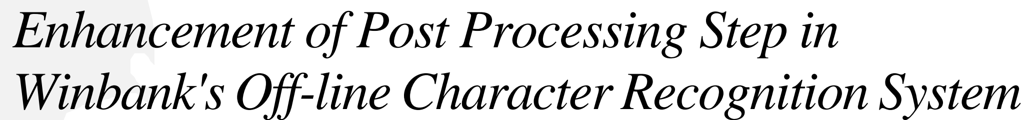
Tony Wu
Advisor: Dr. Gupta

6.199 Advanced Undergraduate Project



Enhancement of Post Processing Step in Winbank's Off-line Character Recognition System

Introduction

Winbank is a project under Sloan's Productivity From Information Technology Initiatives (PROfIT) group. It is a software application capable of locating the courtesy amount box on a scanned bitmap image of a check and then recognizing the handwritten numerical amount in the box. This application would be very useful to banks, as more than 60 billion checks are processed each year in the United States alone. However, banks do not use such technology today because the recognition rates for most automated check processing products[1] are still not deemed to be adequate for broad commercial use.

There are three steps in Winbank's handwriting recognition routine. The first step involves preprocessing of the scanned bitmap image. After the image has been preprocessed, neural net recognition is performed on the normalized images of each character. Finally, post processing examines the numeral bitmap and output of the neural net to produce certain confidence measures about the prediction by the neural net.[2]

This report discusses the design and development for the Winbank application, from the flow of the different modules to the training process for the neural net. Also discussed are the details of each of the three steps involved in recognition. A discussion of the improvements made in the post processing step and its performance follows. Results of an attempt to test on thousands of Brazilian checks will also be discussed, followed by suggestions for further improvements on the software application.

[1] Results of demos obtained from Mitek and Orbograph had around 50-60% recognition rates
[2] http://scanner-group.mit.edu/winbank.html

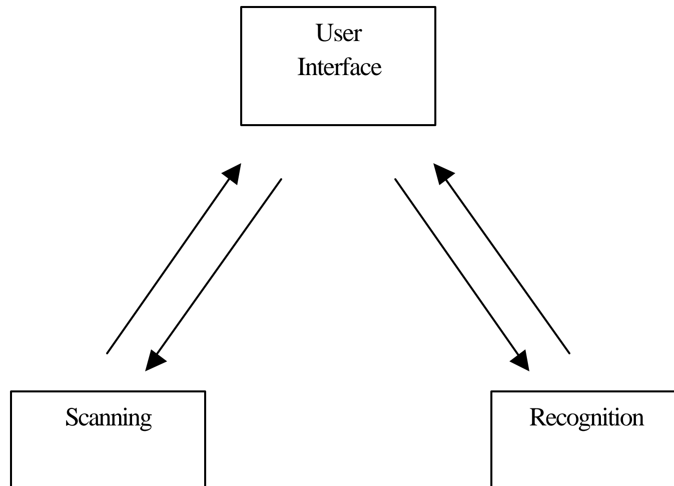
Winbank Application Design and Development

The Winbank Application was first developed in 1993 in C for use on the Windows 3.1 platform. Later, in 1997, it was ported to Windows 95/NT. The 34 files containing functions used for developing Winbank can be divided into three categories: Scanning, User Interface (Windows Programming), and Recognition. The table below lists the files belonging to each category.

Scanning Modules	User Interface Modules	Recognition Modules
Scanner.cpp	Proto2.cpp	Thinning.cpp
Scl.cpp	Proto2.rc	Thicken.cpp
Decomp.cpp	Aboutbox.cpp	Slant.cpp
Winmem.cpp	Testbox.cpp	Th3.cpp
Winscan.cpp	Scanbox.cpp	Str2.cpp
	Percentb.cpp	Segutil.cpp
	Optionsb.cpp	Segments.cpp
	Font.cpp	Profile.cpp
	File.cpp	Postproc.cpp
	Bitbox.cpp	Normal.cpp
	Size.cpp	Hit2.cpp
		GetBlob2.cpp
		Doscan2.cpp
		Delimit.cpp
		Check_ch.cpp
		Cents.cpp
		Back.cpp
		Utility.cpp

Table 1: Files under each category

The Scanning Modules were provided by Hewlett-Packard and are necessary for the scanning operations. The User Interface Modules create the interface for Winbank and responds appropriately to user input. Finally, the Recognition Modules contain various algorithms to preprocess, recognize, and post process the check image.



