Learning and Upgrading Rules for an OCR System Using Genetic Programming

David Andre
Stanford University and Canon Research Center of America
P.O. Box 5402
Stanford, CA 94309
Email: phred@leland.stanford.edu

Abstract: Rule-based systems used for Optical Character Recognition (OCR) are notoriously difficult to write, maintain, and upgrade. This paper describes a method for using Genetic Programming (GP) to evolve and upgrade rules for an OCR system. The language of the evolved programs was designed such that human hand-coded rules can be included into the initial population in order to upgrade for a new font. The system was successful at learning rules for large character sets consisting of multiple fonts and sizes, with very good generalization to test sets. In addition, the method was found to be successful at updating hand-coded rules written in C for new fonts. This research demonstrates the successful application of GP to a difficult, noisy, real-world problem.

1. Introduction

Rule-based systems used in OCR are difficult and time-consuming to write, maintain, and upgrade. There is a rule set for each character that is supposedly true only for that character. Thus, any changes in a rule set must be tested on very large character sets to insure that the rule set accepts all examples of the character and rejects all others. A system that could automate the process of learning these rule sets therefore would be greatly desirable.

Genetic Programming has been successfully applied to simple OCR classification problems (For information on GP, see [Koza 1992]). John Koza [1993] evolved programs that could recognize an 'L' and a 'I" using co-evolved Boolean templates and control code for moving the templates. Andre [1993] extended this work by evolving programs that were successful at recognizing low resolution digits using co-evolved two-dimensional feature detectors. However, both of these approaches involved very low-resolution characters (5x5 and 4x6) and were only capable of recognizing very limited character sets. This paper presents an approach that uses GP to evolve programs to recognize noisy multi-font and multi-size characters using decision rule sets operating on the boundaries of the characters, using a segment breakdown method [Amos, 1993]. This approach has the additional advantage of using a language compatible with the human coded C language decision rule sets; using such a language allows the approach to extend the power or generalizability of hand-coded rule sets.

Section 2 describes the hand-coded recognition system: it explains the preprocessing steps and provides examples of the types of rule sets that are used to recognize characters. Section 3 discusses the language used to represent the rule sets and delineates the steps for using GP on the problem. Section 4 presents results from three experiments: Experiment 1, which was to validate the approach on a trivially small character set; Experiment 2, which was to attempt to evolve a successful indiviudal using a full character set with three sizes of three different fonts; and Experiment 3, which was to test the expansion capability of the system. Section 5 discusses the applications and potential limitations of this approach and presents some conclusions.

2. The system without GP

One of the difficulties in OCR is the sheer number of bits of information present in every character. Often, prior to rule evaluation, the bitmaps are simplified into combinations of more general features. This simplification allows for a set of fewer rules to specify a character but also typically loses some information because feature extraction of very general features such as lines or curves is a many-to-one transformation. One attempted solution has been to attempt to use features of the bitmap that can allow reconstruction of the character. The system in this paper uses a feature breakdown that contains pixel information for only the boundaries of the character.

2.1 Pre-processing

The first step in the recognition of any character is to extract the boundary pixels from the character bitmap. This is done using a quick one-pass raster scan method [Andre, 1993] that provides output in the form of a bi-directional circular linked list for the boundary of the character and for each interior hole. Each element in a list contains row and column information for a boundary pixel. The second step in processing is to close holes that are small enough to be accounted for by noise. The outer boundary of the character is then split into four segments (top, left, right, and bottom) using a technique that is robust to moderate levels of noise [Amos, 1993]. An example

0-7803-1899-4/94 $4.00 ©1994 IEEE

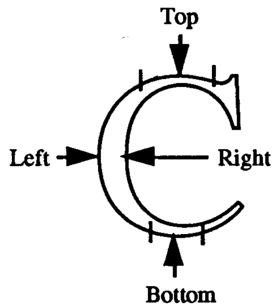

Figure 1. Segmentation of a character boundary

of this segment breakdown is shown in Figure 1. At this point, simple bounding-box values (i.e. the maximum and minimum row and column) are calculated for each hole, for each segment, and for the character as a whole. In addition, the number of pixels in each hole boundary is stored, and the segments are ranked according to their number of pixels. Thus pre-processing produces a simplified packet of information consisting of a number of linked lists and a number of simple statistics for each list: the number of pixels and the bounding box values.

