A Program to Solve the Pentomino Problem by the Recursive Use of Macros

JOHN G. FLETCHER
Lawrence Radiation Laboratory, University of California, Livermore, California

A coding technique is described in which certain macro-instructions are given lists as arguments and are thereby used recursively. The discussion covers primarily an example in which the technique is used to solve the pentomino problem —the problem of fitting 12 pentominos without overlapping into a plane area formed of 60 elemental squares.

Introduction

The purpose of this report is to describe a technique for coding with macros which may have some general application. The technique can be made clear most readily by describing its use in a particular instance, namely, in a program to solve the pentomino problem. This program is written in the FAP language for the IBM 7094.

The Pentomino Problem

Just as a *do*mino is a plane figure formed of two contiguous equal squares, a *pento*mino is a plane figure formed of five contiguous equal squares. A fixed size for the elemental squares being assumed, there are 12 different pentominos, *different* meaning noncongruent. These are each identified by a number from 1 to 12 (see Figure 1). The pentomino problem is the problem of fitting the 12 pentominos without overlapping into a plane area (the *box*) formed of 60 elemental squares. The box may be rectangular (3×20, 4×15, 5×12 or 6×10) but is not necessarily so.

In the program, the box is viewed as a portion of a large arena, 16×32 squares; these squares are numbered from 1 to 512, 1 to 16 proceeding from left to right across the first row, 17 to 32 across the second, etc. Each square corresponds to one of 512 consecutive storage locations, square one corresponding to the uppermost, which is just below a location called ARENA. An input routine stores 0 in the 60 locations corresponding to the box; the number 13 is stored in the remaining locations. To avoid difficulty at the boundaries of the arena, the box may never include any square numbered 1–15 or 497–511 or with a number divisible by 16.

When placed in the box, the pentominos numbered 1, 3, 4, 5 and 11 may assume eight different orientations, numbers 6, 7, 8, 9, and 12 may assume four, number 2 two, and number 10 one. Thus there are 63 different *patterns* which may be placed in the box. A pattern may be uniquely identified by a set of four numbers as follows: The pattern is placed anywhere in the arena and the numbers of the five squares covered are noted. The smallest number, which corresponds to a square called the *lead*, is then subtracted from each of the other four. The result is the required four numbers. In practice it is convenient to add a fifth identifying number, namely, the number of the pentomino to which the pattern corresponds. Thus for example, the illustrated orientation of pentomino number 7 is the pattern 16, 17, 18, 2, 7. The 63 patterns are assigned a canonical order in a manner described below.

To the computer, placing a pattern in the box consists of storing an identifying number in each of the five storage locations corresponding to the squares thus occupied and in a storage location corresponding to the particular pentomino to which the pattern corresponds. (These latter locations, 12 in number, are initially set to zero by the input routine and are consecutive and immediately below a location called MINO.) The pattern may not be placed if any of these locations already contains a nonzero value, for this would mean that the pattern is falling outside the box, that it is overlapping a pattern previously placed,

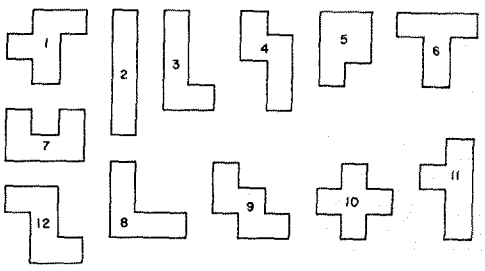


FIG. 1. The twelve pentominos

The work performed in this paper was supported by the U.S. Atomic Energy Commission.

and/or that it corresponds to a pentomino already used in another orientation. Removing a pattern from the box consists in zeroing the corresponding six locations. The identifying numbers are assigned from 1 to 12 in the order that the patterns are placed.

The search for solutions consists of an orderly exhaustion of possible placings of patterns. The locations corresponding to the arena are searched downward until an empty one is found. The patterns are then searched in order until one is found that may be placed so as to have this location at its lead. Then the downward search for an empty location continues until another is found, and another pattern is placed with the new location as lead, and so forth. If the list of patterns is exhausted, the search returns to the previous choice of lead, the pattern with that lead is removed from the arena, and the search is continued with the next pattern on the list. When 12 patterns have been placed, a solution has been found. An output routine then uses the information in the ARENA and MINO portions of storage to make an external record of the solution, after which the last pattern is removed and the search continues for a new solution. The program terminates when all arrangements have been attempted.

It is inefficient to test the six locations corresponding to each pattern every time a new pattern is considered. The lead location of course is tested only once. The lead location minus 1 is then tested. If it is not empty a large number (29) of patterns are eliminated, and the lead location minus 16 is tested. If this location is not empty, no pattern can be placed. On the other hand if lead minus 1 is empty, then lead minus 2, 16 and 17 are tried, and so forth. The MINO location corresponding to a pattern is tested only after the other five locations have been found empty. Thus the search of patterns has a tree-like structure.

