SURREAL NUMBERS AND THE N-QUEENS GAME

Hassan A Noon.

A Thesis submitted to the Faculty of Bennington College,
Bennington, Vermont, in partial fulfillment of the requirements
for the degree of Bachelor of Arts.

May 2002

Recommended to the Faculty of Bennington College for acceptance by:

_____ Thesis Advisor _____ Date
Glen Van Brummelen

_____ Thesis Reader _____ Date
Ruben R. Puentedura

Acknowledgements

First and foremost, I would like to thank my thesis advisor Glen Van Brummelen who agreed to venture down this road with me; for letting me show up last minute and turn his schedule inside out; for his unlimited patience in helping me understand all the simple stuff; for allowing me to submit this thesis at the most ridiculous hour.

I would also like to thank Ruben Puentedura, my advisor and thesis reader, who was always able to answer any question I had. Always walking down the hallway whenever I needed anything from him.

Special thanks to Inam-ur-Rahman who pointed me to the N-Queens problem, which much of this thesis stems from. Nirjan and Sourya who would listen to me talk about queens, numbers and math jargon for hours on end, yet still seem interested. Zain for all the his help with Java. Sara Syed for allowing me to vent my frustrations by listening to me whine and groan throughout my senior year. The abstract algebra class which patiently sat through at least two of my presentations on the N-Queens game. The Media Center crew, those who are and those who have been, especially those who would wake me up at 5 a.m. so I could get back to work.

There are many others who of course should be included, I'm sure you know who you are and will inform me shortly.

Table of Contents

Abstract

First proposed by Max Bezzel in 1848, the 8-Queen problem can be generalized to placing *n* Queens on an NxN chessboard so that no one of the pieces attacks another. The problem can be extended to a game involving two or more players, each successively attempting to place a queen in a non-attacking position on the chessboard. Using the idea of games corresponding to a generalized or Surreal number as introduced by Elwyn Berlekamp, John Conway and Richard Guy, we produce strategies to our n-Queens game and test them in computer simulations.

0 - Introduction

Donald E. Knuth coined the term 'Surreal Numbers' [24] for the class of numbers discovered by John H. Conway [8] in 1976. These surreal numbers have remarkable properties. For all the simplicity of their creation, the axioms established by Conway generate a rich field that includes not only the well-known real ($\mathbb{R}$) and ordinal numbers, but also unfamiliar infinite and infinitesimal numbers. More astonishing is the fact that the surreal numbers are only a simplification of 'numbers' that can be associated with certain combinatorial games; namely two player games with complete information such as chess or checkers. These games distinguish themselves from classical game theory by forcing two players move to alternately as supposed to simultaneously and not allowing chance moves.

In this paper I present a new game derived from a well known mathematical problem and attempt to analyze it in light of Conway's theory of surreal numbers. The N-Queens problem asks us to find a way to place *n* queens on an NxN board so that all the queens are in a non-attacking position. The N-Queens game, on the other hand, asks players to place queens alternately on a given board so that they do not attack any of the queens already placed on the board. This game is much too complex to be analyzed with paper and pencil for large values of *n*. Standard backtracking computer algorithms to find the complete solution set for the N-Queens problem are known but require exponential time. The N-Queens game contains all the complexities of the N-Queens problem, and a great deal more. I implement the game on a computer in order to determine winning and losing positions. My program uses elements from group theory and dynamic programming to minimize the number of calculations

required to run the simulation. Finally, I present a winning strategy for the first player in the N-Queens game for boards with odd-length sides and verify it with the simulation.

1 - Surreal Numbers

Dedekind constructed real numbers by partitioning the rational numbers into two sets. The two sets, say, L and R, are defined so that all numbers in L are less than all numbers in R, and no member of L is its greatest member. This provided, for the first time in the modern period, a satisfactory axiomatic construction of the real numbers. Georg Cantor's ordinals [6] are defined as the order type of a well ordered set [8 Pg. 4,10,11]. The order type of $\{ 0,1,2 \}$ is 3. That of $\{ 0,1,2, \dots \}$, is ω the first transfinite ordinal. That of $\{ 0,1,2, \dots , \omega \}$ is $\omega + 1$ and so on. The ordinals were the first logical conception of transfinite numbers. Conway's definition of Surreal numbers encompasses both these approaches, and proves to be simpler to state.

1.0 - Definition

> *If* L, R *are any two sets of numbers, and no member of* L *is* $\geq$ *any member of* R, *then there is a number* $\{L|R\}$. *All numbers are constructed in this way.*

For a number $x = \{L|R\}$, we refer to x^L as a typical member of L, the left set and x^R as a typical member of R, the right set. We give the name **No** to the class of all numbers created in this way. Now we must define some of the operations on members of **No**. In particular we must determine what we mean by being $\geq$ or $=$ a number.

x *is* $\geq y$ *iff (no* $x^R \leq y$ *and* $x \leq$ *no* y^L*),* $x \leq y$ *iff* $y \geq x$
$x \not\leq y$ *means* $x \leq y$ *does not hold*
$x = y$ *iff (* $x \geq y$ *and* $y \geq x$*)*
$x > y$ *iff (* $x \geq y$ *and* $y \not\geq x$*),* $x < y$ *iff* $y > x$
$x + y = \{ x^L+y , x+y^L \mid x^R+y , x+y^R \}$
$-x = \{ -x^R \mid -x^L \}$

Note that the definitions shown above are based on the existence of sets of numbers as opposed to the existence of numbers themselves. When attempting to prove a proposition about x, we need to consider simpler questions about the same proposition for all x^L and x^R. By applying induction we may eventually reduce any problem to one dealing with the set of no numbers or the empty set. Also note that equality is a defined relation. As a result, two different constructions may yield the same number, for example both $\{ 0 \mid 1 \}$ and $\{ 0,¼ \mid 1 \}$ will represent the number ½.

1.1 - Examples of Numbers

Because the axioms do not assert the existence of any numbers we begin by letting L and R be the empty set $\varnothing$. We write this number as $\{ \mid \}$ and give it the name zero, or symbol 0. We say that 0 was 'created on the zero-th day.' Let us verify that 0 is a number under this definition and that it behaves like the number 0 in regular arithmetic.

Is 0 a number?

> By our definition in section 1.0. we must ask if any member of the left set of $0 \geq$ any member of the right set of 0. There are no such members since both sets are empty. Thus 0 is a number.

Is $0 \geq 0$?

There are no members or the right set of zero (which is empty) that are ≤ 0. $0 \leq$ no members of the left set of zero (which is empty). Thus $0 \geq 0$, further more $0 = 0$.

What is $0 + 0$?

We see that each of the simpler problems we need to solve this problem turns out to be of the form $0 + \varnothing$ or $\varnothing + 0$. There is not such quantity. The left and right sets of $0 + 0$ must be empty. Thus $0 + 0 = 0$

Thus 0 behaves as it should.

We may now use both the number 0 and the empty set to construct the left and right sets of some numbers. The only possibilities are: $\{0|\ \}$, $\{\ |0\}$ and $\{0|0\}$. But we know that the last case cannot be a number, since $0 \geq 0$. We call $1 = \{0|\ \}$ and $-1 = \{\ |0\}$. We say that these numbers were created on the first day. Let us verify the names given to these numbers:

Is $1 \geq 0$?

Is there a $1^R \leq 0$ or is $1 \leq$ any 0^L? Once again, comparisons of members of the empty set with other sets tell us that there are no such relations. Thus $1 \geq 0$. Note that $0 \leq$ a 1^L , namely 0 so $0 \ngeq 1$ as we would expect it to be. So $1 > 0$.

Is $-1 \geq 0$?

It is unless there is a $(-1)^R \leq 0$ or unless $-1 \leq$ some 0^L. We see that $0 \leq 0$ so the first property does not hold. $-1 \not\geq 0$.

Is $0 \geq -1$?

It is unless there is a $0^R \leq -1$ or unless $0 \leq$ some $(-1)^L$. Because both 0^R and $(-1)^L$ are empty, we can conclude that $0 > -1$.

Now that we have verified the existence of these three numbers, we use them to create more. In particular, the following options may be used for both the left and right sets:

$$\{\ \},\ \{-1\},\ \{0\},\ \{1\},\ \{-1,0\},\ \{-1,1\},\ \{0,1\},\ \{-1,0,1\}$$

Examining some of the possibilities, we see that $1 = \{0|\ \}$ is likely to be the same number as $\{-1,0|\ \}$. From the definition we know that this number must be greater than 0. But we already know that $0 > -1$. So the existence of -1 in the left set does not affect the value of this number.

SIMPLICITY THEOREM:

Suppose for $x = \{x^L|x^R\}$ that some number z satisfies $x^L \not\geq z \not\geq x^R$ for all x^L, x^R [1]but that no option of z satisfies the same condition. Then $x = z$

[1] The theorem is proved in ONAG [8], Page 23

The theorem above suggests that two numbers, defined differently, can in fact be equal. Two numbers x and y are called identical ($x \equiv y$) if their left and right sets are the same. In particular, the theorem tells us that x is always the simplest number between x^L and x^R where simplest means earliest created. For instance, let us look at the numbers created on the second day:

$-2 = \{ \mid -1\} = \{ \mid -1,0\} = \ldots$
$-1 = \{ \mid 0\} = \{ \mid 0,1\}$
$-½ = \{-1 \mid 0\} = \{-1 \mid 0,1\}$
$0 = \{ \mid \} = \ldots$
$½ = \{0 \mid 1\} = \{-1,0 \mid 1\}$
$1 = \{0 \mid \} = \{-1,0 \mid \}$
$2 = \{1 \mid \} = \{0,1 \mid \} = \ldots$

We can verify the names assigned to these numbers by calculating values such as $1+1$.[2] We see that this value should be $\{0+1,1+0 \mid \}$. Should $0+1 = 1+0 = 1$, we would get our desired result $\{1 \mid \} = 2$. We have:

$x+0 \equiv \{x^L+0 \mid x^R+0\} \equiv \{x^L \mid x^R\} \equiv x$ and

$0+x \equiv \{0+x^L \mid 0+x^R\} \equiv \{x^L \mid x^R\} \equiv x$

(Note that the second step in both cases is an inductive procedure which makes use of the fact that x^L and x^R are both numbers created on an earlier day, and that by applying this process over and over, we may reduce the problem to one that involves just the empty set.)

