An Improved Method for Finding Knight Covers

Frank Rubin
Master Software Corporation
59 DeGarmo Hills Road
Wappingers Falls, NY 12590

Abstract: A two-step approach to finding knight covers for an NxN chessboard eliminates the problem of detecting duplicate partial solutions. The time and storage needed to generate solutions is greatly reduced. The method can handle boards as large as 45x45 and has matched or beaten all previously known solutions for every board size tried.

Introduction:

A knight cover for a chessboard is a placement of knights on the board so that every square is either occupied or under attack from one of the knights. The Knight Covering Problem for a given board is to find a knight cover using the minimum number of knights.

An earlier article [5] in Ars Combinatoria described a computer method for generating knight covers. The method had a serious drawback. It generated large numbers of duplicate positions, resulting in a great deal of wasted effort. Detecting and eliminating the duplicate positions required a large amount of storage. Since storage is limited, only a fraction of the duplicates could be removed.

In this paper, a new 2-stage approach is described. The method does not generate any duplicate positions, so it does not require much storage, has far less wasted effort, and can find good covers for larger boards.

Terminology

A *neighbor* of a square A is a square that can be reached by one knight move from A. If B is a neighbor of A, then A is a neighbor of B. The *neighborhood* of square A, denoted N(A), consists of A and its neighbors. Each square may have from 2 to 8 neighbors, depending on how close it is to the edges and corners of the board. Any reference to the 8 neighbors of a square means that extended squares beyond the edge of the board are also being considered.

The program

The program operates in two distinct stages, setup and search. The setup stage chooses the order in which the squares of the chessboard will be examined. The search stage examines the board in the chosen order,

and decides which of the squares will be occupied.

The search stage will be described first, since that will explain the objective of the setup stage. The search stage is a simple recursion.

Here is a brief outline of the search algorithm. The heart of the algorithm is the recursive procedure Place which selects the square for knight K starting at position P, where P is an index into the order table.

Start with an empty board.
Place (1, 1).

Procedure Place (K, P)
Starting from position P, choose the next square A in order.
Repeat
  Try placing a knight on square A.
    If the total number of knights placed so far is more than the best so far, then placing a knight on A is rejected.
    Count the total number of squares that are covered by all of the knights placed so far. If this total is too low, then placing a knight on A is rejected.
    Otherwise, place knight K on square A. Recursively call Place (K+1, A+1). After returning from the recursion, remove the knight from A.
  Test whether square A can be left empty.
    Look at the neighbors of A. For each neighbor B, if all the squares in N(B) have been reached, and all of them are empty, then B can never be covered, so leaving square A unoccupied is rejected.
    Otherwise, leave A empty and continue sequentially to the next square, A+1.
Until the board is covered.

In summary, a partial cover will be abandoned if it does not cover enough squares, or if there is some square on the board that can never be covered. Such a square is called an *orphan*. A square becomes an orphan when its neighborhood has been left empty.

The steps of the program will be described in more detail in the following sections.

Representing the board

The board is represented as an NxN array of integer-valued marks centered inside an (N+4)x(N+4) array of marks. Initially, all squares

within the board are marked 0, meaning "empty," while the surrounding guard squares are marked 20, meaning "out of bounds." When a square is attacked, 1 is added to its mark.

When a square is occupied, 10 is added to its mark. Thus, an empty square attacked by k knights is marked k, while an occupied square attacked by k knights is marked k+10. So an empty square always has a mark from 0 to 8, an occupied square has a mark from 10 to 18, and a guard square has a mark from 20 to 28.

This means placing a knight on a square is simply a matter of adding 10 to its mark, and adding 1 to each of its 8 neighbors' marks. The two tiers of guard squares surrounding the board eliminate the need to check whether the neighbors are in bounds.

Choosing the order

The efficiency of the search depends upon how early unproductive placements can be rejected. The order in which the squares are examined determines how soon any bad placements can be recognized and abandoned.

Ideally, the order would be chosen dynamically according to where knights have already been placed, and how many more knights are required to reject any given square. However, this dynamic calculation takes a great deal of time, and was deemed impractical until the new two-stage algorithm was devised. It is more effective to choose the order in advance, and make a table of which squares to check for coverage.