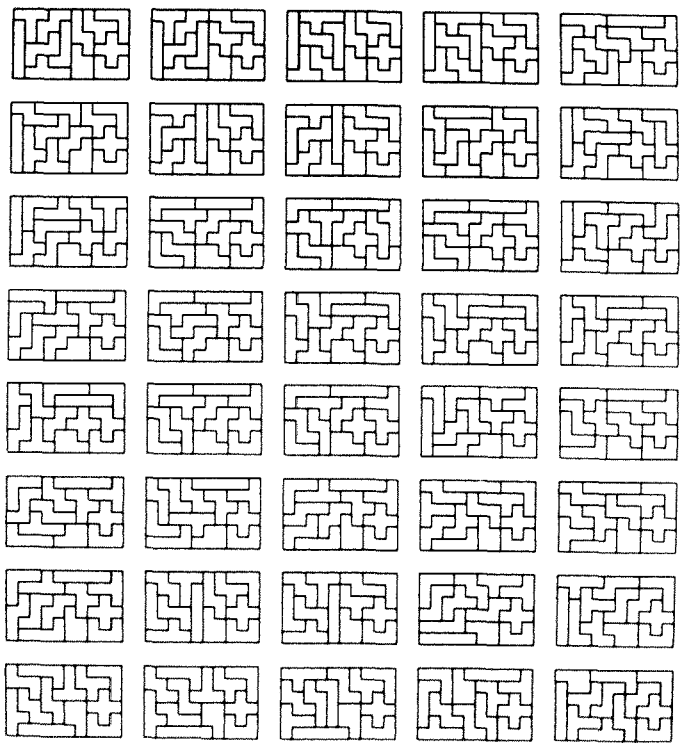


FIG. 2. Some of the solutions obtained by the program

How is this structure to be stored in the machine? It was decided to make a separate piece of coding for each location (relative to the lead) to be searched, these pieces to be nested and re-nested inside each other in accordance with the tree-like structure. This was simply achieved by the technique with macros which is the subject of this report. A study of the complete listing (see Appendix) will reveal how it works. (The arrays LEAD and PATT are essentially pushdown lists to keep track of the leads and patterns used.)

The basic piece of coding is the macro-instruction TEST. The programmer uses this instruction only once in his deck but with a long second argument which describes as compactly as possible the necessary tree-like structure (and thereby assigns the canonical order to the patterns). Within TEST, this argument, which is of the form of a list, is handed to an IRP which breaks it into its separate items and hands them in turn as argument pairs to TEST again and the process repeats. A counter (K) keeps track of the depth of nesting, and at the appropriate place causes the macro-instruction FTEST to be used in place of TEST an thus terminates the nesting. It may be remarked that in problems for which the nesting terminates at various depths (so that a counter cannot be used), the termination may be signaled by a unique symbol appropriately placed throughout the long second argument of the original TEST instruction, this symbol to be recognized by an IFF.

It should be noted that the behavior of the macro TEST is very similar to the operation of a list-processing language such as LISP. In fact, the second argument of TEST has a very LISP-like format. Moreover, the unique terminating symbol referred to above has the same function as the LISP NIL. However, unlike LISP, information cannot be handed "up;" it can only be handed "down."

Results

It may be of some interest, although not relevant to the main point, to know the results of the pentomino program. The program used in practice had a refinement by which solutions differing only by rotation or reflection from a solution already obtained were eliminated. (For example, it was demanded that pentomino 10 occur in the upper left quadrant of the 6×10 rectangular box.) Also, it sorted the solutions obtained into a canonical order and then output them by drawing on a CRT[1] (see Figure 2). It was found that there were 2 solutions for the 3×20 box, 368 for the 4×15 box, 1010 for the 5×12 box, 2339 for the 6×10 box, 2 for a pair of 5×6 boxes, and 65 for an 8×8 box with a 2×2 hole in the center. (Solutions differing by rotation or reflection are not multiply counted.) Total

[1] A sample page of this output is shown in Figure 2. The output routine made use of DD80 subroutines written by Alex Cecil.

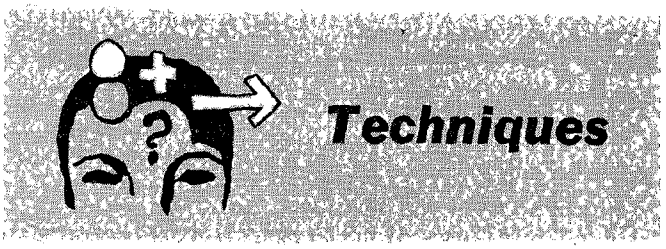
running time for the longest problem (6×10 box) is about 10 minutes.

