


[More](#) [Next Blog»](#)
[Create Blog](#) [Sign In](#)

THE JAVA MATHEMATICIAN

Saturday, February 14, 2015

The N-Queens Puzzle and 0-1 Integer Linear Programming

We saw in [the last post](#) how we can tackle the N-Queens Puzzle recursively by considering each column in turn. This way, we can generate all of the possible solutions to the puzzle (eventually!)

However as we'll see today, we can also consider the puzzle in terms of an *optimization*, according to some constraints upon the rows, columns, and diagonals of the chessboard. By formulating the N-Queens puzzle as an instance of a *0-1 Integer Linear Program*, we can pummel it with the full force of modern combinatorial optimization tools!

So that's the plan. First we'll develop a mathematical formulation for the N-Queens puzzle. Then we'll see how we can translate the variables and constraints of the problem into a matrix representation in Java code, which we can pass into some open-source linear programming tools: the SCPSolver together with the GNU Linear Programming Kit (GLPK).

What is a 0-1 Integer Linear Program?

To start with, a *linear program* is an expression for a problem where we are trying to optimize some linear function (the *objective function*, or *cost function*) according to some *constraints* on its variables.

As a quick example, we might wish to maximize the value of the linear function $f = 2x_1 - 3x_2 + 7x_3$ subject to the constraints that $x_1 + x_2 + x_3 \leq 5$ and $-x_2 + x_3 \leq 1$, with x_1, x_2, x_3 non-negative. Or, written in a more conventional form:

$$\begin{array}{ll} \max & f = 2x_1 - 3x_2 + 7x_3 \\ \text{s.t.} & x_1 + x_2 + x_3 \leq 5 \\ & -x_2 + x_3 \leq 1 \\ & x_1, x_2, x_3 \geq 0 \end{array}$$

Now this example seems contrived (and it absolutely is), but it turns out that we can model many real-world optimization decisions as linear programs. And almost half a century's worth of algorithms and techniques have been fine-tuned to solve for their optimal solutions!

Perhaps the most famous algorithm for solving linear programs is George Dantzig's *Simplex algorithm*. This algorithm is quite efficient: it has a polynomial* $O(n^2)$ running time for most practical cases, where n is the number of input variables.

(* It actually has a worst-case exponential $O(2^n)$ running time, but you've got to *really* go out of your way in constructing a pathological example just to see such behaviour.)

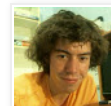
For that example we just saw, the variables x_1, x_2, x_3 can be any real numbers. When we additionally require that the variables take only *integer* values in the solution, we have an *Integer Linear Program*.

Somewhat surprisingly, while the Simplex algorithm (and many others) are fine for solving linear programs with continuous real-number variables, they often fail to work when we want only integers instead! In fact, finding a solution to an Integer Linear Program is considered an *NP-hard problem*, and no polynomial-time algorithm for solving them has been found yet.

When we further restrict the variables to have only the binary values 0 or 1, we have the *0-1 Integer Linear Program*, which is what we'll be using today. Restricting the variables to be binary doesn't actually help us much in terms of computational complexity – we still have an *NP-complete* class of problem here.

It's not all bad news, though. We've still got ways of dealing with such problems, and the

About Me


Russell Edson
[g+](#) [Follow](#) [1](#)

I'm a Mathematician with a strong passion for programming and the Java Virtual Machine!

My areas of mathematical interest include Linear Algebra, Operations Research, Discrete Mathematics, Algorithms and Computation.

As for the programming part, I'm loving all of the JVM languages. I can program well in Java, Scala, Clojure and JRuby; getting competent with Jython and Groovy are on the todo list.

I'm also a long time Linux user (though not as hardcore about it as I used to be). I particularly enjoy Debian, which I've used on-and-off for the last few years. I'm currently giving openSUSE a whirl at the moment.

Also an amateur guitarist, though I don't practice nearly as much as I should. Can *almost* play Classical Gas properly.

[View my complete profile](#)

Blog Archive

- ▼ [2015](#) (11)
 - ▼ [February](#) (2)
 - [The N-Queens Puzzle and 0-1 Integer Linear Program...](#)
 - [The N-Queens Puzzle and Recursion](#)
 - ▶ [January](#) (9)

Blog Posts by Programming Language

- [Clojure](#) (3)
- [Java](#) (4)
- [JRuby](#) (1)
- [Jython](#) (1)
- [Scala](#) (4)

operations research community is actively at work developing bigger and better methods!

We'll be using the fruits of their endeavours today. The guys over at the [GNU Project](#) have compiled many of the useful techniques for solving all kinds of linear programming problems into an open-source C library called the GNU Linear Programming Kit (GLPK). We'll be able to use a Java front-end (the SCPSolver) to call that library from our Java code and solve our 0-1 Integer Linear Programs!