The goal of the ordering is to have, at each step of the search, as many squares as possible where all of their neighbors have been considered. That way, orphans can be detected and eliminated as early as possible. To pick the order, each square is given a score according to how many of its neighbors have not yet been visited. The fewer unvisited neighbors, the higher its score, and the more valuable it is to choose its neighbors soon. The value of a square is the total of its score, plus the scores of its neighbors.

At each step the square with the highest value is chosen next. In case of ties, the square nearest the upper left corner is chosen next. This tie-breaking rule results in better orderings than random tie-breaking, or using the row-dominant tiebreaking rule.

Two different score functions were tried. For a square which has U unreached neighbors, the function 1/U proved to give a poorer order than the function 5^{8-U}. That is, bad placements were detected earlier, on average, using the scoring function 5^{8-U}.

Since squares near the corners of the boards have the highest scores initially, this procedure leads to orders that skip from corner to corner and work inwards from all 4 directions. That is a very inefficient process. It proved necessary to force the order into a single corner by placing 2 or 3 squares near one corner into the ordered list initially. Numerous combinations were tried. The best orders were obtained by choosing squares (1,2) and (2,2) as the first two on the list.

Once the order for visiting the squares has been chosen, it is a routine matter to find when all of the neighbors of each square have been considered. That is the point at which the square needs to be tested to see if it is an orphan. These are precomputed and stored in a table, so checking for orphans is very rapid.

Setting the bounds

The algorithm rejects any placement where the number of covered squares is too low. This is done by keeping a running total of the number of covered squares, and comparing this total to a table of bounds. The n-th element in the bounds table b(n) is the number of squares that must be covered after n knights have been placed. If the number of covered squares c(n) is less than b(n) the partial placement is rejected.

Setting these bounds suitably is critical for obtaining good placements in a reasonable amount of time. If the bounds in the table are too low, the program will run too long, perhaps for years. If the bounds in the table are too high, good knight covers will be missed. The number of bounds, that is, the size of the b table, is the number of knights that are expected to be needed. The number for the (N+1)x(N+1) board can be estimated from the number needed for the NxN board. For a large chessboard the number of knights is high, and it takes many runs of the program to adjust all the bounds.

In the earlier paper [5], it is briefly mentioned that it is possible for the program to adjust its own bounds. This process is called *training*. The training process has now become fairly sophisticated, so that it is possible to let the program set its own bounds without intervention.

During each run the program counts the number of placements p(n) of n knights that were generated. The counts p(n) are compared to a desired range L to U set at the start of the first run. Typically, L=U/10 for large boards. At the end of each run if p(n)<L the bound is decreased, and if p(n)>U, the bound is increased. The amount of the increase or decrease is proportional to log(L/p(n)) or log(p(n)/U), respectively.

The adjustment process operates in several distinct phases. For small

n, p(n) is correspondingly small. During this early phase the bound b(n) is decreased whenever p(n)<2L. This phase ends once a count p(n)>U is encountered. In the middle phase, b(n) is decreased whenever p(n)<L, and increased whenever p(n)>U. When p(n)>U, subsequent bounds b(n+i) are also increased as long as p(n+i)>U/2.

The middle phase ends when a count p(n)=0 is found. The corresponding b(n) is decreased by 3. The remaining bounds are set using biased interpolation. If E is the expected number of knights, then b(n) is reset to b(n-1)+[((b(E)-b(n-1))/(E-n+1))+1.25] where [x] is the integer part of x. The reason for the bias is that knights in the middle of the board can cover more squares than knights at the edges or in the corners. The final knights to be placed will be in the corner and along the edges opposite the starting corner.

After each run the program increases L and U by a factor of 10/9 and immediately proceeds to the next run. The program can run continuously until the researcher decides that no further improvement in the knight cover is likely within a reasonable additional running time.

Results

The program has matched the previous best covers [5] for board sizes from 10x10 through 22x22. For all board sizes 23x23 and larger that have been tried the program has matched or improved on the previous best covers, as follows:

Board size	23	24	25	26	27	28	29	30	35	40	45
Fisher [1]	84	88	97	--	--	--	--	--	--	--	--
Lemaire [3]	84	88	98	106	--	120	126	138	--	235	292
Rubin [5]	83	88	96	102	--	--	--	--	--	--	--
Rubin (new)	82	88	96	102	111	119	126	136	182	233	291

The program also beat the simulated annealing solutions of Jackson and Pargas [2] for all boards from 16x16 through 20x20 by 2 to 3 knights.