A general flow diagram between the three modules

On the Winbank prototype, the user can input an image in two ways. First is by scanning in a check, and second by loading a pre-scanned image. The courtesy amount block is stored as a 300x100 bitmap and then segmented. The segmented digits are saved in a file so users are allowed to see the pre-normalized segmentation effects. After each phase of the preprocessing steps, the results are stored in files so users can see the effects of each step.

Another option users have of testing Winbank's neural net accuracy is through the "Input-Test" option. The user is allowed to input a 16x16 image and it is passed through to the Recognition Engine and the digit with the highest confidence level is displayed.

Neural Net Training

The weights for the neural net need to be retrained whenever the preprocessing steps are altered. This is due to the fact that the training of the neural net takes place after the preprocessing steps. Also, the order of the standardization stages in the preprocessing steps must be maintained in order for the neural network to function properly since different standardized images are produced when the ordering changes. There are six

possible combinations involving the normalization, slant correction, and thinning/thickening and it was determined from previous research that the NST (normalization first, slant correction second, thinning/thickening third) combination yields the best results.

Currently, the neural net for Winbank is trained on numbers provided in the NIST (National Institute of Standards and Technology) database. Separate programs have been written to segment the images provided by NIST and preprocess the images for neural net training. Also, images from actual checks have been scanned in and used as images for training and have yielded slightly better results.

The preprocessed images used for training are usually stored as text files in the following format: First, the actual number of an image is shown, followed by a 16 x 16 image. Each file contains several entries of different numbers and its corresponding images. An example of an entry is shown below:

```
0
0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0
0 0 1 1 1 1 0 0 1 1 1 1 1 0 0 0
0 0 1 1 0 0 0 0 0 0 1 1 1 1 0 0
0 1 1 1 0 0 0 0 0 0 0 0 1 1 1 0
0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1
1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1
1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1
1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1
1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1
1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0
1 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0
0 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0
0 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0
0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0
```

Example of an entry for a 0

This text file is fed into the neural net training program called back2.c, which in turn generates the weight files used by the neural net in Winbank. It is important to remember to separate the training sample from the testing sample as the training sample will always improve its accuracy after each epoch, while for the testing sample, the accuracy actually increases up to a global maximum at a certain epoch and then decreases afterwards.

Pre-processing steps

After the check is scanned, the courtesy amount is saved as a bitmap. The bitmap is then preprocessed for recognition through various algorithms to separate individual characters and to standardize the characters.

Segmentation

The first step in the preprocessing stage is segmentation. Segmentation divides each numeral and symbol into its own separate unit. Each segment is then analyzed individually. Before the actual optical character recognition stage, punctuation marks such as commas and periods are identified based on their location, alignment, and size within the segment image.[3]

Standardization

Now that the characters have been separated, the next step in the preprocessing involves passing each digit through various algorithms to standardize them. The character is normalized to a standard size of 16x16 pixels, followed by a process of slant correction during which a slant correction algorithm corrects the angle of the slant to an

[3] Sparks, Nagendraprasad, and Gupta

upright, vertical position by rotating the number until it reaches its minimum width.[4] Following this are two complementary processes: thinning and thickening. Thinning involves turning the numeral into a bare skeleton of one pixel thickness. A good skeleton retains the connectivity and structural features of the original pattern and, once defined, is then thickened to a thickness of two pixels.[5] The thickened character is now ready for neural net recognition.

Neural Net Recognition

The neural network consists of 256 input nodes corresponding to each of the 16x16 positions in the image, 40 hidden nodes, and 10 output nodes corresponding to each possible digit. Two passes are made through the neural network, first with positive nodes weight files and then the complement nodes[6] weight file. The results of the two passes are used to calculate the confidence level for each output node. Winbank is given the location of the weight files and loads them appropriately.