2.2 Architecture of recognition system

The recognition system contains a rule set for each character found in English printed material. Each rule set contains rules that express relations among values in the simplified packet of information produced by pre-processing. Each of these rules must test true in order for the rule set to return true. Thus, only one rule set should fire for each test character. If two rule sets do fire, then a response is chosen by likelihood of occurrence of the characters, but an uncertainty flag is raised indicating that the confidence of this classification is low. Examples of rules in pseudo-code are given in Figure 2. An example in C is given in Section 3.3. It is important to note that these rules may include complicated loops and multiple operations on the linked lists of pixels (rules 4, 5; Figure 2) or they may simply be quantitative statements of simple bounding box statistics (rules 1, 2, 3; Figure 2).

3. Applying GP to the problem

There are five main steps in preparing to use GP. One must choose: (1) the set of terminals, (2) the set of functions, (3) the fitness measure, (4) the parameters for controlling a run, and (5) the method for designating a result and the criterion for terminating a result.

3.1 Functions (1) and Terminals (2)

There were two motivations for choosing the function and terminal sets. First, the sets had to provide the necessary power to solve the problem. Second, the function and terminal sets had to be capable of expressing the hand-coded versions of the rules. Complicated functions to simulate looping, pointers, and storage were included in the function set. Although the function set may appear overexpressive for learning from scratch, several preliminary experiments indicated that dramatically smaller function sets were not successful. The function and terminal sets are shown in Figure 3. These function and terminal sets were capable of expressing the hand-coded rule sets.

1. Right Segment (rseg) is longer than any other.
2. Number of rows in Tseg is less than 10% of those in Lseg.
3. The mid-column of Lseg to the left of either edge of Lseg.
4. Starting at the intersection of Tseg and Rseg, the top point on Rseg near the middle column is reached before reaching the middle row and after a downward spike.
5. On the lower half of Rseg, between the lowest point in the inner curve and the maximum column point, there is no point at which there is a run of 4 vertical pixels, nor any place where the boundary doubles back in the horizontal direction.

Figure 2. Examples of Rules for the Letter 'C'

Terminals:

General:	first_row, first_col, last_row, last_col	-These correspond to the bounding box statistics for the the entire character.
For each Hole(2):	NumPixels, first_row, first_col, last_row, last_col.	-These correspond to the bounding box statistics and the number of pixels for the holes.
For each Segment(4):	Rank, first_col, last_col, first_row, last_row.	-These correspond to the bounding box statistics and the rank for the linked-lists of the segments.
Constants:	random_int, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.	-Simple constants. The random integer is an ephemeral random constant.

Functions:

add(**a**,**b**), minus(**a**,**b**), times(**a**,**b**), divide(**a**,**b**)	-The simple mathematical functions. Proteced division returns a 1 if the **b** is 0.
LessThanElse(**a**,**b**,**c**,**d**)	-If **a** <= **b** then executes **c**, else executes **d**.
LessThanReturn(**a**,**b**,**c**)	-If **a** <= **b** then execute **c**, else return 0.
VHits(**a**,**b**)	-This returns the number of times the segment specified by (**a** mod 4) crosses the column **b**.
HHits(**a**,**b**)	-This returns the number of times the segment specified by (**a** mod 4) crosses the row **b**.
goto(**a**)	-Moves the Current-Pointer to the start of the segment specified by (**a** mod 4).
GotoandDo(**a**,**b**)	-Moves the Current-Pointer to the start of the segment specified by (**a** mod 4), executes **b**, and then returns the pointer to its previous location.
GoForwardUntil(**a**,**b**,**c**,**d**)	-Moves the Current-Pointer forward until **a** <= **b**, or until the end of the segment is reached. Then, if **a** <= **b**, executes **c**, else executes **d**.
GoBackwardUntil(**a**,**b**,**c**,**d**)	-Like GoForwardUntil, except backwards.
Progn(**a**,**b**)	-Executes **a**, then executes and returns **b**.
SetX(**a**)	-Sets a storage variable to **a**.
GetX()	-Returns the value of the storage variable.
CurrentRow(), CurrentCol()	-Returns the current value for row or column.
(Row/Column)(Forward/Backward)FromStart(**a**) (e.g. RowForwardFromStart(**a**), ColForwardFromStart(**a**), etc.)	-Returns the row/column of the pixel that is **a** steps forward/backward from the start of the current segment.
(Row/Column)(Forward/Backward)FromEnd(**a**)	-Does the same as above for the end of the segment.
(Row/Column)(Forward/Backward)FromCurrent(**a**)	-Does the same as the above for the current pointer.