The original plan for this research was to try all square board sizes up to 30x30. However, after the Lemaire paper [3] was published, a decision was made to try some larger boards in the hope of finding a best cover using Monier's pattern [3]. Monier's pattern is one of the two densest known covering patterns [6]. It is very effective for covering the middle of the board, but it is difficult to cover the remaining squares near the edges. No sizeable area using Monier's pattern had ever been observed in any of the covers generated for boards up through 30x30.

(A Monier-pattern solution for the 18x18 board was discovered by Morgenstern [4] as this paper was being prepared for publication.)

Lemaire has predicted that Monier's pattern will dominate for boards larger than 130x130.

Board sizes of 35x35, 40x40 and 45x45 were tried. Each board was covered twice, once running the covering program without constraints and once forcing a Monier pattern in the center, with the following results:

	35x35	40x40	45x45
Unconstrained	182	233	292
Monier	183	234	291

These results suggest a crossover between the 40x40 and the 45x45 boards. Given the irregular jumps that covering patterns take, however, it is possible that there are boards smaller than 40x40 where Monier is best, and a few boards larger than 45x45 where it is not.

What's next?

The question naturally arises, are these covers optimal? Which boards are most likely to have smaller covers? Optimality has been proven [3] for all boards up to 15x15. The following table can help detect which larger boards might be improved. It shows the density, or mean number of squares covered by each knight for various board sizes 1x1 through 45x45.

Anywhere the density of the (N+1)x(N+1) board is significantly lower than the density of the NxN board is a likely place for improvement. It appears that 27x27 and 30x30 are the most likely.

Table 1 The best coverings now known. N=board size, K=number of knights required, D=density=n^2 /K.

N	K	D	N	K	D	N	K	D
1	1	1	12	24	6	23	82	6.451
2	4	1	13	28	6.036	24	88	6.545
3	4	2.25	14	32	6.125	25	96	6.510
4	4	4	15	36	6.25	26	102	6.627
5	5	5	16	40	6.4	27	111	6.568
6	8	4.5	17	46	6.283	28	119	6.588
7	10	4.9	18	52	6.231	29	126	6.675
8	12	5.333	19	57	6.333	30	136	6.618
9	14	5.786	20	62	6.452	35	182	6.731
10	16	6.25	21	68	6.485	40	233	6.867
11	21	5.762	22	75	6.452	45	291	6.959

Improved Covers

For large boards the number of minimal covers tends to be very large, so it is not yet feasible to list and categorize all of them. One cover is given for each square board size where the new method beat the previous best cover. Both unconstrained and Monier covers are shown for board sizes 35x35, 40x40 and 45x45.

References

[1] David C. Fisher, "On the NxN Knight Cover Problem," Ars Combinatoria 69(2003) pp 255-274.

[2] Anderson H. Jackson, Roy P. Pargas, "Solutions to the NxN Knights Covering Problem," J. Recr. Math. 23(1991), pp 255-267.

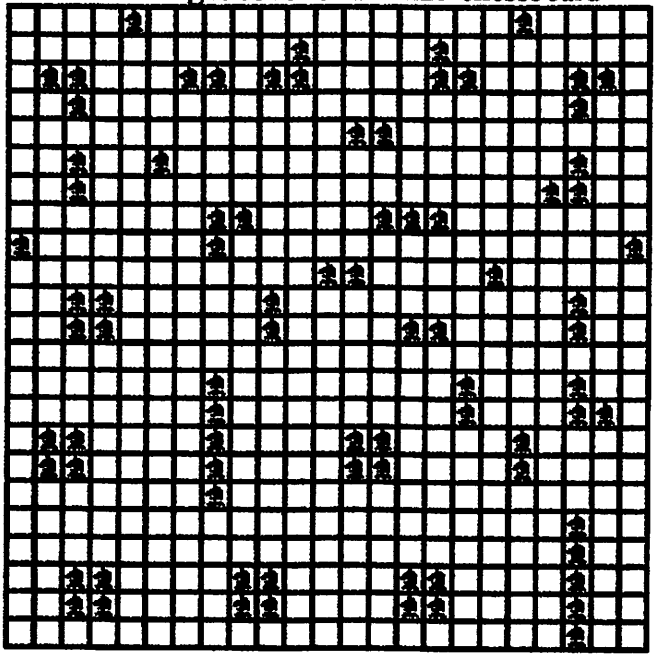
[3] Bernard Lemaire, "Knights Covers on NxN Chessboards," J. Recr. Math. 31(2003), pp 87-99.

