PROGRAMMING A COMBINATORIAL PUZZLE

by

Dana S. Scott

Technical Report No.1
10 June 1958

This work was performed at Princeton University under a contract with the Information Systems Branch of the Office of Naval Research [(Nonr 1858(22)]. Reproduction of this report in whole or in part is permitted for any purpose of the United States Government.

Department of Electrical Engineering
Princeton University
Princeton,New Jersey

Summary

In studying the difficulties of communicating well-posed problems to digital computers, it seems desirable to gain experience not only with engineering problems, which are principally arithmetical, but also with problems that are principally logical. For this purpose a variant of the geometrical placement puzzle called "pentominos" was programmed for the von Neumann computer. This report is an account of the logical difficulties encountered and how they were solved. It incidentally includes solutions to the puzzle. The work was performed under a contract with the Information Systems Branch of the Office of Naval Research [(Nonr 1858(22)].

Introduction

The problem solved is a kind of jigsaw puzzle called pentominos in which 12 different shaped pieces are fitted into a square. Many different solutions are possible, so that a satisfactory program must not stop merely upon finding one of them. Before actually giving the details of this puzzle, it may help focus attention upon the principal difficulties if we pose an auxiliary problem.

Suppose you have been asked to construct a list of all English words with exactly ten letters. Obviously such an extensive list could only be needed for a very worthy purpose. No doubt you have been asked to do this by a government agency who will use the list to promote international understanding. Now there are several ways you can set about making this list. First, you could simply collect together all the reliable dictionaries and tell your secretary to go through them page by page typing out just those words with ten letters. There are several things wrong with this plan: the government agency could have thought of this themselves, and in any case even when the list is typed you cannot be sure it contains all possible ten letter words. So you must be more systematic. The second plan is to tell your secretary to type out all combinations of the twenty-six letters of the alphabet into ten letter groups. Then you will only have to read through the whole list crossing out those words that are not English. Assuming 400 words to a page, there will be more than ten billion pages necessary for this initial list. In all probability your secretary will have left her job to get married long before the work is complete, and you could never explain to the new girl how to continue. This plan is just too systematic.

What ought to be done here is to construct rules for generating a list of words which then can be written out in a reasonable length of time and which you can be absolutely sure contains all the necessary words. English is a rather complicated system, and the author has no idea how to construct rules for such a list. However, the pentamino puzzle is an exactly similar combinatorial problem of about the same order of magnitude for which the rules are easier to state.

A general plan of attack for a class of such problems is explained first under the name of the "back-track" method. Next, the specific puzzle is presented in detail. Two different ways of programming the problem are then described, and some aspects of the actual machine coding are discussed. A complete table of all the solutions of the one form of the problem which was run on the MANIAC is included as an appendix.

The author would like to take this opportunity of expressing his warmest thanks to Dr. Hale F. Trotter of Princeton University. Dr. Trotter not only spent several hours in discussing the problem, but made many helpful suggestions in coding and in planning the program. Further, his assistance in operating the MANIAC was invaluable.

Back-tracking

Abstracting from the fictitious example of the introduction, let us suppose that we have m objects which may as well be the numbers 0, 1, 2, ... , $(m-1)$. For English, $m = 26$. We wish to make a list of sequences of these numbers where there will be a fixed length for the sequences, say n. In the example for the government agency $n = 10$. Clearly the maximum possible length for a list without repetitions is m^n, but we hope never to come even close to this maximum. We suppose that there is an exact criterion for the sequences that we eventually want to find, so that there is no ambiguity as to what the answer to the problem is. Further, the decision as to whether a sequence is to be put into the final list must be independent of anything else already in the list. That is, given an n-termed sequence the criterion for acceptance should give the answer yes or no without referring to the previously accepted sequences. This condition on the problem is necessary to make the solution independent of the order in which the problem is solved. The reason for such a restriction is that the "back-track" method is based on a special order of listing possibilities, and this particular order should not have any influence on the validity of the final results.

The order used in back-tracking is the simple lexicographical order of sequences. This can be easily explained by using numbers written to the base m. There are exactly m^n non-negative integers which have at most n digits when written in the base m. These n-digit numerals are clearly the same as sequences using the numbers 0, 1, ... , $(m-1)$, and the natural order of the numerals is just the lexicographical order of the sequences.