From the results, we know that $1+1 = \{1 \mid \} = 2$.[3]

---

[2] According to the definition of addition given in Section 1.0
[3] It took Russel & Whitehead [29] several hundred pages of logical build-up before being able to show 1+1=2

Sequentially repeating this process we can generate further numbers. The numbers generated on the third day are ¼, ¾, 1½, 3 and their negatives. On day 4, we have ⅛, ⅜, ⅝, ⅞, 1¼, 1¾, 2½, 4 and their negatives and so on for succeeding days. However the system continues beyond finite day numbers; consider the number ω defined to be $\{0,1,2,3,\ldots|\ \}$. It is the first transfinite ordinal and is born on the day ω. From the respective operation definitions, we see that $-\omega = \{\ |0,-1,-2,-3,\ldots\}$ and $1/\omega = \{0|½,¼,⅛,\ldots\}$ are also born on the same day. In addition some ordinary numbers such as $\frac{1}{3} = \{¼, ¼+\frac{1}{16}, ¼+\frac{1}{16}+\frac{1}{64},\ldots|½, ½-⅛,\ldots\}$ are also born on day ω. Of course, we call this number $\frac{1}{3}$ because $¼<¼+\frac{1}{16}<¼+\frac{1}{16}+\frac{1}{64}<\ldots<\frac{1}{3}<\ldots<½-⅛<½$. In fact, using the dyadic rationals and procedures akin to Dedekind sections, we can create all rationals and reals. Thus numbers like $\sqrt{2}$, e, and π are also born on this day.

Cantor's ordinal numbers are also embedded in the surreal numbers; they correspond to the members of **No** with empty right sets; for example $\omega+1 = \{\omega|\ \}$. On the other hand, surreal numbers go well beyond ordinal numbers by allowing the existence of infinite numbers like $\omega-1 = \{\ 0,1,2,3,\ldots\ |\ \omega\ \}$ or $\omega/2 = \{\ 0,1,2,3,\ldots\ |\ \omega,\omega-1,\omega-1,\ldots\ \}$.

2 - Surreal Numbers and Combinatorial Games

By dropping the requirement that no member of the left set can be ≥ any member of the right set, we can broaden our theory of surreal numbers to include general combinatorial games. These games are not identical to those studied in classical game theory. We are concerned instead with well defined games with no hidden information or chance moves with one winner and one loser. Thus we are usually not concerned with economic game theory; i.e. games that allow different levels of payoffs for particular moves. Our games terminate when one player has no legal moves; the other player is then declared the winner.

2.0 - Rules of a Game

1. There are two players, often called Left and Right.
2. There are several, (usually finitely) many positions and often a particular starting position.
3. There are clearly defined rules that specify the moves that either player can make from a given position to its options.
4. Left and Right move alternately.
5. Both players have complete information concerning the game state.
6. There are no chance moves, such as rolling dice or shuffling cards.
7. In the normal play convention, a player unable to move loses.
8. The rules are such that play will eventually come to an end with a player unable to move; thus there are no games which are drawn by repetition of moves.

2.1 - Outcome Classes in Games

Analysis of combinatorial games is, in essence, the assignation of a value to the current position of a game. From this position, either of the two players may make a move, thus transforming the game to what is in effect another game. The value associated with a game state can fall into one of several classes telling us which player can win the game from its current state. The game where neither player can make a move is said to have value 0 and is thus a losing game for the first player. In general, the value P of a particular position or game is determined by its left and right sets L and R, where L is the set of positions available to Left and R is the set of positions available to Right. We write this value as $P = \{L|R\}$. For example, consider the game of Hackenbush.

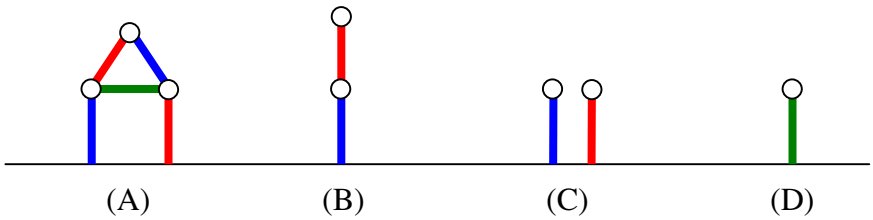


Figure 2.1.0 - Hackenbush

The game begins with a number of line segments connected to a base line. A legal move is to cut or erase one of the edges. Left may cut any blue edge; Right may cut any red edge; and either player may cut a green edge. If any portion of the game is not connected to the base directly or indirectly, it is immediately erased. In Figure 2.1.0 (A), for example, if Left erases the bottom left edge and Right the bottom right edge, the roof of the house along with the green edge completely disappears. In a situation such as this, where the game reduces to no edges at all, its value is said to be 0 since no moves remain. We can verify this with the definition of $P = \{L|R\}$, where because L and R are empty, we have $P = \{\,|\,\} = 0$, just as

we have seen in surreal numbers. Thus since $P = 0$, the second player has a winning strategy.

Suppose the game has a single blue edge. Then Left can make one move to the game with value 0 and Right can make no moves. Hence the value of this game is $\{0|\ \} = 1$. Similarly, the game with a single red edge has value -1. Next consider Figure 2.1.0 (B); Left can move to the game with value 0 (his only move erases the entire tree), and Right to the game with value 1 (his only move leaves a single blue edge standing). The value of this game is $\{0|1\} = ½$. The value of the same game with colors reversed is $-½$. Note that positive games always leave Left with at least one move ≥ 0, whereas Right, if he has a move, must move into a positive game. We may conclude that Left can always win a positive game and similarly Right can always win a negative game, assuming the players use optimal strategies (for instance in the game $\{-1,0|\ \}$ we may assume that Left will move to 0 rather than -1 and win).

Now consider Figure 2.1.0 (C). Left may move to a game with value -1 and Right to a game with value 1. This game has value $\{-1|1\} = 0$.[4] This game consists of two trees, and may be thought of as two games being played simultaneously. We may calculate the value of this game by finding the value of each individual game and adding the results. Say one of these

games is $G = \{G^L|G^R\}$ and the other $H = \{H^L|H^R\}$. By definition, the sum of these games is

$$G + H = \{G^L+H, G+H^L | G^R+H, G+H^R\}$$

The valid moves for Left in the sum of the games are to move in G to G^L or in H to H^L. The resulting sums are G^L+H or $G+H^L$ respectively.

Finally consider Figure 2.1.0 (D). Both players may move into the game with value 0. Obviously the first player to move will win. The value of this game is $\{0|0\}$, which we call $*1$. This game is said to be *fuzzy* to 0, written $*1 \parallel 0$ since neither $G \geq 0$ nor $G \leq 0$ hold.

There is always a winning strategy for the first player in a fuzzy game. In summary:

$G > 0$ *(is positive) if there is a winning strategy for Left.*
$G < 0$ *(is negative) if there is a winning strategy for Right.*
$G = 0$ *(is zero) if there is a winning strategy for the second player.*
$G \parallel 0$ *(is fuzzy) if there is a winning strategy for the first player.*

2.2 - When Hackenbush is Nim

Impartial games form an important subclass of combinatorial games. The theory of Nim is central to that of impartial games. *Impartial games* observe the rules of games outlined in Section 2.0, with the added stipulation that the moves available to a player are the same as those available to his opponent at any point during the game. This the left and right sets of the value of an impartial game are identical. In Hackenbush for example, all games with

[4] By the simplicity theorem in section 1.1

green edges (but no blue or red edges) are impartial. We can immediately see that impartial games can have values that are either zero or fuzzy.

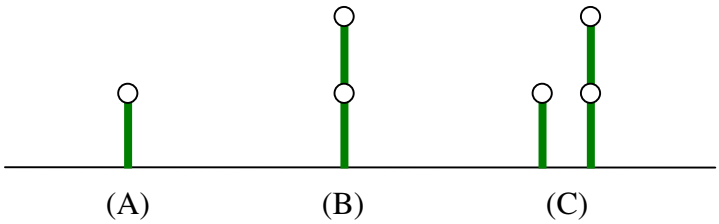


Figure 2.2.0 - Green Hackenbush

The game of Nim, unexpectedly central to the theory of impartial games, is played with several heaps or piles of counters. A legal move in Nim is to remove any number of counters from any one pile. A Nim game with one pile of one counter and another of two counters is identical to the Hackenbush game shown in Figure 2.2.0 (C). In fact, clearly all Hackenbush games with strings of green edges are actually Nim games.[5] The value of the game in Figure 2.2.0 (A) is $*1 = \{ 0 \mid 0 \}$. The value of the game in Figure 2.2.0 (B) is $*2 = \{ 0,*1 \mid 0,*1 \}$.

We can assign a Nim value, also called a Nimber, to any Nim heap or Hackenbush string with *n* counters; $*n = \{ 0,*1,*2,\ldots,*(n-1) \mid 0,*1,*2,\ldots,*(n-1) \}$. Thus Nim games are either fuzzy (have a "Nimber" value $*1,*2,\ldots$) or have value 0.

One variation of Nim allows its players to add counters to any heap if they choose to do so. Suppose that this game G has *n* counters and is being played simultaneously with games H and K. Then, if either player has a winning strategy in $*n+H+K$ and that strategy calls for him to do so, he will move in $*n$. If his opponent decides to make use of this extra freedom

[5] Berlekamp, Conway and Guy [3] show that all green Hackenbush are easily converted into a standard Nim

and add counters to that heap, the winning move in $*n$ is still available to the first player. If the player wishes to reverse the effect of his opponent's move, he may remove as many counters from the game as his opponent added, leaving the situation unchanged. Thus the availability of these new positions from the heap does not alter its value, $*n$. We conclude that for the impartial game $G = \{*a,*b,\ldots|*a,*b,\ldots\}$, the value of the game is $*$(Mex), where Mex is the least integer not present in $(a,b,\ldots)$. R.P. Sprague [30] and P.M. Grundy [17] independently showed that all impartial games are actually Nim games with reversible moves in disguise. This startling result reduces the vast array of possible impartial games to only Nim games, greatly enhancing our analytic power.