Post-processing

After the neural net has produced confidence levels for all possible digits, the confidence levels are examined to see if any reach the required tolerance level. In the special case where the confidence levels are high for two potential digits, a special post processing algorithm examines the image and determines which of the two digits is the

[4] Nagendraprasad, Gupta, and Feliberti
[5] Nagendraprasad, Wang, and Gupta
[6] Mui, Agrawal, Gupta, and Wang

actual digit. The post processor is activated when the highest confidence is greater than a certain pre-set threshold, and the second highest confidence is within a pre-set difference of the highest confidence

Of the potential 45 distinct pairs of digits, 29 are checked using the functions in check_ch.cpp. Each function uses one of thirteen different types of algorithms. The following table lists each distinct pair, whether it is checked by the post processing routine, and if so, which type of algorithm it uses.

First Digit	Second Digit	Checked? Type	First Digit	Second Digit	Checked? Type
0	1	Yes: 1-A	3	4	Not Checked
0	2	Yes: 1-A	3	5	Yes: 5
0	3	Yes: 1-B	3	6	Yes: 6
0	4	Yes: 1-A	3	7	Not Checked
0	5	Yes: 1-A	3	8	Yes: 7
0	6	Yes: 1-C	3	9	Yes: 8
0	7	Yes: 1-A	4	5	Not Checked
0	8	Yes: 3	4	6	Not Checked
0	9	Yes: 1-A	4	7	Not Checked
1	2	Yes: 2	4	8	Not Checked
1	3	Yes: 2	4	9	Yes: 9
1	4	Yes: 2	5	6	Yes: 10
1	5	Yes: 2	5	7	Not Checked
1	6	Yes: 1-A	5	8	Yes: 11
1	7	Yes: 2	5	9	Yes: 12
1	8	Yes: 1-A	6	7	Not Checked
1	9	Yes: 1-C	6	8	Not Checked
2	3	Not Checked	6	9	Not Checked
2	4	Not Checked	7	8	Yes: 1-A
2	5	Not Checked	7	9	Yes: 13
2	6	Yes: 4	8	9	Yes: 1-A
2	7	Not Checked			
2	8	Not Checked			
2	9	Not Checked			

Table 2: Pairs of digits checked by the post processing scheme

Common Attributes

Each of the functions for post processing is called checkXandY, where X and Y are the distinct digits being checked. The input to the function is the 16x16 bitmap,

represented as an array of 256 integers. The functions return an integer whose value is X or Y, and -1 in some cases when the digit cannot be determined.

General Changes Made to Improve the Post Processing Step

The original check_ch.cpp file failed to compile because it caused more than 300 warnings. Since the compiler (Borland C++ 5.02) refuses to compile any file with more than 100 warning messages, the file had to be modified to eliminate the error messages. It seemed that whoever worked on the original file made the same mistakes over and over again. The mistakes include declaring variables that were never used, assigning values to variables that were not used, and allowing a function to not return a value when it needs to. All the warning messages were examined and eliminated. Another mistake fixed that the indexing to the array is incremented at the wrong time.

Another improvement made was to eliminate the unnecessary loops through the bitmap image, since most of the code checks for only a few rows or columns. By doing so, the speed of each function is increased.

Also, there were no comments to indicate what each function was doing. After careful examination of the code, each function is now properly commented so that future developers will have a good comprehension of the way each function works. The comments on each of the functions are discussed below as well as any changes made to improve each function.

Type 1: Checks for Loops

The Type 1 algorithm checks for loops in the digit. For example, the difference between a 0 and a 6 is that on the top half of the 6, there is no loop. So the algorithm scans the first 6 rows of the bitmap and checks how many times the bits are flipped. The bitmap for the fourth row of each digit would look like this: 0000000110000000 for the 6 and 0011000000111100 for the zero. In the case of the 6, we have a string of zeros, followed by 2 ones, and then back to a string of zeros, which results in two distinct bit flip cases (0->1 and 1->0), while for the 0, there are four bit flip cases. As soon as the bit flip counter for a row hits 3, then there is a loop in that row. If there are more than 2 loops counted, the algorithm returns the number that should have a loop in the area checked. Otherwise, it returns the number that should not have a loop.

```
0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0          0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0
0 0 1 1 1 1 0 0 1 1 1 1 1 0 0 0          0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0
0 0 1 1 0 0 0 0 0 0 1 1 1 1 0 0          0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0
0 1 1 1 0 0 0 0 0 0 0 0 1 1 1 0          0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1          0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1    vs.   0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1          0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1          0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1          0 0 1 1 0 0 0 0 0 0 0 1 1 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0          0 0 1 1 1 0 0 0 0 0 0 0 1 1 0 0
1 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0          0 0 1 1 1 0 0 0 0 0 0 0 1 1 0 0
0 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0          0 0 0 1 1 1 0 0 0 0 0 0 1 1 0 0
0 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0          0 0 0 0 1 1 0 0 0 0 0 1 1 0 0 0
0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0          0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0
```

0 vs. 6 case

The Type 1 algorithm can be further divided into three different categories, A, B, and C, depending on where the algorithm is checking for loops. Type 1-A checks the bottom six rows, while 1-B checks the middle six rows, and finally 1-C checks the first six rows.