Aside from the fact that there are too many n-termed sequences, there is no difficulty in writing them all down. Think of numerals to the base m again and start with 00...0 (n times). By systematically adding one to the previous result and making the carries in the proper way for base m arithmetic, one obviously works through <u>all</u> the n digit numerals in their correct lexicographical order. Even programming such a system on a digital computer presents no problem at all, except for space and time.

The most obvious reason why this exhaustive procedure is silly in the case of English is that most of the labor in writing the list of all m^n possibilities is concentrated in the least significant positions in the words. It is quite clear that after about three letters from the left, one can tell if there is going to be any hope of obtaining an English word. For example, the three letters "ooz" are the beginning of the four letter word "ooze", but it seems unlikely that they could occur at the beginning of a ten letter word. Possibly better, the three letter combination "ibm", rather more subtle than such combinations as "yyw" or "tcx", is to the author's best knowledge not the beginning of any English word. Thus, there is absolutely no point in making a list of all ten letter combinations since vast stretches of such a list can be eliminated simply by looking at fairly short initial segments.

The back-track method, then, is based on having a set of rules for eliminating big portions of the full lexicographical list. The rules must be such that, looking at an initial segment, they determine whether all sequences beginning with that segment should be discarded or whether another letter should be added. Of course, when the sequence is built up to the maximum length of n letters, a final decision to accept or reject the complete word must be made.

Suppose the rules have been set up. Let the various letters be the numbers 0, ... , (m-1). Start by writing down the first letter in the left hand position 0. The rules will then determine whether this is a good start. If it is, adjoin one more letter, namely, the first letter in the natural order of the single letters, obtaining 00 . Continuing in this way, consulting the rules at each step, it might be possible to build up to the sequence 000000 . The next step would be to write 0000000 . Here the rules might say that there was no point in going on because no acceptable sequence begins in seven zeros. So now one forgets about seven zeros and tries instead 0000001. If that is still wrong, one tries 0000002. Suppose now that the rules do not forbid going on; the next step is to write 00000020. Proceeding in this way one might build up to 000000231. At the following steps the rules might say that each of the sequences 0000002310, 0000002311, 0000002312, ... , 000000231(m-1) is wrong. Thus, in an indirect way, we have found that the initial segment 000000231 is wrong. The rules did not tell us this fact in the first place because the situation was too complex for this particular contradiction to be built into the rules. For a set of rules to lead to efficient solving of the problem, they must not only eliminate possibilities early in the game, but also they must be simple, since otherwise it will take too long to use them. Hence, we must expect to find indirectly that certain initial segments are impossible.

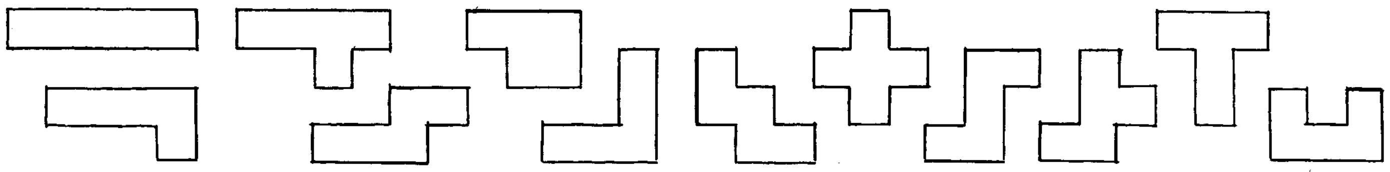
When indirect contradictions are found, the only thing to do is to back-track. Thus we had already written down the sequence 000000231, and only at the next step found that this led to an impossible situation. So we have to go back and erase the 1 at the end and try the initial sequence 000000232. It may turn out that all of the numbers put into the ninth position will indirectly lead to a contradiction. Thus, there will come a point in the back-tracking when we find the sequence 00000023(m-1). Now we cannot erase the number (m-1) and put something else in the ninth position, because everything has already been tried in that position and found to lead to failure. There is nothing to do now but back-track further and start trying from the sequence 00000024. Eventually it is going to be possible to work to the very end, and an acceptable sequence will be found, say 0000002450728. Having obtained the first success, we should not stop, not start over, but simply go ahead and try the sequence 0000002450729. And so on, back-tracking every time all the m digits have been tried in any position. In this way we never lose ground. Finally it will be discovered that further back-tracking is impossible because we have arrived at the place where we would have to modify the initial segment (m-1); then the problem is finished. Of course records of all the successes have been kept, and this list will be the complete answer to the problem with all the solutions given in proper lexicographical order.

