Math 381: Assignment 2: Knights Problem

Ella Kim

October 15 2021

1 Introduction

Recently, I found chess boards of various sizes and knight pieces of 3 different colors (black, white, and green). Along with the chess boards and knights, I want to create a game to play with my two other friends. I also want to analyze and see how many knights max I can have of each team on the various chess boards.

Let b represent the black team (knights), w represent the white team (knights), and g represent the green team (knights). To identify eachsquare on the chess board, let integer i represent the ith row and integer j represent the jth column number of each particular spot. In other words, let $b_{ij}, w_{ij}, and g_{ij}$ be values of only 0 or 1 (binary): empty or has the corresponding color's knight on the ith row jth column square.

I plan to maximize the number of black knights there are on the chess board (which in turn is maximizing the number of other knights for the white and green teams, explained in the constraints after this): Objective Function (LP):

$$max \sum_{i=1}^{m} \sum_{j=1}^{n} b_{ij}$$

* Full LP in input file shown in section 2

As this is a game, I want to be fair for each team and make sure each team (i.e. each color) starts off with the same number of knights. So the total number of knights for each team/color should be equal:

$$\sum_{i=1}^{m} \sum_{j=1}^{n} b_{ij} = \sum_{i=1}^{m} \sum_{j=1}^{n} w_{ij} = \sum_{i=1}^{m} \sum_{j=1}^{n} g_{ij}$$

This constraint is binding, as, while the total can be any integer value between 1 to the total number of squares on a n by m chess board / 3, these totals must equal one another.

Following the rules of normal chess, I want to make sure that there is no more than one knight on one square at a time. For every i row and j column spot on the chess board, the total amount of knights on each should be either 0 or 1.

$$b_{ij} + w_{ij} + g_{ij} <= 1$$

This constraint is non-binding, as there can always be a square that has or does not have a knight on it, so the sum can be either 0 or 1.

It would not be fair for the very first move for any team to be a move to attack another team's knights, so I need to ensure that, all the starting positions do not have a different color team being able to attack on the first move. For each square, I need to ensure that there is either a black knight there, or that there is not because a white or green knight is on a square that can attack the square I am testing. I also need to make sure that wherever there is a white knight, there is no green knight in another square where they can attack (or that there is no white knight at a particular square because a green knight can attack/be attacked. By ensuring that there is no white or green knight attacking a black knight and also ensuring no white knight is being attacked by a green night, this allows all three teams to avoid starting in an advantageous spot; so for each square position on a board at ith and jth square and each rth row and sth column square that can attack the ith and jth square:

$$b_{ij} + w_{rs} + g_{rs} <= 1 \; b_{ij}, \text{ and } w_{ij} + g_{rs} <= 1$$

Similar to the constraint of having only one knight on a square at a time, this constraint is non-binding, as there does not have to be a knight on either of the two positions being summed, so the sum(s) can be 0 or 1.

I also add another constraint for speeding up the run-time of the different chess boards, where no green or white knight will ever be in the 1st row 1st column square. This could have been any corner of the chess board, but, as seen later with my Python code for creating my LP and constraints, I start my for loops from the 1st row 1st column, and would be the first square in the constraints above that is analyzed. To further the explanation, this would mean that the first square would either have a black knight or no knight. If the former was true, and because there is no advantage of where the knights are, the placements for black knights can easily be switched by white knights or green knights and will hold the same effect:

$$w_{11} + g_{11} = 0$$

This constraint is binding, as this only allows these two values (white knight and green knight in the 1st row 1st column square) to be 0 to hold true.

