

6.199 - ADVANCED UNDERGRADUATE PROJECT REPORT

IMPROVING THE AUTOMATIC PROCESSING
OF HANDWRITTEN BANK CHECKS

WINNIE CHAN <WCHAN@MIT.EDU>
AUP ADVISOR: AMAR GUPTA
MIT EECS '02
CAMBRIDGE, MA

6.199 REPORT

IMPROVING THE PROCESSING OF HANDWRITTEN BANK CHECKS

I. INTRODUCTION

Paper checks are the most commonly used method of payment, other than cash, in our society today [3]. However unlike the use of credit and debit cards, current technology does not allow paper checks to be automatically processed because the amount to be paid is usually handwritten. In answer to this issue, PROFIT (Productivity from Information Technology) of MIT Sloan School of Management, headed by Professor Amar Gupta, has developed a system to automate the reading of checks. This automation would eliminate both the time needed for an individual to manually enter the check transaction, as well as the time for a second individual to verify that data entry. The successful implementation of such a system would have a great impact on the banking industry, as it would make check processing more efficient and also reduce the number of human errors.

If bank checks were designed to be read by document understanding systems (DUS) with identification codes and position marks included, then the recognition of isolated digits would not be very difficult. This is because document forms intended to be read by DUS often restrict the writer to print every character into designated boxes which are imprinted with a special ink invisible to the scanner. Since the underlying boxes encourage writers to write neatly and carefully, digits are usually well isolated. Even if the characters are touching each other, those characters can be easily separated by using the underlying boxes as a guideline. However, in the case of bank checks today, the numbers are written in a totally unconstrained way, so that it is very common to find connected digits and other problems associated with freehand writing. As illustrated in figure 1, the most problematic cases found in the courtesy amount of a check are:

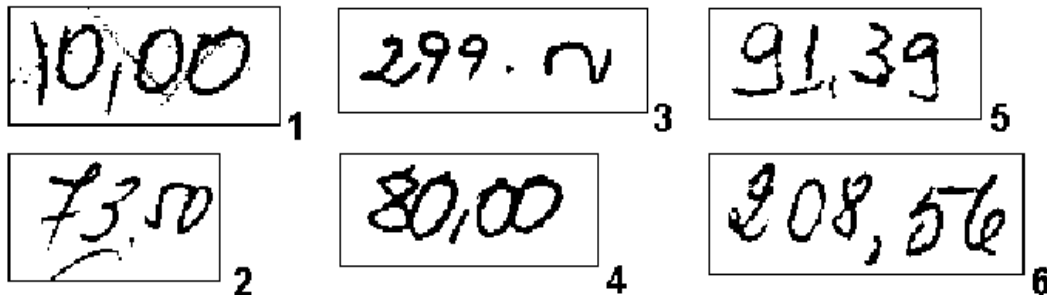


Figure 1: Most prominent problems in extracting digits from the image of the courtesy amount string

1. The scanning and conversion into black and white introduce noise into the image. A check having a busy background image often makes it difficult to distinguish background from text. This kind of noise can be easily eliminated using standard functions to remove noise or by extracting the connected components smaller than a predefined size.
2. Text from the date field or from the legal amount sometimes invades the courtesy amount area. As long as these pieces are not touching any of the digits, it is easy to eliminate these strokes by studying the connectivity with the points surrounding the courtesy amount area.

3. Digits can overlap each other, such as the '2' and the '9'. In this case, it is not possible to extract isolated digits from the string number just by searching for columns of white pixels.
4. Digits can touch each other, such as the '8' and the '0'. A splitting algorithm must be applied to isolate the digits.
5. Digits are broken into fragments, such as the vertical stroke and the horizontal stroke in the number '1'. Sometimes a digit can be fragmented as a result of a bad conversion of the scanned image from gray scale to black and white so that small pieces of the stroke which is lighter or thinner may be lost.
6. The fragment of one digit is touching a different digit, such as the top horizontal stroke of the '5' not being connected to '5' itself but rather connected to its neighbor digit '6'. This is a common case, as check writers often scribble quickly and pay little attention to make sure the digits are properly aligned. Segmentation based on connected components will pull apart the fragment from the corresponding digit so the system may fail in the recognition of the broken digit as well as in the recognition of the disturbed digit and all the possibilities of a split.

II. BACKGROUND ON CURRENT SYSTEM

The current system uses the scanned image of a bank check and obtains the amount in the courtesy box to be paid. After a check is scanned, it is represented as a matrix where each entry of the matrix is the value of the pixel, and each pixel represents the brightness of the corresponding point in that page. The system then follows four steps:

1. Detection of the courtesy box according to heuristic rules

In the case of Brazilian checks, the courtesy amount is always located at the upper right corner and is delimited by vertical and/or horizontal lines. Several combinations of lines may be used to delimit the courtesy amount area. The location of the courtesy amount is accomplished by searching for horizontal and vertical lines in the upper right part of the image. Based on the relative position of these lines, the area of interest is then defined as a rectangle embedded within the lines.

2. Binarization of the image: converting gray scale pixels (256 levels of brightness) into black and white pixels

The pixels of the image will be assigned values of ones and zeroes depending on whether the color was closer to black or white. The current system computes a dynamic threshold based on the analysis of the histogram of the image. It then applies a median filter to reduce the noise in the image.

3. Segmentation of the string into individual characters.

Connected components are extracted from the courtesy amount string and stored into blocks. Each block's segment is then classified as 'DIGIT', 'MULTIPLE', 'NONSENSE', 'PUNCTUATION'...etc. If digits are connected, then the program attempts to separate them using several segmentation algorithms, including a Min-Max contour analysis (based on the one described by Blumenstain, et al [4]), the Hybrid Drop Fall (HDF) algorithm [5][6] and the Extended Drop Fall algorithm [2] from each of the four corners. As Drop Fall methods can be applied from each of the four corners, there are 9 different methods to generate different ways (henceforth known as paths) of separating touching characters, simple heuristics are used to rank these paths. The best path is then used to separate the touching characters, and the result is passed into the recognition module. The segmentation process is performed in a feedback loop that takes the results from the recognition module as inputs and is iterated until the recognition has exceeded a confidence level.