Formulating the N-Queens Puzzle as a 0-1 Integer Linear Program

Recall our last look at the N-Queens puzzle. We'd worked out that a *solution* to the puzzle is a configuration of the chessboard where:

- We have exactly N queens positioned on the board.
- There is exactly one queen in every row.
- There is exactly one queen in every column.
- Any diagonal we could draw over the board should hit, *at most*, one queen.

These will actually serve directly as the constraints for our formulation!

Let's start by defining some *decision variables*. We're interested in the squares on the chessboard. More precisely, we're interested in whether or not a queen occupies a given square in a solution. Suppose we index the rows of the board with i and the columns of the board with j . Then we could define our variables $x_{i,j}$ as follows:

$$\text{Let } x_{i,j} = \begin{cases} 1 & \text{if a queen occupies the square in the } i\text{th row, } j\text{th column,} \\ 0 & \text{if the square in the } i\text{th row, } j\text{th column is vacant.} \end{cases}$$

So our $x_{i,j}$ variables will tell us the locations of the queens on the $N \times N$ board as i ranges across the rows $(1, 2, \dots, N)$ and j ranges across the columns $(1, 2, \dots, N)$.

Our objective function is rather simple: we want to maximize the number of queens placed on the board. So since our $x_{i,j}$ variables will be 1 when a queen exists in the given square and 0 otherwise, we can keep track of the number of queens on the board by just summing up across all of the i, j :

$$\max f = \sum_{i=1}^N \sum_{j=1}^N x_{i,j}$$

In fact, we won't really be worried about our objective function much. As you'll see, it's our constraints that will actually drive us toward a solution to the puzzle in this case.

It is worth mentioning though that if we cared about the particular solution that we reached (say we *really* wanted a queen in the 2nd row, 4th column, if possible), we could weight the variables in this objective function accordingly, and this would yield such a solution for us.

Let's now consider our first constraint: that we require N queens to be positioned on the board for a solution. As we've seen above, the number of queens on the board is simply that sum of the $x_{i,j}$ across all of the rows and columns, so we want to make sure that this total is equal to N:

$$\sum_{i=1}^N \sum_{j=1}^N x_{i,j} = N$$

But wait! If we consider the problem a little more closely, we can see that this constraint is not even necessary. If we find a solution that satisfies the row and column constraints (ie. there is exactly 1 queen in each row, and exactly 1 queen in each column), we will automatically have N queens positioned on the board anyway.

So in fact that first constraint is not necessary for our formulation – it'll be a *consequence* of the formulation instead.

So on to the next constraint: We require exactly one queen in every row. Keeping in mind that we iterate across the rows with i and move across the columns with j , then we simply need to sum across all of the columns in a given row and make sure the sum is equal to 1. And we do

this for every row:

$$\sum_{j=1}^N x_{i,j} = 1 \quad \text{for all } i = 1, 2, \dots, N.$$

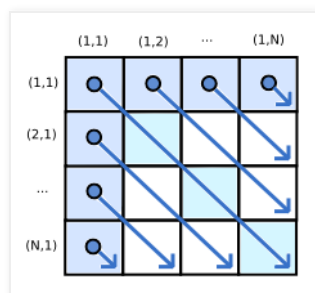
Looks short enough, but keep in mind that right here we already have N linear constraint equations, and each one deals with a different subset of our N^2 variables! So you can see how finding solutions to large linear programs can be computationally demanding.

Similarly, we have the requirement that we have exactly one queen in each column – this time we sum across all of the *rows* in a given column to check that the sum is equal to 1, which we do for every column.

$$\sum_{i=1}^N x_{i,j} = 1 \quad \text{for all } j = 1, 2, \dots, N.$$

Now the last set of constraints, for the diagonals, is kind of tricky. We'll work through them as follows.

First, we'll consider the "forward diagonals"; that is, those diagonals where we move down and to the right (ie. the column number *increases* with the row number). Note that we can actually get all of the possible forward diagonals by moving left/right across the first row, and up/down along the first column, as shown in the following diagram:



So that's what we'll do!

Our first set of constraints will deal with moving along the top row. That is, given a square $x_{1,j}$ in the first row, we'll sum along the forward diagonal from that position; ie. those squares $x_{1+m,j+m}$ as m ranges from 1 to $N-j$ (the remaining columns to the right.)

For the sum, we require that we hit at most one queen. This should hold for each square in the top row, too. So we have the following inequality constraint:

$$x_{1,j} + \sum_{m=1}^{N-j} x_{1+m,j+m} \leq 1 \quad \text{for all } j = 1, 2, \dots, N.$$

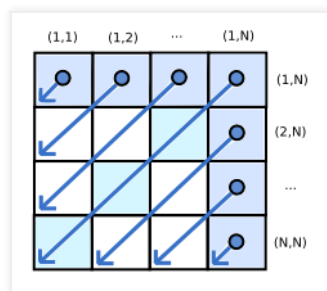
Similarly for the forward diagonals along the first column: we start from a given square $x_{i,1}$ in the first column and sum across the forward diagonal squares $x_{i+m,1+m}$ as m ranges from 1 to $N-i$ (the remaining rows below.)