The original code checked for loops incorrectly. It counts the number of bit flips in each row as it goes through each column in the row. And when the number of bit flips

is greater than 3, the number of loops is incremented. This is wrong because it will double count for any further bit flips because a loop usually has 3 or more bit flips. The code was changed to increment the loop counter only once, when the number of bit flips reaches 3 at any row. Testing of the code showed that it was rarely used a high thresholds[7].

Type 2: Checks for 1's

The second algorithm is a relatively simple. It is generally used for checking between a 1 and another number. Since a 1 is usually relatively thin, there are usually few pixels that are on at the outside columns. The algorithm runs through the first and last five columns and keeps track of the number of on pixels[8]. If that sum is more than six, then it most likely is not a 1 and is instead the other number.

```
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0          0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0          0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0          0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0          0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0    vs.   0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0          0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0          0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0          0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0          0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0          0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0          0 0 0 1 1 0 0 0 0 0 0 0 1 1 0 0
0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0          0 0 0 0 1 1 0 0 0 0 0 1 1 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0          0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0
```

1 vs. 5 case

---

[7] High threshold is > 0.5

[8] An "on pixel" is a 1 in the bitmap, and an "off pixel" is a 0

In the example above, the 1 has four pixels on in the outside columns, while the 5 has 33 pixels on in the outside columns. Originally, the algorithm returned a 1 if there were less than two on pixels on the outside columns. However, this would not handle the case of the 1 shown above because of the horizontal line at the bottom. Testing showed that this was correct about half the time (6 out of 12) at high thresholds. An example of the post processor being correct is as follows. If the actual number is a 1, and the neural network's highest confidence was 5 and its level is above the threshold. Then if its second highest was 1 (and its level is within the difference), and the post processor checks this case and returns a 1 instead, then it was correct. Conversely, if the neural network's output was correct but then the post processor returns the second highest confidence output, then it is said to be wrong.

Type 3: Checks 0 and 8

The third algorithm deals specifically with the case of checking between a 0 and an 8. Since a 0 is much like an 8 except that its middle is hollow, the algorithm checks a 3x3 square in the middle of the bitmap. If the number of on pixels in the square is greater than 2, the algorithm returns an 8. Otherwise, the square is relatively empty and it returns a 0.

No specific changes were made to the original code, and testing of the code showed that it was rarely used at high thresholds.

Type 4: Checks 2 and 6

The third algorithm deals specifically with the case of checking between a 2 and a 6. This algorithm works on the premise that the on pixels at the midsection of a 2 are located towards the right, while for a 6, they are towards the left. This code basically compares the number of pixels that are on from the left side to the ones from the right side. If there are more pixels that are on for the left, than it returns a 2. If there are more on pixels on the right, a 6 is returned. Otherwise, if the numbers of on pixels are the same for left and right, a -1 is returned to indicate that the post processing could determine whether it was a 2 or a 6.

No changes were made to the original code, and testing showed that the code was correct compared to the neural network four times and wrong only twice at high thresholds.

Type 5: Checks 3 and 5

The third algorithm deals checks between a 3 and a 5 by going from the 4th row to the 7th row and summing the location (column) which an on pixel appears. It then divides by the number of on pixels encountered to determine the average of the location of those pixels. If the average on pixel's location is below 8.5, then it is towards the left, and returns a 5. This is because of the tendency of the top middle segment of a 5 to be towards the left. Otherwise, if the average is above or equal to a 7.5, then it returns a 3 because the top middle segment of a 3 is usually on the right side.

The original divided the sum of the locations by sixteen to get the average. However, this is incorrect because it assumes that there are only sixteen on pixels in the rows. An extra variable to count the number of hits was added and used to get the correct

average. Testing showed that it was wrong most of the time (3 out of 8) at high thresholds.