4. Recognition of characters based on a Directional Distance Distribution neural network.

To reduce noise, each character first undergoes thickness normalization, size normalization, and slant correction. Afterwards the classifier attempts to determine the identity of the digit. If a decision is reached with high confidence, then a syntactic verification is carried out for correctness. If a decision cannot be made with a high degree of confidence, then the segment is rejected and is looped back to the segmentation step, where a new segmentation of the amount string is performed and the whole process is repeated.

Of the 6 problematic cases presented in section I, case 1 and 2 can be easily dealt with by replacing the old filter function in the step 2 (binarization) because a function that will filter out the noise in case 1 and 2 can be independently implemented. The problems of cases 3, 4, 5, 6, however, involve the feedback loop of repeating segmentation and recognition. In particular, when dealing with segments classified as connected digits (as in case 3 and 4), the current system will rank all the possible character separation methods according to some simple heuristics, tries these separation and stops at the first separation that leads to a recognition exceeding a certain minimum confidence threshold. Since the heuristics used for ranking is not very accurate, often the separation methods that will lead to successful recognition are not examined. Unfortunately, the current system abandons the other possible separations and the feedback loop has no flexibility to somehow store the results of the other separations. Upon careful examination of the existing code, visiting scholar Rafael Palacios decided that it was necessary to redesign the segmentation loop. My advanced undergraduate project is to assist him in the redesign that will allow the system to consider all separations exhaustively, thereby achieving a higher recognition accuracy. The new design should also take the need for merging (as in case 5 and 6) into consideration and include the future implementation of a merging function into the core structure of the loop.

III. DESIGNS OF THE NEW SEGMENTATION LOOP

Isolating the preprocessing (detection of courtesy amount box, binarization, and the extraction of connected components), the new segmentation loop takes a list of connected components as input. Then, for each connected component (henceforth known as a segment), testing for the level of

overlapping is performed. If a segment is analyzed to be overlapping with its neighbor, then these two segments are merged into one segment. In the case of isolated fragments (as in the vertical stroke and the horizontal stroke in the digit '1' of case 5), the fragments are usually connected to the correct digit and the problem is solved. If the fragment is touching another digit (as in the top horizontal stroke of the digit '5' not touching '5' but touching '6') and the overlap is evident (the horizontal stroke overlaps with the digit '5' by a lot), then the two segments are merged into one.

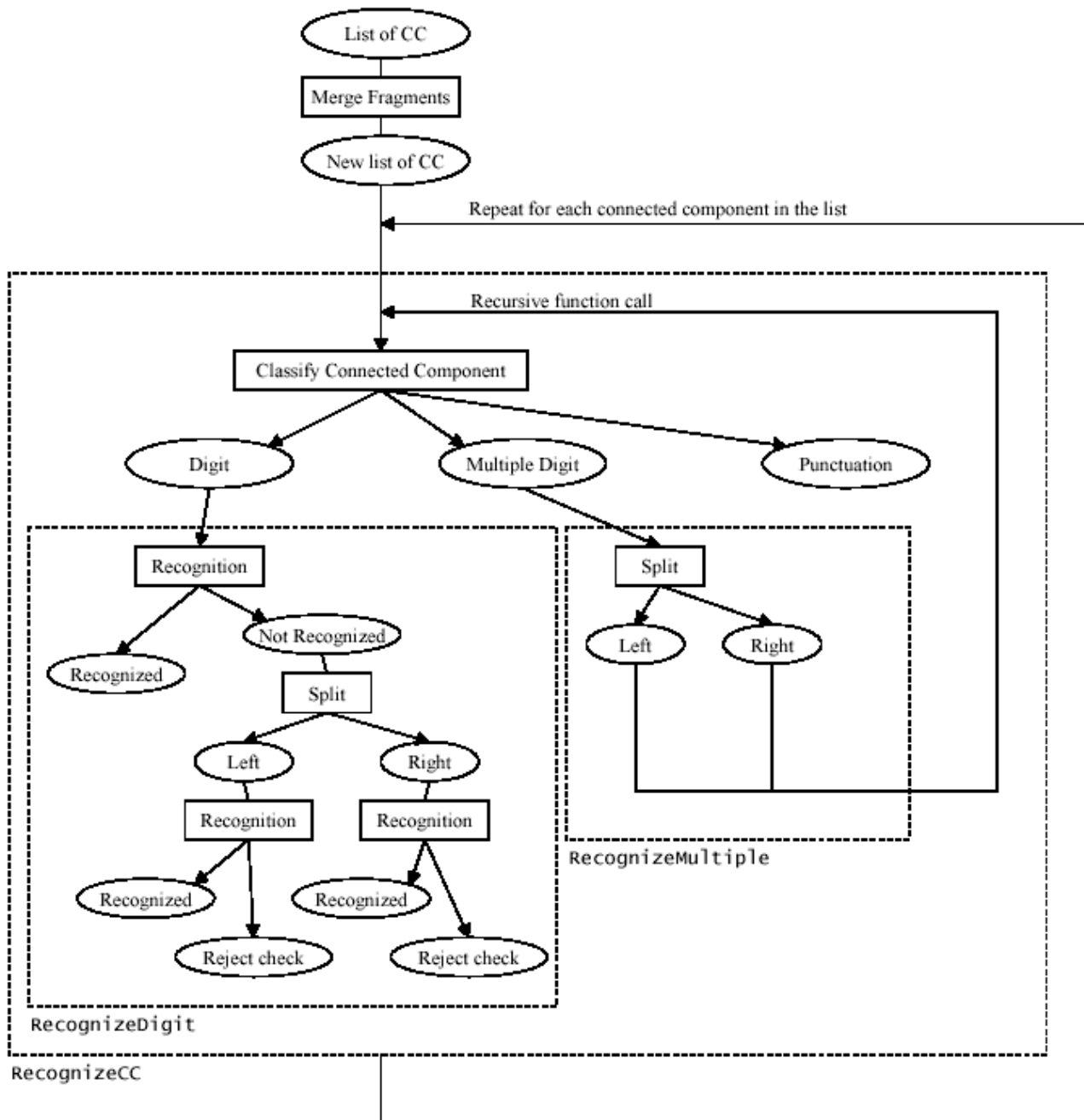


Figure 2: General scheme of the recursive segmentation approach

After the detection of overlapping and any necessary merging is done, the new list of connected components is iteratively passed into the main recursive function that will determine the value of the courtesy amount string. First, a segment is passed into the classifier. Using simple heuristics, a segment is classified as either a digit, a multiple of digits, or a punctuation. Depending on the classification, a different recognizer is called. In the case where separation of digits is required, all combinations of splits are systematically recognized and all the results are stored. At the end of the search, an evaluation module will decide which is the solution with the highest degree of confidence among the options.