Again, the sum should be less than or equal to 1, and we do this for every square in the first column:

$$x_{i,1} + \sum_{m=1}^{N-i} x_{i+m,1+m} \leq 1 \quad \text{for all } i = 1, 2, \dots, N.$$

Now let's consider the "backward diagonals", where we move down and to the *left* (so the column number *decreases* as the row number increases.)

Similar to the forward diagonal case, we can actually get all of the possible backward diagonals by moving left/right on the first row, and up/down along the last column, as shown.



So for the first row, we start at a given square $x_{1,j}$ and sum along the backward diagonal from that position until we reach the left-hand edge of the board: that is, we sum across the squares $x_{1+m,j-m}$ as m ranges from 1 to $j-1$.

We want to hit at most one queen, as always. We do this for all squares in the top row:

$$x_{1,j} + \sum_{m=1}^{j-1} x_{1+m,j-m} \leq 1 \quad \text{for all } j = 1, 2, \dots, N.$$

Finally, for the backward diagonals down the last column, given a square $x_{i,N}$ from the last column we sum down the diagonal squares $x_{i+m,N-m}$ as m ranges from 1 to $N-i$ (the remaining squares below.)

For all of the squares down the last column, we should hit at most 1 queen on the diagonal:

$$x_{i,N} + \sum_{m=1}^{N-i} x_{i+m,N-m} \leq 1 \quad \text{for all } i = 1, 2, \dots, N.$$

So that handles all of our constraint equations!

As one final note, we want our decision variables to take only the binary values 0 and 1 of course, so we conventionally make that explicit too.

Our complete 0-1 Integer Linear Program formulation for the N-Queens Puzzle is as follows:

$$\begin{aligned} \max f &= \sum_{i=1}^N \sum_{j=1}^N x_{i,j} \\ \text{subject to:} \\ \sum_{j=1}^N x_{i,j} &= 1 \quad \forall i = 1, 2, \dots, N \\ \sum_{i=1}^N x_{i,j} &= 1 \quad \forall j = 1, 2, \dots, N \\ x_{1,j} + \sum_{m=1}^{N-j} x_{1+m,j+m} &\leq 1 \quad \forall j = 1, 2, \dots, N \\ x_{i,1} + \sum_{m=1}^{N-i} x_{i+m,1+m} &\leq 1 \quad \forall i = 1, 2, \dots, N \\ x_{1,j} + \sum_{m=1}^{j-1} x_{1+m,j-m} &\leq 1 \quad \forall j = 1, 2, \dots, N \\ x_{i,N} + \sum_{m=1}^{N-i} x_{i+m,N-m} &\leq 1 \quad \forall i = 1, 2, \dots, N \\ x_{i,j} &\in \{0, 1\} \quad \forall i, j = 1, 2, \dots, N. \end{aligned}$$

So all up, we have $6 \cdot N$ linear constraint equations in N^2 variables, all of which are binary (0 or

1).

Awesome! Now that the formulation is done, our next challenge is translating this into a form that we can more readily work with in Java code.

The SCPSolver and the GLPK

As mentioned above, we'll be using the GNU Linear Programming Kit (GLPK) library to handle the hard work of solving our Integer Linear Program. But since we're also using Java, we'll want an easy way to make calls to the subroutines of that library.

This is where the SCPSolver comes in! Developed by Hannes Planatscher and Michael Schober, the SCPSolver provides us with an easy class-based interface to the GLPK library.

All we need to do is set up our Integer Linear Program formulated above in terms of the SCPSolver classes, and it will handle all of the actual communication with the GLPK.

Now the SCPSolver, like most linear programming tools, can operate in terms of matrices/arrays. Our linear objective function can simply be a size N^2 array, with each of the array indices corresponding to a particular $x_{i,j}$ variable.

And in that case, the $6*N$ constraints we have above can be turned into a $(6*N) \times N^2$ constraints matrix, where each row of the matrix corresponds to a single linear constraint equation.

With the objective function array and the constraints matrix, the SCPSolver can then call the GLPK library to solve our program. We'll get back a size N^2 array corresponding to the values for the $x_{i,j}$ that make up the solution to the N-Queens puzzle.

So the last real trick here will be finding an easy way to programmatically construct the rows for the constraints matrix (since we *definitely* don't want to have to enter all of them by hand!)