The value of the game in Figure 2.2.0 (C) is simply the sum of the games $*1$ and $*2$. We can apply our definition of sums to these Nimbers, and calculate the result through recursion. An easier way to get the same result is to create a Nim-Addition table, such as:

0	1	2	3	4	5	6	7	8
1	0	3	2	5	4	7	6	9
2	3	0	1	6	7	4	5	10
3	2	1	0	7	6	5	4	11
4	5	6	7	0	1	2	3	12
5	4	7	6	1	0	3	2	13
6	7	4	5	2	3	0	1	14
7	6	5	4	3	2	1	0	15
8	9	10	11	12	13	14	15	0

Table 2.2.0 - A Nim-Addition Table

Consider the value 6 in the third column and the fifth row. It corresponds to the Nim game with two heaps, one with value three and the other five. If we were to make a move in the three heap in this game, we could move to $*2+*5$, $*1+*5$ or $0+*5$, that is, all the preceding

values in the fifth row. If we were to make a move in the five heap, we could move to $*3+*4$, $*3+*3$, $*3+*2$, $*3+*1$ or $*3+0$, that is, all the preceding values in the third column. The value of $*3+*5$ is the first integer not present in these preceding rows and columns; Thus $*3+*5=*6$.[6] All the values in the table are calculated in this way. For instance, from the table above we see that the Nim value of the game in Figure 2.2.0 (C) is $*1+*2=*3$.

[6] Nim addition can also be performed by working with a binary representation of Nimbers [3, Pg. 73-74]

3 - The N-Queens Game

The N-Queens problem was proposed by Max Bezzel in 1848 [2] for a regular 8x8 chessboard. The queen is a chess piece that attacks in horizontal, vertical and diagonal directions. The problem asks to place *n* queens on a board of size NxN so that none of the queens attacks any of the others. Gauss proved in 1850 that there are 92 solutions to this problem on an ordinary chessboard [5,16]. Twelve of these solutions are distinct; the remaining eighty are generated through rotations and flips of the original twelve solutions. The problem becomes increasingly complex as the board size increases. It is often used as an illustration of backtracking algorithms in computer science courses, where solutions are generated by attempting to place queens in available squares. Such algorithms are known to take exponential time to generate all solutions of the board. For our N-Queens game, we will be concerned with all the ways that *any* number of queens, not specifically *n* queens, can cover the entire board without attacking one another. The smallest number of queens possible to achieve this on a 8x8 chessboard is five; it can be done in 91 distinct ways.

N	Solutions	Distinct Solutions	Minimum Queens	Distinct Solutions
1	1	1	1	1
2	0	0	1	1
3	0	0	1	1
4	2	1	3	2
5	10	2	3	2
6	4	1	4	17
7	40	6	4	1
8	92	12	5	91
9	352	46		
10	724	92		
11	2680	341		
12	14200	1787		
13	73712	9233		
14	365596	45752		
15	2279184	285053		
16	14772512	1846955		
17	95815104	11977939		

Table 3.0 - Solutions to the N-Queens Problem

Source: Mathworld.com [6]

Table 3.0 demonstrates the near-exponential rise in the number of solutions to the N-Queens problem as the board size increases.

The N-Queens game is a two player game played on a chessboard where each player moves alternately. A legal move for any player is to place a queen on the board so that it does not attack any of the queens already in place. The first player unable to make a move loses the game. Georg Schrage [31] presented a similar game with severe restrictions that allowed an easier implementation of the game in a computer program. His rules are such that the players are required to move into successive columns of the board starting from the left.

We can see immediately that the N-Queens game is an impartial game since it conforms to the conditions in Sections 2.0 and 2.2. Any position in the game is an N-position (Next player winning) or a P-position (Previous player winning). We may regard the N-Queens problem as a one person version of the N-Queens game where the player tries to maximize the number of moves he can make in order to delay the eventual losing outcome.

A representation of the N-Queens game in terms of Nim heaps is not as obvious as it was for Green Hackenbush. On the other hand, the game resembles Nim-Squared [15]; a game played with an array of counters arranged in a grid. A legal move in Nim-Squared is to remove any number of counters from any column or row of the grid. The N-Queens game may be viewed as a variant of Nim-Squared where a player must remove all the counters in a row, column and diagonal of any one counter.

4 - Analysis of the N-Queens Game

The N-Queens game ends when no unattacked squares remain on the board. Such a position has value 0, since no moves can be made from it. Knowing this end condition, we may start with an empty board and consider all possible routes to any ending position, and thus calculate the Nimber associated with the board. We may apply this technique easily to boards of size up to 4x4. Larger boards become much too complex to analyze with pen and paper, so we must use other methods to analyze them.

4.0 - The Smaller Boards

Let us examine the 1x1 N-Queens game. There is only one move available, which takes us to an end position. Thus the value of the empty 1x1 board is $\{ 0 \mid 0 \} = *1$. For the 2x2 board, we have four available moves, all of which lead to an end game. The value of this game, then, is $\{0,0,0,0|0,0,0,0\} = *1$. The symmetries of the board are such that we needed to consider only one of the four available moves. For larger boards we may take advantage of the symmetries of the board to eliminate duplicate calculations like these. Let us now consider the 3x3 case. A move to the middle of the board leads to the end game; This option has value 0. From the symmetries of the board, clearly we need only consider two more moves; to a corner square, and to an edge. These options are examined in Figure 4.0.0. (C) and (D).

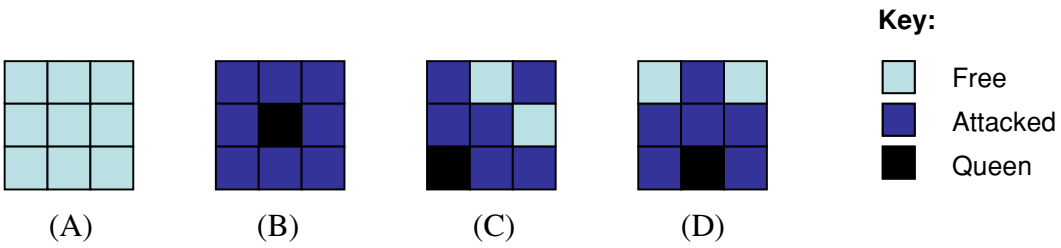


Figure 4.0.0 - The N-Queens game on a 3x3 Board

The light squares indicate the unattacked squares, hence still available. A black square represents a queen. After counting reflections and rotations, the only unique positions available from 4.0.0 (A) are to 4.0.0 (B), 4.0.0 (C) and 4.0.0 (D). But the only positions available from 4.0.0 (C) and 4.0.0 (D) are to the endgame, hence the values for 4.0.0 (C) and 4.0.0 (D) are $*1$. Thus the only moves available from 4.0.0 (A), the empty board, are to those with value 0, $*1$ or $*1$. Using our definition of Nimbers, we see that the empty 3x3 board has value $*2$.

4.1 - Strategies

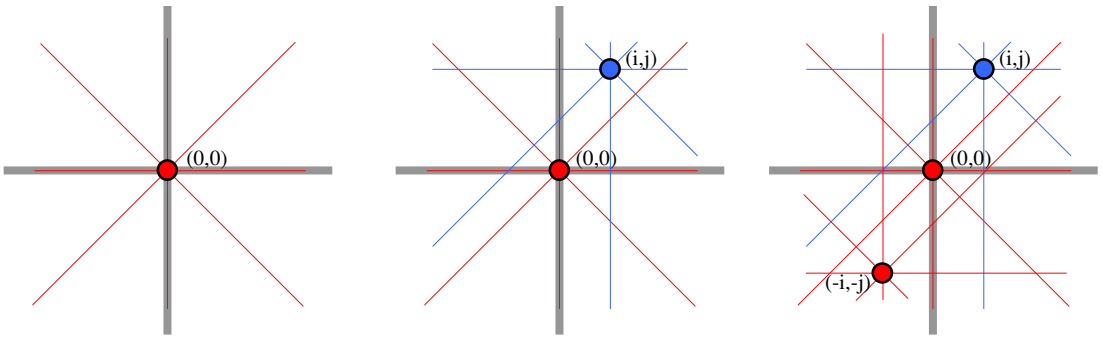
We already know the value of any N-Queens game will be 0 or a Nimber. If a given position has value 0, it is said to be a P-position (Previous player winning) because the first player to move will lose under optimal strategies. If the game value is a Nimber, it is said to be a N-position (Next player winning) because at least one of the current player's moves will force his opponent into a losing position, namely, one with value 0. A player with a winning strategy, to play optimally, must move to a position with value 0. We have already seen such moves in the first three boards. In each case, moving to the middle guaranteed a win. It can be easily shown that moving to the middle on an empty 4x4 board also is a winning move.

THEOREM:

On an empty nxn board, for odd *n*, the middle position is a winning strategy (i.e. has value 0).

Proof:

Map each square on a given nxn board to a point in $\mathbb{Z}^2$, where the middle square on our board will map to the point (0,0). Place the first queen on this middle point (0,0). We denote all the points attacked by this queen with horizontal, vertical and diagonal lines passing through the origin. That is to say, we may not place a queen on any point that passes through any of the lines y=x, y=−x, y=0 and x=0. Note that this situation, represented in Figure 4.1.0 (A), has 180° rotational symmetry around the origin. In particular, a 180° rotation of any of the free points on the plane produces another free point.



(A) (B) (C)

Figure 4.1.0 - Winning Strategy on Boards with Odd Side

Suppose that the opponent places his piece on the point (i,j) (Figure 4.1.0 (B)). Clearly the point (-i,-j), a 180° rotation of (i,j) around the origin, is free, since it is not

on one of the attack lines of the central queen. Player 1 thus places a queen in position (-i,-j).

