MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence
Memo. No. 176

May 1969

Discovering Good Regions for Teitelman's Character Recognition Scheme

Patrick Winston

Teitelman's Scheme:

Warren Teitelman presented a novel scheme for real time character recognition in his master's thesis submitted in June of 1963. A rectangle, in which the character is to be drawn, is divided into two parts, one shaded and the other unshaded. Using this division, a computer converts characters into ternary vectors in the following way. If the pen enters the shaded region, a 1 is added to the vector. When the unshaded region is entered, a 0 is appended. Finally, if the pen is lifted from the writing surface, a w is generated. Figure 1 illustrates the basic idea he used. Thus, with the shading shown, the character V is converted to 1 0 w 1 0.* A V drawn without lifting the pen would yield a 1 0 1. A T gives 1 0 w 1, and so on.

Notice that each character may yield several vectors, depending on just how it is drawn. The vectors to be stored, then, depend upon the style of the user as well as the division of the rectangle into shaded and unshaded regions.

In order to conserve storage space and reduce search time, the character vectors of Teitelman's scheme are stored in a tree-like structure like that shown in figure 2. Notice that the tree is essentially binary--only

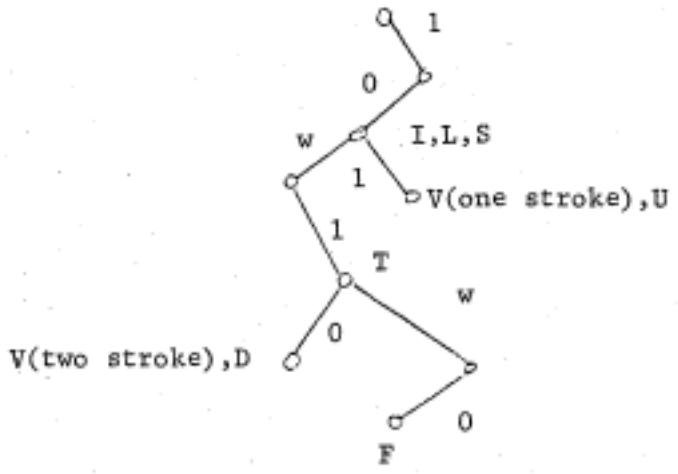
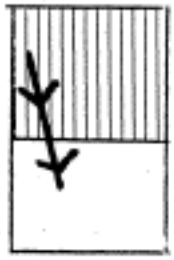


Figure 2. A vector storage tree.

*Since all figures are completed by lifting the pen, it shall be the convention to drop the final w.

1

When the pen is first applied, it lands in the shaded region, thus yielding an initial 1.

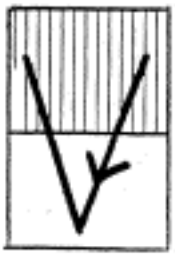


1 0 w

The pen enters the unshaded region, yielding a 0, and then leaves the writing surface, generating a w.

1 0 w 1

The second stroke begins in the shaded region.



1 0 w 1 0

And the pen enters the unshaded region, completing the character V.

Figure 1. Teitelman's Scheme.

two branches can grow from a single node. This follows since a w may be followed only by a 1 or 0; 0 only by 1 or w; and 1 only by 0 or w.

Since several characters often end up at the same position on the tree, Teitelman elected to resolve ambiguity by combining the information from several different region partitionings and their associated trees. The regions he used are shown in figure 3. His program takes a weighted look at the characters suggested by each tree.