Constructing the Constraints Matrix

Let's look at our first constraint (for the rows) again:

$$\sum_{j=1}^N x_{i,j} = 1 \quad \text{for all } i = 1, 2, \dots, N.$$

Suppose for a second that we're dealing with the $N=4$, 4×4 board case. This gives us the equations:

$$\begin{aligned} x_{1,1} + x_{1,2} + x_{1,3} + x_{1,4} &= 1 \\ x_{2,1} + x_{2,2} + x_{2,3} + x_{2,4} &= 1 \\ x_{3,1} + x_{3,2} + x_{3,3} + x_{3,4} &= 1 \\ x_{4,1} + x_{4,2} + x_{4,3} + x_{4,4} &= 1 \end{aligned}$$

And since we'll represent a row of the constraint matrix as the coefficients for

$$x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4}, x_{2,1}, \dots, x_{3,1}, \dots, x_{4,1}, x_{4,2}, x_{4,3}, x_{4,4}$$

in that order, then the part of the matrix corresponding to these row constraints looks like this:

$$\begin{array}{cccccccccccc} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}$$

Now this same general pattern will hold, regardless of what value of N we want! So we can design a method to construct such a matrix as follows.

First, recall that in terms of our arrays, we start at row $i=0$ for that first row, and column $j=0$ for the first column, since we index everything starting from 0. This actually helps us though:

- Notice that for each of the rows $i=0, 1, 2, 3$, the first 1 that appears in that row is at index $N*i$. For instance, the first 1 in row $i=2$, (ie. the 3rd row), is at index $4*2=8$, (ie. the 9th column).
- From there, we simply place N lots of 1's, and then fill in the rest of the row with 0's.

We can automate this pattern in code as follows:

```

1  /**
2   * Adds the "one queen per row" constraints to the linear program,
3   *   ie. the sum from j=1 to n of xi,j = 1 for all i=1,...,n.
4   */
5  public static void makeQueenRowConstraints(int n, LinearProgram lp) {
6      double[][] rowConstraintsMatrix = new double[n][n*n];
7      for (int row = 0; row < n; row++) {
8          for (int column = n*row; column < n*row + n; column++) {
9              rowConstraintsMatrix[row][column] = 1;
10         }
11     }
12
13     for (double[] row : rowConstraintsMatrix) {
14         lp.addConstraint(new LinearEqualsConstraint(row, 1, ""));
15     }
16
17     //printConstraintsMatrix(rowConstraintsMatrix);
18     //System.out.println();
19 }

```

QueensLP_RowConstraints.java hosted with ♥ by GitHub

[view raw](#)

If we were to uncomment the `printConstraintsMatrix()` method there and check the output (which you should feel free to do! The method is included in the code for the final program), we'd see that it exactly matches up with the pattern we have above.

The rest of the method simply passes the constraints row-by-row to the SCPSolver `LinearProgram` instance. We want each of those constraints to be an equality to 1, so we add a new `LinearEqualsConstraint` as shown.

Similarly, recall that our column constraints were the following:

$$\sum_{i=1}^N x_{i,j} = 1 \quad \text{for all } j = 1, 2, \dots, N.$$

For our $N=4$, 4×4 board case, we have the equations:

$$\begin{aligned}
 x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} &= 1 \\
 x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} &= 1 \\
 x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} &= 1 \\
 x_{1,4} + x_{2,4} + x_{3,4} + x_{4,4} &= 1
 \end{aligned}$$

And so we have the following pattern for the column constraints matrix:

```

1 0 0 0  1 0 0 0  1 0 0 0  1 0 0 0
0 1 0 0  0 1 0 0  0 1 0 0  0 1 0 0
0 0 1 0  0 0 1 0  0 0 1 0  0 0 1 0
0 0 0 1  0 0 0 1  0 0 0 1  0 0 0 1

```

We can generate *this* pattern as follows.

- Note that for each of these rows, the first 1 is in the corresponding column index. (eg. we have a 1 at index $(0,0)$, $(1,1)$, $(2,2)$, ...)
- To reach the other 1's on the same row, we simply move right by N positions each time, until we reach the end of the row.

So here's a method to automate our creation of the column constraints:

```

1  /**
2   * Adds the "one queen per column" constraints to the linear program,
3   *   ie. the sum from i=1 to n of xi,j = 1 for all j=1,...,n.
4   */
5  public static void makeQueenColumnConstraints(int n, LinearProgram lp) {
6      double[][] columnConstraintsMatrix = new double[n][n*n];
7      for (int row = 0; row < n; row++) {
8          for (int column = row; column < n*n; column += n) {
9              columnConstraintsMatrix[row][column] = 1;
10         }
11     }
12 }

```

```

13     for (double[] row : columnConstraintsMatrix) {
14         lp.addConstraint(new LinearEqualsConstraint(row, 1, ""));
15     }
16
17     //printConstraintsMatrix(columnConstraintsMatrix);
18     //System.out.println();
19 }

```

QueensILP_ColumnConstraints.java hosted with ♥ by GitHub

[view raw](#)

Next, we come to the diagonal constraints. Just as it was slightly tricky to work with them in the formulation, it's also fiddly to automate the pattern construction.