Type 6: Checks 3 and 6

This algorithm checks between a 3 and a 6. It works on the premise that a 6 has a curve on the left side, so the first on pixel of each row would be relatively close to the left edge, while a 3's first on pixel for its middle rows would be farther to the right. The algorithm goes through the 9th, 10th, and 11th row to find the first on pixels on those rows. Then, it averages the location of the two rightmost on pixels in those three rows and returns a 3 if it is greater than 5 and 6 otherwise.

No changes were made to the original code, and testing its performance showed that this case was never entered.

Type 7: Checks 3 and 8

The Type 7 algorithm deals specifically with the case of checking between a 3 and an 8. It operates on the assumption that a 3 is like an 8 except that it is missing the left curves in the middle rows. Therefore, the first on pixels for those rows are likely to be farther away. The fourth, fifth, tenth, and eleventh rows that are checked, and the average of the first on pixels of those rows are calculated. If the average is below 5.5, then it returns an 8 because it means that it is relatively close to the left edge. Otherwise, it returns a 3 due to the lack of the left curve causing the first on pixel to be farther out.

Only the general changes were made to the code. Testing the code revealed that the code was correct 3 out of 3 times at high thresholds.

Type 8: Checks 3 and 9

The eighth algorithm deals specifically with the case of checking between a 3 and a 9. It works much like the Type 7 algorithm except that it checks only the fourth and fifth rows, since a 9 only has the left curve on the top half. After determining the average of the first on pixels in those rows, it returns a 9 if it is below 5.5 and a 3 otherwise.

Testing showed that this case was never checked and no specific changes were made to the code.

Type 9: Checks 4 and 9

This algorithm deals specifically with the case of checking between a 4 and a 9. It does so by exploiting the fact that the top part of a 9 is closer to edge than it would be for a 4. The 4 has a peak in the middle really close to the top edge but then it slopes downwards on the left side. In this algorithm, four middle columns (7th, 8th, 9th, and 10th) are examined to find the location of the first on pixel in each column. The average of the four locations is computed, and if it is greater than 2.5, then a 4 is returned since it is not very close to the top. Otherwise, a 9 is returned since those pixels are pretty close to the top.

In this case too, the code was not changed specifically and testing showed that the case was not checked at all at high thresholds.

Type 10: Checks 5 and 6

The tenth algorithm deals specifically with the case of checking between a 5 and a 6. It works on the premise that 5's are missing the left curve on the bottom half that 6's

have. So, this algorithm calculates the average column of the first on pixels from the ninth to thirteenth rows, where the missing curve would most likely be. If the average is less than or equal to 4, then it returns a 6. If the average is greater than or equal to 5, then it returns a 5. If it is between 4 and 6, it checks the location of the latest first on pixel in those rows. If that is greater than 6, then a 5 is returned since it is far away from the left edge. Otherwise, the function returns a 6.

Testing showed that the post processor was correct for this case 9 out of 10 times, and the original code was not altered except for the general changes.

Type 11: Checks 5 and 8

The Type 11 algorithm deals specifically with the case of checking between a 5 and an 8. It assumes that from the fourth row to the sixth row, the last on pixel will be close to the right border for an 8 and close to the left border for a 5. If the average in these rows is less than 7, then it returns a 5. If it is greater than 9, then it an 8 is returned. If it is between 7 and 10, then the next assumption is checked. The first on pixel from the ninth row to the twelfth row would be closer to the right border for a 5 and for an 8, the left border. The average of the first on pixels in these rows is calculated and if it is less than 7, then an 8 is returned. Otherwise, a 5 is returned.

Testing of this check showed that it was correct 10 out of 11 times. No specific changes were made to this code.

Type 12: Checks 5 and 9

This algorithm works on checking between a 5 and a 9. First, it obtains the first and last on pixels from the third row to the seventh row. Then, it computes two averages, one for the last on pixels (last_avg), and another for the difference between the last and first on pixels (diff_avg). Then, if last_avg is less than 8, then it returns a 5 because 5's have a vertical line on the left side. Otherwise, if last_avg is greater than or equal to 11, then it returns a 9 since 9's have a right curve.