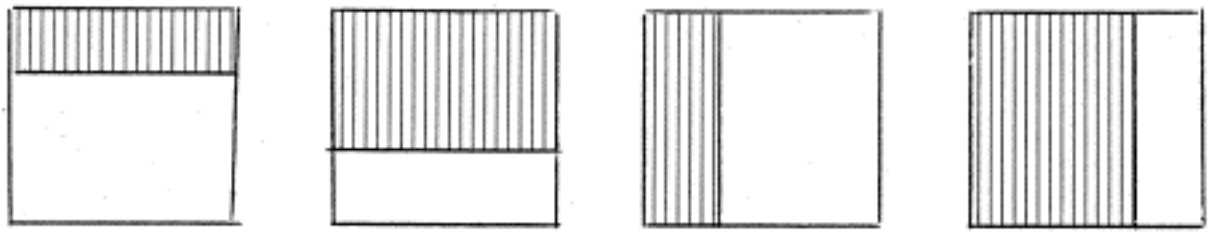


Figure 3. Teitelman's regions.

Comment:

Teitelman claims his scheme is superior to other character recognition methods because the program uses the order in which parts of the character are drawn. While this is no doubt an important factor, I feel that something should also be said about the program's sensitivity to connectedness. By this I mean that the program notes not only the presence of lines, but also the general areas they connect. For example, Teitelman's program would respond to figure 4-a by indicating "there is a line connecting the upper left corner to the lower left corner," whereas to figure 4-b the response would be quite different. Recognition schemes based on template

Figure 4. A connected and an unconnected sample.

matching might equate the two figures indicating "there are vertical lines in the upper left, middle left, and lower left corners."

Finding Good Partitions:

Suppose the input rectangle is divided into nine "atomic" areas as in figure 5. If the part to be shaded is selected from combinations of

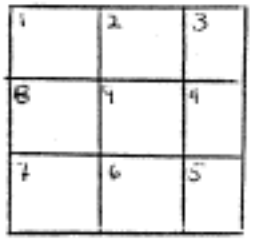


Figure 5. Atomic areas.

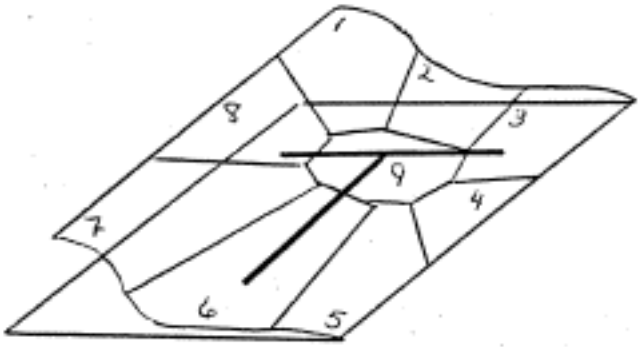
these atomic regions, one must suffer with 2^9 or 512 possibilities.* The best combinations will obviously depend upon the character set to be recognized. But to construct and evaluate all possible partitions and associated trees would be exceedingly tedious. On the other hand, finding any sort of analytic method of optimum partitioning seems equally formidable, if not impossible. Hence the problem of finding a partition that is in some sense good seems best approached heuristically.

Two simplifications were made here. First, I assume only one partitioning is available for the recognition process. And second, that only one version of each character is to be learned and recognized. The results are sufficiently enlightening that these assumptions seem justified.

*If duals formed by reversing shaded and unshaded regions are eliminated, there are still 256 possibilities.

Input:

The characters were converted to number strings by hand. First the character set to be recognized was printed into rectangles. Then a clear plastic template bearing the numbered atomic regions was placed on each character in turn. The course of the pen through the atomic regions can then be easily read off and put into the machine in the form of a number list. See figure 6.



T converts to 2 9 6 w 1 2 3

Figure 6. Conversion of characters to number sequences.

Atomic Regions:

Rather than build the shaded regions from the atomic regions of figure 5 as Teitelman did, I chose those of figure 7.

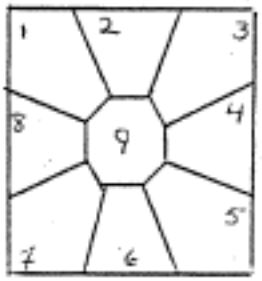


Figure 7. Atomic areas used.