* * *

The objective function and the constraints all come together to my full LP, where it uses the same variables (b, w, and g as black, white, and green knights in the integer i row and integer j column):

Maximize:

$$b_{11} + b_{12} + b_{13} + ... + b_{mn-2} + b_{mn-1} + b_{mn}$$

(all possible black knight positions on a m by n chess board, so mn different black knight positions)

Which is subject to:

$$b_{11} + w_{11} + g_{11} <= 1$$
$$.$$
$$.$$
$$b_{mn} + w_{mn} + g_{mn} <= 1$$

(mn of the same type; ensure that there is only one knight on a single square)

$$b_{11} + w_{23} + g_{23} <= 1$$
$$.$$
$$.$$
$$b_{mn} + w_{rs} + g_{rs} <= 1$$

(mn of the same type; ensure that there is either a black knight or a white/green knight in an attacking pair, where int row r and int row s refer to a possible attacking pair of squares)

$$w_{11} + g_{23} <= 1$$
$$.$$
$$.$$
$$w_{mn} + g_{rs} <= 1$$

(mn of the same type; ensure that there is either a white knight or a green knight in an attacking pair of squares, where int row r and int row s refer to a possible attacking pair of squares)

$$b_{11} + b_{12} + b_{13} + ... + b_{mn} = w_{11} + ... + w_{mn-2} + w_{mn-1} + w_{mn}$$

$$w_{11} + w_{12} + b_{13} + ... + w_{mn} = g_{11} + ... + g_{mn-2} + g_{mn-1} + g_{mn}$$

(mn of the black, white, and green variables; ensure that the totals of each knight color/team are equal to one another)

$$w_{11} + g_{11} = 0$$

(ensure that there is no white or green knight in the very first square (1st row, first column) of any chess board)

$$b_{11}, b_{22}, ..., b_{mn}, w_{11}, ..., w_{mn}, g_{11}, ...g_{mn} \in \{0, 1\}$$

(mn of black, white, and green knight variables; ensure that all variables are binary)

2 Experiments: Square Chess Boards

To start experimenting, I first write out the LP to find the max number of knights for all three teams as well as their positions subject to the constraints explained in Section 1: Introduction.