The main problem in applying the back-track method is to formulate the proper set of rules about impossible initial segments. Since the method is really very simple and general, it is hard to talk about it in the abstract. Thus, in the next section a fairly complex example of this kind of problem is given. There are two quite natural and distinct ways to apply the back-track method to the particular problem, and it will be seen that one works much faster than the other. This illustrates the point that there are generally many ways to attack the problem, and the first that suggests itself may very well not be the best. There can be no very general rules given for using the method, because it will be seen from the example how many special features were used to set it up. It is hoped, however, that the example is suggestive enough to give ideas applicable to other kinds of listing problems.

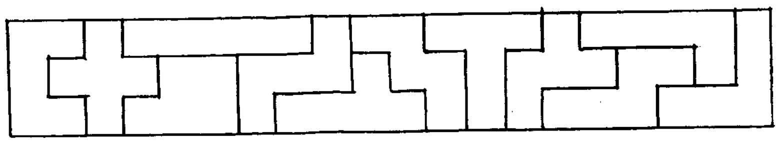
The back-track method is very natural when one starts thinking about this kind of problem, and the author used it in an early version of a program for a special case of the pentominos puzzle. However, the idea was not looked upon as a general method. It seems that the first person to suggest the system as a method was Professor R. J. Walker who read a paper entitled "An enumeration technique for a class of combinatorial problem" at the New York meeting of the American Mathematical Society on April 26, 1958, which the author had the pleasure of hearing. The name "back-track" is due to Professor D. H. Lehmer.

Pentominos

The problem of fitting pentominos together to form various configurations was first brought to the attention of the author by Professor and Mrs. R. M. Robinson, who had read about the puzzle in the article "Checker Boards and Polyominoes" by S. W. Golomb, American Mathematical Monthly, vol. 61 (1954) pp. 675-682. Just as a domino consists of two adjacent squares, a pentomino consists of five adjacent squares. It turns out that there are twelve ways to make distinct pentominos allowing for identifications of the pieces by rotation and reflection. The twelve pieces are as follows:

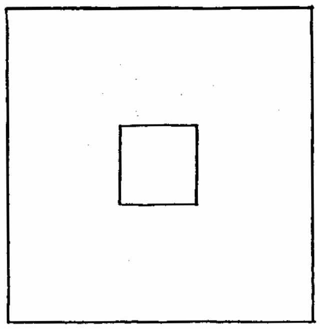


The problem is to fit these twelve pentominos together into a rectangle making use of just one of each kind. Since 12x5 = 60 and the number 60 can be factored in many different ways, there are many different forms that the puzzle can take. For example, 60 = 3x20 and the twelve pieces can actually be put together to form a 3x20 rectangle as follows:



The factorizations 1x60 and 2x30 obviously lead to impossible rectangles, but all the other factorizations 3x20, 4x15, 5x12, 6x10 correspond to rectangles that can be made.

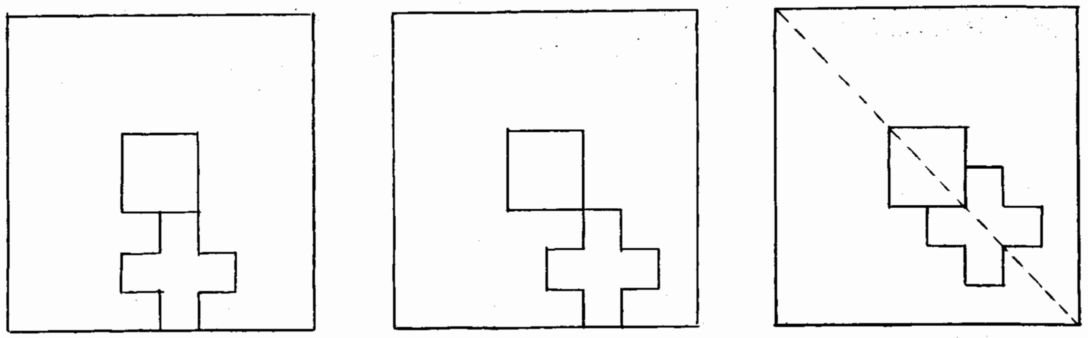
Another form of the puzzle makes use of the checker board. An amusing story about the puzzle can be found in the book "The Canterbury Puzzles" by Henry Ernest Dudeney (London, 1919), pp.119-121. In this form the twelve pieces have a checker board coloring which introduces an additional constraint in the problem that we do not want to consider here. However, the checker board suggests another shape in which we can put the plain pentominos. A checker board has 8x8 = 64 squares while the twelve pieces only add up to 60 squares. Hence, four squares will have to be left out. The most symmetric place to leave out four squares is the 2x2 square in the center.



