Discovery of Optical Character Recognition Algorithms using Genetic Programming

Polina K. Spivak
10861 Santa Teresa Drive
Cupertino, CA 95014
polina@eeeyore.com

Abstract: Optical character recognition is a trivial problem, at least for literate humans. However, creating a good character recognition program is not so simple—a single character can have significant variability when considered across many fonts. Some characters are misleadingly similar, especially considering their incarnations in multiple fonts.

This paper discusses a technique, as well as some pitfalls, for automatically evolving optical character recognition programs using genetic programming. The problem-specific information required for this technique is a set of training characters, e.g. { 0, …, 9, a, …, f }, and GIF incarnations of those characters in different fonts. The result is a set of algorithms that can determine which character is represented by an image. The algorithms can be applied to those images used to evolve the algorithms, as well as new images that represent one of the characters in a new font not seen during evolution.

Introduction

Optical character recognition serves an important function in today's world. It allows existing paper documents to be easily converted into the digital domain, which enables efficient storage, searching, manipulation and distribution of the documents. Optical character recognition has other practical uses, such as helping vision-impaired people read printed matter. Clearly, this is a compelling software problem.

The approach taken in this paper is to use genetic programming to evolve algorithms for recognizing characters. Many other approaches are possible. For example, all the known characters, in each known font, can be stored in a database. However, this would take up a large amount of space (not feasible for small machines such as PDAs) and cannot handle new fonts that are not yet in the database. Algorithms for character recognition can be written by software developers, but this is error prone because people necessarily cannot evaluate as many "trial" characters as computers to form a basis for their algorithm's decisions. Another solution would be to use a very good typist, but of course this is not cost effective since human labor is vastly more expensive than computer processing power for this sort of application. So, we return to the idea of letting computers do our programming work for us, and evolve the optical character recognition algorithms.

The scope of this paper was limited to a set of 16 characters, namely the lower-case hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f). Each character was represented by multiple 10x10 GIF images (each using a different font). The goal of the algorithm was to take as input an image, and determine which of the sixteen known characters it represents. The specific images used for this paper are shown in Appendix A, although there is nothing special about this choice of characters and fonts.

Methodology

This section discusses the most successful approach taken by the author to solve the 16-character recognition problem. Other approaches, and the reasons they were less successful, are described in the Pitfalls section.

At a high level, each individual in a population of optical character recognizers evolved by genetic programming must take as input an image, and produce a result that specifies which character is represented by the image. There are two ways to structure the individuals—each individual can produce one or many outputs. If the individual produces one output, then the output is interpreted as the single character that the algorithm believes is represented by the image. For example, the individual might produce a single integer, which represents the ASCII code of a character. If the algorithm uses boolean rather than integer values, then the output would be the integer's binary encoding, which would correspond to a 4-bit result (interpreted as a single result value) for the 16-character set used in this paper. The advantage of this approach is that an individual uniquely identifies a particular character. When the character is identified correctly, there is no ambiguity. However, when a character is identified incorrectly, there is no "fallback" character, so it's not possible to determine what other character it might be. Also, this algorithm gives no preference to checking common characters first, which could be an important optimization when performing character recognition on a lengthy document.

The second structure, and the one used in this paper, is to have individuals that produce one boolean result per character, in this case 16 boolean results. The n^{th} result specifies whether the individual believes the image corresponds to the n^{th} character. For example, if result 8 is "true" for a given image, then the individual believes the image represents the character "8". Ideally, for every image, exactly one of the results will be "true" and the others will be "false", uniquely identifying the correct character. However, if some new image is encountered, and the individual is not quite certain which of two or three characters it represents, the individual can have multiple "true" results. For example, "true" values for results 0 and 8 indicate that the character is one of "0" and "8". This allows for false positives, which can be a helpful hint when reading whole words and using a spell-checker in conjunction with single character recognition. For example, if the character set included "1" (one) and "l" (the lower-case letter L), the individual might respond positively to both characters and let the spell-checker choose the letter (lower case L) because the following characters were "engths", forming the word "length".

Having settled on the multiple result producing branches approach (one branch per recognized character, 16 in this case), we note that each branch is completely independent of every other branch. So, instead of evolving an individual with 16 branches, we can achieve equivalent results by evolving 16 individuals with one result producing branch and a fitness function that takes into account the index of the individual being evolved. This greatly simplifies the complexity of the genetic programming problem, and very good solutions emerge with far less computation. The Pitfalls section expands on the fallacy of using a single individual that can be broken down into multiple independent individuals.