The general scheme of this approach is pictured as a flow chart in figure 2. The main functions are shown in dotted lines, grouping the objects that they implement. As shown in the diagram, the core of the diagram is the recursive function RecognizeCC, named after Recognize Connected Component, which is iteratively called to recognize each connected component of the image.

When a segment is classified as a 'DIGIT', the system will make two attempts to recognize the digit. If the first attempt fails, then the digit-recognizer will try to split the segment and recognize the left half and the right half. If either the left half or the right half is not recognized, then the check is rejected. Once a segment has been classified as 'digit', at most one round of splittings can occur. In case there was a misclassification of a pair of touching digits classified as a single digit, this one round of splittings will allow for this pair to be classified. However, the splitting is not recursive and can only happen at most once to eliminate the infinite splitting of a segment. Rejecting a check means the machine is requesting human recognition; it is always better for the machine to ask for help than to accept the check and return a wrong recognition.

When a segment is classified as a 'MULTIPLE', the multiple-recognizer will use different splitting algorithms to generate different pairs of left segments and right segments. Then the multiple-recognizer will pass the results of the separated segments in a recursive call to RecognizeCC. As the process is repeated for every split algorithm implemented in the system, great flexibility is provided for the selection of any splitting algorithms.

When a segment is classified as a 'PUNCTUATION', it does not need to be passed into the neural network for recognition, so the location of the punctuation is saved in the solution string. This information is needed in the evaluation module, which is responsible for picking the best solution. In the evaluation of a solution, in addition to analyzing the individual confidences of each digit in the solution, the arrangement of digits and punctuations can be used as yet another heuristic. Take the example in figure 3.

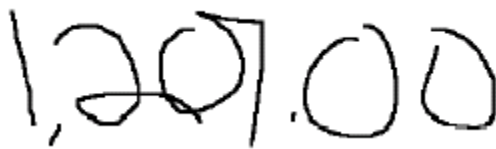


Figure 3: String containing three connecting digits.

It is made of 6 connected components: [1] [,] [207] [.] [0] [0]. Since the third component contains three touching digits, a different splitting algorithm is applied to separate the digits. Figure 4 shows one possible tree of solutions generated during the recognition using two different splitting algorithms.

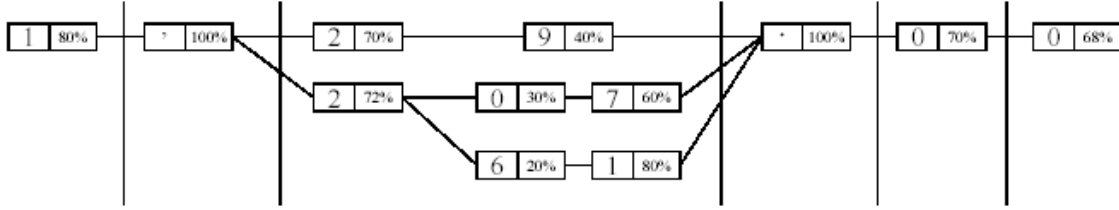


Figure 4: Set of solutions generated by the segmentation algorithm

Depending on the split algorithm used, [207] can be broken into two or three digits: First the third connected component is separated between number '2' and '07' by both split algorithms. Then the segment '07' obtained from the first split function is recognized as the number '9', while the segment '07' obtained from the second split function is recognized as a 'multiple segment' and split again. Although it is difficult to establish which separation is better just by watching the confidences in the segmentation process, the evaluation module will take contextual information into account. In the example shown in figure 4, [1,207,00] is picked as a better solution than [1,29,00] because there must be 3 digits between punctuation marks.

IV. IMPLEMENTATION DETAILS

THE STRUCTURE OF SOLUTION

The most challenging aspect of the implementation is to define a structure for representing possible solutions. In the base case of recognizing a single digit, a good representation should contain information about what the recognized value of the digit is, as well as the associated confidence. In the final form a complete solution, a good representation should contain information about the whole string of characters (for example, [1 , 2 0 7 , 0 0] and each character's associated confidence. [.8 1.0 .72 .3 .6 1 .7 .68]).

If splitting is ever performed, every split function produces a pair of digits slightly different than the pair of digits produced by the other algorithms, and sometimes the results from the recognition module may not concur, thereby creating alternative solutions. An image containing a group of three digits is first divided into one digit and another multiple digit, which is then divided in a deeper level of recursion. However, the process is different depending on the algorithm used to split characters, and there is no split algorithm that always performs the best in all the possible situations that are found in bank checks. Thus a good representation should not contain just a single solution, but rather a set of solutions obtained as a consequence of applying different splitting algorithms. For example, if there are 9 different splitting algorithms, then applying two rounds of splitting on a segment will generate 81 optional solutions!

Therefore, during the segmentation process, it is necessary for the solution structure to store the information about all 81 solutions while not losing the sequential ordering of each solution's individual digits. Furthermore, as shown in the previous example, different splittings can generate solutions of variable lengths. Clearly, a good representation of the solution structure should not only allow the number of solutions to grow, but also the length of a solution to vary.

After careful consideration of all the constraints, the basic structure of a solution is finalized to be composed of a matrix of recognized characters and a matrix of respective confidences. Each line of the matrix will represent an option, so every time a branch is produced, the database duplicates its size. Figure 5 shows the solution structure generated by the previous example of the string 1,207.00.