This form of the puzzle has the advantage of having many solutions (65 in all) without the number of combinations becoming unreasonably large. The 3x20 puzzle has even fewer places that pieces can fit, but it unfortunately has just two distinct solutions, which makes the combinatorial problem somewhat uninteresting. Of course, we do not consider solutions to any of these puzzles to be distinct if one can be obtained from another by rotation or reflection of the whole configuration.

There are other forms of the 8x8 problem that are interesting. For example, the 2x2 square can just be considered as a thirteenth piece and allowed to appear anywhere within the big square. There must be a vast number of distinct solutions to this more general puzzle, for Mr. Robert Gibbon, an undergraduate at Princeton University, has catalogued over 700 distinct solutions working them out by hand, and he believes he may have as many as 600 more that have not been fully sorted yet. The author would like to record here his indebtedness to Mr. Gibbon whose remarkable proficiency and enthusiasm were most encouraging.

For the purposes of this report we shall now restrict all our attention to the symmetric form of the 8x8 puzzle. The problem is to devise a systematic procedure for enumerating all the possible distinct ways of solving the puzzle. Since there would be much wasted effort if we listed solutions which differed only by a rotation or a reflection, the first question is how to take the symmetry out of the problem without destroying our chances of finding all the solutions. This is easily done by considering the pentomino in in the shape of a cross. If there were any solution to the puzzle, then by rotating or reflecting the 8x8 square it would be possible to obtain an equivalent solution with the cross in one of the following three positions:



Thus, by considering these three cases separately we have eliminated almost all chance of getting non-distinct solutions. Only "almost" because the third position of the cross above still has an axis of symmetry about one of the diagonals. This last bit of symmetry can finally be eliminated by restricting the possible positions of one of the totally unsymmetric pieces when working on the third case. We are now ready to see how the back-track method can be applied to this problem.

Programming Pentominos

For simplicity only the first position of the cross will be discussed. The other cases are treated quite similarly. (In fact the program was simply run three times with the different positions of the cross put in as three different initial conditions.)

In applying the back-track method it is necessary to encode the problem into numbers; that is, an interpretation of the two quantities n and m must be given. Then rules must be set up for eliminating initial segments. This will be done in two different ways for this particular puzzle. In both cases we shall have $n = 11$, since after the cross has been placed, there are only eleven pieces left to fit in. The simplest way to proceed is to give each of the remaining pieces an index of one of the numbers 0, 1, ... , 10. Then each of the pieces is tried in all possible ways in the square, and a list is made of all the ways it can fit in. Each of the lists is ordered in some convenient way and numbers are assigned in succession 0, 1, 2, ... The number m is then defined to be the maximum length of any one of these eleven lists. Since in this problem the lists are not all of the same length, let us use the notation m_i to denote the length of the i^{th} list. Thus, m_i is the total number of ways the i^{th} piece can be placed in the 8x8 square.

With these conventions, a sequence of eleven numbers

$$k_0, k_1, k_2, \dots , k_{10}$$

can be interpreted as meaning that the first piece has been put in its k_0^{th} position, the second piece in its k_1^{th} position, and so on. Obviously, this sequence of numbers represents a solution to the puzzle if, and only if, there is no overlapping of the various pieces when they are put in the required positions. Of course, every solution can be represented in this way.

These considerations at once suggest the rules for eliminating obviously impossible initial segments: <u>The initial segment $k_0, k_1, \ldots, k_p$ is considered impossible if either $k_i > m_i$ for some $i \leq p$, or if the positions of these first p+1 pieces overlap with one another in any way</u>. The way to arrange the computation is to keep an accumulated "total" running all the time of those squares which have already been filled up. Then when a new piece is tried it has only to be checked against this total. The rules are complete; the back-track method will now generate a list of all the possible solutions.