[4] Lee Morgenstern, June 2004, www.contestcen.com/kn18.htm.

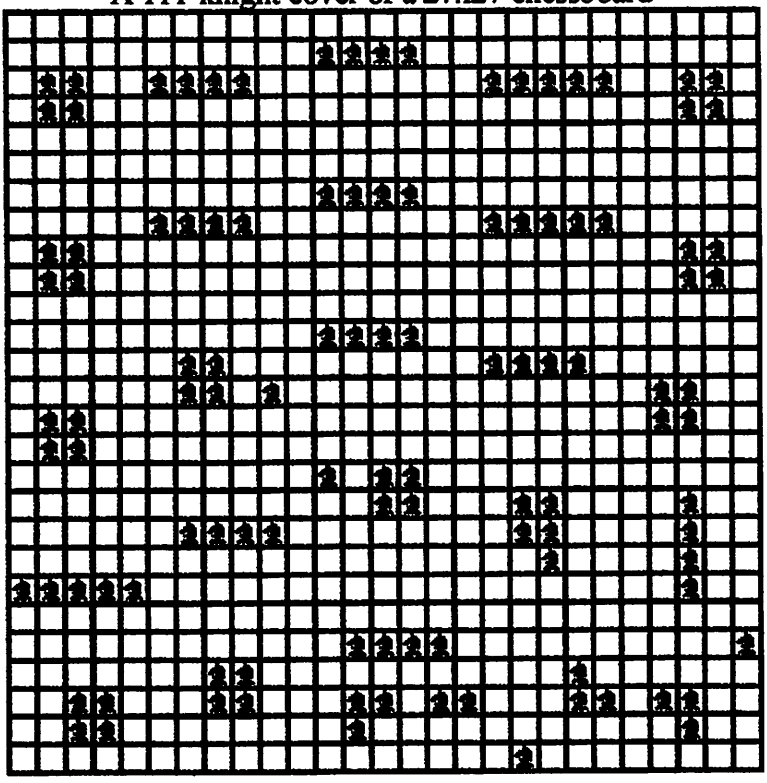
[5] Frank Rubin, "Improved Knight Coverings," Ars Combinatoria 69(2003) pp 185-196.

[6] Frank Rubin, "An Efficient Knight Covering Pattern," J. Recr. Math. to appear.

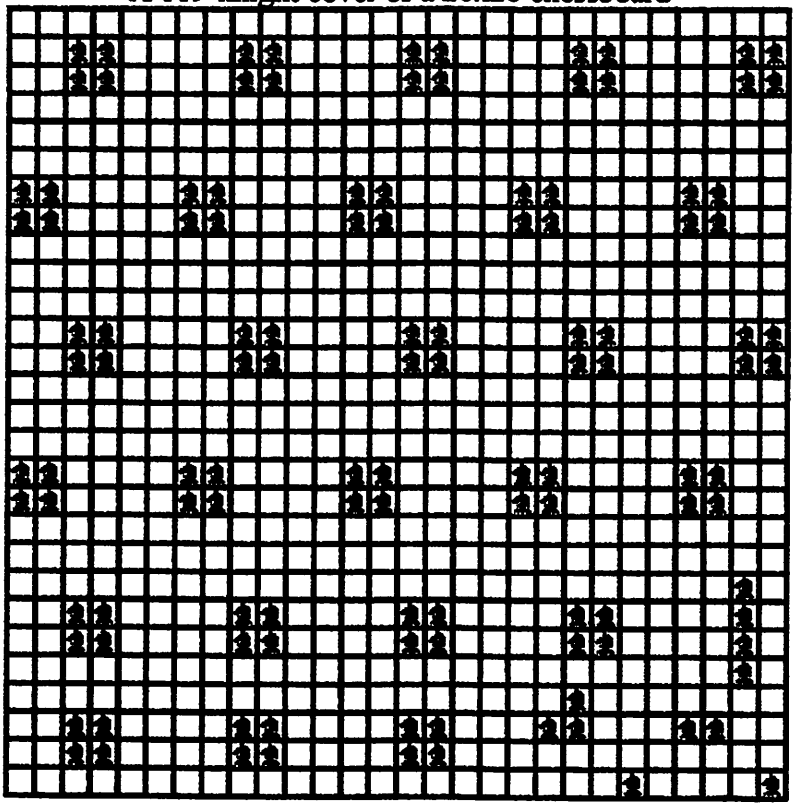
An 82-knight cover of a 23x23 chessboard