I think this set is somewhat better in accomodating letters with diagonal strokes such as K, M, N, R, V, W, X, and Y. The advantage is that minor variants are not split into different sequences of input numbers. This is desirable since it renders the assumption of only one version per character somewhat less unrealistic. X, for example, yields the sequence 1 9 5 w 3 9 7 with my scheme, even if its position and slant are slightly altered. But using Teitelman's layout, the first stroke alone yields four variants as shown in figure 8. Coupled with four

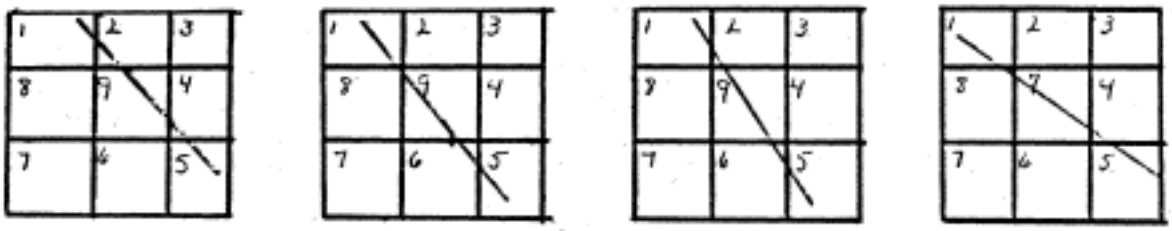


Figure 8. Variants of first stroke of X.

for the second stroke, X would have 16 possible sequences.

Tree Criteria:

As various partitionings are proposed, there must be some measure of how good the trees they generate are. After some thought, two qualities were selected as most important: first, the number of branches should be small; and second, instances of more than one character at a node should be rare. Subroutines were written to examine tree structures for these qualities and return with a pair of numbers. The first number was just the number of branches. The second was the probability of error,

given that all characters are equally likely and that when several characters are found at a node, one is selected at random.

The branch number, B, and the probability-of-error number, P, are combined into an overall badness factor, W, by the following formula: $W = B^2 + c_1 P$. This formula was arrived at by certain intuition-guided considerations designed to provide a reasonable balance between probability of error and number of branches. Ideally there would be just as many branches as there were characters and no characters would be confused.

B was squared since the penalty for branches should be small until there are somewhat more branches than characters, but then the penalty should rapidly become severe. P was entered linearly since there seemed to be no good reason for another form. The weighting constant, c_1, was selected so that the contribution from each term is the same when there are twice as many branches as characters and when the probability of error is 1/10, both conditions seeming equally bad to me.

$W = P\ (c_2 B)^2$ is an attractive alternative formula I have not explored.

The Heuristics:

The program itself was straightforward. A flowchart is given in figure 9. Its goal was to take an existing partitioning and improve it by either adding or deleting a single atomic area. It was guided in this by four simple heuristics -- two were specifically designed to reduce the number of branches, and two to reduce the probability of error. Which pair was tried first depended on the particular weaknesses of the existing tree. Recall that the inspection routine returns not with just an overall merit factor, but rather with a branch number and a probability-of-error number. If the contribution from the branches to the badness factor is greater than that of the probability-of-error, then $B^2 - cP > 0$ and the branch heuristics are tried first. Otherwise the probability heuristics are tried first.

The first branch reduction heuristic is simply to remove from the shaded region an atomic area that has no common border with any other in the shaded region except possibly the center. Intuitively this should reduce the length of many vectors as it does for the D vector in figure 10.