Figure 3. The function and terminal sets.

3.2 Fitness measure (3)

The third step in preparing to use GP on a problem is to determine the fitness function. The fitness cases consist of a set of bitmaps, where the exact number of bitmaps varies for each experiment. In each experiment, the set of fitness cases is split into positive and negative cases. The positive set of fitness cases consisted of bitmaps of the letter 'C' in several different fonts and sizes and the negative fitness cases consisted of bitmaps of various other characters. To score perfectly, an individual must return a value greater than 0 when tested on a 'C', and must return a value equal to or less than 0 when tested on any other character. Penalties are assessed for incorrect responses. If an individual mistakenly classifies a bitmap as a 'C', the individual is penalized for a false positive. If an individual mistakenly fails to classify a bitmap as a 'C', the individual is penalized for a false negative. In addition, if any individual classified all fitness cases as the same, an additional penalty equal to the number of fitness cases was added. The penalties for each experiment are shown below.

Experiment	# Positive Cases	# Negative Cases	False Positive Penalty	False Negative Penalty
1	10	60	1	6
2	20	600	1	30
3	3002	7000	3	7

In C: (cptr is a pointer to a pixel)

```
(1)     if (Rseg->rank != 1) return(0);
(2)     cptr = Lseg->Start;
        mrow = (Lseg->first_row + Lseg->last_row) / 2;
        while (cptr != Lseg->End && cptr->row < mrow)
                cptr = cptr->forward;
        if (Lseg->End->col < cptr->col) return(0);
        if (Lseg->Start->col < cptr->col) return(0);
```

In GP-tree form:

```
(1)     (LessThanReturn 1 Rseg_rank (LessThanReturn Rseg_Rank 1 1))
(2)      (GotoAndDo 2 (Progn   (GoForwardUntil  (divide (plus Lseg_first_col Lseg_last_col) 2)
                                        (CurrentRow) 1 1)
                              (LessThanReturn (CurrentCol)(GotoAndDo 2 (ColForwardFromStart 0))
                               (LessThanReturn (CurrentCol)(GotoAndDo 1 (ColBackwardFromEnd 0)) 1))))
```

Figure 4: Example rules for the letter 'C', in both the C programming language and in GP-tree form

3.3 Examples of translated rules

Figure 4 shows C and GP-tree versions of two rules, which were among those translated into GP-tree form for the letter 'C'. The first is quite simple; it is true if and only if the right segment is the longest segment. The second rule is more complicated. The rule first finds the pixel on the left segment that is halfway down. Then, it returns true if this pixel is to the left of both the start and end of the left segment. In a sense, this rule insures that the curvature of the left segment matches that of a 'C'. The left segment of an 'A', for example, fails this test because the midway pixel is to the right of the lower end of the left segment. These rules are clearly human-designed, but they are similar in complexity to those evolved by GP.

3.4 Values of parameters (4) and success predicate (5)

The fourth step in preparing to use GP is to determine the values for the various run parameters. Initial random S-expressions were allowed to be only six levels deep. Evolved S-expressions were limited to a depth of 20. Tournament selection reproduction was used. The straight-copy fraction was 0.1, the crossover-at-any-point fraction was 0.2, and the crossover-at-function-point fraction was 0.7. The rate of mutation for Experiments 1 and 2 was 0, for Experiment 3 the mutation rate was 15%. For more information about these parameters, see [Koza, 1992]. The fifth step in preparing to use GP is to specify termination conditions and to choose a method for designating the result of the run. Runs were terminated if an individual with zero fitness was found, or after the maximum number of generations. The result of the run is the best-so-far individual -- the best individual obtained during the run.