The input to the algorithm is a monochrome 10x10 GIF image. This is translated (by a Java program) into an array of 100 boolean values, one for each pixel, where true and false correspond to black and white, respectively. Each of the 100 boolean values corresponds to one of the 100 boolean "pixel" terminals, p0 through p99.

Table 1 specifies the tableau used in this paper's solution to the 16-character 10x10 pixel recognition problem.

Table 1: Tableau for Single-Character Optical Character Recognition

Objective	For each character in the set { 0, …, 9, a, …, f }, Find a minimal strategy for recognizing the character. "Minimal" refers to the size of the individual. "Recognizing" the character means responding "true" when the input is an image of the character, and responding "false" otherwise.

Terminal Set	Boolean values p0, …, p99, where pN represents the Nth pixel in the input image.
Function Set	Boolean functions: Not (1 parameter) And (2 parameters) Or (2 parameters) IfThenElse (three parameters)
Fitness Cases	96 monochrome 10x10 pixel images, described in Appendix A.
Raw Fitness	Sum of the error, over the 96 fitness cases, plus the size of the individual (which is only added once, *not* once per fitness case). The error for a single fitness case is: 200 * 0 for a correct response 200 * 1 for a false positive 200 * 20 for a false negative A false positive is when the algorithm should respond "true" only to an image of character X, but responds "true" to an image of character Y (where X ≠ Y). A false negative is when the algorithm should respond "true" only to an image of character X, but responds "false" to an image of character X.
Standardized Fitness	Same as raw fitness. Smaller errors and smaller individual sizes are better.
Hits	Number of fitness cases that had correct responses. This is equivalent to the raw fitness, minus the size of the individual.
Wrapper	None.
Parameters	G = 200 M = 2000 Probability of crossover = 0.1 Probability of reproduction = 0.9 Basic selection method is tournament selection, with seven individuals participating in the tournament. Initial population is created using the Ramped Half-and-Half method, with trees of size 2 to 6.
Success Predicate	None. The evolution continues for the specified number of generations, to allow smaller (simpler) individuals to evolve.

Using an appropriate raw fitness function was crucial to achieving good results. The fitness function takes into account three factors: size of the individual, false positives, and false negatives. False negatives (when a character was *not* recognized) do the most damage to the individual's score. False positives (when an incorrect character is mistakenly recognized) also contribute heavily, since the individual's primary goal is correctness. However, mistakenly recognizing the wrong character corresponds to the individual being uncertain about which of a set of characters is the correct one. As mentioned earlier, this error can be mitigated by a

spell-checker and is not as serious as completely failing to recognize a correct character image. Any correctness errors are the primary factor in an individual's score—any error on any image is valued 200 times more important than the last factor, which is the individual's size. This allows the population to initially select primarily for correctness.

When enough correct individuals have been found, the smaller, simpler individuals are then preferred (and any individuals with less than perfect correctness are on the path to extinction). This aspect of fitness is worth much less than correctness—focusing the evolutionary pressure on individual size too early in the evolutionary process works against the population diversity that is required to find perfectly correct solutions. The smaller individuals are better for two reasons. The superficial reason is that we humans can easily understand how they work, and can more easily write a paper about simpler individuals. However, there is a more important reason, which is that they are more likely to look for the minimal "key" set of attributes of a character, increasing the chances of recognizing these key attributes in a new image of the character that was not available during evolution. In other words, smaller algorithms are more likely to adapt well to an environment with incarnations (fonts) of known characters.

The population size was also important in finding solutions quickly. There are 100 pixels in each image, so an initial tree of depth 2 to 6 will not use many of these pixels. If the population size is small, then a small number of pixels will be represented. If the population size is very large (in this case, 20 times the number of pixels), it is very likely that every important pixel is represented many times and will be available for evolution.

Results

Table 2 shows the best individual character recognizers, taken from the best results of 24 runs (for each character).