LEMMA:

If the free spaces on the board originally had 180^o rotational symmetry, the placement of queens at (i,j) and (-i,-j) preserves that symmetry.

Proof:

Suppose the square (a,b) was free prior to the placement of queens at (i,j) and (-i,-j) and is not free now. Then the queen at either (i,j) or (-i,-j) attacked it. Suppose it was the one at (i,j). Then it is easy to show that the queen at (-i,-j) attacks (-a,-b), and thus both squares at 180^o rotational symmetry (a,b) and (-a,-b) are not free.

Suppose the square (a,b) remained free after the placement. If (-a,-b) is attacked, then (since the original board has 180^o rotational symmetry) it must be attacked by the queen at either (i,j) or (-i,-j). Suppose it is attacked by (i,j); then obviously the queen at (-i,-j) attacks (a,b). But (a,b) is known to be free. (-a,-b) must remain unattacked.

Adopting this strategy, Player 1 is guaranteed to have a move after his opponent moves. Because there are only finitely many moves possible, play must come to an end and it can do so only after Player 1's turn.

Notice that in the strategy outlined above, our first move ensured that no later moves can be made to an axis of symmetry. Furthermore, our move maintained 180^o rotational symmetry of the unchecked squares on the board. This strategy fails on a board with n even, since 180^o

rotational symmetry is not maintained. Moving to the middle of an even board does, however, create a reflective symmetry around the main diagonal in which the queen is placed. A tit-for-tat strategy similar to the one outlined above, attempting to exploit this reflective symmetry, fails immediately. The reflection of the opponents move about the axis is no longer available, since it is on a diagonal to the square chosen by the opponent.

5 - Programming the N-Queens Game

The N-Queens problem is most often solved by a backtracking algorithm where a computer program attempts to place a queen in all available rows in each successive column and thus cover all possible solutions to a given board. The program presented in this paper uses a similar approach to calculate the Nimber associated with a given empty board, hence solving the N-Queens game. Our program iterates through all the squares on the chessboard and creates a new board with a queen on each available square. It then repeats the process till it reaches the end condition, when no more queens can be placed on the board. The value of this last board is 0. Once we know all the different end positions, we can calculate the values of all the intermediate positions and work our way up to the starting position of the empty board.

The algorithm described above is highly inefficient; it redundantly calculates Nimbers for board positions symmetrical to positions already computed. This is most obvious by observing that placing a queen in any corner of the empty board is guaranteed to find the same value. If we were to take into account these symmetries, our algorithm might work much faster; and given the exponential increase in processing time required, this saving may be critical. The algorithm also treats the case of moving to square *a* and then to square *b* distinctly from moving to *b* and then to *a*. Our program may be further enhanced if we do not differentiate between these two cases.

5.0 - Symmetries of a Chessboard

If we ignore the coloring of the squares, the group of symmetries of the chessboard is the dihedral group D_4. We are interested instead in the group of symmetries of the free squares on a chessboard. If, for example, this group is the group of 180^o rotations, then in a given position we know that the value of the option of moving to one square is the same as the option of moving to the square at a 180^o rotation to it. We need calculate only one of these values. The possible symmetry groups to which the free squares of a given N-Queens game belong is shown below.

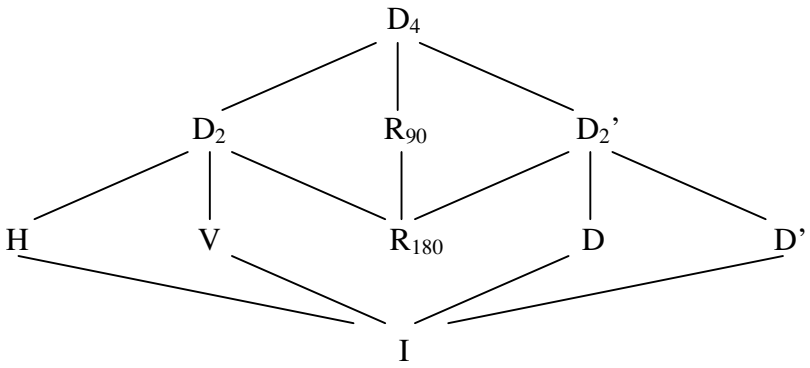


Figure 5.0.0 - Lattice Diagram of D_4

Figure 5.0.0 is the lattice diagram of the dihedral group D_4. The free squares of a given board will always be in one of the symmetry groups outlined above, since they are at least always in I, the identity transformation. The labels H and V in the diagram stand for horizontal and vertical flips of the board respectively, the D and D' for flips along the main diagonal or the bottom left to top right diagonal, and R_{180} and R_{90} for 180^o and 90^o rotations around the center of the board. If we determine the group of symmetries of free squares on our chessboard to be one of the bottom five values in Figure 5.0.0 (namely H, V, R_{180}, D and D', all of which have order 2), we can cut our work by up to half by considering the appropriate

elements in the orbits of the squares[7]. If the group of symmetries is D_2, R_{90} or D_2' (all of which have order 4), we need only consider up to one fourth of the total number of free squares. Finally, if the group is D_4, we can cut the number of calculations by a factor of eight. Determining the group to which a given board belongs costs some time, but the reduced game tree that results speeds up the overall time it takes to find our solution. Rodney W. Topor [33] has used a similar technique to determine solutions to the N-Queens problem.

5.1 - Using Dynamic Programming for an Optimized Solution

Dynamic programming is an algorithmic technique in which an optimization problem is solved by caching sub-problem solutions (memoization) rather than recomputing them. We noticed earlier that our program treats the cases of moving to square *a* and then *b* as a distinct case from moving to *b*, then *a*. After three moves, our program will look at 3! different cases where it could have calculated only one, and then proceeded to look up the other values from the first cached value.

We store our board in an array of size *n*, $Q = (q_1,q_2,q_3,\ldots q_n)$ where for every q_i, *i* denotes the column number and *q* the row number a queen is in. If there is no queen in a particular column then the *q* value is set to zero. We may store the value of this particular board in a separate *n*-dimensional array. The value of the board Q will be in the q_1th element in the first dimension, q_2th element in the second dimension, …, and the q_nth element in the *n*th dimension. If we come across the same arrangement of queens in the search in our game tree,

[7] When a group G acts on a set X, it permutes the elements of X. Any particular element X moves around in a fixed path, which is called its orbit. In the notation of set theory, a group orbit can be defined as
$G(x) = \{gx \in X : g \in G\}$

we will find the value already stored in the appropriate place and we will not need to calculate it again.

This approach to dynamic programming can use a considerable amount of memory and prove to be unfeasible to implement for large values of *n*. Note that even though we do not use all the elements of the *n*-dimensional array, each one of the empty elements will still reserve space in memory. We may work around this problem by using a hashtable or a similar data structure to avoid allocating unused space. An implementation with the hashtable will typically[8] be able to find solution for board of side eight or less. In order to solve the 9x9 case, we can make the fifteen distinct moves on the empty board and solve for each option individually. We can use our results to find the value of the empty board.

[8] Computers with little free memory may not be able to solve for the 8x8 case.

6 - Results

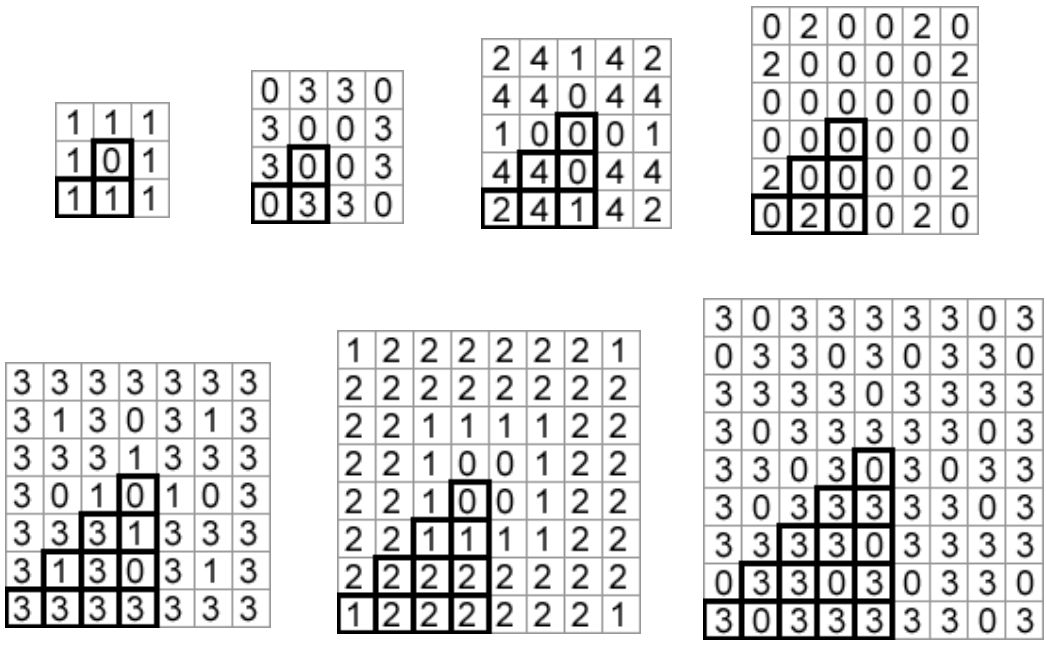


Figure 6.0 - Options on the Empty Board

The Figure above shows empty boards of size 3 to size 9 and their option values for the N-Queens game. The number in each square corresponds to the Nimber value of the board with a queen placed in that square. The symmetry of the empty board implies that only the squares in the triangular region need be calculated. The Nimber value of an empty board is the minimal excluded integer (Mex) not present in any of its options. For example, the Nimber value for the 7x7 board is $*2$. The table below shows the values of empty boards for $n = 1,2,\ldots,10$.