4. Experiments and results

All experiments were run using a modified version of the Simple Genetic Programming Code, written by Walter Tackett. The code was changed to run on a network of seven 486 PC's running Microsoft NT. Character bitmaps used in this study were obtained by scanning in printed characters in binary format. The positive character sets consisted of 'C's; the negative character sets consisted of non 'C's. In all experiments, the Times Roman, Courier, and Helvetica fonts in sizes 8, 10, and 12 were used. For Experiment 3, other fonts were included.

4.1 Experiment 1: Pilot study

The goal of this experiment was to validate the system, and to verify that GP could evolve rule sets capable of correctly classifying full size characters with a character set similar in size to that previously used in applying GP to OCR [Andre, 1994; Koza, 1993]. Thus, the positive character set included 10 'C's, one each of Times Roman, Courier, and Helvetica of sizes 8, 10, 12, with two size 12 Courier 'C's. The negative set included non-'C' characters that were chosen to maximize similarity with the 'C'. Characters such as 'G', '(', '{', '[', '2', 't', 'T', 'S' were included, as were characters such as the 'A', 'B', and 'E'. The population size varied from 500 to 5000, and run time varied from one hour to several hours. The maximum number of generations was 100. Initial populations were random.

All 10 runs were successful. A perfect individual was found as early as generation 30 and as late as generation 97. The perfect individuals were each tested on several pages of test data, and generalization scores were obtained. It was found that these individuals generalized poorly, correctly recognizing only 60-80% of the test data correctly. The low number of fitness cases encouraged overfitting: non-generalizable properties of this small set of 'C's were often utilized by the successful individuals.

4.2 Experiment 2: Full character set

The purpose of Experiment 2 was to attempt to produce solutions for the 'C' that would generalize well to never before seen test data. To avoid the overfitting problem observed in Experiment 1, a larger character set was used, which consisted of a full set of characters from Helvetica, Courier, and Times Roman of sizes 8, 10, and 12. There were 20 positive examples and 600 negative examples. Both sets were 'filled out' by including multiple examples of some fonts and sizes. Again, the positive set contained only 'C's, and the negative set was biased to include those characters likely to be confused with a 'C'. The population size was 8000; run times varied from three to eight days. The maximum number of generations was 200 and the initial populations were random.

This problem was much less tractable. Out of 5 attempted runs, only one perfect individual was found, on generation 180 of a run lasting six days. Over many pages of test data, it was found to generalize fairly well, correctly classifying between 96% and 99% of the characters on a page. Thus, the overfitting problem was greatly reduced by increasing the size of the character set. The exact nature of the solution was difficult to ascertain, but it appeared in many respects to resemble the translated version of the hand-coded rule set. For example, the predominant structure in both the evolved and the hand-coded individual was a set of nested LessThanReturn functions.

4.3 Experiment 3: Upgrading an existing rule to handle a new font

This experiment was designed to test the system's ability to upgrade a previously existing hand-coded rule set to be capable of handling a new font. A version of the best 'C' rule set was translated by hand into the GP-tree language, and was tested to have identical performance to the C language version. The rule set for the 'C' was tested on many new fonts, where it was found to do quite well in general, but performed badly on a font called Eurostyle. The 'C's in that font were often rejected, and many of the 'G's in that font were accepted as 'C's. One possible explanation for this is that the 'G's in Eurostyle are very similar to the 'C's; the 'G' lacks the horizontal crossbar that is so characteristic of the 'G' in other fonts. A set of fitness cases was developed that captured the fonts that the hand-coded version of the 'C' could accurately classify. This set included 3000 positive and 7000 negative cases. This set contained many different fonts with examples from at least five different sizes in each font. The translated version scored 100% on this set prior to upgrading. To attempt to upgrade for the Eurostyle font, two examples of a Eurostyle 'C' were added to the positive fitness cases. Thus, there were 3002 positive cases, and 7000 negative. The initial population consisted of the translated individual, each of the separate rules found in the translated individual, and random individuals. The mutation rate was set at 15% to increase the genetic diversity of the populations. The population size was 5000 and run times varied from several hours to several days. The maximum number of generations was 100.