Table 2: Best Single-Character Optical Recognition Algorithms

Character	Individual	Accuracy on New Font
0	`(& (? p24 p58 p47) (& p77 (~ (| p35 p54))))`	93.75%
1	`(? (| p56 p36) p70 (? p74 p95 p75))`	100%
2	`(& (? (? p83 p56 p22) p74 p22) (~ p57))`	93.75%
3	`(& (~ (| (| p73 p52) p42)) (? p57 p5 p66))`	93.75%
4	`(& (? p25 p75 p24) p56)`	93.75%
5	`(& (? p87 p7 p67) (& (~ p62) (~ p47)))`	93.75%
6	`(& (| p25 p62) (? (| p26 p35) p78 (~ p37)))`	81.25%
7	`(& (? p17 p65 p2) (~ p95))`	100%
8	`(& (& p45 p22) (| p65 (& p37 p62)))`	100%
9	`(& (& (~ p63) p53) (| p38 p27))`	100%
a	`(? p66 (? p96 p21 p73) p88)`	100%
b	`(? (| p15 p5) p69 p13)`	93.75%
c	`(~ (| (| (| p74 p56) p67) (| (| p45 p4) p58)))`	100%
d	`(& (~ p65) (? (& p26 p45) p34 p97))`	100%
e	`(& (& (? p47 p52 p56) (| p34 p47)) (? p84 p53 (~ p96)))`	93.75%
f	`(& (? p73 p33 p24) (& p34 (~ p95)))`	100%

Given a relatively small training set (only 6 different fonts of each character), the best character recognizers did well on a new font that had not been seen during the evolution process. Fully half the recognizers performed completely accurately on the new font. All but one of the rest missed only one out of the 16 new images in the new font. Only one recognizer (for character 6) missed 3 out of 16 images in the new font, but this is still better than 80% accuracy.

As an example, table 3 shows the interpretation of the algorithms for some characters.

Table 3: Interpretation of Select Individuals

Character	Individual	Interpretation
0	(& (? p24 p58 p47) (& p77 (~ (| p35 p54))))	Pixel 24 is the top left corner of the "0"—based on where it is, the algorithm checks the middle right corner pixel. Pixel 77 is the bottom left corner. Pixels 35 and 54 inside the "0" and should not be on.
1	(? (| p56 p36) p70 (? p74 p95 p75))	Pixels 56 & 36 are too far right, so this "IF" branch is useless. For a thick character "1", pixels 74 & 95 are both on. For others, pixel 75 is on, and is rare in other characters.
4	(& (? p25 p75 p24) p56)	Pixels 25 and 24 are at the peak of the 4 (at the top). At least one of them must be on in any character 4. Pixel 75 is in the bottom horizontal row. Pixel 56 is either part of the vertical line, or part of the tail (right part) of the horizontal line.
7	(& (? p17 p65 p2) (~ p95))	Pixel 17 is part of the top right corner of the seven. If it off, then pixel 2 (the top left pixel) must be on. Pixel 2 occurs very rarely in any other character. Pixel 65 represents the middle of the character—it is common in other characters, so it is "combined" with pixel 17 to recognize this character. Pixel 95 is a middle pixel in the bottom row. It is never on for character "7", but requiring this to be off eliminates some other characters (in certain fonts), such as 0, 1, 2, 3, etc.

The evolution of the algorithms for the characters proceeded as expected—the initial population performed very poorly, but the algorithms quickly converged to properly recognizing most characters. Because there was no success predicate, the evolution continued, causing smaller correct individuals to be found. The evolution of the best algorithm for character 0 serves as a good example.

Table 4: Evolution of Best Algorithm for Character "0"

Generation	Mean Fitness	Best Individual	Best Fitness	Interpretation
0	21287	p37	6401	Most of the population failed to recognize most characters. The best individual found a pixel that is on for all images of "0", and off for many other images (all but 32).
1	17696	(& (| p37 p25) (| p89 p77))	4607	The population has improved significantly. The best individual preserves the useful pixel p37 (from generation 0), and finds other useful pixels that serve to eliminate false positives. It responds correctly to all but 23 images.

2	15077	(& (~ p44) (| p91 p37))	3806	The population improved somewhat. The best individual again preserves pixel 37, and uses some new pixels to improve eliminating characters other than 0.
3	13785	Too big to fit: size of individual is 90 nodes	2091.0	The population again improved slightly. The best individual still preserved pixel 37 as an important part of its function, but added a very large number of nodes to eliminate characters other than 0. It has traded size and simplicity for better accuracy—almost twice as good as the best individual of the previous generation.
8	5613	Too big to fit: size of individual is 61 nodes	61	The population has steadily improved. A perfectly correct individual is found in this generation. It preserves pixel 37, which was identified early on as a great marker for character "0". However, this individual is overly large and complicated, so the evolution continues.
28	6740	(& (~ p35) (& (? (| p17 p57) (| p47 p39) (? p7 p4 p68)) (~ (| p65 p54))))	19	The population has gotten worse, but the best individual has gotten much better—it is three times smaller than in generation 8. It no longer uses pixel 37, and has found other pixels nearby that serve a similar purpose (e.g. pixels 35 and 47).
50	7140	(& (~ p35) (& (? p57 p47 p68) (~ (| p65 p54))))	12	The best individual is found. Perhaps a better one exists, but it is not obvious and cannot be much better than this one.