Characters								Confidences							
1	,	2	9	,	0	0		.80	1.0	.70	.40	1.0	.70	.68	
1	,	2	0	7	,	0	0	.80	1.0	.72	.30	.60	1.0	.70	.68
1	,	2	6	1	,	0	0	.80	1.0	.72	.20	.80	1.0	.70	.68

Figure 5: Structure of solution generated by the segmentation algorithm

MAINTAINING THE SOLUTION

There are two main functions used to maintain the database of solutions: **ExtendSolution** and **AddNewEntry**. The former function is used to add more characters to the current list; these characters are the result of the analysis of the following connected component in the list of segments. The partial solution to be added with **ExtendSolution** is not an alternative to the current solution; therefore it is appended to all the current options. In the previous example, this function was last used to add the zero with a confidence of 68% that resulted from the analysis of the final connected component. When the partial solution to be added has more than one entry (i.e. we are adding optional solutions) the size of the resulting solution will be the number of entries of the current solutions multiplied by the number of entries of the merged solution. In the previous example this situation occurred by adding the partial solution of the third connected component. The partial solution was the three row matrix [2 9; 2 0 7; 2 6 7], and the current solution at that point was the single row [1 ,], consequently resulting in a new solution with three rows: [1 , 2 9; 1 , 2 0 7; 1 , 2 6 7].

On the other hand, the function **AddNewEntry** is used to create a branch, for example to add the partial solutions generated with several split algorithms. Every call to **AddNewEntry** results in a new solution with a number of entries equal to the sum of the entries in the current solution and the added partial solution. In the previous example this function was used to merge the partial solution [0 7] resulting from the first split function with the solution [6 7] resulting from the second split function in the analysis of the second half of the third connected component.

OPTIMIZATION – EARLY REJECTION

If one of the segments cannot be recognized, then the partial solution must be rejected. If a segment undergoes splitting and the left half or the right half cannot be recognized, then the partial solution must be rejected. One of the biggest consequences of the exhaustive splitting approach is that it generates a quite a number of solutions. As previously pointed out, two rounds of splitting on a single segment can generate 81 solutions. Therefore, this algorithm can take a long time to run, and the implementation of early rejection is extremely important. For details of the implementation, please refer to the code included in Appendix A.

V. AN EXAMPLE

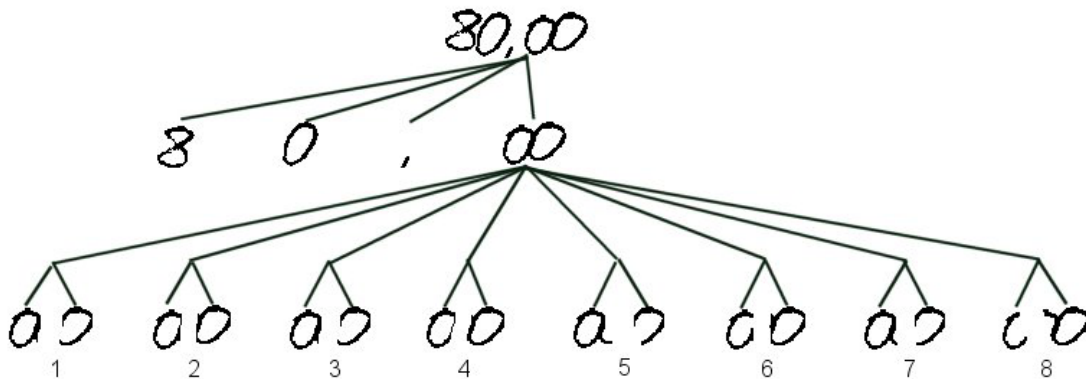
1) A check is scanned and the courtesy amount box is located.



2) The courtesy amount “80.00” is converted into a black and white image

3) The connected components [8] [0] [,] [00] are extracted and stored as segments.

4) RecognizeMain() calls RecognizeConnectedComponent() iteratively for each segment, which will classify [8] and [0] as ‘DIGIT’, [,] as ‘PUNCTUATION’, and [00] as ‘MULTIPLE’.



1 Extended Drop Fall, from top left
3 Extended Drop Fall, from top right
5 Hybrid Drop Fall, from top left
7 Hybrid Drop Fall, from top right

2 Extended Drop Fall, from bottom left
4 Extended Drop Fall, from bottom right
6 Hybrid Drop Fall, from bottom left
8 Hybrid Drop Fall, from bottom right

Figure 6 shows that a segment containing touching digits such as [00] will undergo splitting using various algorithms. The eight cases shown here are the actual outputs from the new recognition module that applied the Extended Drop Fall and Hybrid Drop Fall algorithms with various starting points.

5) Various splitting algorithms are applied to the segment [00] as shown in Figure 6. As a result, each of these will be recognized as different digits and be stored as different entries in the solution structure.

VI. ALTERNATIVES CONSIDERED

In the new design described in the paper, the problems related to broken fragments are solved before entering the recursive process by attaching each fragment to the corresponding digit. The alternative model involves having the overlap and merge function performed inside the recursive function. Basically, if a segment is classified as a fragment, then the system will try to merge this fragment with its neighbor.