A successful individual was found on the fourth attempted run, in generation 12. The individual was tested on several pages of multiple sizes of Eurostyle font, and was found to be perfect at distinguishing a 'C' from a non-'C'. Even though the individual had only been trained on two examples of one size, the solution generalized to many sizes. Further analysis indicated that the upgraded individual was very similar to the hand-coded individual, but with a change in the part of code that discriminated between a 'C' and a 'G' -- the rule set had evolved to accept Eurostyle 'C's and reject Eurostyle 'G's.

5 Discussion and conclusions

The results from the experiments described in this paper indicate that a GP-OCR system can successfully evolve general rule sets that accurately classify printed characters. The letter 'C' was chosen for its high number of 'look-alikes', and thus it can be expected that rule sets would be as easy to learn for the other characters. While it would take a long time to learn a generalizable rule set for all characters, the experiments presented here suggest that GP is capable of it. In addition, it was shown that a GP-OCR system can successfully update hand-coded rule sets to handle new fonts. After updating, these rule sets can be translated back into C so they could easily be integrated with human-coded rule sets.

There are several limitations inherent in the current work that require additional research. Available computer time restricts the size of the character set, which in turn restricts the generalizability of the solutions. In

addition, the 'lack of inertia' problem with GP evolved solutions is very apparent in this domain. For example, although the evolved successful individual in Experiment 3 was perfect on the 10,002 fitness cases, changes made to accommodate the new Eurostyle characters affected generalization abilities. The hand-coded version was approximately 99.999% accurate on Courier, for example, but after upgrading for Eurostyle the new individual was only about 99.95% accurate on Courier. GP-OCR is limited to the information contained in the training sets, whereas human programmers have a generalized notion of what makes a character a 'C' that has more inertia -- learning a new font does not ruin previously learned information. Because computer time is limited, the fitness sets cannot grow without end, and some sort of fitness case management system is required. One proposed system would prune characters from the character set when they add nothing to the generalization ability of the learned rule sets. However, a large amount of computer time is needed for this type of system and research into improving the generalizability of GP learned solutions is also needed. Future work will examine the use of Automatically Defined Functions (ADF's) to improve generalizability. In addition, the effects of allowing individuals to use various forms of long term individual and collective memory will be investigated.

One important facet of this research is that the GP language was designed to interact with and be equivalent to the language used by human programmers. The high number of functions and terminals used in this research was not for the mere sake of solving the problem, but for increased interaction with human-written programs. In fact, when GP started from a random population, very few successful individuals used all of the functions or terminals. However, the large lexicon was required to allow translation from programs written in C. The possibility of easily combining evolved code and human written code is exciting because it indicates that the different programming advantages of GP and humans could perhaps be combined in a single system. This research not only suggests that GP evolved code can successfully intereact with human written programs, but also indicates that GP is capable of handling the difficult, noisy, real-world problem of OCR.

Acknowledgements

I am grateful to Lanre Amos, John Koza, Scott Coon, and my family for invaluable advice and support.

Bibliography

Amos, L. (1993). A method for robust segmentation of the boundaries of printed characters. (Canon Research Center Technical Report), Palo Alto, CA: Canon Research Center of America.

Andre, D. (1994). Automatically defined features: The simultaneous evolution of 2-dimensional feature detectors and an algorithm for using them. *Advances in Genetic Programming*. (Kim Kinnear, Ed.). Cambridge, MA: MIT Press.

Andre, D. (1993). A fast one pass raster-scan method for boundary extraction in binary images. (Canon Research Center Technical Report), Palo Alto, CA: Canon Research Center of America.

Koza, J.R., (1993). Simultaneous discovery of detectors and a way of using the detectors via genetic programming. *1993 IEEE International Conference on Neural Networks, San Francisco.* Piscataway, NJ: IEEE 1993. Volume III. p 1794-1801.

Koza, J.R., (1992). *Genetic Programming: on the programming of computers by means of natural selection,* Cambridge, Mass: MIT Press.

Tackett, W. (1993). *Simple Genetic Programming Code*. Obtained via FTP from the author.