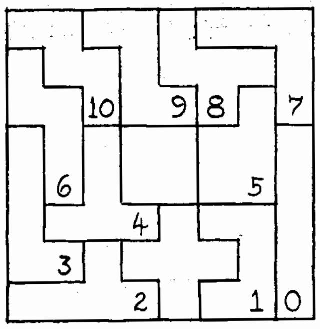
The main trouble with this application of the back-track method is the very large number of possible positions for each piece in the square. In getting the problem on the machine, the programmer will either have to write out as fixed data all the positions of all eleven pieces, or he will have to invent a way of having them generated inside the machine. Either way is a lot of trouble. In the case of the 3x20 puzzle the method was actually programmed in this way with the machine generating the possible positions. It worked, but was rather slow. It quickly became apparent that for the 8x8 configuration there was simply too much freedom inside the square to make this plan at all practical, so another way to apply back-tracking to the problem was devised.

<u>A Second Program</u>

In the system just described, we proceeded by placing the first piece first, the second piece second, the third piece third, and so on. For example, at one stage it would be necessary to consider a configuration where each of the first four pieces were placed in four separate corners. In continuing forward from such a configuration it will be found only after much work that the initial four positions indirectly lead to an impossible situation. Of course, no reasonable set of rules will eliminate all impossibilities at once, but it seems that allowing the various pieces to be spread out at large distances from one another makes the impossibilities much less obvious. No doubt in working the puzzle by hand these impossible configurations will become apparent to the eye very quickly, but communicating to the machine the rules for recognizing these configurations is a very complicated problem. The idea that occurred to the author at this point was not to try to put the pieces into the square in a fixed order, but rather to try to put them in in bunched-up configurations that would lead to many overlappings as quickly as possible. Several plans were constructed and discarded before the following simple procedure was discovered: Fill the square up a row at a time.

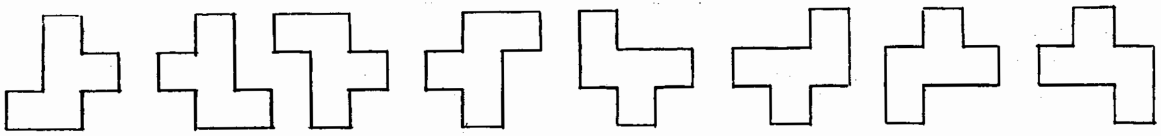
When the problem was finally coded, it turned out that it was more convenient for the machine to fill the square up from the bottom right-hand corner reading from right to left up to the top left-hand corner. Clearly the choice of one direction or the other makes no difference in the theory behind the method. It should be noticed, however, that the positions of the cross were chosen to give contradictions as early as possible in the placing of the pieces.

As an example of how this scheme works, the following drawing is the first solution that was found by the machine, where the numbers indicate the order from bottom to top in which the pentominos were placed. It must be remembered that before this solution was found, many different pieces were tried in their various positions before the right piece was finally fit in and allowed to stay in each of the numbered squares.



In the appendix at the end of this report a table is given of the 65 solutions to the puzzle. The order of the solutions is exactly the order in which they came out of the machine. Inasmuch as the machine only preserved the successes, the back-tracking cannot be exhibited in full, but it can be seen from the complete table how successive solutions were obtained from the preceding one by changes in the "later" pieces. Of course, there are many jumps in the table where very little relation can be seen between the solutions. This simply means that there was an intervening period of extensive back-tracking where none of the earlier placings could be used.

It will now be explained exactly how the method was applied. Each piece was given one of the indices 0, 1, ... , 10. Instead of numbering the possible positions of each pentomino, only the different orientations were numbered. For example, here is a pentomino that has the maximum number of eight orientations:



It will be seen that five out of the twelve pentominos have all eight orientations, while the remaining have fewer since they each have at least one axis of symmetry. Thus, a piece with an orientation is identified with a pair of numbers (i,j) where $i < 11$ and $j < 8$. There is a total of 88 identification pairs, and in applying the back-track method we take $m = 88$. In the actual program the pairs of numbers (i,j) were identified with the integers of the form $i \cdot 8 + j$. Because some of the pieces do not enjoy all eight orientations, not all of the 88 integers are meaningful. The rejection of meaningless pairs was built into the rules. To explain these rules, the notation r_i will be used to denote the number of distinct orientations of the i^{th} piece.