Python code to solve LP in lpsolve:

```
# open pre-made txt file to write in
knight_input = open("knight1.txt", 'w')
# start LP line for maximizing value
knight_input.write("max: ")

# number rows
m = 3
# number cols
n = 3

# make function to get objective function for black knights
def getObj(m, n):
    # go though each square/spot on chess board
    for i in range(1, m+1):
        for j in range(1, n+1):
            # write out all possible black knight spaces
            knight_input.write("+b_" + str(i) + "_" + str(j))
    # end objective function
    knight_input.write("; \n")

# make function to get constraint for different colors not attacking each other
def getColorConst(m, n):
    # go though each square/spot on chess board
    for i in range(1, m+1):
        for j in range(1, n+1):
            # for style, write out all the possible attacking for i_jth spot
            # for conditions with different colors
            getAttacking(i,j,m,n,"b")
            getAttacking(i,j,m,n,"w")


# make function to get constraint for only one chess piece on a board
def getOneConst(m, n):
    # go though each square/spot on chess board
    for i in range(1, m+1):
        for j in range(1, n+1):
            # write out conditions to ensure each square/spot only has one or 0 knights
            knight_input.write("+b_" + str(i) + "_" + str(j))
```

```
            knight_input.write("+w_" + str(i) + "_" + str(j))
            knight_input.write("+g_" + str(i) + "_" + str(j))
            # sum must be <= 1 for condition to hold, end condtion after each square/spot
            knight_input.write(" <= 1;  \n")

# return a list of possible places to get attacked/attack
def getAttacking(i, j, m, n, color):
    # ensure spot attacking from is within the board dimensions
    if (i-1 > 0) and (j-2 > 0):
        # start out by adding black to spot being attacked
        knight_input.write("+" + color + "_" + str(i) + "_" + str(j))
        # write out white or green horse in spot attacking from,
        # depending on whether color and square testing is black
        # to white and green or white to green
        if color == "b":
            knight_input.write("+w_" + str(i-1) + "_" + str(j-2))
        knight_input.write("+g_" + str(i-1) + "_" + str(j-2))
        # end condition for that one spot testing for incoming attacks
        knight_input.write(" <= " + str(1) + "; \n")
    # same procedure, there are 8 max possible spots 1 square could
    # be attacked from, if there are no squares that can attack,
    # then automatically I add a black knight to max black
    if (i+1 <= m) and (j-2 > 0):
        knight_input.write("+" + color + "_" + str(i) + "_" + str(j))
        if color == "b":
            knight_input.write("+w_" + str(i+1) + "_" + str(j-2))
        knight_input.write("+g_" + str(i+1) + "_" + str(j-2))
        knight_input.write(" <= " + str(1) + "; \n")
    if (i-1 > 0) and (j+2 <= n):
        knight_input.write("+" + color + "_" + str(i) + "_" + str(j))
        if color == "b":
            knight_input.write("+w_" + str(i-1) + "_" + str(j+2))
        knight_input.write("+g_" + str(i-1) + "_" + str(j+2))
        knight_input.write(" <= " + str(1) + "; \n")
    if (i+1 <= m) and (j+2 <=n):
        knight_input.write("+" + color + "_" + str(i) + "_" + str(j))
        if color == "b":
            knight_input.write("+w_" + str(i+1) + "_" + str(j+2))
        knight_input.write("+g_" + str(i+1) + "_" + str(j+2))
        knight_input.write(" <= " + str(1) + "; \n")
    if (i-2 > 0) and (j-1 > 0):
        knight_input.write("+" + color + "_" + str(i) + "_" + str(j))
        if color == "b":
            knight_input.write("+w_" + str(i-2) + "_" + str(j-1))
        knight_input.write("+g_" + str(i-2) + "_" + str(j-1))
        knight_input.write(" <= " + str(1) + "; \n")
```

```
    if (i+2 <= m) and (j-1 > 0):
        knight_input.write("+" + color + "_" + str(i) + "_" + str(j))
        if color == "b":
            knight_input.write("+w_" + str(i+2) + "_" + str(j-1))
        knight_input.write("+g_" + str(i+2) + "_" + str(j-1))
        knight_input.write(" <= " + str(1) + "; \n")
    if (i-2 > 0) and (j+1 <= n):
        knight_input.write("+" + color + "_" + str(i) + "_" + str(j))
        if color == "b":
            knight_input.write("+w_" + str(i-2) + "_" + str(j+1))
        knight_input.write("+g_" + str(i-2) + "_" + str(j+1))
        knight_input.write(" <= " + str(1) + "; \n")
    if (i+2 <= m) and (j+1 <= n):
        knight_input.write("+" + color + "_" + str(i) + "_" + str(j))
        if color == "b":
            knight_input.write("+w_" + str(i+2) + "_" + str(j+1))
        knight_input.write("+g_" + str(i+2) + "_" + str(j+1))
        knight_input.write(" <= " + str(1) + "; \n")

# make function to get constraint for equal number of black, white, and green
def getEqCol(m, n):
    # go though each square/spot on chess board
    for i in range(1, m+1):
        for j in range(1, n+1):
            # write out corresponding b_i_j spot/square
            knight_input.write("+b_" + str(i) + "_" + str(j))
    # setting condition that sum of black knights is = to whites
    knight_input.write("=")
    # go though each square/spot on chess board
    for i in range(1, m+1):
        for j in range(1, n+1):
            # write out all sum of w_i_j spots/squares
            knight_input.write("+w_" + str(i) + "_" + str(j))
    # end condition
    knight_input.write("; \n")
    # go though each square/spot on chess board
    for i in range(1, m+1):
        for j in range(1, n+1):
            # white again to make sum equal to green (therefore
            # green is also = to sum of black knights)
            knight_input.write("+w_" + str(i) + "_" + str(j))
    # setting up = condition of sums
    knight_input.write("=")
    # go though each square/spot on chess board
    for i in range(1, m+1):
        for j in range(1, n+1):
```