Results for other characters were very similar. On most runs, a correct algorithm was found that was no more than twice the size of the best solution. About 2% of the runs failed to find a correct solution, giving incorrect responses to one or two characters out of the set of 96 test cases.

Pitfalls

Before the method discussed in the Methodology section was discovered, there were several misguided attempts at finding efficient ways to evolve optical character recognition programs. The interesting cases are described here, because they highlight some aspect of the problem.

At first glance, it seems that generating a single unambiguous integer result is the best solution. It would seem to scale well in the number of characters, because a 16-bit integer value can represent most known characters (by using the UTF16 16-bit Unicode encoding). However, the major problem with this approach was that simple boolean operators like "AND", "OR", "NOT" and the "IF" part of "IF-THEN-ELSE" are crucial to a decision problem. These operators require boolean values, and their results generated a huge number of 0 and 1 values (integer equivalents of boolean values) into a tree that was required to produce a non-boolean value. This made it very difficult for the algorithm to find a correct solution. After 10,000

generations with population size 600, only about 75% of the characters were properly recognized.

With the integer approach, there was also the enticing possibility of using sophisticated functions, such as counting the number of black pixels in a given region (by specifying a bounding box, with integer row and column indices. Humans naturally recognize characters by finding distinguishing markers, such as a long vertical line of connected pixels on the left side for the digit "1". However, the extra functions only made the evolution slower, especially since many of the inputs were results of boolean functions rather than the indices they were meant to be.

Another problem, with both boolean and integer typed individuals, was using a single result-producing individual versus multiple independent individuals. The single result means that a single algorithm must properly recognize *all* known characters, so the algorithm is significantly more difficult to evolve than many smaller single-character recognizers. Each of the smaller, independent single-character recognizer runs found a perfectly correct individual within the first 20 (usually even less) generations.

Performance

The performance bottleneck of the technique described in this paper is that, to recognize a single character, the algorithm for each known character must be run. Most real applications would read whole documents, and require the ASCII character set at a minimum for simple English text. Other languages, even those that use Latin characters, may have additional requirements such as accents, umlauts and other character modifying symbols. Some languages, in particular for Asian locales, have exponentially more characters than the Latin character set. Running a large number of algorithms, even if each one is fairly simple, would be very time consuming and not practical when it occurs per character, for a document that contains thousands (or millions or more) characters.

One simple approach is to run the algorithms in an order based on the character's frequency. For example, the character "e" appears more frequently than most other characters, so the algorithm for recognizing "e" should be executed before the algorithm for recognizing "z".

Another approach is to look for similarities among algorithms. For example, characters "1" and "4" typically have a vertical line on the right side of the character, so the algorithms might both look for this "marker". Such commonalities could be exploited, so that the bit of algorithm that looks for a marker is only executed once. As each algorithm proceeds in evaluating a given character, it could save such "marker" information for use by other algorithms, or use previously found marker information.

Other solutions are undoubtedly possible, and some combination of them is certainly necessary to produce an optical character recognition tool that performs well in practice.

Future Work

Optical character recognition is useful when it can recognize characters within a document, rather than a single image of a single character. One obvious next step would be to write a wrapper program that splits a document into single characters. For example, it could use horizontal uninterrupted whitespace as a heuristic to distinguish lines of text, and then use character connectivity (the vertical whitespace between characters) to separate individual characters and words. The OCR program's determination of the characters, combined with a spell checker for words (to correct small mistakes made by either the OCR program or the wrapper character-separating program), could be an effective and reliable document scanning software tool.

Another possibility for this approach is for character recognition. Many of the currently available handwriting recognition tools, like those used in Palm Pilots and other PDAs, require the user to write the software's version of the characters, which are often quite different than the cursive letter style many people use. If a user were to provide a sample of his or her

handwriting, then the PDA could "learn" to recognize the user's handwriting, even if it is distinct from the norm. This simple application could be lifesaving—doctors' handwriting is notoriously illegible, and people die every year because a pharmacist misread a doctor's scribbles.