A program using the same technique was written to solve the hexiamond (cf. *di*amond) problem. Hexiamonds consist of six contiguous equal equilateral triangles and are 12 in number. It was found, for example, that there is no way to place them in a rhomboidal box 3×12.

APPENDIX. Listing of the Program

```
*
        COUNT   81      THIS PROGRAM GENERATES 765 INSTRUCTIONS.
        TITLE           IT SOLVES THE PENTOMINO PROBLEM BY MACRO RECURSION.
*
TEST    MACRO   C,NEXT,XX CONSIDER SQUARE C (RELATIVE TO LEAD).
        ZET     ARENA-C,1 SEE WHETHER IT IS EMPTY.
        IFF     K-1,0,0   (FOR OTHER THAN LEAD (C NOT 0), DO NEXT ITEM.)
        TRA     XX+1      IF IT IS NOT EMPTY, TRY NEXT CHOICE FOR C.
        IFF     K-1,0,1   (FOR LEAD (C IS 0), DO NEXT ITEM.)
        TCO               DO AS DESCRIBED IN TCO MACRO.
        STQ     ARENA-C,1 INDICATE SQUARE C TO BE OCCUPIED.
K       SET     K+1       (ADVANCE COUNT OF DEPTH OF MACRO NESTING.)
        IRP     NEXT      (ITERATE ON ITEMS IN NEXT, WHICH IS A LIST.)
        IFF     K-5,0,0   (FOR DEPTH OF NESTING NOT 5, DO NEXT ITEM.)
        TEST    NEXT      DO AS IN TEST MACRO WITH ARGUMENTS FROM NEXT.
        IFF     K-5,0,1   (FOR DEPTH OF NESTING EQUAL 5, DO NEXT ITEM.)
        FTEST   NEXT      DO AS IN FTEST MACRO WITH ARGUMENTS FROM NEXT.
        IRP               (CEASE ITERATING ON ITEMS IN NEXT.)
K       SET     K-1       (REDUCE COUNT OF DEPTH OF MACRO NESTING.)
XX      STZ     ARENA-C,1 RESET SQUARE C TO EMPTY.
TEST    END               SQUARE C AND ITEMS IN NEXT NOW ALL CONSIDERED.
*
TCO     MACRO             LEAD SQUARE HAS BEEN TESTED.
        TXI     STEM,1,1  IF NOT EMPTY, TRY NEXT SQUARE AS LEAD.
        PXD     ,1        IF EMPTY, RECORD ITS NUMBER
        STD     LEAD,2      AT PROPER PLACE IN LEAD ARRAY, AND
TCO     END               CONTINUE.
*
FTEST   MACRO   C,M       CONSIDER SQUARE C AND PENTOMINO M.
        ZET     ARENA-C,1 SEE WHETHER SQUARE C IS EMPTY.
        TRA     *+9       IF NOT, TRY NEXT CHOICE FOR C. OTHERWISE,
        ZET     MINO-M    SEE WHETHER PENTOMINO M IS UNUSED.
        TRA     *+7       IF NOT, TRY NEXT CHOICE FOR C AND M. OTHERWISE,
        STQ     ARENA-C,1 INDICATE SQUARE C TO BE OCCUPIED.
        STQ     MIND-M    INDICATE PENTOMINO M TO BE USED.
        STL     PATT,2    SAVE CURRENT POINT IN SEARCH OF PATTERNS, AND
        TXI     ROOT,2,1  PROCEED TO TRY TO PLACE ANOTHER PATTERN.
        STZ     ARENA-C,1 RESET SQUARE C TO EMPTY.
        STZ     MINO-M    RESET PENTOMINO M TO UNUSED STATUS.
FTEST   END               SQUARE C AND PENTOMINO M HAVE BEEN CONSIDERED.
*
        CALL    INPUT,ARENA,MINO    INITIALIZE ARENA AND MINO ARRAYS.
        AXT     1,1           INDEX 1 = NO. OF CURRENT LEAD SQUARE.
        AXT     1,2           INDEX 2 = NO. OF PATTERNS ALREADY PLACED + 1.
        AXC     1,4           INDEX 4 = -1 ALWAYS.
K       SET     1             (K = DEPTH OF MACRO NESTING.)
*
ROOT    TXH     SOLVED,2,12 SOLUTION FOUND IF 12 PATTERNS ALREADY PLACED.
        PXD     ,2          SET DECREMENT
        XCA                   OF MQ TO NEW VALUE OF INDEX 2.
*
STEM    TEST    00((01((02($        BEGIN SEARCH OF PATTERNS.
        ETC     (03((04,02)(16,03)(17,11)(18,11)(19,03)))$
        ETC     (16((15,04)(17,05)(18,07)(32,08)))$
        ETC     (17((18,05)(33,06)))$
        ETC     (18((19,04)(34,08)))))(16($
        ETC     (15((14,04)(31,09)(17,05)(32,01)))$
        ETC     (17((18,05)(32,05)(33,05)))$
        ETC     (32((31,12)(33,07)(48,03)))))(17($
        ETC     (18((19,04)(33,01)(34,09)))$
        ETC     (33((32,07)(34,12)(49,03)))))(16((15($
        ETC     (14((13,03)(30,12)(31,01)(17,11)(32,06)))$
        ETC     (31((30,09)(17,01)(32,05)(47,04)))$
        ETC     (17((18,11)(32,10)(33,01)))$
        ETC     (32((33,01)(48,11)))))(17$
        ETC     (18((02,07)(19,03)(32,06)(33,01)(34,12)))$
        ETC     (32((31,01)(33,05)(48,11)))$
        ETC     (33((34,09)(49,04)))))(32($
        ETC     (31((30,08)(47,04)(33,06)(48,11)))$
        ETC     (33((34,08)(48,11)(49,04)))$
        ETC     (48((47,03)(49,03)(64,02)))))))
*
        TIX     STEMX,2,1   ALL PATTERNS TRIED WITH CURRENT LEAD, SO
        CALL    FINISH      IF NO PATTERNS ALREADY PLACED, EXIT, ELSE
STEMX   CAL     LEAD,2      RECALL PREVIOUS NUMBER OF LEAD SQUARE,
        PDX     ,1          RESTORE IT TO STATUS AS CURRENT LEAD,
        PXD     ,2          RESET DECREMENT
        XCA                   OF MQ TO REFLECT REDUCTION IN INDEX 2, AND
        TRA*    PATT,2      RETURN TO EXAMINATION OF PATTERNS.
*
SOLVED  CALL    OUTPUT,ARENA,MINO   SOLUTION OBTAINED, SO OUTPUT IT,
        AXC     1,4         RESET INDEX 4 TO -1, AND
        TXI     STEMX,2,-1  RETURN TO SEARCH FOR NEW 12TH PATTERN.
*
ARENA   BES     512     THIS ARRAY CORRESPONDS TO THE SQUARES.
MINO    BES     12      THIS ARRAY CORRESPONDS TO THE PENTOMINOS.
LEAD    BES     12      THIS IS A PUSHDOWN LIST OF LEAD SQUARES.
        OUP     1,12    (SINCE INDEX 4 = -1, INDIRECT ADDRESSING THRU THE
        PZE     ,4        PATT ARRAY TRANSFERS TO STORED LOCATION + 1.)
PATT    BES     0       THIS IS A PUSHDOWN LIST OF PATTERNS.
        END             SUBROUTINES REQUIRED ARE INPUT, OUTPUT, FINISH.
*
```