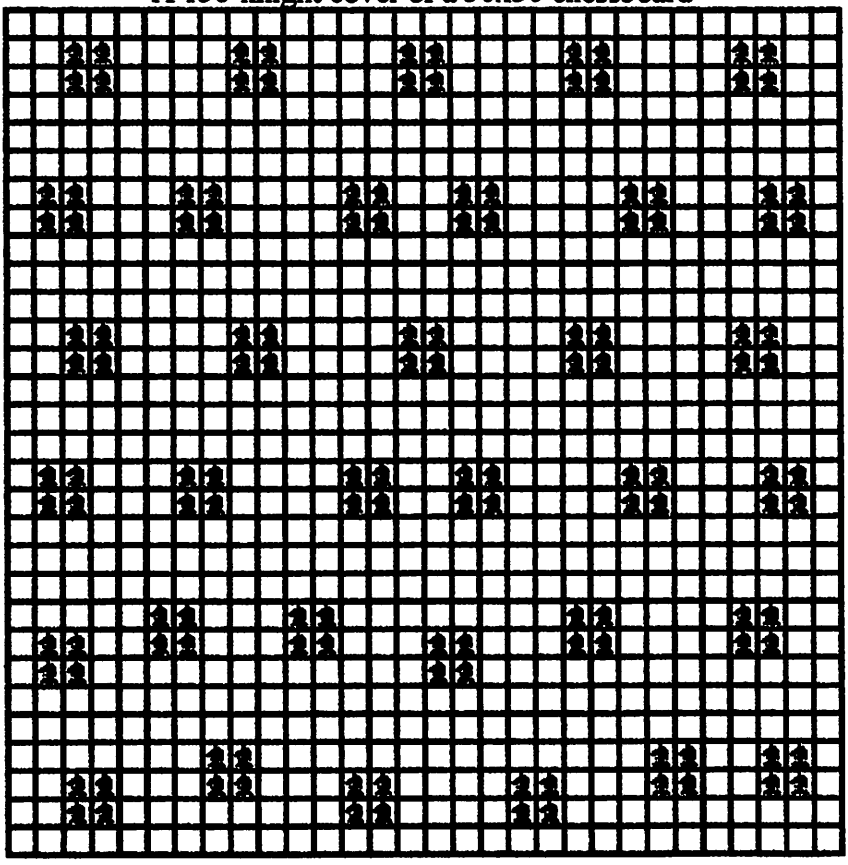
A 111-knight cover of a 27x27 chessboard