N	1	2	3	4	5	6	7	8	9	10
Value of nxn board	$*1$	$*1$	$*2$	$*1$	$*3$	$*1$	$*2$	$*3$	$*1$	0

Table 6.0 - Nim Values of the Empty Board

The Nimber values of boards of size 1, 2 and 3 correspond to our analysis in Section 4.0. Note that the Nimber values of all the odd sized boards are non zero, as we demonstrated

with our winning strategy in Section 4.1. Our strategy corresponds to placement of the first queen in the middle square and indeed the middle square has value 0. Surprisingly, the first nine boards, even those with even sides, are games of N-positions[9]. That is to say the first player can win each of these games. In all cases up to the 9x9 board, a winning move is to place the first queen in the middle of the board (for even boards, one of the four squares nearest the middle). Surprisingly, however, calculations for the empty 10x10 board show that it is a P-position; it has value 0. All the options from this game are non-zero Nimbers.

[9] N-positions are Next player winning and P-positions are Previous player winning (Section 4.1)

7 - Extensions and Open Questions

The algorithm presented in this paper to solve the N-Queens game is not necessarily the optimal one. We consider rotations of the free squares and we take into account some duplicated branches of the tree, but there are other symmetries not yet considered. For example, we may place two queens on the board so that the free squares have no symmetries; but we know that this arrangement of the queens can be rotated by 90^o with no change to the value of the game. Thus we have not taken into account all possible time savings.

The N-Queens game, like most combinatorial games, has simple rules but is difficult to play optimally. The object of this paper was to attempt to provide the players with a methodology to work out winning strategies of a given game. In this section we indicate various alternate approaches to the analysis, open questions, and alternate versions of the game.

7.0 - Analysis Using Graph Theory

L. R. Foulds and D G. Johnston [12] use graph theory and linear programming to solve various N-Chess Piece problems. They queen, they argue, may be considered to be a bishop and a rook combined. So by combining the attacking properties of the two pieces, linear programming may be used to solve the N-Queens problem.

The chessboard may be represented as an undirected graph. Each square is considered to be a vertex; a pair of vertices is adjacent if they lie on the same vertical, horizontal or diagonal on the board. The graph is complicated to draw on paper, but it may be easily manipulated via a computer using its adjacency matrix. Using a backtracking algorithm like the one described

in section 5, we may use this graph rather than an array to represent our board. Consider what happens to the complement of the graph when we place a queen on a vertex v. (Note that the complement of a graph is calculated very quickly, since we simply switch the zeros and ones in the adjacency matrix.) The vertices connected to v in the complement are those that are still unattacked. Now we create a new graph g' with only the unattacked vertices, those connected to v in the complement. We place the next queen in this graph and again simplify the graph by examining the vertices connected to the newly placed queen by only one edge. We continue doing so until there are no vertices left. We have now achieved a cover of the board, i.e., there are no free squares available.

We may also find solutions to the N-Queens game by finding dominating sets on the queens' graph described above. A set of vertices such that all the vertices in the graph are either in this set or are adjacent to a vertex in this set is said to be dominating [27]. Minimal dominating sets thus are really end positions of the N-Queens game.

Restating the problem in this way allows us to apply different tools to the problem but the complexity of the puzzle in its recursive nature remains. An algorithm employing graph theory to solve the N-Queens game is expected to be just as good efficient as the one described in this paper.

7.1 - Other Strategies

There is often more than one winning strategy to a given winning position. We have already seen a winning strategy for all boards of odd length, namely, placement of the first queen in

the middle, but there are other winning moves as well. Also, we know there is a winning strategy for $n = 2,4,6,8$. Is there a winning strategy that works with these and other even boards (we know that the 10x10 cannot be one of these boards)? If so, can it be easily stated? If not, is there an easy to implement strategy for just the 6x6 and 8x8 case?

It is not easy to see an end position from a typical empty board with even *n*, such as the 8x8 board. It may be possible to implement a near-optimal AI strategy at the beginning and switch to an analyzed winning regular strategy once the board becomes simple enough to analyze. For example, one might consider implementing a 'greedy' strategy in which one places a queens so that they attack the *maximum* number of free squares on the board. Similarly, one might attack the *minimum* number of free squares. It is unlikely that either strategy will be near-optimal, but perhaps a combination of the two might prove effective. It may be productive to set a machine against itself repeatedly so that it learns from its past successes and failures, thereby learning to switch from one strategy to the other at an appropriate time.

7.2 - Games with Other Chess Pieces and Other Boards

A variation of the N-Queens problem that has been popular is on the toroidal board, where the top and bottom edges wrap around, as do the left and right edges. The N-Queens game can be extended to such boards. The N-Queens problem on a rectangular board may prove to be more difficult to analyze than the original version. For a long enough rectangle, this game approaches the N-Rooks game, since the diagonals play a lesser role. The N-Queens game on hollow boards (with some squares that are marked unavailable from the beginning of the

game) is also an interesting variant. Whereas a regular board may have a winning strategy for the first player, the second player can use the 'holes' in the board to his advantage to win. For example if playing on a board with odd length, the second player can move to a 180^{o} rotation to the unavailable square so that the first player can no longer use the winning strategy outlined in Section 4.1.

Some more natural extensions of the game are played with other chess pieces. The N-Rooks game is trivial; it will always end after *n* moves. The N-Bishops and N-Knights games may be more interesting.

The N-Bishops game is really two games being played simultaneously. A bishop placed on a black square attacks only black squares, while a bishop placed on a white square attacks only white squares. So the game is really being played on two 'hollow' boards at once. The two boards are of equal size, but all the black squares have been removed from one and all the white squares have been removed from the other (Notice that each of these games is identical to the N-Rooks game played on a diamond shaped board.). To calculate the value of the game, we may find the Nimbers associated with each board and add them using nim addition. For a board with even *n*, we immediately note that the two games our board separates into are identical, up to a rotation or flip. So the nim values of both boards must be exactly the same; but $*x+*x=0$. So the N-Bishops game for even lengths is always a P-Position; that is, the second player can always win. In this case an explicit winning strategy can be stated: if the first player moves in, say, one of the black squares, the winning strategy for the second

player is to make the same move as his opponent, but in the white board. Note that this too is a tit-for-tat strategy, similar to the one for our N-Queens game with odd sides.

Another natural extension is to allow the players to place any chess piece on the board. Pieces like the knight make games like these more difficult to analyze. Placing a knight on the board so that it does not attack another piece already in place does not ensure that the knight itself will not be attacked. Thus we must add another rule i.e., the piece being placed cannot be attacked by any piece already on the board.

7.3 - Some Partizan Games

Let us amend our rules to the N-Queens game so as to make it *partizan*; the left and right options do not have to be identical in these games. A player can place a queen on any square that is not being attacked by his opponent's pieces. The rule is reasonable for it allows a player's pieces to be 'protected' if need be, and reflects the actual play of chess. Were this game to be played with multiple pieces, we would adopt normal chess convention where if for instance a king is in front of my rook, the rook does not attack any squares past the king since the king is blocking its path.

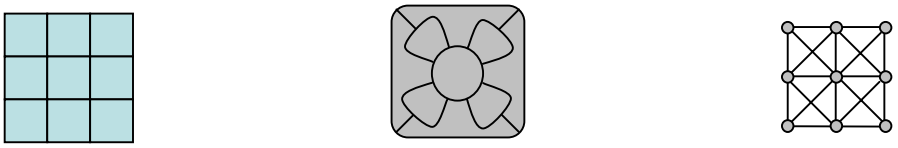


Figure 7.3.0 - Col and Snort on the 3x3 Chess Board

The partizan N-Kings game is really a variant of Snort [4 Pg. 145, 8 Pg. 91] in disguise. Snort is a map coloring game played between two players. The players Left and Right must

paint an area on the map Blue or Red respectively. The players cannot paint an area on the map if the area adjacent to it is painted by an opponent's color. Our partizan N-Kings game is really the same game, except that it allows fields or areas that are joined by corners to be adjacent. Figure 7.3.0 shows the 3x3 N-Queens game board, its representation as a Col map and the identical graph obtained my replacing an area in the diagrams with a vertex and a boundary between two areas with an edge between the vertices they represent. All Snort maps form planar graphs whereas all partizan N-Kings games played on boards of size 4x4 or more are really non-planar graphs. In fact, we may view any partizan N-Chess Piece game as a variation of Snort which can be played on non-planar graphs.

Another partizan game to consider is a variant of Col [4 Pg. 37, 8 Pg. 91]. Col is a map coloring game similar to Snort, where a player may not color a field or area if it is adjacent to an area of his own color. The equivalent version of Col in the partizan N-Chess piece game would allow a player to move to a square being attacked by his opponent but not by one of his own pieces.

7.4 - Extensions of Other One-Player Games

The N-Queens game derives from the N-Queens problem; in essence the N-Queens problem is the one-player version of the N-Queens game. It is possible to convert other solitaire puzzles and create interesting impartial or partizan games. An obvious example is peg solitaire. Each move in peg solitaire eliminates one peg from the game. The object of the game is to leave as few pegs as possible (just one if you can) on the board. A two player peg game can be devised similarly to our N-Queens game; the loser is the first player who cannot

move a peg. The object of the game is thus to force the opponent into this position. In general, if the object of a one-player puzzle with complete information and no chance moves is to maximize the number of steps it takes to complete that puzzle, it can be extended easily into a two-player game.

8 - Conclusions

The N-Queens problem, like most combinatorial games is difficult to play well. By applying Conway's [4,8] surreal numbers to this game, we can simplify a given position to an extent and make moves in the game accordingly.

N	1	2	3	4	5	6	7	8	9	10
Value of nxn board	*1	*1	*2	*1	*3	*1	*2	*3	*1	0

Table 8.0 - Nim Values of the Empty Board

The table above shows the such values for the empty boards of size 1,2,3,…,10. They show that the first player to move will always have a winning strategy all of the boards except the 10x10. The resulting sequence of numbers is peculiar and has no obvious pattern to predict values of larger boards.