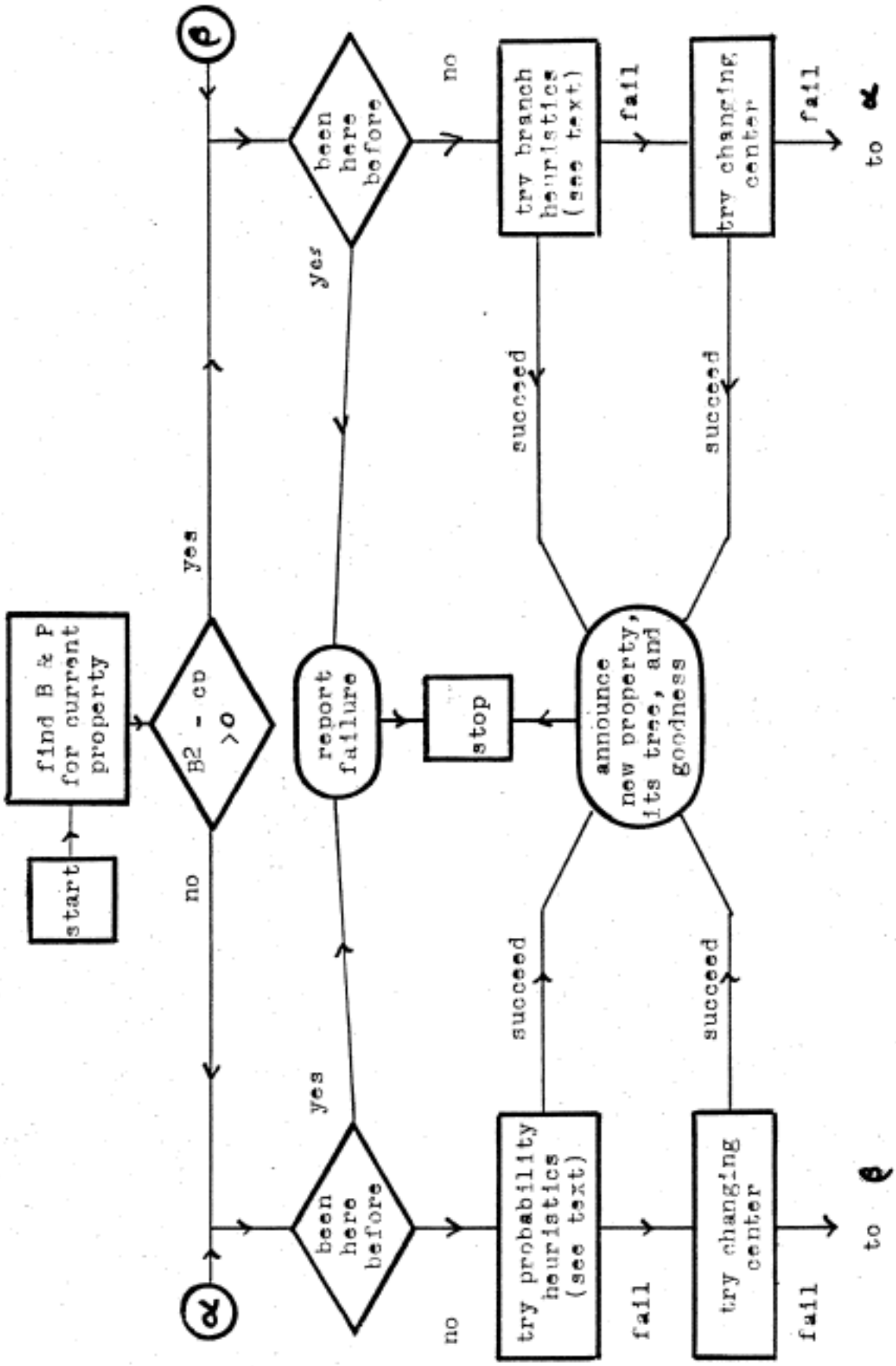
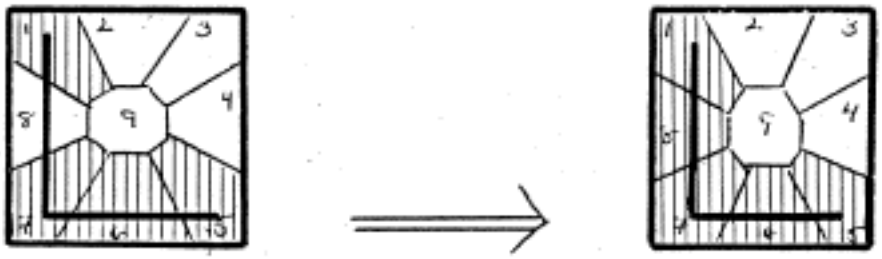


Figure 9. Program flowchart.