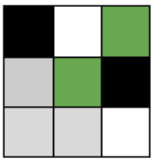
```
            knight_input.write("+g_" + str(i) + "_" + str(j))
    knight_input.write("; \n")

# set all values as binary
def getBin(m, n):
    knight_input.write("bin b_1_1")
    # go though each square/spot on chess board
    for i in range(1, m+1):
        for j in range(1, n+1):
            # condition because I already wrote b_1_1 above for foramtting
            if (i != 1) and (j != 1):
                knight_input.write(",b_" + str(i) + "_" + str(j))
    # go though each square/spot on chess board
    for i in range(1, m+1):
        for j in range(1, n+1):
            knight_input.write(",w_" + str(i) + "_" + str(j))
    # go though each square/spot on chess board
    for i in range(1, m+1):
        for j in range(1, n+1):
            knight_input.write(",g_" + str(i) + "_" + str(j))
    knight_input.write(";")

# now call all the functions above to write out the .txt input file
getObj(m,n)
getOneConst(m,n)
getColorConst(m,n)
getEqCol(m,n)
# add condition to make calculations faster and condition that realized had
knight_input.write("w_1_1+g_1_1=0;  \n")
getBin(m,n)

# to ensure correctness/sanity check
knight_input = open("knight1.txt", 'r')
```

* * *

I use the same code for each chess board, and only change the integer n row and integer m column for the size of the chess board. The input file is similar to the summary of the LP in Section 1: Introduction, and is shown for the 5 by 5 chess board later in this section, as well as the output from lpsolve.

As knights move 1 diagonal and then 2 horizontally or vertically, the smallest square board I can start with is a 3 by 3 board. Below is the last portion of the output from lpsolve, where the total run-time in seconds is written, as well as the i row and j column that have knights on the corresponding square.

Output visualization (3 by 3):

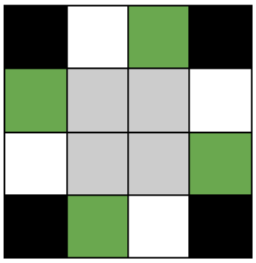


3 by 3 chess board final max layout

When I observe these visually, I have to remind myself that these are not the concrete answers, in that this answer follows the LP I have set up and follows the same constraints (besides the efficiency constraint for no white or green knight in the first square) when the chess board is rotated 90, 180, and 270 degrees. It also solves the same LP and constraints when the black, white, and green knights are interchanged in their positions (i.e. green can be switched with black knights and would still satisfy the constraints). I conclude that the max possible number of knights each team can have is 2

Like mentioned before the Python code, I use the same script for a 4 by 4 chess board, except for the m and n variables as the corresponding row and column values.

Output Visualization (4 by 4):



4 by 4 chess board final max layout

The resulting chess board for the 4 by 4 is particularly interesting, as, by the pattern and the ability to switch a team's position, the same conditions would be met (besides the efficiency constraint for no white or green knight in the first square) when you rotate the outer ring counter-clockwise/clockwise one to three spaces. I conclude that there are a max 4 number of knights each team can have.

The final square chess board I have results for is the 5 by 5 chess board (reasons are explained after the 5 by 5 result and analysis). As mentioned previously, I now show the lpsolve input file and the output as reference to how I computed and got the visualizations/results for the 5 by 5 chess board:

* * *

lpsolve input file:

max: $+b_{11} + b_{12} + b_{13} + ... + b_{53} + b_{54} + b_{55}$;
(25 black knight variables; all possible black knight positions on 5 by 5 chess board)
$b_{11} + w_{11} + g_{11} <= 1$;
.
.
$b_{55} + w_{55} + g_{55} <= 1$;
(25 of the same type; ensure that there is only one knight on a single square)
$b_{11} + w_{23} + g_{23} <= 1$;
.
.
$+b_{55} + w_{34} + g_{34} <= 1$;
(90 of the same type; ensure that there is either a black knight or a white/green knight in an attacking pair)
$+w_{11} + g_{23} <= 1$;
.
.
$+w_{55} + g_{34} <= 1$;
(90 of the same type; ensure that there is either a white knight or a green knight in an attacking pair of squares)
$+b_{11} + b_{12} + b_{13} + ... + b_{mn} = w_{11} + ... + w_{53} + w_{54} + w_{55}$;
$+w_{11} + w_{12} + b_{13} + ... + w_{mn} = g_{11} + ... + g_{53} + g_{54} + g_{55}$;
(25 of the black, white, and green variables each; ensure that the totals of each knight color/team are equal to one another)
$+w_{11} + g_{11} = 0$
(ensure that there is no white or green knight in the very first square (1st row, first column) of any chess board)
$b_{11}, b_{22}, ..., b_{55}, w_{11}, ..., w_{55}, g_{11}, ...g_{55} \in \{0, 1\}$
(25 of black, white, and green knight variables; ensure that all variables are binary)

* * *

Inputting the above file into lpsolve resulted in the output below (only the last portion of the output is displayed here, as the only important information is the resulting positions of knights and the total run-time).

