Integer Programming and Chessboard Puzzles

Derryl Sayo

```python
import numpy as np
import cvxpy as cp
```

```
(CVXPY) Apr 15 12:00:09 PM: Encountered unexpected exception importing solver GLOP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected < 9.8.0. Please open a
(CVXPY) Apr 15 12:00:09 PM: Encountered unexpected exception importing solver PDLP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected < 9.8.0. Please open a
```

Chessboard Puzzles

The problem statement for a *chessboard puzzle* (or more specifically *chessboard non-attacking puzzles*) is as follows:

> Given a large number of non-pawn chess pieces (rook, bishop, knight, queen, king) of a single type and none of the others, how do we place the **maximum** number of pieces such that **no capturing can occur** if the pieces are played legally according to the standard rules of chess

This entry aims to summarize, comment, and implement some of the methods described in "An Application of Graph Theory and Integer Programming: Chessboard Non-Attacking Puzzles" (see Foulds and Johnston 1984) to better understand how we can mathematically model this family of problems to find exact solutions.

Why is this Problem "Hard"?

Although the premise is very simple once you understand the movement rules of each piece, a non-trivial question would be "how many **distinct ways** are there to place n-queens"? In particular, the space of possible placements is enormous. Even for the 8-queens problem on an 8×8 board, there are a total of

$$\binom{64}{8} = 4,426,165,368$$

positions with only **92** possible solutions. If you simply try all possible placements and then verify they work, it is too computationally expensive for a brute force search. There are some shortcuts to reduce the search space such as using the fact that we can only place a single queen in each column (reduces the 8-queens possibilities to $8^8 = 16,777,216$) but this is still exponentially growing and reaches the limits of a standard computer's computational power very quickly. For example, the search space under this assumption for the 13-queens problem is 13^{13} possibilities or

$$13^{13} = 3.0287511 \times 10^{14}$$

Although we won't be writing the code to return the total number of solutions, it is worth noting the difficulty of finding solutions for larger chess boards.

Solutions using Integer Programming

Our objective is to **place** (ie. **assign**) chess pieces to locations on a chess board such that they do not attack each other **constrained by the legal attacking moves of chess**.

This can be recognized as a canonical LP assignment problem with chess specific constraints.

Although the paper seeks to build up the problem from the basic principles of linear programming and matrix properties, I found that there is a gap of information missing in explaining how the problem is intuitively formulated mathematically if I wasn't already familiar with the canonical formulation of the problem.

Below, we will walk through the intuition on formulating any problem which involves placing things into a finite space under certain restrictions as an assignment IP problem.

Decision Variables

The core to the assignment problem is to find a way to mathematically represent a way to say "I want to place some item at some location uniquely".

If we think about how we would organize objects in real life, there is really only one decision to be made for each object in question: Should it be placed here or not?

Before even considering how the matrix will look, we know that a chess board is a perfect square and can be uniquely indexed by a row i and a column j. For each of these coordinates (i, j), we want to place some value (let's say 1) if a piece is placed there and another value (let's say 0) if a piece is not placed there.

Let x_{ij} be the assignment of a piece at location (i, j). Then,

$$x_{ij} = \begin{cases} 1 & \text{the piece is placed at location } (i, j) \\ 0 & \text{the piece is not placed at location } (i, j) \end{cases}$$

This binary decision variable formulation is quite powerful. Routing problem solutions use this idea as well by asking "should I take this road or not?" and the knapsack problem makes decisions based on "should I take this item or not?"

Objective Function

Our goal to solve the chessboard puzzle is to fit **as many** of the specified piece on an $n \times n$ chess board. In mathematical terms we want to **maximize** the total number of assigned pieces:

$$\max \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} x_{ij}$$

Although in this case we are unconstraining the number of pieces and letting the algorithm decide, some problems wish to simply find a feasible solution and can be represented by the objective function:

$$\max 1$$

Constraints

We will formulate the constraints for the n-queens problem. To find a feasible solution, we must satisfy the following:

1. Decision variables x_{ij} cannot be assigned a negative value (ie. must be assigned 0 or 1)
2. Only a single queen can exist in each row, column, and diagonal

The first constraint is typical for linear/integer programming. If we flatten the matrix of decision variables into a large vectors, we want each

$$x_{ij} \geq 0$$

for each element of the aggregated X vector.

The flavor of the second constraint is often found in assignment problems and follows from the design of the decision variables.

The rows and columns constraint is quite intuitive since a chess grid can be visualized easily. The idea is as follows for the rows: Consider *only* a single row i. Then, we want to look at every column in that row and count how many assignment occurences are found. Since we know that the variables can only take two values, the assignment will be *unique* if it ends up summing to 1. Thus:

$$\sum_{i=0}^{n-1} x_{ij} = 1,\ i = 1, \dots, n$$

A similar idea is used for the columns

$$\sum_{j=0}^{n-1} x_{ij} = 1,\ j = 0, \dots, n-1$$

The diagonals are a little more complicated. We use the idea that each diagonal lies on some "level" where the coordinates summed/subtracted equal some number.