The rule for rejecting an initial sequence

$$k_0, k_1, \ldots, k_p,$$

where $k_t < 88$ and $p \leq 10$, depends on writing each integer k_t in the form

$$k_t = i_t \cdot 8 + j_t ,$$

where $j_t < 8$. The sequence is <u>rejected</u> under any one of the following conditions:

Either (a) $j_t \geq r_{i_t}$ for some $t \leq p$,

or (b) $i_t = i_u$ for some t,u where $t < u \leq p$,

or (c) if the pieces with indices $i_0, i_1, \ldots, i_p$ are placed successively in the orientations $j_0, j_1, \ldots, j_p$ in such a way that at each placing the empty square farthest to the right and nearest to the bottom is filled, then either some pieces overlap or some pieces stick over the edge or into the hole in the middle.

Referring back to the diagram on page 8, it is seen that the numbers 0, 1,...,10 have nothing at all to do with the indices given to the pieces but only refer to the <u>order</u> in which the pieces were placed. In this use of the back-track method the interpretation of the integers n and m is very different from that explained on pages 6 ff. Notice also that the rules for rejection are more complicated.

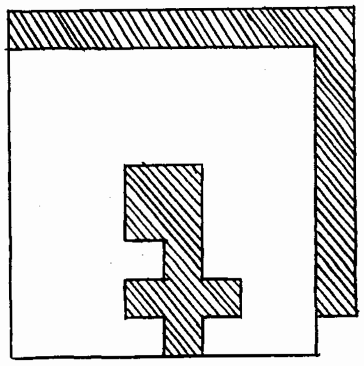
That the rules for the second form of the back-track method are more efficient than those for the first form will only become really clear when the programming technique is explained in the next section. However, it can be seen now that if several lists of indices and accumulated totals are kept running at the same time, then the checking of the rule upon the adjunction of a new piece requires only some quick comparisons of numbers rather than any extensive calculation.

Coding

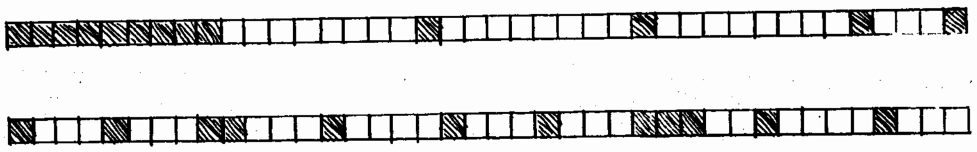
From the description given above it can be seen that the main problem in the program was to handle several lists of indices that were continually being modified. Some of the indices were simply numerical and could be incremented and decremented in the natural way, but others were to be used to indicate which values of indices had been already used up and how much of the puzzle had been filled up, and this is non-numerical information. The whole code will not be described in detail, but only certain features peculiar to the problem.

First, we must decide on the representation of the geometric configuration in the machine. Clearly since this is a question of filling up squares, a 0 will represent an empty square, and a 1 a filled square. The binary character of the MANIAC made this choice seem even more natural. Now generally the words in the machine are thought of as a linear pattern, 40 bits per word for the MANIAC, reading from left to right. The problem under consideration is two-dimensional, so it has to be cut up to fit into the machine. A simple way would be to think of each of the 8x8 square as put into a different word. This is obviously wasteful because the words are 40 bits long, and only 8 bits per word would get used. It follows, then, that we shall try to get several rows of the square into one 40 bit word. This is a good idea except for the fact that one has to keep track of where one row begins and the other ends. Since the nature of the problem is to start with an empty pattern and fill it up little by little, the obvious suggestion is to put an already filled bit between the rows in the word to keep them apart.

To understand this last point more clearly, think of the two-dimensional configuration first. We must not only be careful not to let our pentominos fall over the edges but also not to let them stick into the 2x2 hole in the center. So the edges and hole are filled up to begin with, and the pieces can only be placed in the empty squares. Note that it is not necessary to put a border all around the square but only on two sides. Thus, the initial configuration with the first position of the cross already filled in will look like this:

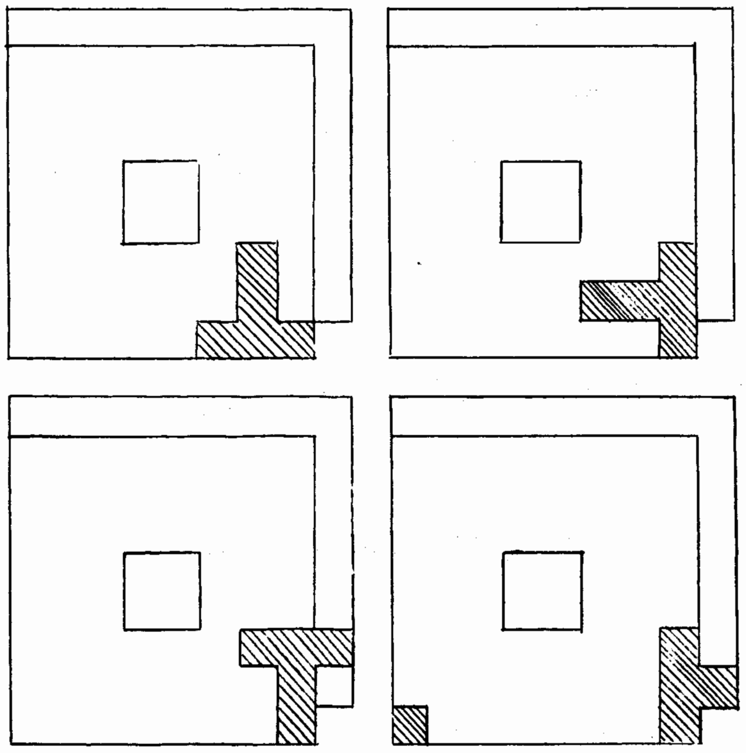


There will be, of course, two other initial configurations corresponding to the other two positions of the cross. Finally, to put this information into the machine, the configuration is cut into strips, and they are laid end to end. Notice that exactly 80 bits are necessary, which make two words as follows:



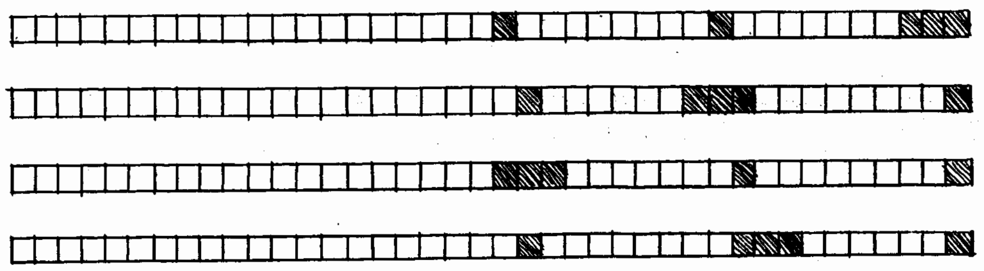
So far, all we have in the machine is that which is to be filled and nothing to fill it with. This brings us to the next problem.

For the sake of illustration, only the one pentomino in the shape of a T will be considered. The coding of the orientations of the other pieces is exactly similar, and the T has the advantage of only needing four orientations. Before cutting into strips, the four positions of the T look as follows:



It will be seen at once that in two of the above orientations the piece overlaps with the border and in one it overlaps with the cross. It would appear then that only one was necessary. But this conclusion is wrong, because all the different orientations that the T can assume must be represented, so that all the relevant positions can be obtained from them by shifting. Shifting is a very simple procedure on the MANIAC, as it is on most machines.

It was noted above that the representation of the complete 8x8 square required two machine words. It turns out, just by chance, that since none of the pentominos are more than five squares long, all the positions of all the pieces fit into just one word. This fortunate circumstance was a great help in coding since it cut down on the amount of data handling. For example, in the case of the T the four orientations give us these four words, after cutting into strips and skipping a square every eight squares to correspond to the border:



Each of the eleven pieces was treated in the same way. We now have in the machine everything that is needed to fill the puzzle, and the final problem is to get the information into the correct positions.

When the smaller 3x20 puzzle was programmed, the procedure was simply a shifting of each of the orientations of each of the pieces one square to the left at a time. After each shift, a test against the partially filled rectangle was made to see if there was any overlapping. Such a test is easy on the **MANIAC**, because one of its arithmetical operations is a bit-by-bit product (sometimes referred to as an "extract" order). After a bit-by-bit product, a zero test will detect the presence of any overlapping. The drawback of this method lies in the fact that a shift-test-shift-test loop in the program is rather slow in getting to a usable result.