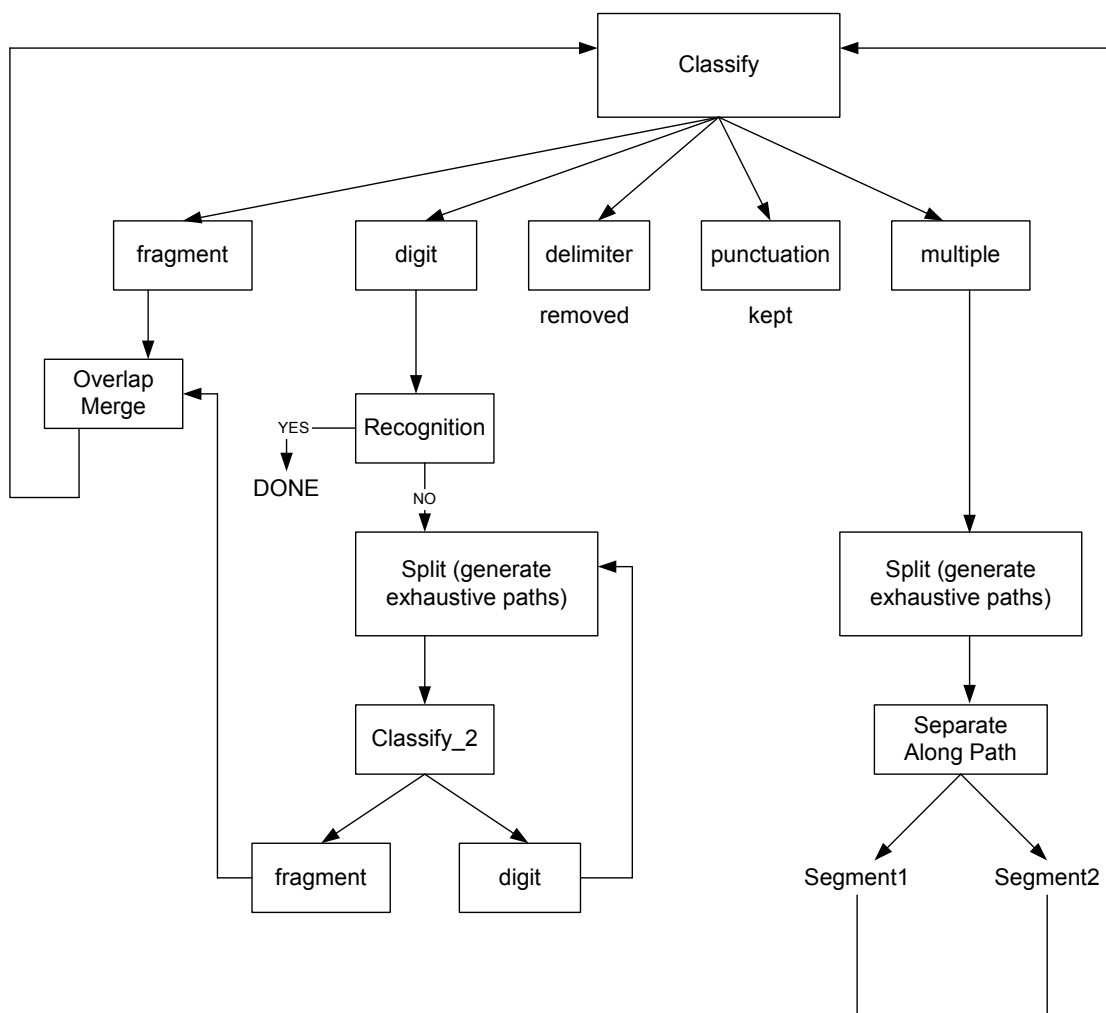


Figure 5: Structure of solution generated by the segmentation algorithm

One of the merits of the current design having merging take place prior to entering the recursive loop is that this design is conceptually easy (and therefore easy to code). If merging happened inside the loop, then there would be many special cases to deal with, such as what happens if, as a result of a splitting, the left segment was classified as a digit and recognized with high degree of confidence, while the right segment was classified as a fragment and required merging with the left segment that was already “successfully” recognized? Such situations would probably lead to the rejection of a check that could have been recognized if merging was performed earlier.

One of the disadvantage of performing merging before the recursive loop is that it tends to “over-merge”. Although over-merging has no negative effects on accuracy, it does lead to major inefficiency. As previously pointed out, just two rounds of splits to a segment leads to 81 solutions. The consequence of over-merging is that a significant amount of unnecessary splitting is performed. Each additional split generates many branches. In the case of multiple overlapping digits, the recursive splits required would be even more expensive. However, although over-merging can be inefficient, it leads to a higher recognition rate. Considering that accuracy is more important than speed, the system will attempt to join all the fragments to the corresponding digit before entering the recursive segmentation loop.

VI. FUTURE WORK

Although the exhaustive approach would produce more accurate results, it can be very slow. Yet another alternative approach is to implement a “fuzzy boundary”. If the character is recognized with outstanding confidence, then the segmentation loop terminates. If the recognition barely passes the confidence level, then the incumbent solution is kept and other segmentation methods are considered. If the recognition is below the confidence level, then the incumbent solution is rejected from consideration. Clearly, the execution speed and recognition accuracy are tradeoffs, and one of the main objectives of the ongoing research is to find the optimum tradeoff between execution speed and recognition accuracy in the bank check processing system.

As discussed above, choosing the right representation for the solution structure is quite challenging. There are several improvements that can be made to the solution structure. First, the solution structure can also be extended with an additional matrix to store information about the split algorithm used. This information may help to study which split algorithm performs better than others. Secondly, the solution structure of the current system is programmed with static memory to reduce programming errors. However, this limits the maximum size of the solution structure; using dynamic memory will provide greater flexibility. Thirdly, many of the solutions produced by different split algorithm are equivalent. To be more efficient, a cleanup function should be used after adding all the solutions of different split algorithms for the same segment. This way, less memory is required, and less time will be spent updating and maintaining the database of solutions.

VII. CONCLUSION

The most important lesson learned from this project is the importance of source code documentation. On the first day of work, this author was handed 40 source files with no documentation. Trying to make sense out of the code was quite challenging; fortunately the previous authors used long descriptive names for functions and variables, which helped mitigate the problem. Picking the “right place” for the new implementation and reusing as much code as possible was particularly challenging. It was not obvious where one should draw the line of using old code vs. writing new ones to replace them. On one hand, this author worried about the compatibility of the new code with the old code because no one in the winbank team had a complete understanding of the system (the original designers had graduated 3 years ago). On the other hand, a complete redesign of the system seemed more like a three year project than an AUP. Another major obstacle in the project was that there were over 5 copies of the code, and it was not immediately obvious which one was the good copy. Since there is no revision control system, the only way of backup was to make copies of the code. 40% of this author’s time was spent recovering lost code and putting fragments of code together.

However, overall this project was a great opportunity to experience some of these software design issues. It is this author's feeling that 6.033 (Computer Systems Engineering) taught her to appreciate the value of a simple design, and 6.170 (Software Engineering Laboratory) has helped her how to handle the challenges of the actual implementation. In particular, working with Rafael Palacios was both very educational and very rewarding. He had a lot of great ideas, but was always open for discussions as well. His help and constructive comments were the key to the success of the new segmentation design.