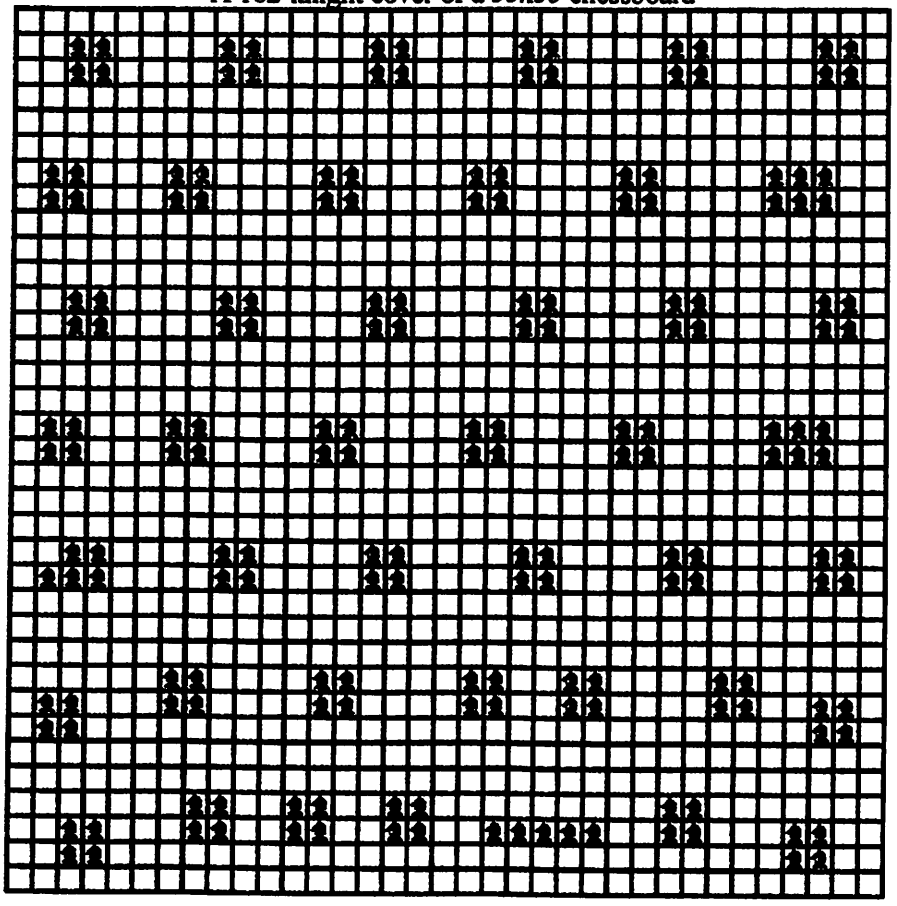
A 119-knight cover of a 28x28 chessboard



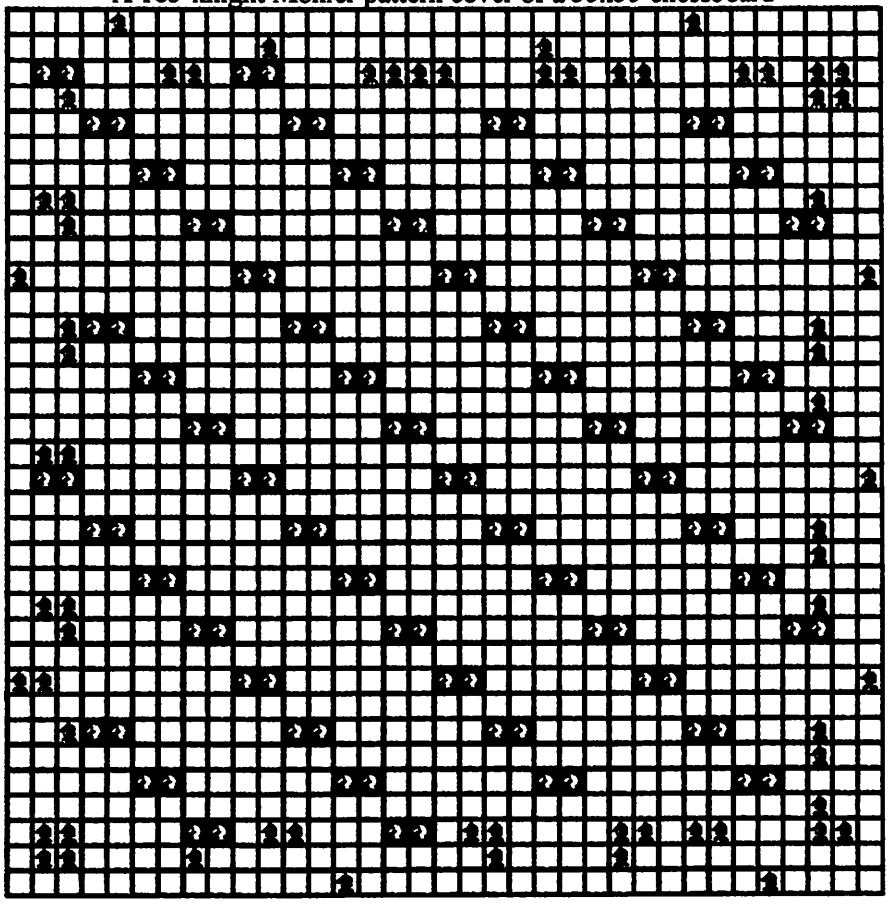
A 136-knight cover of a 30x30 chessboard