D: 1 w 1 0 1 0 1 D: 1 w 1 0 1

Figure 10. Branch heuristic 1.

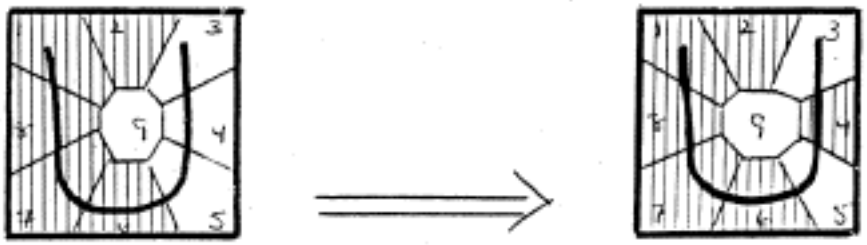
The second branch reduction heuristic adds an atomic area that shares both its non-center borders with areas already in the region. Note how this reduces the length of the L vector in figure 11, for example.



L: 1 0 1 L: 1

Figure 11. Branch heuristic 2.

Since reducing the number of branches increases the liklihood of finding more than one character at a node, as one might expect, the probability-of-error reducing heuristics complement the branch reducing heuristics. The first probability heuristic involves adding a separated region; the second involves dropping an interior region. See figures 12 and 13.



C: 1 0 C: 1 0
U: 1 0 U: 1 0 1 0

Figure 12. Probability heuristic 1.

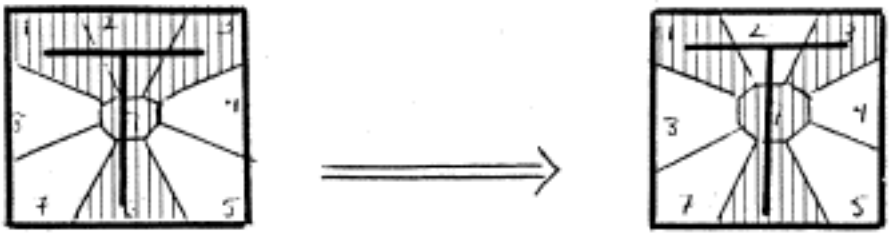


Figure 13. Probability heuristic 2.

Note that if the first pair of heuristics tried fail, then the program tries adding or deleting the center region. Then, in desperation, it enters the remaining pair of heuristics. If none of these work, it reports that it has failed.

RESULTS

English Block Capitals:

The program was used on the 26-letter English alphabet with encouraging results. Starting with nothing in the shaded region, the program evolved the 2-3-4-6-8 region of figure 14. Only 63 branches

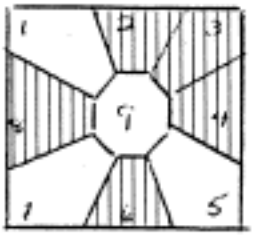
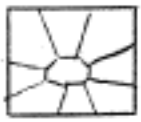


Figure 14. Block capital region.

were used of which 12 were w branches that cannot bear letters. The L is confused with Z, and W with V. The program looped 8 times and tried 17 of the possible 512 partitionings before finding the best it could come up with. The region changes are indicated below.