* * *

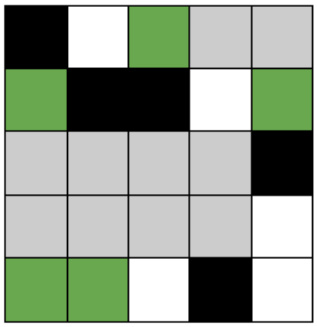
Output from lpsolve for 5 by 5:

```
Time to load data was 0.001 seconds, presolve used 0.007 seconds,
 ... 40.650 seconds in simplex solver, in total 40.658 seconds.
```

```
Value of objective function: 5.00000000

Actual values of the variables:
b_1_1                          1
b_2_2                          1
b_2_3                          1
b_3_5                          1
b_5_4                          1
w_1_2                          1
g_1_3                          1
g_2_1                          1
w_2_4                          1
g_2_5                          1
w_4_5                          1
g_5_1                          1
w_5_2                          1
w_5_3                          1
g_5_5                          1
```

* * *

Output visualized (5 by 5):



5 by 5 chess board final max layout

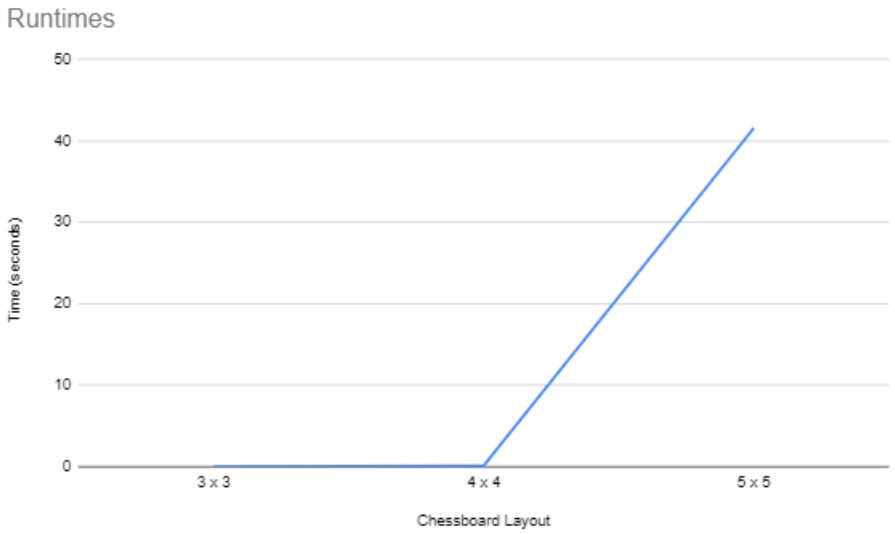
Observing the output, there does not seem to be a clear of a pattern as the 4 by 4 chess board. I conclude that the max number of knights each team can have on this board is 5.

When attempting to solve for a 6 by 6 chess board, however, the run time became too long (had to terminate my command, as it was taking more than 10 minutes to solve). I now switch to analyzing non-square chess boards.

Here, I compiled the run-time of the square chessboard into a chart (where x is int x row/column):

3 x 3	0.012
4 x 4	0.104
5 x 5	41.509

x by x chess board run-times



x by x chess board run-times

As seen in the table and the graph (for visual analysis), there is a extremely sharp intake in run-time from the 4 by 4 square to the 5 by 5 square, and I was not able to compute an output for a 6 by 6 under 10 minutes, which confirms that the run-time continues to exponentially grow. As of now, the only explanation I can guess on is that the increase in row and column by one greatly increases the possibles, resulting in a long run-time.

* * *

I have also compiled the max number of knights for each of the square chessboard into a chart (where x is int x row/column):

3 x 3	2
4 x 4	4
5 x 5	5

x by x chess board run-times

Although I cannot say much from 3 different chess boards, as of now there is a gradual increase in the max number of knights each team can have on a chess board (for 4 by 4 and 5 by 5, the max is equal to the dimension).