For starters, let's look at the case where we want the "forward diagonals" obtained by moving along the first row:

$$x_{1,j} + \sum_{m=1}^{N-j} x_{1+m,j+m} \leq 1 \quad \text{for all } j = 1, 2, \dots, N.$$

When we expand the sum out in the N=4 case, we end up with the following four constraint inequalities:

$$\begin{aligned}
 x_{1,1} + x_{2,2} + x_{3,3} + x_{4,4} &\leq 1 \\
 x_{1,2} + x_{2,3} + x_{3,4} &\leq 1 \\
 x_{1,3} + x_{2,4} &\leq 1 \\
 x_{1,4} &\leq 1
 \end{aligned}$$

Which yields the following matrix pattern:

```

1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0

```

To create this pattern, notice that we start out placing 1's in the same positions we did for the column constraints (ie. indices (0,0), (1,1), (2,2), (3,3).) And we always move exactly N+1 places to the right for the next 1 in the row; except that we stop earlier as we move down the rows. Note that for a given row index *i*, we stop at N*i places from the right-end of the row.

So we can construct such a constraint matrix as follows:

```

1  /* Here we create the constraints for the "forward diagonals" where
2  * we move along the first row.
3  * ie. x1,j + the sum of x1+m,j+m is <= 1 for all j=1,...,n.
4  */
5  double[][] rowForwardConstraintsMatrix = new double[n][n*n];
6  for (int row = 0; row < n; row++) {
7      for (int column = row; column < n*n - row*n; column += (n+1)) {
8          rowForwardConstraintsMatrix[row][column] = 1;
9      }
10 }
11
12 for (double[] row : rowForwardConstraintsMatrix) {
13     lp.addConstraint(new LinearSmallerThanEqualsConstraint(row, 1, ""));
14 }

```

QueensILP_FirstRowForwardDiag.java hosted with ♥ by GitHub

[view raw](#)

With the forward diagonals, we also want to work up/down the first column:

$$x_{i,1} + \sum_{m=1}^{N-i} x_{i+m,1+m} \leq 1 \quad \text{for all } i = 1, 2, \dots, N.$$

Expanding this out for N=4, we get inequalities that look like this:

$$\begin{aligned}
 x_{1,1} + x_{2,2} + x_{3,3} + x_{4,4} &\leq 1 \\
 x_{2,1} + x_{3,2} + x_{4,3} &\leq 1
 \end{aligned}$$

$$\begin{array}{rcl} x_{3,1} + x_{4,2} & \leq & 1 \\ x_{4,1} & \leq & 1 \end{array}$$

Which means we want the following pattern for the constraint matrix:

```
1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0
0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
```

In this pattern, the placement of the first 1 in each row i is at $N*i$, where we count the row indices starting from 0 of course. Then we simply move $N+1$ places to the right for each new 1 in a row, until we reach the end. We can code this process up like this:

```
1 /* Here we create the constraints for the "forward diagonals" where
2  * we move along the first column.
3  * ie.  $x_{i,1} + \text{the sum of } x_{i+m,1+m} \leq 1$  for all  $i=1, \dots, n$ .
4  */
5 double[][] columnForwardConstraintsMatrix = new double[n][n*n];
6 for (int row = 0; row < n; row++) {
7     for (int column = n*row; column < n*n; column += (n+1)) {
8         columnForwardConstraintsMatrix[row][column] = 1;
9     }
10 }
11
12 for (double[] row : columnForwardConstraintsMatrix) {
13     lp.addConstraint(new LinearSmallerThanEqualsConstraint(row, 1, ""));
14 }
```

QueensILP_FirstColForwardDiag.java hosted with ♥ by GitHub

[view raw](#)

Finally, we want to code up the constraints for the "backward diagonals". The backward diagonals obtained by moving along the first row are given by that constraint equation we saw earlier:

$$x_{1,j} + \sum_{m=1}^{j-1} x_{1+m,j-m} \leq 1 \quad \text{for all } j = 1, 2, \dots, N.$$

Expanding out the sum for the $N=4$ case, we get four constraint inequalities:

$$\begin{array}{rcl} x_{1,1} & \leq & 1 \\ x_{1,2} + x_{2,1} & \leq & 1 \\ x_{1,3} + x_{2,2} + x_{3,1} & \leq & 1 \\ x_{1,4} + x_{2,3} + x_{3,2} + x_{4,1} & \leq & 1 \end{array}$$

With the following matrix pattern:

```
1 0 0 0 0 0 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 1 0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 1 0 0 1 0 1 0 0 0 0
```