References

- [1] Palacios, Rafael. "*Segmentation Algorithm to read courtesy amount in bank checks*," working paper.
- [2] Punnoose, Jibu. "*An Improved Segmentation Module for Identification of Handwritten Numerals*," October 1999.
- [3] United States Check Study. Results of 1999 Research. Edited by The Green Sheet, Inc. <http://www.greensheet.com/CheckStudy/index99.html>, oct 25, 2001
- [4] Blumenstein, M. and Verma, S. (1998) "*A Neural Based Segmentation and Recognition Technique for Handwritten Words*". IEEE International Conference on Neural Networks Vol. 3, 1738-1742.
- [5] Khan, S. "*Character Segmentation Heuristics of Check Amount Verification*" Master Thesis, Massachusetts Institute of Technology. (1998)
- [6] Dey, S. "*Adding Feedback to Improve Segmentation and Recognition of Handwritten Numerals*" Master Thesis, Massachusetts Institute of Technology. (1999)

Appendix A

```
////////////////////////////////////  
//  
// SOLUTION.H  
//  
// This file contains the new main loop of recognition  
//  
#include "Classify.h"  
#include "WinBank.h"  
#include "CABValue.h"  
#include "Segment.h"  
#include "BWImage.h"  
#include "Recognize.h"  
#include "Path.h"  
#include "StructProcessor.h"  
#include "Arbiter.h"  
#include "DDDNet.h"  
#include "PosNet.h"  
  
#define SOL_MAX_ROW 10  
#define SOL_MAX_COL 7  
  
typedef struct {  
    int N; //number of entries  
    char digit[SOL_MAX_ROW][SOL_MAX_COL];  
    double conf[SOL_MAX_ROW][SOL_MAX_COL];  
    int len[SOL_MAX_ROW];  
    // char digit[1000][20]; //value from recognition of digit [i][j]  
    // double conf[1000][20]; //confidence of digit [i][j]  
    // int len[1000]; //length of entry[i]  
} t_solution;  
  
bool isRejected(t_solution *sol);  
void PrintSolution(bool accepted, t_solution *sol);  
void RecognizeOneSegment(Segment *seg);  
  
t_solution RecognizeMain(CABValue* CAB);  
t_solution RecognizeConnectedComponent(Segment* seg);  
t_solution RecognizeDigit(Segment* seg);  
t_solution RecognizePunctuation(Segment* seg);  
bool LastRecognize(Segment *left_seg, Segment *right_seg, t_solution *sol);  
t_solution RecognizeMultiple(Segment* seg);  
void CombineLeftRight(Segment* left_seg, Segment* right_seg, t_solution *cumulative_sol);  
Status SplitWithContour(Segment* original_seg, int vmid, Segment** left_seg, Segment**  
right_seg);  
Status SplitWithDropFall(Segment* original_seg, int vmid, int dir, Segment** left_seg,  
Segment** right_seg, bool special_case);  
  
void ExtendSolution(t_solution *curr_sol, t_solution *new_sol);  
void AddNewEntry(t_solution *curr_sol, t_solution *new_sol);
```

```

////////////////////////////////////
//
// SOLUTION.CPP
//
// This file contains the new main loop of recognition
//
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <time.h>
// #include <math.h>
#include <windows.h>

#include "assert.h"
#include "Solution.h"
#include "Classify.h"
#include "WinBank.h"
#include "CABValue.h"
#include "Segment.h"
#include "BWimage.h"
#include "Recognize.h"
#include "Path.h"
#include "StructProcessor.h"
#include "Arbiter.h"
#include "DDDNet.h"
#include "PosNet.h"

// function:   RecognizeMain
//
// purpose:   the main function for segmentation and recognition of all
//            the connected components in the block list
//
// return:    t_solution.  IMPORTANT NOTE: if the N
//
// author:    Winnie Chan    (Jan 24, 2002)
//
t_solution RecognizeMain(CABValue *CAB)
{
    t_solution curr_sol, new_sol;
    curr_sol.N = 0;
    new_sol.N=0;

    //overlap and merge

    //Get Matrix From Block List
    Block* curr_block = CAB->head;

    while (curr_block!=NULL) {

        Segment* curr_seg = curr_block->seg;
        char* filename = "test_segment.txt";
        curr_seg->OutputSegment(filename);
        new_sol = RecognizeConnectedComponent(curr_seg);
        if (!isRejected(&new_sol))
        {
            curr_sol.N=-1;
            return curr_sol;
        }
        else
        {
            ExtendSolution(&curr_sol, &new_sol);
            curr_block = curr_block->next;
        }
    }
    return curr_sol;
}

void PrintSolution(bool accepted, t_solution *sol)
{
    MessageBox(NULL, "Printing to test_solution.txt", "Printing to file", 0);
    int i, j;
}

```

```

ofstream file("test_solution.txt"); // Open the file
if (!file.is_open()) return;

if (accepted)
    file << "Accepted" << "\r\n";
else file << "Rejected" << "\r\n";

file << sol->N << "\r\n";

//print the digits
file << "Digits" << "\r\n";
for (i=0; i<sol->N; i++) {
    for (j=0; j<sol->len[i]; j++) {
        file << sol->digit[i][j] << "\t";
    }
    file << "\r\n";
}

//print the confidences
file << "Confidences" << "\r\n";
for (i=0; i<sol->N; i++) {
    for (j=0; j<sol->len[i]; j++) {
        file << sol->conf[i][j] << "\t";
    }
    file << "\r\n";
}
file.close();
}

void RecognizeOneSegment(Segment *seg)
{
    seg->digit = 8;
    seg->ArbConf[seg->digit]=.28;
}

// function:  isRejected
//
// purpose:   determines if a solution is rejected
//
// return:    returns true is check should be rejected
//
// author:    Winnie Chan    (Jan 29, 2002)
//
bool isRejected(t_solution *sol)
{
    if (sol->N<0)
        return true;
    else return false;
}

// function:  RecognizeConenctedComponent
//
// purpose:   Recursive function to recognize connected components
//
// author:    Winnie Chan    (Jan 25, 2002)
//
t_solution RecognizeConnectedComponent(Segment* seg)
{
    assert(seg!=NULL);
    t_solution new_sol;
    new_sol.N=0;

    int class_type = Classify(seg->pixel, seg->height, seg->width);

    switch(class_type) {
    case DIGIT:
        MessageBox(NULL, "classified as digit!", "winnie3", 0);
        new_sol = RecognizeDigit(seg);
        break;
}
}