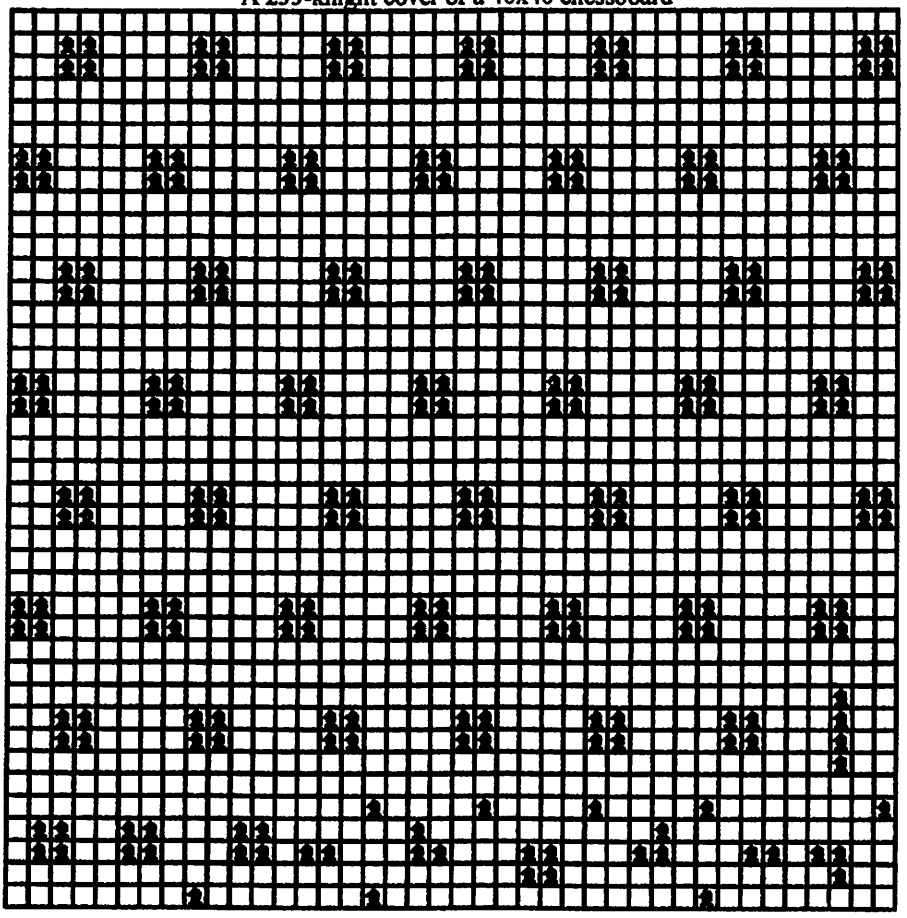
A 182-knight cover of a 35x35 chessboard



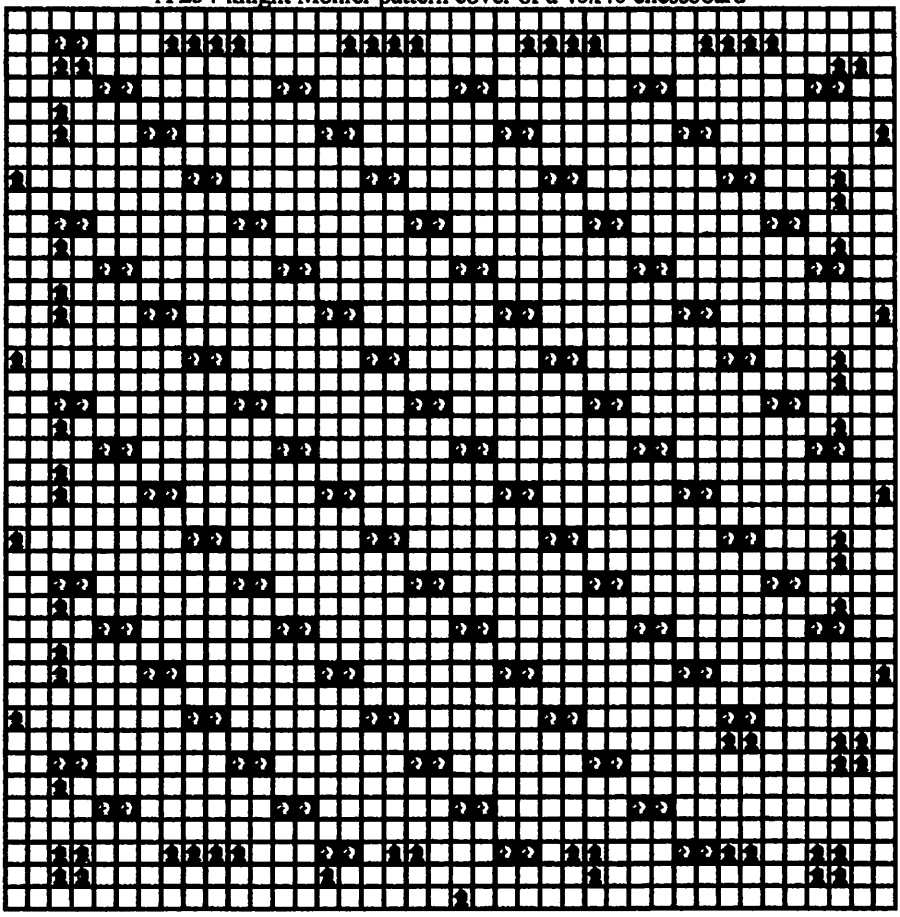
A 183-knight Monier pattern cover of a 35x35 chessboard