3 Experiments: Non-square Chess Boards

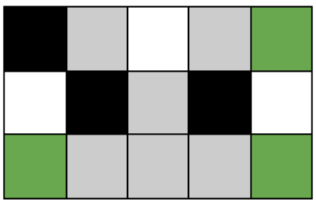
Like the changes I made to the Python code for each of the square chess boards, I only changed the integer m and integer n values to the m row and n column dimension of the non-square chess boards I am experimenting with.

Output visualization (3 by 4):

3 by 4 chess board final max layout

With the output given, this 3 by 4 visual reveals that the result is the same as the 3 by 3 matrix, only in a different colors in the positions, but we already know that which team is in what positions does not matter. I therefore conclude that the max knights each team can have on this board can have is 2.

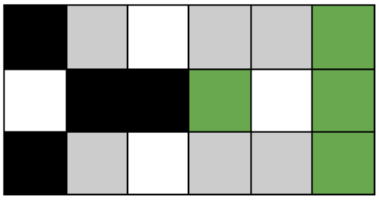
Output visualization (3 by 5):



3 by 5 chess board final max layout

With the output given, this 3 by 5 visual initially made me believe this was a symmetrical board through the center, but the off value of the 5 made me realize it is not. I conclude that there is not a significant pattern and that the max knights this board can have is 3.

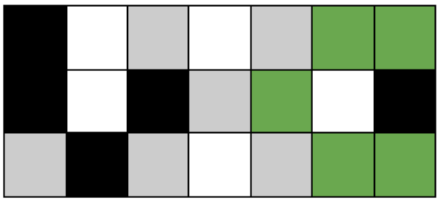
Output visualization (3 by 6):



3 by 6 chess board final max layout

With the output given, this 3 by 6 visual also initially made me believe this was a symmetrical board through the center, but the nature of a knight's move makes it impossible to be. I conclude that there is also not a significant pattern in this chess board and that the max knights this board can have is 4.

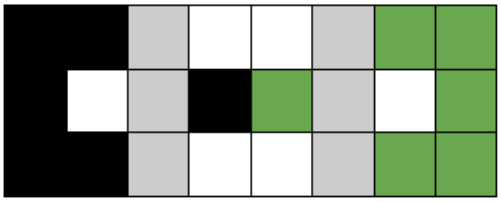
Output visualization (3 by 7):



3 by 7 chess board final max layout

With the output given, this 3 by 7 visual again is very similar but not a symmetric pattern. I now realize from the past few boards that there is always the shape of the white knights to some extent in the larger boards, with the black and green on the left and right edges and center. I conclude that the max knights this board can have is 5.

Output visualization (3 by 8):

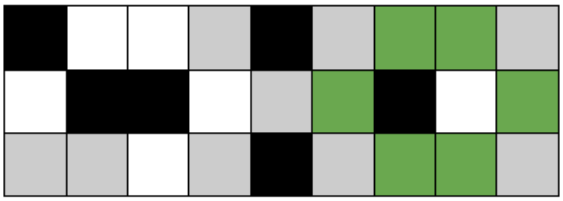


3 by 8 chess board final max layout

With the output given, this 3 by 8 visual confirms the shape of the white knights but with double in the center, with the black and green on the left and right edges and center. The chess board is also finally symmetrical down the middle I conclude that there is also not a significant pattern in this chess board and

that the max knights this board can have is 6.

Output visualization (3 by 9):



3 by 9 chess board final max layout

With the output given, this 3 by 9 visual breaks the pattern of the white knights being in the center and instead has a similar "almost" symmetrical pattern. I conclude that the max knights this board can have is 6, which is odd to me as, up until now, there has been a one increase in max knights.

Here, I have included a table of the 3 by int x column chess board and its max number of knights per team. Again it seems interesting that there is a plateau in the max number of knights from 8 to 9 in column count.