The N-Queens problem is known to be a complex problem. It takes exponential time to find the complete solution set of the problem using standard backtracking algorithms. This problem is a subset of the N-Queens game, which makes the game even more complex. By finding more efficient algorithms to solve this game, we can also solve the N-Queens problem, and other games related to the N-Queens problem[10], more efficiently.

[10] Several games related to the N-Queens Problem are highlighted in Section 7.

Appendix

```
/* ========================================================================
Copyright 2002 Hassan A. Noon
CODE FOR N-QUEENS GAME
WRITTEN IN JAVA DEVELOPED BY SUN MICROSYSTEMS USING BREEZYGUI LIBRARIES
======================================================================== */

//////////////////////////////////////////////////////////////////////////
///     Coordinates class - Coordinates.java
//////////////////////////////////////////////////////////////////////////
abstract public class Coordinates extends Object
{
        private int x;
        private int y;

        // Constructors
        public Coordinates()
        {
                x = 0;
                y = 0;
        }
        public Coordinates(int a, int b)
        {
                x = a;
                y = b;
        }

        // Methods
        public int getX() {return x;}
        public void setX(int nmb) {x = nmb;}
        public int getY() {return y;}
        public void setY(int nmb) {y = nmb;}
}



//////////////////////////////////////////////////////////////////////////
///     Queen class - Queen.java
//////////////////////////////////////////////////////////////////////////
public class Queen extends Coordinates
{
        private boolean playerOne;

        // Constructors
        public Queen()
        {
                super();
                playerOne = true;
        }

        public Queen(int a, int b)
        {
                super(a,b);
                playerOne = true;
        }

        public Queen(int a, int b, int c)
        {
                super(a,b);
                setPlayer(c);
        }


        // Methods
        public boolean isPlayerOne() {return playerOne;}
        private void setPlayer(int nmb)
        {
                if (nmb == 1)
                {
                        playerOne = true;
                }
                else
                {
```

```
                        playerOne = false;
                }
        }

        public Object copy()
        {
                Queen copyQueen = new Queen(getX(),getY());
                copyQueen.playerOne = playerOne;
                return copyQueen;
        }
}



////////////////////////////////////////////////////////////////////////
///     Patch class - Patch.java
////////////////////////////////////////////////////////////////////////
public class Patch extends Coordinates
{
        private boolean checked;
        private boolean covered;

        // Constructors
        public Patch()
        {
                super();
                checked = false;
                covered = false;
        }

        public Patch(int a, int b)
        {
                super(a,b);
                checked = false;
                covered = false;
        }


        // Methods
        public boolean isitChecked() {return checked;}
        public boolean isitCovered() {return covered;}
        public void Checkit() {checked = true;}
        public void Coverit() {covered = true;}

        public Object copy()
        {
                Patch copyPatch = new Patch(getX(),getY());
                copyPatch.checked = checked;
                copyPatch.covered = covered;
                return copyPatch;
        }
}



////////////////////////////////////////////////////////////////////////
///     Board class - Board.java
////////////////////////////////////////////////////////////////////////
import java.util.*;

public class Board extends Object
{
        private String group;
        private Patch[][] board;
        private Vector queens = new Vector();
        private int[] boardqueens;
        // Constructors
        public Board()
        {
                group = "D4";
                board = new Patch[8][8];
                boardqueens = new int[8];
                for (int i=0; i < 8; i++)
                {
                        boardqueens[i] = 0;
```

```
                for (int j=0; j < 8; j++)
                {
                        board[i][j] = new Patch(i,j);
                }
        }
}

public Board(int n)
{
        group = "D4";
        board = new Patch[n][n];
        boardqueens = new int[n];

        for (int i=0; i < n; i++)
        {
                boardqueens[i] = 0;
                for (int j=0; j < n; j++)
                {
                        board[i][j] = new Patch(i,j);
                }
        }
}


// Accessors
public boolean isChecked(int i, int j) {return board[i][j].isitChecked();}
public boolean isCovered(int i, int j) {return board[i][j].isitCovered();}
public String groupType() {determineGroup(); return group;}


// Mutators
public void Check(int i, int j) {board[i][j].Checkit();}
public void Cover(int i, int j) {board[i][j].Coverit(); boardqueens[i] = j + 1;}
public void determineGroup()
{
        if ( (noFreePatches() % 2) == 1)
        {
                group = "I";
        }
        else
        {
                boolean horizontal = isH();
                boolean diagonal = isD();
                if (horizontal == true && diagonal == true)
                {
                        group = "D4";
                }
                else
                {
                        boolean vertical = isV();
                        if (horizontal == true && vertical == true)
                        {
                                group = "D2";
                        }
                        else
                        {
                                boolean diagonalp = isDp();
                                if (diagonal == true && diagonalp == true)
                                {
                                        group = "D2p";
                                }
                                else
                                {
                                        if (horizontal == true)
                                        {
                                                group = "H";
                                        }
                                        else
                                        {
                                                if (vertical == true)
                                                {
                                                        group = "V";
                                                }
                                                else
                                                {
                                                        if(diagonal == true)
```

```
                                                        {
                                                                group = "D";
                                                        }
                                                        else
                                                        {
                                                                if (diagonalp == true)
                                                                {
                                                                        group = "Dp";
                                                                }
                                                                else
                                                                {
                                                boolean rotOneEighty = isOneEighty();
                                                                if (rotOneEighty == false)
                                                                        {
                                                                        group = "I";
                                                                        }
                                                                        else
                                                                        {
                                                        boolean rotNinety = isNinety();
                                                                if (rotNinety == true)
                                                                                {
                                                                        group = "R90";
                                                                                }
                                                                                else
                                                                                {
                                                                        group = "R180";
                                                                                }
                                                                        }
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }
                }
        }
}

//Methods
public int Size()
{
        double dim =  board.length;
        return (int) dim;
}

public int noFreePatches()
{
        int tmp = 0;
        for (int i=0;i<board.length;i++)
        {
                for (int j=0;j<board.length;j++)
                {
                        if( isChecked(i,j) == false)
                        {
                                tmp = tmp +1;
                                /*Patch[] biggerArray = new Patch[iAmFree.length + 1];
                                for(int k = 0; k < iAmFree.length; k++)
                                {
                                        biggerArray[k] = iAmFree[k];
                                }
                                iAmFree = biggerArray;
                                Patch p = new Patch (i,j);
                                iAmFree[iAmFree.length - 1] = p;*/
                        }
                }
        }
        return tmp;
}

public void placeQueen(int i, int j, int player)
{
        if (isChecked(i,j) == false)
        {
                Queen placeThisQueen = new Queen (i,j,player);
                queens.addElement (placeThisQueen);
                checkPatches(i,j);
```

```
                }
        }

        private void checkPatches (int x, int y)
        {
                for (int i=0;i<board.length;i++)
                {
                        for (int j=0;j<board[y].length;j++)
                        {
                                if (i == x || j == y || x - i == y - j || x - i == j - y)
                                {
                                        Check(i,j);
                                }
                                if (i == x && j == y)
                                {
                                        Cover(i,j);
                                }
                        }
                }
        }

        public String queenArray ()
        {
                String a = "";
                for (int i = 0; i < boardqueens.length; i++)
                {
                        a = a + boardqueens[i] + "-";
                }
                return a;
        }

        public Object copy()
        {
                Board copyBoard = new Board(board.length);
                for (int i = 0; i < queens.size(); i++)
                {
                        copyBoard.queens.addElement(((Queen) queens.elementAt(i)).copy());
                }
                for (int i = 0; i < board.length; i++)
                {
                        copyBoard.boardqueens[i] = boardqueens[i];
                        for (int j = 0; j < board.length; j++)
                        {
                                copyBoard.board[i][j] = (Patch) board[i][j].copy();
                        }
                }
                return copyBoard;
        }

//---------- Is Board Horizontally Symmetrical ---------\\
        private boolean isH ()
        {
                boolean isSymm = false;
                for (int i=0; i< Size(); i++)
                {
                        for (int j=0; j< Size(); j++)
                        {
                               if(isChecked(i,j) == true && isChecked(i , Size() - j - 1 ) == true)
                                {
                                        isSymm = true;
                                }
                                else
                                {
                                        isSymm = false;
                                        i = Size();
                                        j = Size();
                                }
                        }
                }
                return isSymm;
        }
//-----------------------------------------------------\\
//----------- Is Board Vertically Symmetrical ----------\\
        private boolean isV ()
        {
                boolean isSymm = false;
```

```
                for (int i=0; i< Size(); i++)
                {
                        for (int j=0; j< Size(); j++)
                        {
                              if(isChecked(i,j) == true && isChecked( Size() - i - 1 , j ) == true)
                                {
                                        isSymm = true;
                                }
                                else
                                {
                                        isSymm = false;
                                        i = Size();
                                        j = Size();
                                }
                        }
                }
                return isSymm;
        }
//-------------------------------------------------------\\
//----------- Is Board Diagonally Symmetrical ----------\\
        private boolean isD ()
        {
                boolean isSymm = false;
                for (int i=0; i< Size(); i++)
                {
                        for (int j=0; j< Size(); j++)
                        {
                if(isChecked(i,j) == true && isChecked( Size() - j - 1  , Size() - i - 1 ) == true)
                                {
                                        isSymm = true;
                                }
                                else
                                {
                                        isSymm = false;
                                        i = Size();
                                        j = Size();
                                }
                        }
                }
                return isSymm;
        }
//-------------------------------------------------------\\
//----------- Is Board Symmetrical Diagonally ----------\\
        private boolean isDp ()
        {
                boolean isSymm = false;
                for (int i=0; i< Size(); i++)
                {
                        for (int j=0; j< Size(); j++)
                        {
                                if(isChecked(i,j) == true && isChecked( j , i ) == true)
                                {
                                        isSymm = true;
                                }
                                else
                                {
                                        isSymm = false;
                                        i = Size();
                                        j = Size();
                                }
                        }
                }
                return isSymm;
        }
//-------------------------------------------------------\\
//------- Is Board Symmetrical under 180 rotation ------\\
        private boolean isOneEighty ()
        {
                boolean isSymm = false;
                for (int i=0; i< Size(); i++)
                {
                        for (int j=0; j< Size(); j++)
                        {
                  if( isChecked(i,j) == true && isChecked(Size() - i - 1 , Size() - j - 1) == true)
                                {
                                        isSymm = true;
```

```
                                }
                                else
                                {
                                        isSymm = false;
                                        i = Size();
                                        j = Size();
                                }
                        }
                }
                return isSymm;
        }
//-------------------------------------------------------\\
//------- Is Board Symmetrical under 90 rotation -------\\
        private boolean isNinety ()
        {
                boolean isSymm = false;
                for (int i=0; i< Size(); i++)
                {
                        for (int j=0; j< Size(); j++)
                        {
                               if( isChecked(i,j) == true && isChecked(Size() - j - 1 , i) == true)
                                {
                                        isSymm = true;
                                }
                                else
                                {
                                        isSymm = false;
                                        i = Size();
                                        j = Size();
                                }
                        }
                }
                return isSymm;
        }
//-------------------------------------------------------\\

}



////////////////////////////////////////////////////////////////////////
///     StartGame class - StartGame.java
////////////////////////////////////////////////////////////////////////

import java.awt.*;
import BreezyGUI.*;

public class StartGame extends GBDialog
{