Conclusions

This paper presented a proof-of-concept—genetic programming can be used to automatically generate competent, simple algorithms for the optical character recognition problem. For the characters used in this paper, the automatically generated algorithms were able to recognize the characters in the "training set" of images 100% accurately, and also performed quite well on characters that had not been seen during the algorithm's evolution.

The technique described here is not limited to the concrete set of characters used to produce the results. It should work on a wider and more diverse set of other characters, other fonts and other sizes, to produce a more comprehensive solution to the problem. With some additional programming, as mentioned in the Future Work section, the algorithms derived by this technique can form the building blocks of a program that performs optical character recognition of entire documents.

Acknowledgements

The ECJ8 suite is a simple, powerful, and extraordinarily well-documented genetic programming implementation. It was especially useful for this paper because ECJ8 is implemented in Java, and Java has built-in support for handling images.

ECJ8 is available at http://www.cs.umd.edu/projects/plus/ec/ecj/. This paper's results were obtained on a Windows PC, but any platform that supports Java can be used.

References

Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT Press 1992.

Appendix A: Character Images

Table 5 shows the images that were with the algorithm. The first six fonts were used to train the algorithms. The seventh font was not used to train the algorithms, but rather to test the algorithms on previously unseen representations of the known characters.

Each image is a 10x10 monochrome bitmap (100 pixels, or 100 bits), though it is shown here at approximately three times the size to enable the reader to clearly identify the characteristics of each image. Each image was represented to the algorithm as 100 boolean inputs (pixels), where true and false correspond to black and white, respectively. The boolean inputs were named p0 … p99, representing the bits in row-order fashion (e.g. B90 is the pixel in the bottom left corner, which is always white/false).

Table 5: Character images used as test cases

	Font 1	Font 2	Font 3	Font 4	Font 5	Font 6	Font 7
"0"							
"1"							
"2"							
"3"							
"4"							
"5"							
"6"							
"7"							
"8"							
"9"							
"a"							
"b"							
"c"							
"d"							
"e"							
"f"							

The images were generated by typing characters into a drawing program and clipping a 10x10 pixel square from the image, centered on the character, and saved in GIF format. To apply the technique in this paper to a wider set of fonts and characters, it would be necessary to find a ready source of images rather than creating them by hand.

To convert the GIF images into valid program inputs (100 boolean values), a Java program was used. It read content from a GIF file, created an Image object, and translated each pixel to a boolean value depending on whether the pixel was black or white.

Appendix B: ECJ8 Setup

The ECJ8 software requires a parameters file to run the evolution. The following is the parameter file used to achieve the results in this paper.

```
parent.0 = C:/polina/ECJ8/ec/gp/koza/koza.params

# Random Seed
seed.0 = 152874921

# Uncomment here to use fitness proportionate selection
#breed.reproduce.source.0 = ec.select.FitProportionateSelection
#gp.koza.xover.source.0   = ec.select.FitProportionateSelection
#gp.koza.mutate.source.0  = ec.select.FitProportionateSelection
stat.file = $out.stat

generations = 600
pop.subpop.0.size = 2000

# Specifies the character that the individual should recognize
eval.problem.char = 9
eval.problem.numchars = 16
eval.problem.numfonts = 6

# Define the function set
gp.fs.size = 1
gp.fs.0.name = f0
gp.fs.0.info = ec.gp.GPFuncInfo
gp.fs.0.size = 104
gp.fs.0.func.0 = gp.func.Pixels$C0
gp.fs.0.func.0.nc = nc0
gp.fs.0.func.1 = gp.func.Pixels$C1
gp.fs.0.func.1.nc = nc0
# Also need functions for pixels 2 to 98, omitted here for simplicity.
gp.fs.0.func.99 = gp.func.Pixels$C99
gp.fs.0.func.99.nc = nc0

gp.fs.0.func.100 = gp.func.Not
gp.fs.0.func.100.nc = nc1
gp.fs.0.func.101 = gp.func.And
gp.fs.0.func.101.nc = nc2
gp.fs.0.func.102 = gp.func.Or
gp.fs.0.func.102.nc = nc2
gp.fs.0.func.103 = gp.func.IfThenElse
gp.fs.0.func.103.nc = nc3

# We specify our problem's custom Java code here
eval.problem = gp.OCRProblem
eval.problem.data = gp.OCRData
eval.problem.stack.context.data = gp.OCRData

pop.subpop.0.species.ind.numtrees = 1
pop.subpop.0.species.ind.tree.0 = ec.gp.GPTree
```