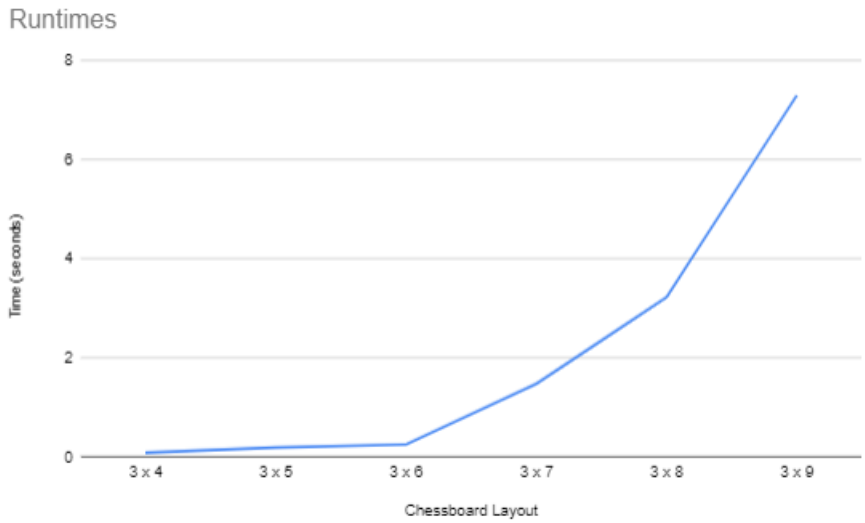
3 x 4	2
3 x 5	3
3 x 6	4
3 x 7	5
3 x 8	6
3 x 9	6

3 by x chess board max knights count

Here, I have also included a table of the 3 by int x column chess board and its corresponding run-time. On the next page is a visual graph of the run-time:

3 x 4	0.093
3 x 5	0.199
3 x 6	0.258
3 x 7	1.485
3 x 8	3.23
3 x 9	7.298

3 by x chess board run-times (seconds)



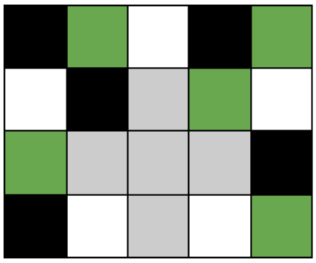
3 by x chess board run-times

As observed in this graph compared to the run-time graph of the square chess boards, the run-time for 3 by x chessboards do not exponentially increase as rapidly. I have come to realize why the run-time is so different. As previously stated, in order to move from one side of the board to the other vertically, a knight would need at least 3 rows. To only have 3 rows would significantly cut the possible placements of the knights, which is why, as long as the number of rows stays at 3, the increase in column count is not as a huge increase in run-time as it was to increase both the row and column count.

* * *

I now transition to 4 by x columns, as the 3 by 10 chess board took longer than 10 minutes to compute a solution. Like the previous chess board types, I only changed the m and n variables in my Python code to create the LP with the corresponding constraints to solve in lpsolve:

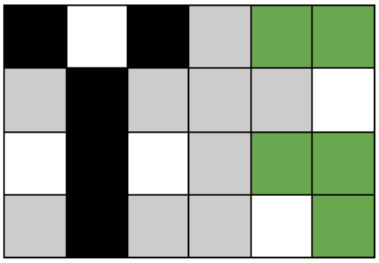
Output visualization (4 by 5):



4 by 5 chess board final max layout

The visualization of the output from lpsolve shows that this chess board is almost symmetrical in that the green and black could be switched in the opposite positions vertically down the center and solve the same LP (excluding the already mentioned claim that the teams can be changed in position and still solve the exact same LP). I conclude that this board has a max 5 knights per team.

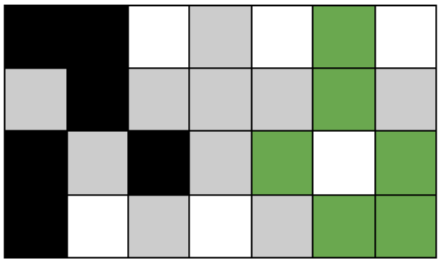
Output visualization (4 by 6):



4 by 6 chess board final max layout

The visualization of the output from lpsolve shows that this chess board is very unlike the last chess board and does not seem to have any apparent pattern from previous chess boards or symmetry on an axis. I conclude that this board also has a max 5 knights per team, again something intriguing, as I had predicted that this would have a 1 increase in max count.

