topology with *a*. Misclassifications for the same reason may be observed for *i*, *l* and *r*. The letter *e* is mostly misclassified with *v* and *r*, because they have a similar handwritten topology. The misclassifications of *n* do not display a proper pattern, because *n* can be easily distorted during handwriting, and hence, may have a similar topology with many letters.

5 Conclusion

In this paper we proposed a constructive learning automaton approach to the recognition of handwritten characters. Promising results were obtained after testing our system. Recognition rates reached vary between 71.94% for the best and 93.79% for the top five results. For unconstrained characters segmented from words, performance of our approach is similar to those reported in [21, 22]. A similar approach was used in [3]. The authors reached an average top five recognition rate of 93.78% for online English handwritten character recognition.

As can be concluded from the results the weakest point in our application is the representation technique of smooth curves that are usually encountered in letters *c*, *ç*, *g*, *ğ*, *s* and *ş*. The direction of possible future work is the revision of the representation technique for smooth curves. A new technique for placement of reference points may be focused on the sharp changes of the degree of curvature along the presented curve. This will lead to robustness against slant variation and increase the performance.

Acknowledgements. This work has been supported by the Boğaziçi University Research Fund under the grant number 09A107D.

References

1. N. Arica and F.T. Yarman-Vural. An overview of character recognition focused on off-line handwriting. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions*, 31:216–233, 2001.
2. Y. Assabie and J. Bigun. Ethiopic character recognition using direction field tensor. *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, 3:284–287, 2006.

3. V. S. Chakravarthy and B. Kompella. The shape of handwritten characters. *Pattern Recognition Letters*, 24:1901–1913, 2003.
4. D.A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach.* Prentice Hall, 2003.
5. R. Gonzalez and M. Thomason. *Syntactic Pattern Recognition: An Introduction.* Addison-Wesley, 1978.
6. C.M. Holt, A. Stewart, M. Clint, and R.H. Perrott. An improved parallel thinning algorithm. *Communications of the ACM*, 30(2):156–160, 1987.
7. K.-Y. Huang. *Syntactic Pattern Recognition For Seismic Oil Exploration.* World Scientific, 2002.
8. S. Kantawong. Road traffic signs detection and classification for blind man navigation system. *Control, Automation and Systems, 2007. ICCAS '07. International Conference on*, pages 847–852, 2007.
9. L. Lam, S.-W. Lee, and C.Y. Suen. Thinning methodologies - a comprehensive survey. *Pattern Analysis and Machine Intelligence*, 14(9):869–885, 1992.
10. J. Llados and G. Sanchez. Symbol recognition using graphs. *Image Processing, 2003. ICIP 2003. Proceedings. 2003 International Conference on*, 2:49–52, 2003.
11. K. Narendra and M. Thathatchar. *Learning Automata: An Introduction.* New York. Addison-Wesley, 1989.
12. H. Nishida and S. Mori. Algebraic description of curve structure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14:516–533, 1992.
13. M.R. Ogiela. Automatic understanding of medical images based on grammar approach. *Imaging Systems and Techniques, 2007. IST '07. IEEE International Workshop on*, pages 1–4, 2007.
14. B.R. Okombi-Diba, J. Miyamichi, and K. Shoji. Segmentation of spatially variant image textures. *16th International Conference on Pattern Recognition (ICPR'02)*, 2:20917, 2002.
15. A. Papoulis. *Probability, Random Variables and Stochastic Processes.* McGraw-Hill Companies; 3rd edition, 1991.
16. W. Paul and J. Baschnagel. *Stochastic Processes: From Physics to Finance.* Springer, 1999.
17. J. Rocha and T. Pavlidis. A shape analysis model with applications to a character recognition system. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:393–404, 1994.
18. J.C. Simon. Off-line cursive word recognition. *Proceedings of the IEEE*, 80:1150–1161, 1992.
19. M.B. Tümer, L.A. Belfore, and K.M. Ropella. A syntactic methodology for automatic diagnosis by analysis of continuous time measurements using hierarchical signal representations. *Systems, Man, and Cybernetics, Part B, IEEE Transactions on*, 33(33):951–965, 2003.
20. A. Ustimov and B. Tümer. Construction of a learning automaton for cycle detection in noisy data sequences. *Lecture Notes in Computer Science*, pages 543–552, 2005.
21. B.A. Yanikoglu and P.A. Sandon. Off-line cursive handwriting recognition using style parameters. *Department of Mathematics and Computer Science, Dartmouth College*, 1993.
22. B.A. Yanikoglu and P.A. Sandon. Recognizing off-line cursive handwriting. *Proc. Computer Vision and Pattern Recognition*, pages 397–403, 1994.