Here, the first 1 for each row occurs at the corresponding row-column index (ie. $(0,0)$, $(1,1)$, $(2,2)$, ...), and we move along exactly $N-1$ spaces to place the next 1 each time. However, for a given row index i , we don't move beyond the $(N*i)+1$ position!

```
1 /* Here we create the constraints for the "backward diagonals" where
2  * we move along the first row.
3  * ie.  $x_{1,j} + \text{the sum of } x_{1+m,j-m} \leq 1$  for all  $j=1, \dots, n$ .
4  */
5 double[][] rowBackwardConstraintsMatrix = new double[n][n*n];
6 for (int row = 0; row < n; row++) {
7     for (int column = row; column < n*row + 1; column += (n-1)) {
8         rowBackwardConstraintsMatrix[row][column] = 1;
9     }
10 }
11
12 for (double[] row : rowBackwardConstraintsMatrix) {
13     lp.addConstraint(new LinearSmallerThanEqualsConstraint(row, 1, ""));
14 }
```


QueensILP_FirstRowBackwardDiag.java hosted with ♥ by GitHub

[view raw](#)

Last of all, we have the backward diagonals obtained by moving down the last column:

$$x_{i,N} + \sum_{m=1}^{N-i} x_{i+m,N-m} \leq 1 \quad \text{for all } i = 1, 2, \dots, N.$$

In our N=4 case, we get the following inequalities when we expand out the sums:

$$\begin{aligned} x_{1,4} + x_{2,3} + x_{3,2} + x_{4,1} &\leq 1 \\ x_{2,4} + x_{3,3} + x_{4,2} &\leq 1 \\ x_{3,4} + x_{4,3} &\leq 1 \\ x_{4,4} &\leq 1 \end{aligned}$$

And the pattern for the constraint matrix looks as follows:

```
0 0 0 1 0 0 1 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0 1 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
```

Now to generate that pattern, note that the position of the first 1 in each row is a little bit trickier here. What we want to do is this: given a row index i , start at $N*(i+1)$, and then move back one space. From then on we simply move forward $N-1$ spaces as usual until we reach the end.

But there's a slight hiccup this time around! Notice that if we applied this rule precisely as we've said it here, we'd always end up with a 1 in the last column of the top row, wouldn't we? (I've bolded the relevant position in the pattern above.)

However, since we know that this will *always* happen and exactly *where* it happens, we can manually fix it up afterward.

So our final code snippet for the constraint matrix generation is as follows:

```
1  /* Here we create the constraints for the "backward diagonals" where
2   * we move along the last column.
3   * ie.  $x_{i,N} + \text{the sum of } x_{i+m,N-m} \leq 1$  for all  $i=1, \dots, N$ .
4   */
5  double[][] columnBackwardConstraintsMatrix = new double[n][n*n];
6  for (int row = 0; row < n; row++) {
7      for (int column = n*(row+1) - 1; column < n*n; column += (n-1)) {
8          columnBackwardConstraintsMatrix[row][column] = 1;
9      }
10 }
11
12 /* Note: Our iteration here always gives us an extra '1' at the end of
13 * the top row, which we can simply remove here.
14 */
15 columnBackwardConstraintsMatrix[0][n*n - 1] = 0;
16
17 for (double[] row : columnBackwardConstraintsMatrix) {
18     lp.addConstraint(new LinearSmallerThanEqualsConstraint(row, 1, ""));
19 }
```

QueensILP_LastColBackwardDiag.java hosted with ♥ by GitHub

[view raw](#)

When we put everything together (including setting up the linear program in the SCPSolver and defining the objective function), we'll wind up with a neat little program to solve the N-Queens puzzle!

[Originally the whole program was going to be a Gist (like the code snippets you see above), but given that it's quite complicated I figured I'd separate it out into its own GitHub project. That way I can also include the SCPSolver and GLPK libraries* too, and everything you'd need to actually run the code is right there with it.

* Those libraries are licensed under the [GNU GPLv3](#), so note that the complete N-Queens solver program is also licensed under the GNU GPLv3. Which works just fine for us! Sharing is caring, and all that jazz.]

Solving the N-Queens Puzzle

Here we go, it's what we've all been waiting for. Let's see this program in action!

Running the program to solve the N=4 case (with the 4x4 chessboard) gives us the following output (in no time at all, too!):

```
//> GLPK Simplex Optimizer, v4.51
//> 24 rows, 16 columns, 72 non-zeros
//>      0: obj =  0.000000000e+00 infeas =  8.000e+00 (8)
//> *   13: obj =  4.000000000e+00 infeas =  0.000e+00 (1)
//> OPTIMAL SOLUTION FOUND
//> GLPK Integer Optimizer, v4.51
//> 24 rows, 16 columns, 72 non-zeros
//> 16 integer variables, all of which are binary
//> Integer optimization begins...
//> +   13: mip =      not found yet <=                +inf
//> (1; 0)
//> +   18: >>>>  4.000000000e+00 <=  4.000000000e+00 < 0.1%
//> (2; 1)
//> +   18: mip =  4.000000000e+00 <=      tree is empty  0.0%
//> (0; 5)
//> INTEGER OPTIMAL SOLUTION FOUND
//>
//> x1,1=0 x1,2=0 x1,3=1 x1,4=0
//> x2,1=1 x2,2=0 x2,3=0 x2,4=0
//> x3,1=0 x3,2=0 x3,3=0 x3,4=1
//> x4,1=0 x4,2=1 x4,3=0 x4,4=0
```

As you can see, it performs the optimization process and returns us the $x_{i,j}$ values for the solution. If we represent queens on the chessboard as X's and use 0's for the empty spaces like we did in the last blog post, then we have the following solution:

```
0 0 X 0
X 0 0 0
0 0 0 X
0 X 0 0
```

Which is great, since that's definitely a solution to the 4-Queens problem!