RECEIVED FEBRUARY, 1965



R. M. GRAHAM, Editor

A Fast Storage Allocator

KENNETH C. KNOWLTON
Bell Telephone Laboratories, Inc., Murray Hill, N. J.

A fast storage bookkeeping method is described which is particularly appropriate for list-structure operations and other situations involving many sizes of blocks that are fixed in size and location. This scheme, used in the LLLLLL or L[6] (Bell Telephone Laboratories Low-Level List Language), makes available blocks of computer registers in several different sizes: the smaller blocks are obtained by successively splitting larger ones in half, and the larger blocks are reconstituted if and when their parts are simultaneously free.

Introduction

A storage allocating and bookkeeping mechanism is described which makes available blocks of computer registers in several different sizes, the smaller blocks being obtained by successively splitting larger ones in half, and the larger blocks being reconstituted if and when their parts are simultaneously free. Each block contains a few bits of information indicating its size and whether it is presently free or in use; the rest of the block may be formatted as the user wishes. The operations involved in obtaining blocks from and returning them to the free storage lists are very fast, making this scheme particularly appropriate for list structure operations and for other situations involving many sizes of blocks which are fixed in size and location. This is in fact the storage bookkeeping method used in the Bell Telephone Laboratories Low-Level List Language[1] (LLLLLL or L[6], pronounced "L-sixth").

Free Storage Lists

The method to be described is illustrated by an example in which block sizes are one, two, four and eight machine words.[2] Figure 1a depicts the original layout of storage,

[1] KNOWLTON, K. C. A programmer's description of LLLLLL, Bell Telephone Laboratories low-level list language. Bell Telephone Laboratories, Inc., Mar. 1965. This language provides for blocks of the following lengths: 1, 2, 4, 8, 16, 32, 64 and 128 words.

[2] Upon completing this paper the author has learned that a similar storage handling system is used in SIMSCRIPT where block sizes are in fact one, two, four and eight words. For a description of the language, see MARKOWITZ, H. M., HAUSNER, B., and KARR, H. W. *SIMSCRIPT—A Simulation Programming Language*. Prentice-Hall, Englewood Cliffs, N. J., 1963.