```

```

    case MULTIPLE:
        MessageBox(NULL, "classified as multiple", "winnie4", 0);
        new_sol = RecognizeMultiple(seg);
        break;
    case PUNCTUATION:
        MessageBox(NULL, "classified as punctuation", "winnie5", 0);
        new_sol = RecognizePunctuation(seg);
        break;
    default:
        MessageBox(NULL, "Classify() failed to classify!", "Error", MB_OK | MB_ICONERROR);
    }
    return new_sol;
}

// function:   LastRecognize
//
// purpose:    recognizes the left segment and the right segment when the original
//             segment was classified as digit. 'Last' because it's the last attempt
//             to recognize before giving up and rejecting the check.
//
// return:     TRUE if recognition is accepted; FALSE if recognition is rejected.
//
// author:     Winnie Chan    (Jan 29, 2002)

bool LastRecognize(Segment *left_seg, Segment *right_seg, t_solution *sol)
{
    bool accepted=false;
    RecognizeOneSegment(left_seg);
    // RecognizeSeg(nn1, nn2, nn3, nn4, sp, arb, left_seg);
    if (left_seg->digit != NOT_RECOGNIZED)
    {
        RecognizeOneSegment(right_seg);
        //RecognizeSeg(nn1, nn2, nn3, nn4, sp, arb, right_seg);
        if (right_seg->digit != NOT_RECOGNIZED) {
            accepted=true;
            int i=sol->N;
            sol->digit[i][0] = '0' + left_seg->digit;
            sol->digit[i][1] = '0' + right_seg->digit;
            sol->conf[i][0] = left_seg->ArbConf[left_seg->digit];
            sol->conf[i][1] = right_seg->ArbConf[right_seg->digit];
            sol->len[i] = 2;
            sol->N++;
        }
    }
    return accepted;
}

// function:   RecognizePunctuation
//
// purpose:    Adds punctuation to database of solution
//
// author:     Winnie Chan    (Feb 1, 2002)
//
t_solution RecognizePunctuation(Segment* seg)
{
    t_solution new_sol;
    new_sol.N=1;
    new_sol.conf[0][0]=1.0;
    new_sol.digit[0][0]='.';
    new_sol.len[0]=1;
    return new_sol;
}

// function:   RecognizeDigit
//
// purpose:    Attempts to recognize digit.
//             If fails, try to split.
//             If second attempt to recognize fails, REJECT.
//
// author:     Winnie Chan    (Jan 25, 2002)
t_solution RecognizeDigit(Segment* seg)

```

```

{
    t_solution new_sol;
    new_sol.N=0;

    //call NN
    RecognizeOneSegment(seg);

    // RecognizeSeg(nn1, nn2, nn3, nn4, sp, arb, seg);
    if (seg->digit == NOT_RECOGNIZED) {
        //MessageBox(NULL, "RecognizeDIGIT: first attempt failed", "winnie6", 0);
        Segment* left_seg=NULL;
        Segment* right_seg=NULL;

        //REJECT only if NONE of the NINE is successful.
        //If at least one split leads to accepted results, do not reject.
        bool atLeastOneAccepted=false;
        int vmid = seg->FindVerticalMidline();

        if (SplitWithContour(seg, vmid, &left_seg, &right_seg) == SEG_OK)
            if (LastRecognize(left_seg, right_seg, &new_sol))
                atLeastOneAccepted=true;

        for (int d=0; d<4; d++) { //direction
            //Extended drop fall
            if (SplitWithDropFall(seg, vmid, d, &left_seg, &right_seg, true) == SEG_OK)
            {
                if (LastRecognize(left_seg, right_seg, &new_sol))
                    atLeastOneAccepted=true;
            }

            //Regular drop fall
            if (SplitWithDropFall(seg, vmid, d, &left_seg, &right_seg, false) == SEG_OK)
            {
                if (LastRecognize(left_seg, right_seg, &new_sol))
                    atLeastOneAccepted=true;
            }
        }

        if (!atLeastOneAccepted)
        {
            //MessageBox(NULL, "RecognizeDIGIT: REJECT CHECK!", "winnie8", 0);
            //REJECT!!!!
            new_sol.digit[0][0] = '?';
            new_sol.conf[0][0] = -1;
            new_sol.len[0]=0;
            new_sol.N=-1;
        }
    } else { //success!
        char d = '0' + seg->digit;
        char s[] = "RecognizeDIGIT: first attempt succeeded: d";
        s[40] = d;
        //MessageBox(NULL, s, "winnie7", 0);
        new_sol.digit[0][0] = '0' + seg->digit;
        new_sol.conf[0][0] = seg->ArbConf[seg->digit];
        new_sol.len[0]=1;
        new_sol.N=1;
    }
    return new_sol;
}

// function:   RecognizeMultiple
//
// purpose:    Applies various splitting algorithms and returns the cumulative
//             solution
//
// author:     Winnie Chan    (Jan 25, 2002)
//
t_solution RecognizeMultiple(Segment* seg)

```

```

{
    t_solution cumulative_sol;
    cumulative_sol.N=0;

    Segment *left_seg=NULL;
    Segment *right_seg=NULL;
    int vmid = seg->FindVerticalMidline();

    if (SplitWithContour(seg, vmid, &left_seg, &right_seg) == SEG_OK)
        CombineLeftRight(left_seg, right_seg, &cumulative_sol);

    for (int d=0; d<4; d++) { //direction
        //Extended drop fall
        if (SplitWithDropFall(seg, vmid, d, &left_seg, &right_seg, true) == SEG_OK)
            CombineLeftRight(left_seg, right_seg, &cumulative_sol);

        //Regular drop fall
        if (SplitWithDropFall(seg, vmid, d, &left_seg, &right_seg, false) == SEG_OK)
            CombineLeftRight(left_seg, right_seg, &cumulative_sol);
    }

    return cumulative_sol;
}

// function:   CombineLeftRight
//
// purpose:    Given the left_seg and the right_seg, this function attempts
//             to recognize them individually and adds the solution to table.
//
// author:     Winnie Chan    (Jan 25, 2002)
//
void CombineLeftRight(Segment* left_seg, Segment* right_seg, t_solution *cumulative_sol)
{
    t_solution left_sol = RecognizeConnectedComponent(left_seg);
    if (!isRejected(&left_sol))
    {
        t_solution right_sol = RecognizeConnectedComponent(right_seg);
        if (!isRejected(&right_sol))
        {
            ExtendSolution(&left_sol, &right_sol); //appends right_sol to the end of
left_sol
            AddNewEntry(cumulative_sol, &left_sol); //adds left_sol to the whole table.
        }
    }
}

// function:   SplitWithContour
//
// purpose:    Given the original_seg and the vertical midline distance, computes
//             the path and separates it according to the contour algorithm.
//             The resulting segments are stored as left_seg and right_seg.
//
// author:     Winnie Chan    (Jan 25, 2002)
//
Status SplitWithContour(Segment* original_seg, int vmid, Segment** left_seg, Segment**
right_seg)
{
    Status success = SEG_ERROR;
    Block *tmp_block = new Block(original_seg);
    Path *tmp_path = tmp_block->GetContourPath(vmid);
    if (tmp_path==NULL)
        return success;

    success = tmp_block->SeparateRegionAlongPath(tmp_path);
    original_seg->OutputSegment("test_original_seg.txt");
    if (success == SEG_OK) {
        *left_seg = tmp_block->bList1->seg;
        (*left_seg)->OutputSegment("split_left_0.txt");
        *right_seg = tmp_block->bList2->seg;
        (*right_seg)->OutputSegment("split_right_0.txt");
    }
}