An excellent solution to the problem was suggested by Dr. Trotter. When a human tries to solve this puzzle, each piece is picked up and brought over to the partially completed configuration. When a machine solves the puzzle, it is immaterial whether the piece is shifted to fit the configuration or the configuration is shifted to fit the piece. It was already decided that the square would be filled up a row at a time from the bottom. In the strips, this means filling up the holes in succession from right to left. If the pieces were being shifted, a large string of 1's would be collecting on the right. However, if the partially filled configuration is being shifted to fit around the pieces, it is quite all right to drop the consecutive 1's off the end of the word, because they are known to be there anyway. This simple remark makes the program much more efficient since the data handling now only requires calling for the various orientations and immediately testing them against an already shifted configuration.

By now it should be clear that the convention of cutting the square into strips and putting them end to end allows a continual right shift of the main configuration to be interpreted not only as a right to left motion but also as a gradual bottom to top motion of the pieces relative to the configuration.

All the essential problems of machine reading and handling of the data are thus solved. The rest of the coding consisted mainly of keeping the lists of indices employed in back-tracking up to date. The details need not concern us here. However, it was a good feature of this problem that the logic of the program did not require very many single steps in the various loops of the code. Hence, it was possible to use several tricks that took up a lot of memory space but did not require much in the way of internal formation of orders, thus saving considerable running time. The final result was that one run with one initial position took just a little over one hour of machine time on a machine that performed approximately 4000 single address instructions per second. The whole problem required only about three and one-half hours. It was very hard to estimate beforehand the time needed, because it was difficult to obtain a clear idea of the total number of combinations that would be tried. The rather short period of three hours was a very pleasant surprise.

Conclusion

The experience gained while solving this puzzle led to two conclusions: that a binary machine is essential for the effieicnt solving of combinatorial problems and that some additional non-arithmetic machine orders are desirable.

Consider first the question of a binary machine. In coding pentominos for the machine, the general plan is to give a representation of a geometric configuration or pattern in the language of the machine. The use of 0's and 1's explained above would seem to be the simplest solution to the problem. It is conceivable that on a decimal machine one could use a 0 to denote an empty square, a 1 a black square, a 2 a red square, a 3 a green square, and so on, to enable coding of a more complicated problem. However, if the digits are not used in an ordinary arithmetic mode, there is no reason to prefer any one base to any other. The binary base will always be the simplest not only in the construction of the machine but also in the explanation of the non-arithmetic orders. From the more practical standpoint of using existing machines, it should be pointed out that the binary machines usually have longer words than decimal machines. It is a distinct advantage to be able to handle as many bits as possible in one operation because the factor of time is very acute in combinatorial problems. Even in the MANIAC with 40 bit words, it was necessary to deal with two-word units in the particular form of the pentomino puzzle that was programmed. A slight variation of this problem can easily lead to a program requiring three-word units for the representation of the full geometric configuration. Thus, any device that either lengthens words or makes double-precision coding easy would be of considerable help.

With respect to non-arithmetic orders, the particular operations that were most useful on the MANIAC were the shift orders, which used two 40 bit registers at a time, and the extract or bit-by-bit product operation. By now these are standard orders on most machines. Orders which allowed words to be broken apart more easily could have been very useful in two different contexts. In the first place, when storing a two-dimensional square in one-dimensional words the configuration had to be broken up into rows. It might turn out in the program that a change-over to columns would be necessary. More generally, it is a problem of transposing a matrix when the entries of the matrix are not stored in separate locations but several are packed into one word. Fortunately we were able to side-step this issue, but it strikes the author as a question that will have to be dealt with sooner or later.

In the second place, it will be noticed in the description of pentominos that symmetry plays an important role. Restricting attention even to two-dimensional squares, it will be seen that reflections cause considerable trouble. If the square is stored by rows, then a top for bottom reflection is just an interchange of rows placing the first last and the last first. However, a left for right reflection is a real nuisance. Either the problem of the transposition of the square has to be solved, or there has to be an order that inverts words end for end. Most desirable would be to have both methods available. Dr. Trotter suggested an operation that would shift left out of one register and right into a second register. Such an operation would be more flexible than a complete inversion since it would also permit an efficient division of the word into two portions of variable length.

In programming this problem only a few difficulties were encountered - and these have been discussed above. It is quite clear, however, that further attempts to solve geometrical and combinatorial problems will raise additional logical difficulties which will not be easily solved by the standard arithmetic operations.