Next, let's find a solution for the 8-Queens puzzle:

```
//> GLPK Simplex Optimizer, v4.51
//> 48 rows, 64 columns, 272 non-zeros
//>      0: obj =  0.000000000e+00 infeas =  1.600e+01 (16)
//> *   30: obj =  8.000000000e+00 infeas =  2.220e-16 (1)
//> OPTIMAL SOLUTION FOUND
//> GLPK Integer Optimizer, v4.51
//> 48 rows, 64 columns, 272 non-zeros
//> 64 integer variables, all of which are binary
//> Integer optimization begins...
//> +   30: mip =      not found yet <=                +inf
//> (1; 0)
//> +  107: >>>>  8.000000000e+00 <=  8.000000000e+00 < 0.1%
//> (11; 0)
//> +  107: mip =  8.000000000e+00 <=      tree is empty  0.0%
//> (0; 21)
//> INTEGER OPTIMAL SOLUTION FOUND
//>
//> x1,1=0 x1,2=0 x1,3=0 x1,4=1 x1,5=0 x1,6=0 x1,7=0
//>      x1,8=0
//> x2,1=0 x2,2=0 x2,3=0 x2,4=0 x2,5=0 x2,6=0 x2,7=0
//>      x2,8=1
//> x3,1=1 x3,2=0 x3,3=0 x3,4=0 x3,5=0 x3,6=0 x3,7=0
//>      x3,8=0
//> x4,1=0 x4,2=0 x4,3=0 x4,4=0 x4,5=1 x4,6=0 x4,7=0
//>      x4,8=0
//> x5,1=0 x5,2=0 x5,3=0 x5,4=0 x5,5=0 x5,6=0 x5,7=1
```

```
//> x5,8=0
//> x6,1=0 x6,2=1 x6,3=0 x6,4=0 x6,5=0 x6,6=0 x6,7=0
//> x6,8=0
//> x7,1=0 x7,2=0 x7,3=0 x7,4=0 x7,5=0 x7,6=1 x7,7=0
//> x7,8=0
//> x8,1=0 x8,2=0 x8,3=1 x8,4=0 x8,5=0 x8,6=0 x8,7=0
//> x8,8=0
```

Again, we get a solution (pretty quickly, too!)

```
0 0 0 X 0 0 0 0
0 0 0 0 0 0 0 X
X 0 0 0 0 0 0 0
0 0 0 0 X 0 0 0
0 0 0 0 0 0 X 0
0 X 0 0 0 0 0 0
0 0 0 0 0 X 0 0
0 0 X 0 0 0 0 0
```

So you can see that we can use our program to easily get solutions to the N-Queens Puzzle. In fact, we can use this program to easily find solutions to much larger chessboard configurations, too!