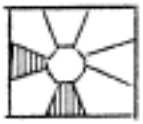
first figure is B, number of branches
second figure is P, probability of error

1  (0 25/26)

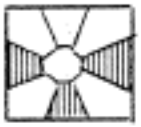
probability heuristic called + used

2 (19 16/26)

probability heuristic called + used

3  (32 10/26)

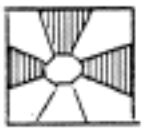
probability heuristic called + used

4  (46 7/26)

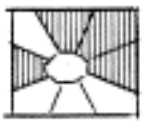
probability heuristic called + used

5 (65 4/26)

branch heuristic called + used

6  (54 5/26)

probability heuristics called and not used
center change did no good
desperation branch heuristic used

7  (53 4/26)

probability heuristic called + used

8 (64 2/26)

failed

Greek Lower Case:

Results were even better using the lower case Greek letters in the form seen in figure 15. The final tree had 52 branches with only ϛ and ϐ sharing a node. Again, the program was called 8 times and again the first heuristic tried was used in 6 or the 7 successful attempts at improvement. Only 12 configurations were tried before the winner was found. The winning tree is shown in figure 16.

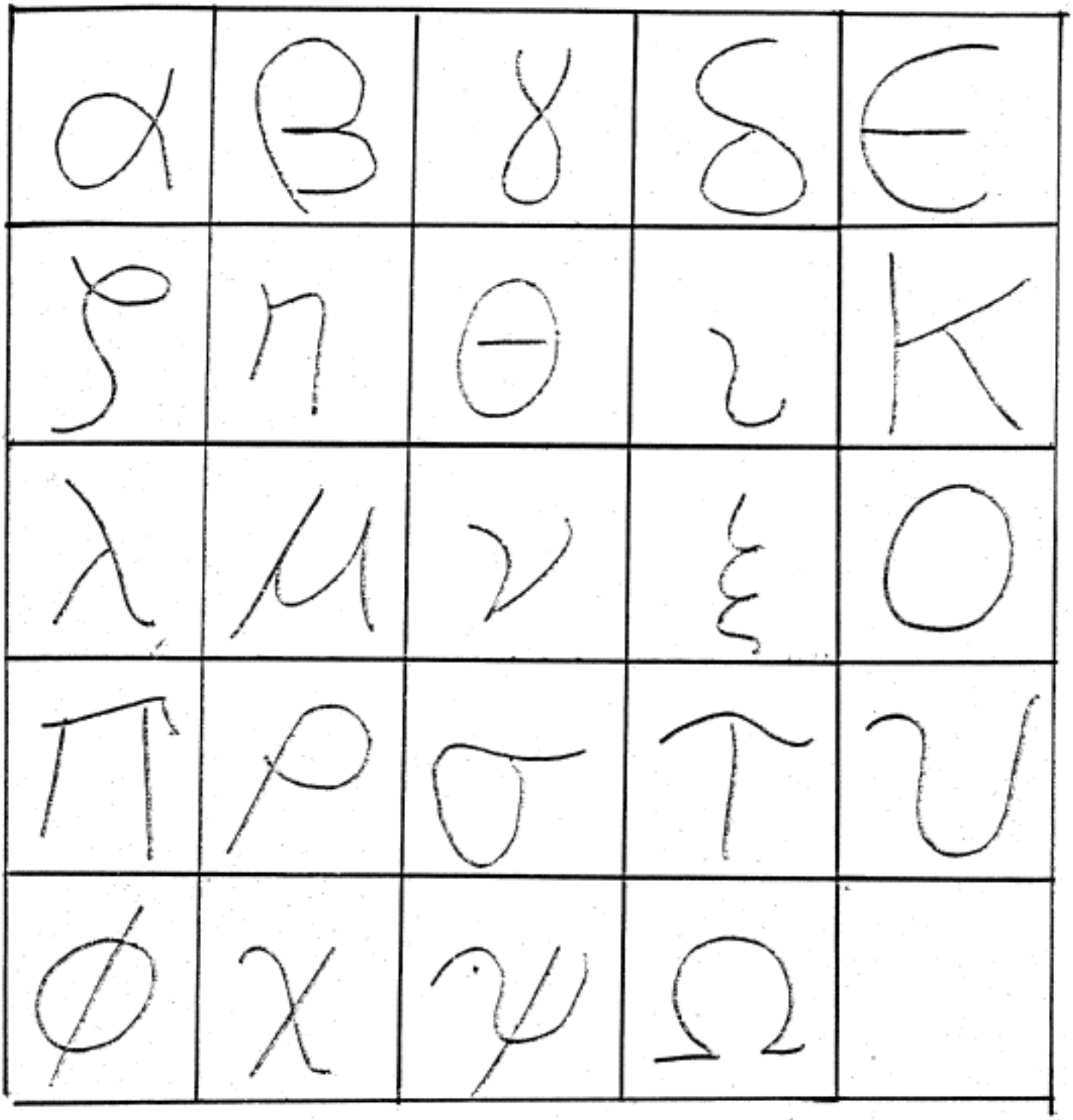
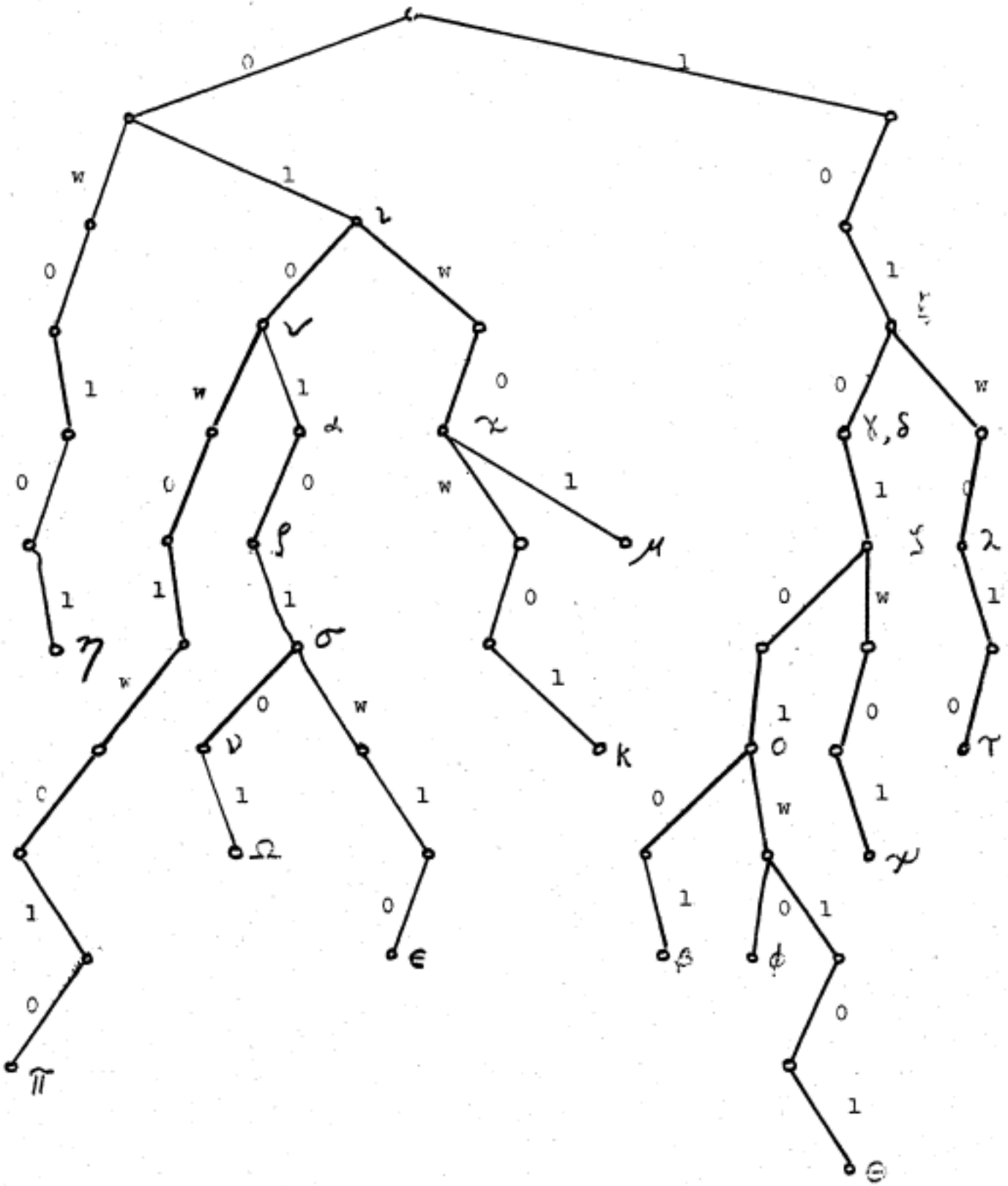
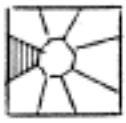