        Label sizeLabel = addLabel("Board Size", 1, 1, 1, 1);
        IntegerField sizeField = addIntegerField(8, 1, 2, 1, 1);
        Button playBtn = addButton ("Play",2,1,2,1);

        public int n;

        public StartGame(Frame f, int a)
        {
                super(f);
                setSize(150,75);
                setTitle ("Choose Board Size");
                n = a;
        }

        public void buttonClicked (Button buttonObj)
        {
                n = sizeField.getNumber();
                setDlgCloseIndicator ("Play");
                dispose();
        }

}



////////////////////////////////////////////////////////////////////////
///     My Program - game.java
```

```
//////////////////////////////////////////////////////////////////////
import java.awt.*;
import java.awt.Graphics;
import java.awt.Color;
import java.util.*;
import BreezyGUI.*;
import java.io.*;

public class game extends GBFrame
{
        Label aLabel = addLabel("", 1, 1, 2, 1);
        Label bLabel = addLabel("", 2, 1, 2, 1);
        Label cLabel = addLabel("", 3, 1, 2, 1);
        Label dLabel = addLabel("", 4, 1, 2, 1);
        Label eLabel = addLabel("", 5, 1, 1, 1);
        Label fLabel = addLabel("", 6, 1, 2, 1);
        Label gLabel = addLabel("", 7, 1, 1, 1);
        Label hLabel = addLabel("", 8, 1, 1, 1);
        Label iLabel = addLabel("", 9, 1, 1, 1);
        Label jLabel = addLabel("", 10, 1, 1, 1);
        Label kLabel =
addLabel("\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\
t\t\t\t\t", 11, 1, 1, 1);

        public Button newGame = addButton ("New Game", 5, 2, 1, 1);
        IntegerField colField = addIntegerField(1,     7, 2, 1, 1);
        IntegerField rowField = addIntegerField(1,     8, 2, 1, 1);
        public Button userMove = addButton ("Move",    9, 2, 1, 1);
        public Button gameValueButton = addButton ("Value of Game",    10, 2, 1, 1);
        public Button boardValueButton = addButton ("Value of Board",    11, 2, 1, 1);

        Board myBoard;
        int marginX = 20;
        int marginY = 20;
        int bSize = 0;
        Hashtable valueTable = new Hashtable();

//----------------- Playing the game ------------------\\

        // Get Board Size and set it up

        public void game()
        {
                StartGame gamedlg = new StartGame (this,bSize);
                gamedlg.show();
                if (gamedlg.getDlgCloseIndicator().equals ("Play"))
                {
                        if (bSize != gamedlg.n)
                        {
                                valueTable.clear();
                        }
                        bSize = gamedlg.n;
                        myBoard = new Board(bSize);
                        repaint();
                }
        }
//-----------------------------------------------------\\




//----------- Do this when Button Is Clicked -----------\\
        public void buttonClicked (Button buttonObj)
        {
                if (buttonObj == newGame)
                {
                        game();
                }
                else if (buttonObj == userMove)
                {
                        if (myBoard.isChecked(colField.getNumber() - 1,rowField.getNumber() - 1) ==
true || colField.getNumber() - 1 < 0 || rowField.getNumber() - 1 < 0 || colField.getNumber() - 1 >
myBoard.Size() || rowField.getNumber() - 1 > myBoard.Size())
                        {
                                messageBox ("Illegal Move");
                        }
```

```
                        else
                        {
                        myBoard.placeQueen (colField.getNumber() - 1,rowField.getNumber() - 1,1);
                        repaint();
                        }
                }
                else if (buttonObj == gameValueButton)
                {
System.out.println ("Board Size = " + bSize + " & Value of Game = " + myBoardValue(myBoard));
                        repaint();
                }
                else
                {
                        Board blankBoard = new Board(bSize);
System.out.println ("Board Size = " + bSize + " & Value of Board = " + myBoardValue(blankBoard));
                        repaint();
                }
        }
//-------------------------------------------------------\\



//---------------------- Draw Board ----------------------\\
        public void paint (Graphics g)
        {
                if (bSize != 0)
                {
                        for (int i=0;i<bSize;i++)
                        {
                                for (int j=0;j<bSize;j++)
                                {
                                        drawPatch(i,j);
                                }
                        }
                }
        }
//-------------------------------------------------------\\



//--------------Set and Draw a patch color -------------\\
        private void drawPatch (int x, int y)
        {
                Graphics asquare = getGraphics();
                if (myBoard.isCovered(x,y) == true)
                {
                        asquare.setColor(new Color (0,0,0));
                }
                else if ((x + y) % 2 == 0 && myBoard.isChecked(x,y) == false)
                {
                        asquare.setColor(new Color (153,153,153));
                }
                else if ((x + y) % 2 == 1 && myBoard.isChecked(x,y) == false)
                {
                        asquare.setColor(new Color (204,204,204));
                }
                else if ((x + y) % 2 == 0 && myBoard.isChecked(x,y) == true)
                {
                        asquare.setColor(new Color (40,40,40));
                }
                else if ((x + y) % 2 == 1 && myBoard.isChecked(x,y) == true)
                {
                        asquare.setColor(new Color (60,60,60));
                }
                asquare.fillRect(marginX + x*20, myBoard.Size()*20 - y*20, 20, 20);
        }
//-------------------------------------------------------\\



//-------------------- Recursively ----------------------\\
//------------- Find the Value of the Game --------------\\
        public int myBoardValue (Board inputBoard)
        {
```

```
        int returnThis = gameValue(inputBoard);
        return returnThis;
}



public int gameValue (Board thisBoard)
{
        int freeOptions = thisBoard.noFreePatches();
        String thisBoardtoString = thisBoard.queenArray();

        if (freeOptions == 0)
        {
                return 0;
        }
        else if(freeOptions == 1)
        {
                return 1;
        }
        else if(valueTable.get(thisBoardtoString) != null)
        {
                return ((Integer) valueTable.get(thisBoardtoString)).intValue();
        }
        else
        {
                int[] optionVals;
                String mygroup = thisBoard.groupType();
                if ( mygroup == "I" )
                {
                        optionVals = new int[freeOptions];
                        int counter = 0;
                        for (int i = 0; i < bSize; i++)
                        {
                                for (int j = 0; j < bSize; j++)
                                {
                                        if (thisBoard.isChecked(i,j) == false)
                                        {
                                                Board subBoard = (Board) thisBoard.copy();
        // Make a new Board with
                                                subBoard.placeQueen(i,j,1);
                        // a queen in the ith Patch
                                        int thisSubboardValue = gameValue(subBoard);
                valueTable.put(subBoard.queenArray(),new Integer (thisSubboardValue));
                                                optionVals[counter] = thisSubboardValue;
                                                counter = counter +1;
                                        }
                                }
                        }
                }
                else if (mygroup == "D4")
                {
                        optionVals = new int[freeOptions];
                        int counter = 0;
                        for (int i = 0; i < bSize / 2; i++)
                        {
                                for (int j = i; j < bSize / 2; j++)
                                {
                                        if (thisBoard.isChecked(i,j) == false)
                                        {
        Board subBoard = (Board) thisBoard.copy();              // Make a new Board with
        subBoard.placeQueen(i,j,1);                             // a queen in the ith Patch
                                        int thisSubboardValue = gameValue(subBoard);
                valueTable.put(subBoard.queenArray(),new Integer (thisSubboardValue));
                                                optionVals[counter] = thisSubboardValue;
                                                counter = counter +1;
                                        }
                                }
                        }
                }
                else if (mygroup == "D2" || mygroup == "R90")
                {
                        optionVals = new int[freeOptions];
                        int counter = 0;
                        for (int i = 0; i < bSize / 2; i++)
                        {
                                for (int j = 0; j < bSize / 2; j++)
```

```
                        {
                                if (thisBoard.isChecked(i,j) == false)
                                {
                                        Board subBoard = (Board) thisBoard.copy();
// Make a new Board with
                                        subBoard.placeQueen(i,j,1);
                // a queen in the ith Patch
                                int thisSubboardValue = gameValue(subBoard);
        valueTable.put(subBoard.queenArray(),new Integer (thisSubboardValue));
                                        optionVals[counter] = thisSubboardValue;
                                        counter = counter +1;
                                }
                        }
                }
        }
        else if (mygroup == "D2p")
        {
                optionVals = new int[freeOptions];
                int counter = 0;
                for (int i = 0; i < bSize / 2; i++)
                {
                        for (int j = i; j < bSize - i; j++)
                        {
                                if (thisBoard.isChecked(i,j) == false)
                                {
                                        Board subBoard = (Board) thisBoard.copy();
// Make a new Board with
                                        subBoard.placeQueen(i,j,1);
                // a queen in the ith Patch
                                int thisSubboardValue = gameValue(subBoard);
        valueTable.put(subBoard.queenArray(),new Integer (thisSubboardValue));
                                        optionVals[counter] = thisSubboardValue;
                                        counter = counter +1;
                                }
                        }
                }
        }
        else if (mygroup == "R180" || mygroup == "H")
        {
                optionVals = new int[freeOptions];
                int counter = 0;
                for (int i = 0; i < bSize ; i++)
                {
                        for (int j = 0; j < bSize / 2; j++)
                        {
                                if (thisBoard.isChecked(i,j) == false)
                                {
                                        Board subBoard = (Board) thisBoard.copy();
// Make a new Board with
                                        subBoard.placeQueen(i,j,1);
                // a queen in the ith Patch
                                int thisSubboardValue = gameValue(subBoard);
        valueTable.put(subBoard.queenArray(),new Integer (thisSubboardValue));
                                        optionVals[counter] = thisSubboardValue;
                                        counter = counter +1;
                                }
                        }
                }
        }
        else if (mygroup == "V")
        {
                optionVals = new int[freeOptions];
                int counter = 0;
                for (int i = 0; i < bSize / 2; i++)
                {
                        for (int j = 0; j < bSize ; j++)
                        {
                                if (thisBoard.isChecked(i,j) == false)
                                {
                                        Board subBoard = (Board) thisBoard.copy();
// Make a new Board with
                                        subBoard.placeQueen(i,j,1);
                // a queen in the ith Patch
                                int thisSubboardValue = gameValue(subBoard);
        valueTable.put(subBoard.queenArray(),new Integer (thisSubboardValue));
                                        optionVals[counter] = thisSubboardValue;
```

```
                                                counter = counter +1;
                                        }
                                }
                        }
                }
                else if (mygroup == "D")
                {
                        optionVals = new int[freeOptions];
                        int counter = 0;
                        for (int i = 0; i < bSize ; i++)
                        {
                                for (int j = 0; j < bSize - i; j++)
                                {
                                        if (thisBoard.isChecked(i,j) == false)
                                        {
                                                Board subBoard = (Board) thisBoard.copy();
        // Make a new Board with
                                                subBoard.placeQueen(i,j,1);
                                // a queen in the ith Patch
                                        int thisSubboardValue = gameValue(subBoard);
                        valueTable.put(subBoard.queenArray(),new Integer (thisSubboardValue));
                                                optionVals[counter] = thisSubboardValue;
                                                counter = counter +1;
                                        }
                                }
                        }
                }
                else //if (mygroup == "Dp")
                {
                        optionVals = new int[freeOptions];
                        int counter = 0;
                        for (int i = 0; i < bSize ; i++)
                        {
                                for (int j = 0; j < i; j++)
                                {
                                        if (thisBoard.isChecked(i,j) == false)
                                        {
                                                Board subBoard = (Board) thisBoard.copy();
        // Make a new Board with
                                                subBoard.placeQueen(i,j,1);
                                // a queen in the ith Patch
                                        int thisSubboardValue = gameValue(subBoard);
                        valueTable.put(subBoard.queenArray(),new Integer (thisSubboardValue));
                                                optionVals[counter] = thisSubboardValue;
                                                counter = counter +1;
                                        }
                                }
                        }
                }
                return theMex(optionVals);
        }
    }
//-------------------------------------------------------\\
//-------------------------------------------------------\\



//---------------- Are 2 Int Arrays Equal --------------\\
    public boolean boardsAreEqual (int[] a, int[] b)
    {
        boolean isit = true;
        for (int i = 0; i < a.length; i ++)
        {
                if (a[i] != b[i])
                {
                        isit = false;
                }
        }
        return isit;
    }
//-------------------------------------------------------\\
```

```
//------------------ Calculate the MEX ------------------\\
        public int theMex (int[] b)
        {
                int mex = 0;
                for (int i = 0; i < b.length; i ++)
                {
                        for (int j = 0; j < b.length; j ++)
                        {
                                if (b[j] == mex)
                                {
                                        mex = mex + 1;
                                }
                        }
                }
                return mex;
        }
//-------------------------------------------------------\\



//---------- Is Board Horizontally Symmetrical ---------\\
        public boolean isH (Board inBoard, Patch[] freePatches)
        {
                boolean isItH = false;
                for (int i=0; i<freePatches.length; i++)
                {
        if(inBoard.isChecked( freePatches[i].getX() , bSize - freePatches[i].getY() - 1 ) == false)
                        {
                                isItH = true;
                        }
                        else
                        {
                                isItH = false;
                                break;
                        }
                }
                return isItH;
        }
//-------------------------------------------------------\\



//--------------- BreezyGUI Window Stuff ---------------\\
        public static void main (String[] args)
        {
                Frame frm = new game();
                frm.setSize (400,250);
                frm.setVisible (true);
        }
}
//-------------------------------------------------------\\
```