For example, take this 4×4 chess board. For the second anti-diagonal (bottom-left to top-right lines), we have the (0-indexed) grid coordinates $[(2,0),(1,1),(0,2)]$. Summing each $i+j$, we see that this diagonal has "level" $k = 2$. The largest "level" will include the coordinate $(n-1, n-2)$ for level $k = 2n-3$. If we continue aggregate this pattern, we get that the anti-diagonal levels are $i + j = k,\ k = 1, \dots, 2n-3$.

$$\begin{bmatrix} * & * & (0,2) & * \\ * & (1,1) & * & * \\ (2,0) & * & * & * \\ * & * & * & * \end{bmatrix}$$

Similarly, we can use the same idea for the diagonals (top-left to bottom-right lines) but we subtract the coordinates. For the second diagonal, we have grid coordinates $[(0,1),(1,2),(2,3)]$ such that each $i - j = -1$. If we continue this for each subtraction, we get that the levels are $i - j = l,\ l = -(n-2), \dots, n-2$.

$$\begin{bmatrix} * & (0,1) & * & * \\ * & * & (1,2) & * \\ * & * & * & (2,3) \\ * & * & * & * \end{bmatrix}$$

For the anti-diagonals, we get the following constraints where we can have at most 1 queen on each anti-diagonal:

$$\sum_{i+j=k}^{n} x_{ij} \leq 1,\ k = 1, \ldots, 2n-3$$

And the diagonals:

$$\sum_{i-j=l}^{n} x_{ij} \leq 1,\ l = -(n-2), \ldots, n-2$$

Integer Programming Solutions using Python

We will use CVXPY with the above formulation to solve the n-queens problem.

```
def ip_nqueens(n):
        # Define CVXPY variables
        X = cp.Variable((n, n), integer=True)

        # Define objective function
        objective = cp.Maximize(cp.sum(X))

        # Define constraints
        constraints = []

        # Assignment problem decision variables are non-positive 0/1
        constraints = constraints + [X >= 0]

        # Rook constraints
        # All rows have at most 1 queen
        constraints = constraints + [cp.sum(X[i,:]) <= 1 for i in range(n)]

        # All cols have at most 1 queen
        constraints = constraints + [cp.sum(X[:,j]) <= 1 for j in range(n)]

        # "Bishop constraints" - there is a single queen in each diagonal
        # Anti-diagonals (ie. level lines go from bottom-left to top-right) have
        # at most 1 queen
        for k in range(1, 2 * n - 3):
                diagonal = []
                for i in range(n):
                        for j in range(n):
                                if (i + j == k):
                                        diagonal.append(X[i,j])

                        constraints.append(cp.sum(diagonal) <= 1)

        # Diagonal constraints (ie. level lines go from top-right to bottom-left) have
        # at most 1 queen
        for l in range(-1 * (n - 2), n - 2):
                diagonal = []
                for i in range(n):
                        for j in range(n):
                                if (i - j == l):
                                        diagonal.append(X[i,j])

                constraints.append(cp.sum(diagonal) <= 1)

        # Solve problem
        problem = cp.Problem(objective, constraints)
        problem.solve()

        return np.round(X.value, 3)
```

```
ip_nqueens(4)
```

```
array([[-0., -0.,  1., -0.],
       [ 1., -0., -0., -0.],
       [-0., -0., -0.,  1.],
       [-0.,  1., -0., -0.]])
```

```
ip_nqueens(8)
```

```
array([[-0., -0.,  1., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0.,  1., -0.],
       [-0.,  1., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0.,  1.],
       [-0., -0., -0., -0.,  1., -0., -0., -0.],
       [ 1., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0.,  1., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0.,  1., -0., -0.]])
```

```
ip_nqueens(13)
```

```
array([[-0., -0.,  1., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0., -0., -0.,  1., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0.,  1.],
       [-0., -0., -0.,  1., -0., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0.,  1., -0., -0., -0., -0.],
       [-0., -0., -0., -0.,  1., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0.,  1., -0., -0., -0., -0., -0.],
       [-0.,  1., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0.,  1., -0.],
       [-0., -0., -0., -0., -0., -0.,  1., -0., -0., -0., -0., -0., -0.],
       [ 1., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0., -0.,  1., -0., -0., -0.],
       [-0., -0., -0., -0., -0.,  1., -0., -0., -0., -0., -0., -0., -0.]])
```

Conclusion

Above we have detailed a way of solving chessboard non-attacking puzzles with integer programming whose core ideas behind its formulation is able to be expanded into a whole class of combinatorial optimization problems.

Understanding how the n-queens problem relates to the assignment problem is formulated is essential to formulating a variety of other combinatorial optimization problems as an integer programming problems such as vehicle routing and the knapsack packing.

References

Foulds, L. R., and D. G. Johnston. 1984. "An Application of Graph Theory and Integer Programming: Chessboard Non-Attacking Puzzles." *Mathematics Magazine* 57 (2): 95–104. http://www.jstor.org/stable/2689591.