Figure 15. Greek alphabet.


0
1
w
1
0
0
w
1
0
1
w
1
0
0
w
1
0
1
0
w
1
0
1
0
w
1
0
1

1 (0 23/24)

probability heuristic called + used

2  (17 16/24)

probability heuristic called + used

3 (34 10/24)

probability heuristic called + used

4 (43 9/24)

probability heuristic called + used

5 (58 3/24)

branch heuristic called + used

6 (45 4/24)

probability heuristics called + not used
center change did no good
desperation branch heuristic used

7 (42 3/24)

probability heuristic called + used

8 (53 1/24)

failed

A second experiment with the Greek letters was performed to see if the program would home in on the 2-4-5-6-8 region from another starting point. The 9-4-8 region was selected more or less at random as an alternate starting point, and the program successfully reached the same end state, this time in 6 steps. See figure 17.

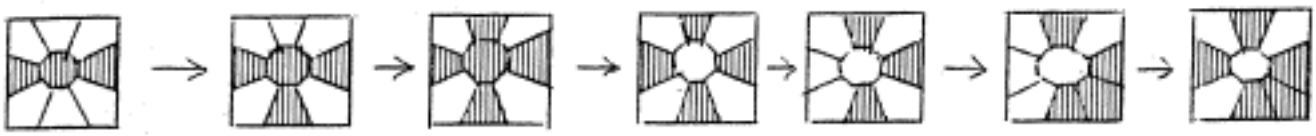


Figure 17. Sequence of areas reported in second Greek alphabet experiment.

A third experiment started from 1-9-5-7-2 but failed without arriving at the former solutions. See figure 18.

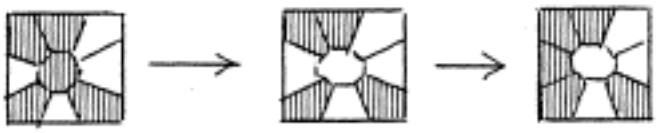


Figure 18. Sequence of areas reported in third Greek alphabet experiment.

CONCLUSION + EXTENSIONS

Conclusions:

The program as it stands seemed to work quite well. In general, the first heuristic tried succeeded in improving the badness factor by reducing the branch or probability factor it was designed for. Experiments with the Greek alphabet indicate that final results do not depend stongly on the starting pattern -- in two cases the same final pattern evolved, and in a third, the final pattern was different, but the badness factor was nearly the same.

Further Work:

It would be interesting to pump in more and more characters to see at what point saturation can be exhibited. That is, about how many characters are required before the program is incapable of reducing the probability of error below some arbitrary figure, say 1/5.

The badness factor, $W = B^2 + c_1 P$, worked surprisingly well. The powers used, the constant c_1, and the general form were a bit arbitrary and might be improved.

Generalizing the problem to the case in which a character is associated with several input sequences would certainly require a more general function.

If more than one tree is allowed for recognition, the problem becomes more difficult. W would have to apply to sets of trees, and heuristics would have to be found to operate on sets of partitionings.