Bibliography

1. Bruce Abramson, Moti Yung, *Divide and Conquer under Global Constraints: A Solution to the N- Queens Problem*, Journal of Parallel and Distributed Computing, **6**, 649-662 (1989)

2. M. Bezzel, *Berliner Schachgesellschaft*, **3** (1848), page 363

3. A. A. Bruen, R. Dixon, *The n-queen problem*, Discrete Math. **12**, (1975), 393-395

4. E. R. Berlekamp, J. H. Conway, R. K. Guy, *Winning Ways for Your Mathematical Plays*, Volume 1, Second Edition, A K Peters, Ltd., 2001

5. Paul J. Campbell, *Gauss and the eight queens problem; a study in miniature of the propogation of historical error*, Historia Math., **4** (1977), No. 4, 397-404

6. G. Cantor, *Über unendliche, lineare Punktmannigfältigkeiten, Arbeiten zur Mengenlehre aus dem Jahren 1872-1884*. Leipzig, Germany: Teubner-Archiv zur Mathematik, 1884.

7. Chapel Hill Conference on Combinatorial Mathematics and Its Applications, *Sphere packing, coding metrics, and chess puzzles*, Proceedubg of the second Chapel Hill Conference on Combinatorial Mathematics and Its Applications (1970) 176-189

8. J. H. Conway, *On Numbers and Games*, Second Edition, A K Peters, Ltd., 2001

9. Cengiz Erbas, Murat M. Tanik, *Linear congruence equations for the solutions of the N-Queens problem*, Information Processing Letters, **41** (1992) 301-306

10. C. Erbas, *Different perspectives of the N-Queens problem*, Proc. ACM Computer Science Conf., Kansas City, MO

11. B. J. Falkowski, L Schmitz, *A note on the queens' problem*, Inform. Process Lett. **23** (1986) 39-46

12. L. R. Foulds, D G. Johnston, *An application of Graph Theory and Integer Programming: Chessboard Non-attacking Puzzles*, Mathematics Magazine, Vol. 57, No. 2, March 1984, 95-104

13. Aviezri S. Fraenkel, Hans Herda, *Never rush to Be First in Playing Nimbi*, Math. Magazine, Vol. 53, No. 1, January 1980

14. A. S. Fraenkel, A. Jaffray, A. Kotzig, G. Sabidussi, *Modular Nim*, Theor. Comput. Sci. (Math Games) **143** (1995) 319-333

15. D. Fremlin, *Well-Founded Games*, Eureka; the Archimedeans' journal, Vol. 36, 1973, 33-37

16. J. Giusburg, *Gauss's arithmetization of the problem of 8 queens*, Scripta Math. **5** (1939) 63-66

17. P. M. Grundy, *Mathematics and Games*, Eureka, **2**(1939) 6-8; reprinted *ibid* **27**(1967) 9-11

18. Jun Gu, Rok Sosic, *A Polynomial Time Algorithm for the N-Queens Problem*, SIGART bulletin, Vol. 1, No. 3 (1990) 7-11

19. Olof Heden, *Maximal partial spreads and the modular n-queen problem*, III, Discrete Math., **243** (2002) No. 1-3, 135-150

20. O. Heden, *On the modular n-queen problem*, Discrete Math., **102** (1992) 155-161

21. E. J. Hoffman, J. C. Loessi, R. C. Moore, *Construction for the solution of the n-queens problem*, Math. Mag. **42** (1969) 66-72

22. David H. Hollander, *An unexpected two-dimensional space-group containing seven of the twelve basic solutions to the eight queens problem*, J. Recreational Math. **6** (1973) No. 4, 287-291

23. T. Kløve, *The modular n-queen problem*, Discrete Math **19** (1981) 289-291

24. Donald E. Knuth, *Surreal Numbers*, Addison-Wesley, 1974

25. Daniel Loeb, *Fundamental solutions of the eight queens problem*, BIT **22** (1982), No. 1, 42-52

26. MSRI publications, *Games of no Chance*, Cambridge Univ. Press, Cambridge, 1994

27. Patric R. J. Östergård, William D. Weakley, *Values of Domination Numbers of the Queen's Graph*, The electronic Journal of Combinatorics, **8** (2001), #R29

28. M. Reichling, *A simplified solution of the N-queens' problem*, Inform. Process Lett **25** (1987) 253-255

29. B. Russel, A. N. Whitehead, B. *Principia Mathematica*. New York: Cambridge University Press, 1927.

30. R. P. Sprague, *Über mathematische Kampfspiele*, Tôhoku Math J., **41**(1935-6), 438-444; Zbl. **13**, 290

31. Georg Schrage, *The eight queens problem as a strategy game*, Int. J. Math. Educ. Sci. Technol., 1989, Vol. 17, No. 2, 143-148

32. I. Rivin, I. Vardi, P. Zimmerman, *The n-queens problem*, Amer. Math. Monthly **101** (1994) 629-638

33. Rodney W. Topor, *Fundamental Solutions of the Eight Queens Problem*, BIT, (1982), 42-52

34. *Eric Weisstein's World of Mathematics* (http://mathworld.wolfram.com/QueensProblem.html)