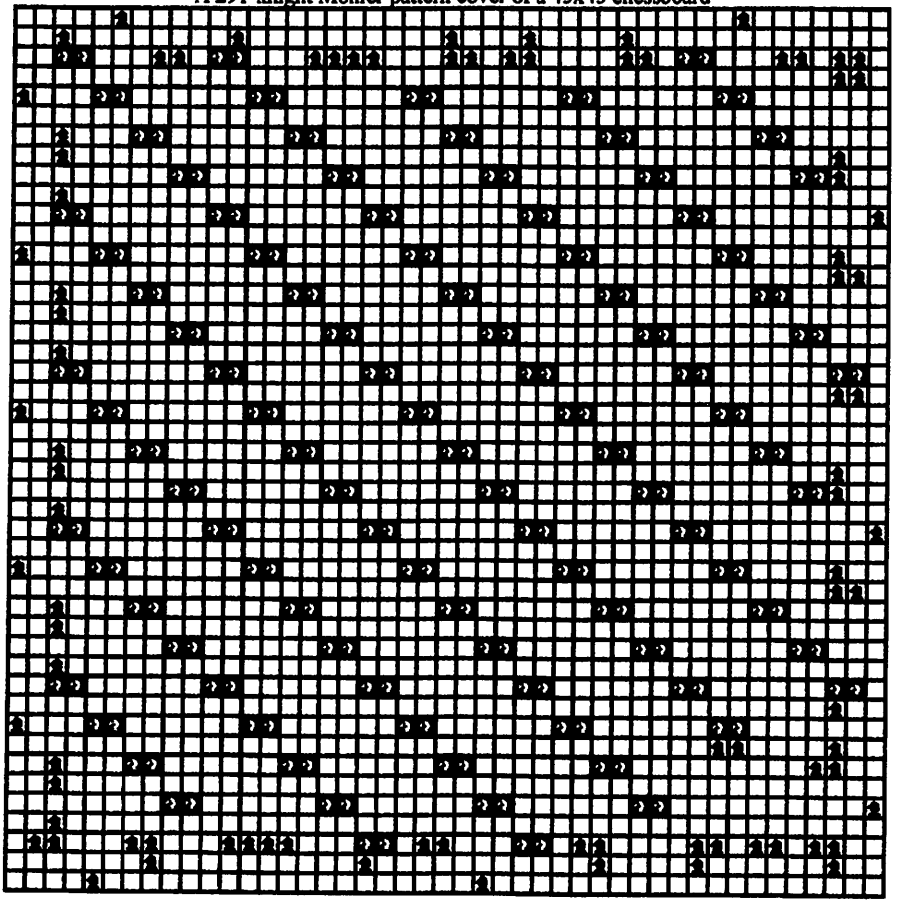
A 233-knight cover of a 40x40 chessboard



A 234-knight Monier pattern cover of a 40x40 chessboard



A 291-knight Monier pattern cover of a 45x45 chessboard



A 292-knight cover of a 45x45 chessboard