```

```

    } else MessageBox(NULL, "Split with contour path failed.", "Error", MB_OK |
MB_ICONERROR);

    //clear memory for tmp_block and tmp_path
    //delete tmp_block;
    //delete tmp_path;
    return success;
}

// function: SplitWithDropFall
//
// purpose: Given the original_seg, the vertical midline distance and the
// direction(SEG_FALL_TOPLEFT, SEG_FALL_TOPRIGHT, SEG_FALL_BOTTOMLEFT,
// SEG_FALL_BOTTOMRIGHT), computes the path and separates it according
// to the dropfall algorithm..
// The resulting segments are stored as left_seg and right_seg.
//
// author: Winnie Chan (Jan 25, 2002)

Status SplitWithDropFall(Segment* original_seg, int vmid, int dir, Segment** left_seg,
Segment** right_seg, bool special_case)
{
    Status success = SEG_ERROR;
    int offset = dir + (special_case ? 1 : 5);
    Block *tmp_block = new Block(original_seg);
    Path *tmp_path = tmp_block->GetDropFallPath(dir, vmid, special_case);
    if (tmp_path==NULL)
        return success;

    success = tmp_block->SeparateRegionAlongPath(tmp_path);
    if (success== SEG_OK) {
        *left_seg = tmp_block->bList1->seg;
        char sl[]="split_left_x.txt";
        sl[11]='0'+offset;
        (*left_seg)->OutputSegment(sl);
        char sr[]="split_right_x.txt";
        sr[12]='0'+offset;
        *right_seg = tmp_block->bList2->seg;
        (*right_seg)->OutputSegment(sr);
    } else MessageBox(NULL, "Split with drop fall failed", "Error", MB_OK | MB_ICONERROR);

    //clear memory for tmp_block and tmp_path
    //delete tmp_block;
    //delete tmp_path;
    return success;
}

// function: ExtendSolution
//
// purpose: Given a curr_sol (of size A x C) and a new_sol (of size B x D),
// _multiplies_ and modifies curr_sol (now size = A*B x C+D)
//
// author: Winnie Chan (Jan 24, 2002)
//
void ExtendSolution(t_solution *curr_sol, t_solution *new_sol)
{
    if (new_sol->N ==0)
        return;

    if (curr_sol->N==0) { //copy new_sol into curr_sol
        for (int i=0; i<new_sol->N; i++) {
            for (int j=0; j<new_sol->len[i]; j++) {
                curr_sol->digit[i][j] = new_sol->digit[i][j];
                curr_sol->conf[i][j] = new_sol->conf[i][j];
            }
            curr_sol->len[i] = new_sol->len[i];
        }
        curr_sol->N = new_sol->N;
    }
}

```

```

    if (curr_sol->N * new_sol->N >= SOL_MAX_ROW)
        MessageBox(NULL, "ExtendSolution: out of memory error!", "Error", MB_OK |
MB_ICONERROR);

    //make copy of curr_sol's len array so that we don't clobber it
    int left_len[SOL_MAX_ROW];
    for (int k=0; k<curr_sol->N; k++)
        left_len[k]=curr_sol->len[k];

    for (int group=0; group<new_sol->N; group++) { // group = row index of new_sol
        for (int i=0; i<curr_sol->N; i++) { // i = row index into curr_sol
            int row = group*curr_sol->N + i; // row = row index into cumulative_sol
            // curr_sol->len[row] = curr_sol->len[i] + new_sol->len[group];
            //for (int j=0; j<curr_sol->len[i]+new_sol->len[group]; j++) { //changed
            for (int j=0; j<left_len[i]+new_sol->len[group]; j++) {
                //if (j<curr_sol->len[row]) {
                if (j<left_len[i]) { //copy from left
                    curr_sol->digit[row][j] = curr_sol->digit[i][j];
                    curr_sol->conf[row][j] = curr_sol->conf[i][j];
                } else { //copy from new_sol
                    curr_sol->digit[row][j] = new_sol->digit[group][j-left_len[i]];
                    curr_sol->conf[row][j] = new_sol->conf[group][j-left_len[i]];
                }
            }
            curr_sol->len[row] = left_len[i] + new_sol->len[group]; //new
            // curr_sol->N++;
        }
    }
    curr_sol->N = curr_sol->N * new_sol->N; //new
}

// function: AddNewEntry
//
// purpose: Given curr_sol (of size A x C) and new_sol (of size B x D)
//           _adds_ and modifies curr_sol (now size = A+B x C_or_D)
//
// author: Winnie Chan (Jan 24, 2002)
//
void AddNewEntry(t_solution *curr_sol, t_solution *new_sol)
{
    if(curr_sol->N + new_sol->N >=SOL_MAX_ROW)
        MessageBox(NULL, "AddNewEntry: out of memory error.", "Error", MB_OK |
MB_ICONERROR);

    for (int i=0; i<new_sol->N; i++) {
        int row = curr_sol->N + i; //row index into the next entry
        curr_sol->len[row]=new_sol->len[i];
        for (int j=0; j<curr_sol->len[row]; j++) {
            curr_sol->digit[row][j] = new_sol->digit[i][j];
            curr_sol->conf[row][j] = new_sol->conf[i][j];
        }
        curr_sol->N++;
    }
}

```