Further checking is done in the cases that last_avg is less than 11 and greater than or equal to 8. This time, diff_avg is checked. Since a 5 will only have a vertical line, the width should be relatively small, meaning diff_avg is relatively small. For a 9, however, it would be bigger since it would have a circle loop on its top half. So, if diff_avg is less than or equal to 3, a 5 is returned and if it is greater than or equal to 6, a 9 is returned. In the case in between, even more checking must be done.

Finally, the last check done on the image to determine between a 5 and a 9. It first calculates the number of last on pixels that occurred in the left half (before column 9, called last_half) and also the number of last on pixels that occurred in the leftmost quarter (before column 5, called last_quarter) in the five rows checked earlier. If last_half is greater than or equal to three, then it is a 5 because a 9 usually has its last on pixels on the right side. If last_quarter is greater than or equal to two, then it is a 5 for the same reason as above. Otherwise, if last_quarter is less than two, then a 9 is returned.

Testing showed that this case was rarely checked at high thresholds. Subsequently, only the general changes were made to this function.

Type 13: Checks 7 and 9

The thirteenth and final algorithm deals specifically with the case of checking between a 7 and a 9. It does so by going three diagonal lines. First, from the position (10, 1)[9] to (1, 10) and subsequently from the (11, 1) to (1, 11) and from (12, 1) to (1, 12). Along each line the number of pixels until the first on pixel is added to the total. Since a 7 will not have anything until the end of the line because of its shape, the location of the on pixels will be high. For a 9, however, it will be different, as the lower end of the loop will be hit relatively early. So, if the total is greater than 16, which would be relatively late, a 7 is returned. Otherwise, a 9 is returned.

```
0 0 0 0 0 0 0 0 0 / / / 0 0 0 0
0 0 0 0 0 0 0 0 / / / 0 0 0 0 0
0 0 0 0 0 0 0 / / / 0 0 0 0 0 0
0 0 0 0 0 0 / / / 0 0 0 0 0 0 0
0 0 0 0 0 / / / 0 0 0 0 0 0 0 0
0 0 0 0 / / / 0 0 0 0 0 0 0 0 0
0 0 0 / / / 0 0 0 0 0 0 0 0 0 0
0 0 / / / 0 0 0 0 0 0 0 0 0 0 0
0 / / / 0 0 0 0 0 0 0 0 0 0 0 0
/ / / 0 0 0 0 0 0 0 0 0 0 0 0 0
/ / 0 0 0 0 0 0 0 0 0 0 0 0 0 0
/ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Three Diagonal Lines used for Type 13

Post Processor Performance

The performance of the post processing step was calculated in the following way. First, the performance of the system without post processing was obtained for various

[9] (10, 1) = 10th row , 1st column

thresholds for confidence levels. Then, the performance was calculated for the system that will activate the post processing for various thresholds and difference thresholds. For example, if the threshold is 0.7 and the difference threshold is .2, then the post processor will be activated if the highest confidence returned by the neural net is greater than 0.7 and the second highest confidence is within 0.2 of the highest. Also, when the post processor's output differs from the neural net's, the number of times the post processor corrected an incorrect neural network output as well as the number of times the post processor was wrong and the neural network output was right were calculated.

Threshold	Difference	Correct	Incorrect	Rejected	Post Process?	Added Value
0.5	0.25	4700	593	526	Yes	0.815%
0.5	0.25	4662	631	526	No	
0.6	0.2	4571	509	739	Yes	0.417%
0.6	0.2	4552	528	739	No	
0.6	0.3	4579	501	739	Yes	0.593%
0.6	0.3	4552	528	739	No	
0.7	0.2	4377	393	1049	Yes	0.321%
0.7	0.2	4363	407	1049	No	
0.7	0.3	4381	389	1049	Yes	0.413%
0.7	0.3	4363	407	1049	No	
0.75	0.25	4255	327	1237	Yes	0.354%
0.75	0.25	4240	342	1237	No	
0.8	0.3	4095	286	1438	Yes	0.269%
0.8	0.3	4084	297	1438	No	
0.9	0.1	3464	175	2180	Yes	0.000%
0.9	0.1	3464	175	2180	No	

Comparison of results with or without post processing

Results from the test showed that the addition of the post processing steps increased the overall accuracy for all the different threshold and difference levels except for the 0.9 and 0.1 case. The trend from the results is that the added value (increase in accuracy rate) of post processing decreases as the threshold level increases and it increases as the difference level increases. The older version of the code did not compile,

so no comparison between the new and old versions can be made. However, the results do show that the overall accuracy was increased with the new version. Therefore, it should be implemented in the current Winbank system.

Brazilian Checks