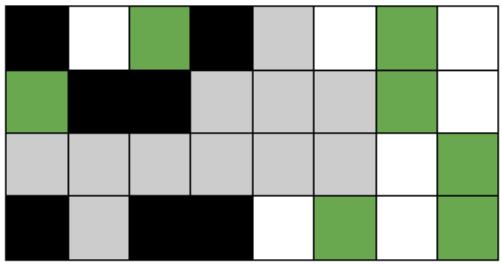
Output visualization (4 by 7):



4 by 7 chess board final max layout

The visualization of the output from lpsolve shows that return to a somewhat symmetrical pattern but, again, is not perfect. The white also does not follow a center pattern/shape as it had in the 3 by x chess boards. I conclude that this board has a max 6 knights per team.

Output visualization (4 by 8):



4 by 8 chess board final max layout

The visualization of the output from lpsolve shows a return to a non-symmetric pattern, and I cannot seem to find another significant pattern in the positions. I conclude simply that the max knights per team is 7.

Here, I have included a table of the 4 by int x column chess board and their max knight count:

4 x 5	5
4 x 6	5
4 x 7	6
4 x 8	7

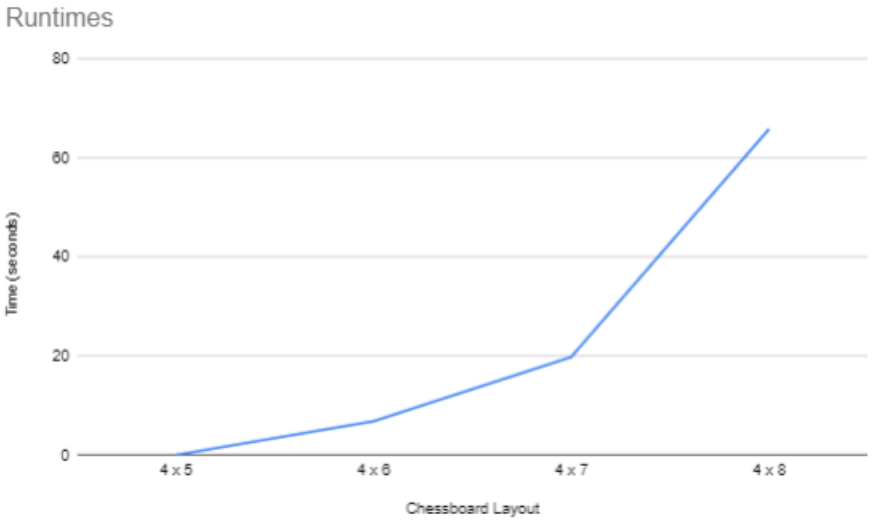
4 by x chess board max knights count

It was interesting to see the opposite pattern compared to the 3 by x chess boards where the plateu happened in larger column counts whereas it occurred from the 5th to 6th column increase. At first I had suspicion as to whether the number of total squares on each chess board, but it did not happen in the same range of total squares.

Here, I have also included a table of the 4 by int x column chess board and its corresponding run-time. On the next page is a visual graph of the run-time:

4 x 5	0.074
4 x 6	6.918
4 x 7	19.876
4 x 8	65.873

4 by x chess board run-times (seconds)



4 by x chess board run-times

As observed in this table and graph compared to the run-time graph of the 3 by x chess boards, the run-time for 4 by x chessboards increased more rapidly exponentially, but not as rapidly as the square chess boards (the graphs might look similar at a glance for the 3 by x and 4 by x, but the vertical axis shows how much greater the increase is in run-time for the 4 by x chess boards). The run-times recorded here confirms my claim in my conclusion about 3 by x chess boards, where the increase in one dimension increases the number of possible positions for knights is slower than increasing both dimensions at once. It also confirms that the increase in possible positions in least in a 3 by x chess board, and the increase in row count for 4 by x chess boards greatly increased the exponential growth in run-time (but, again, not as great as increasing both dimensions at once).