So to end the post, here's the output and solution for the 16x16 chessboard (that was also computed in less than a second.)


```
//> GLPK Simplex Optimizer, v4.51
//> 96 rows, 256 columns, 1056 non-zeros
//> 0: obj = 0.000000000e+00 infeas = 3.200e+01 (32)
//> * 63: obj = 1.600000000e+01 infeas = 8.882e-16 (2)
//> OPTIMAL SOLUTION FOUND
//> GLPK Integer Optimizer, v4.51
//> 96 rows, 256 columns, 1056 non-zeros
//> 256 integer variables, all of which are binary
//> Integer optimization begins...
//> + 63: mip = not found yet <= +inf
//> (1; 0)
//> + 286: >>>> 1.600000000e+01 <= 1.600000000e+01 < 0.1%
//> (17; 1)
//> + 286: mip = 1.600000000e+01 <= tree is empty 0.0%
//> (0; 35)
//> INTEGER OPTIMAL SOLUTION FOUND
//>
//> x1,1=0 x1,2=0 x1,3=0 x1,4=0 x1,5=0 x1,6=0 x1,7=0
//> x1,8=0 x1,9=0 x1,10=0 x1,11=0 x1,12=0 x1,13=0
//> x1,14=0 x1,15=0 x1,16=1
//> x2,1=0 x2,2=0 x2,3=0 x2,4=0 x2,5=1 x2,6=0 x2,7=0
//> x2,8=0 x2,9=0 x2,10=0 x2,11=0 x2,12=0 x2,13=0
//> x2,14=0 x2,15=0 x2,16=0
//> x3,1=0 x3,2=0 x3,3=0 x3,4=0 x3,5=0 x3,6=0 x3,7=0
//> x3,8=0 x3,9=0 x3,10=0 x3,11=0 x3,12=1 x3,13=0
//> x3,14=0 x3,15=0 x3,16=0
//> x4,1=0 x4,2=0 x4,3=0 x4,4=1 x4,5=0 x4,6=0 x4,7=0
//> x4,8=0 x4,9=0 x4,10=0 x4,11=0 x4,12=0 x4,13=0
//> x4,14=0 x4,15=0 x4,16=0
//> x5,1=0 x5,2=0 x5,3=0 x5,4=0 x5,5=0 x5,6=0 x5,7=1
//> x5,8=0 x5,9=0 x5,10=0 x5,11=0 x5,12=0 x5,13=0
//> x5,14=0 x5,15=0 x5,16=0
//> x6,1=0 x6,2=0 x6,3=0 x6,4=0 x6,5=0 x6,6=0 x6,7=0
//> x6,8=0 x6,9=0 x6,10=0 x6,11=0 x6,12=0 x6,13=1
//> x6,14=0 x6,15=0 x6,16=0
//> x7,1=0 x7,2=0 x7,3=0 x7,4=0 x7,5=0 x7,6=1 x7,7=0
//> x7,8=0 x7,9=0 x7,10=0 x7,11=0 x7,12=0 x7,13=0
//> x7,14=0 x7,15=0 x7,16=0
//> x8,1=0 x8,2=0 x8,3=0 x8,4=0 x8,5=0 x8,6=0 x8,7=0
//> x8,8=0 x8,9=0 x8,10=0 x8,11=0 x8,12=0 x8,13=0
//> x8,14=1 x8,15=0 x8,16=0
//> x9,1=0 x9,2=1 x9,3=0 x9,4=0 x9,5=0 x9,6=0 x9,7=0
//> x9,8=0 x9,9=0 x9,10=0 x9,11=0 x9,12=0 x9,13=0
//> x9,14=0 x9,15=0 x9,16=0
```

```
//> x10,1=0 x10,2=0 x10,3=0 x10,4=0 x10,5=0 x10,6=0
//> x10,7=0 x10,8=0 x10,9=0 x10,10=0 x10,11=0 x10,12=0
//> x10,13=0 x10,14=0 x10,15=1 x10,16=0
//> x11,1=0 x11,2=0 x11,3=1 x11,4=0 x11,5=0 x11,6=0
//> x11,7=0 x11,8=0 x11,9=0 x11,10=0 x11,11=0 x11,12=0
//> x11,13=0 x11,14=0 x11,15=0 x11,16=0
//> x12,1=0 x12,2=0 x12,3=0 x12,4=0 x12,5=0 x12,6=0
//> x12,7=0 x12,8=1 x12,9=0 x12,10=0 x12,11=0 x12,12=0
//> x12,13=0 x12,14=0 x12,15=0 x12,16=0
//> x13,1=0 x13,2=0 x13,3=0 x13,4=0 x13,5=0 x13,6=0
//> x13,7=0 x13,8=0 x13,9=0 x13,10=0 x13,11=1 x13,12=0
//> x13,13=0 x13,14=0 x13,15=0 x13,16=0
//> x14,1=0 x14,2=0 x14,3=0 x14,4=0 x14,5=0 x14,6=0
//> x14,7=0 x14,8=0 x14,9=1 x14,10=0 x14,11=0 x14,12=0
//> x14,13=0 x14,14=0 x14,15=0 x14,16=0
//> x15,1=1 x15,2=0 x15,3=0 x15,4=0 x15,5=0 x15,6=0
//> x15,7=0 x15,8=0 x15,9=0 x15,10=0 x15,11=0 x15,12=0
//> x15,13=0 x15,14=0 x15,15=0 x15,16=0
//> x16,1=0 x16,2=0 x16,3=0 x16,4=0 x16,5=0 x16,6=0
//> x16,7=0 x16,8=0 x16,9=0 x16,10=1 x16,11=0 x16,12=0
//> x16,13=0 x16,14=0 x16,15=0 x16,16=0
```

And so a solution to the 16-Queens Puzzle is as follows.

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 X
0 0 0 0 X 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 X 0 0 0 0
0 0 0 X 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 X 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 X 0 0 0
0 0 0 0 0 X 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 X 0 0
0 X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 X 0
0 0 X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 X 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 X 0 0 0 0 0 0
0 0 0 0 0 0 0 0 X 0 0 0 0 0 0 0 0
X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 X 0 0 0 0 0 0 0
```

Posted by Russell Edson at 12:07 AM

 +1 Recommend this on Google

Labels: [Java](#)

No comments:

Post a Comment

Comment as:

[Home](#)

[Older Post](#)

Subscribe to: [Post Comments \(Atom\)](#)

Simple template. Powered by Blogger.