Recently, a representative from Integris, a Brazilian software/hardware integration company brought a CD-ROM containing 5000 Brazilian check images. An attempt was made to train the neural net on 2000 of the images and then check its accuracy for the remaining 3000 checks. However, several problems were encountered in the process. First, the checks did not have a courtesy amount box. Instead, the courtesy amount area was generally on the top right corner of the check. Also, the courtesy amount often included delimiters such as "Re$" before the actual and also a "#" before and after the actual amount. Finally, all of the checks were stamped and sometimes the stamp would cross into the courtesy amount area, making it impossible to read.

Further Improvements

Further analysis can be done on the results of the post processing steps to improve the system's accuracy. The cases where the post processor was incorrect while the neural net was correct can be examined to see why that happened and how to improve it. If certain algorithms are usually more wrong than right, it can be taken out to increase the added value of the post processing step.

Also, there are still some mysterious bugs remaining in the application. Most notably, on Windows NT, the program crashes right after the user presses the "Recognize" button for an input test situation. Also, on Windows 95, the program can not perform the load image function correctly. Before the process is finished, the program tries to access an invalid address, and causes it to crash. Several days have been devoted to debugging this problem but no cause for the bug has been found yet.

Summary

This report has detailed the design and development of Winbank, a software prototype capable of recognizing the written amount in the courtesy amount block of a check. The application is broken down into three modules: Scanning, Recognition, and User Interface. The User Interface module handles the user's input and calls the necessary routines in the two other modules in order to determine the correct output for the user. Training for Winbank's neural net needs to be redone if any of the preprocessing steps are modified.

The preprocessing steps include segmentation and standardization. Segmentation divides the image into separate characters, and determines whether each segment is a digit or a punctuation mark. Each digit is passed to the standardization step is normalized, slant corrected, thinned and thickened. The result is given to the neural network and a confidence level is generated for each potential digit. In the post processing step, a digit is reexamined if the confidence level is high for two potential digits. The code for the post processing step did not compile at first due to the numerous

errors in it. Several general changes were made to improve the post processor in terms of speed, accuracy, and memory usage. Specific changes were also made to some algorithms to correct errors in the original code.

Results showed that the post processing step's added value increases as the threshold level increases and as the difference level decreases. The overall accuracy of the system increases for all the levels tested and should be added to Winbank's system.

Recently, Integris provided 5000 images of Brazilian checks. Attempts to train and test the application on the images encountered several problems such as lack of courtesy amount block area, abnormal delimiters and noise due to stamps made on the checks.

Further improvements to the prototype include further analysis on the effects of the post processor. Some algorithms may fail more than it succeeds and either can be taken out or examined and refined. Also, a few mysterious bugs still remain and should be fixed.

If the necessary modifications can be made to the system, the offline numerical recognition system mentioned here might someday be improved to the point where it is adequate for broad commercial use.

References

Hussein, K., Gupta, A., Agrawal, A., and Wang, P. S. P., "A Knowledge Based Segmentation Algorithm for Enhanced Recognition of Handwritten Courtesy Amounts", Discussion Paper, International Financial Services Research Center (IFSRC) No. 289-94, Sloan School of Management, MIT, 1994

Nagendraprasad, M. V., Gutpa, A., and Feliberti, V., “A New Algorithm for Correcting Slant in Handwritten Numerals”, Discussion Paper, IFSRC No. 215-92, Sloan School of Management, MIT, 1992

Nagendraprasad, M. V., Wang, P. S. P., and Gupta, A., “Algorithms for Thinning and Rethickening Binary Digital Patterns”, *Digital Signal Processing 3*, pp. 97-102, 1993

Sparks, P. L., Nagendraprasad, M. V., and Gupta, A., “An Algorithm for Segmenting Handwritten Numeral String”, Discussion Paper, IFSRC No. 214-92, Sloan School of Management, MIT, 1992

Mui, L, Agrawal, A., Gupta, A., and Wang, P. S. P., “An Adaptive Modular Neural Network with Application to Unconstrained Character Recognition”, Discussion Paper, IFSRC No. 261-93, Sloan School of Management, MIT, 1993