Some Observations about the n-Queens Problem in Higher Dimensions

Jeremiah Barr* Shrisha Rao†
Department of Computer Science
Mount Mercy College
1330 Elmhurst Drive NE
Cedar Rapids, IA 52402

March 14, 2004

Abstract

A well-known chessboard problem is that of placing eight queens on the chessboard so that no two queens are able to attack each other. (Recall that a queen can attack anything on the same row, column, or diagonal as itself.) This problem is known to have been studied by the mathematician Gauss, and can be generalized to an $n \times n$ board, where $n \geq 4$. It can further be generalized to a problem in higher dimensions as well.

We use a generalized implementation of the well-known backtracking algorithm to generate solution counts for certain values of n and higher dimensional spaces.[1] We discuss the problem of generalizing the algorithm and provide details of our implementation. Based on our solution counts, we offer hypotheses for the behavior of the number of possible solutions in higher dimensions, and offer suggestions for further work.

*jrbarr@mmc.mtmercy.edu
†srao@mmc.mtmercy.edu

[1] Sample solutions and code showing our implementation can be viewed at the URL `http://www2.mtmercy.edu/academicsdir/compscience/projects/nqueens/`.

1 Introduction

The 8-queens problem is a well known chessboard problem, whose constraints are to place eight queens on a normal chessboard in such a way that no two attack each other, under the rule that a chess queen can attack other pieces in the same column, row, or diagonal. This problem can be generalized to place n queens on an n by n chessboard, otherwise known as the n-queens problem. The mathematicians Gauss and Polya studied this problem [2], and Ahrens [1] showed that for all $n \geq 4$, solutions exist. This problem can be further generalized to d-dimensions, where two queens attack one another if they lie on a common hyperplane [4]. This problem can be described as the n-queens problem in d-dimensions. (The traditional 8-queens problem, as described above, is 2-dimensional.)

2 A Brief Survey of the Literature

A number of versions of the n queens problems exist. For example, the modular version is the same as the traditional problem, except for the fact that diagonals on the board do not stop at the edge of the board, but continue and wrap around to the opposite side(s) [6]. This results in fewer solutions for a given n. For modular boards, the concept of a partial solution where less than n queens are placed on a board has been introduced. For most values of n, only partial solutions can be found, except for the case where $gcd(n, 6) = 1$ for a modular board of size n by n, conditions which Polya and Hurwitz proved can hold n queens [7].

Numerous methods for solving the problem exist as well. An interesting manner of constructing an n queens board is discussed in [2], where queens board solutions are derived from magic square solutions. Another method involving no combinatorial searching, and also no computer time, is discussed by Bernhardsson [5], where queens are placed based on if n is even and not of form $6k + 2$, if it is even and not of the form $6k$, or if it is odd. However, the traditional method is called backtracking, which is the only known way to find every solution for a given board of size n in d-dimensional space. This method is discussed more below.

3 Describing the Queens' Attack Lines in Cartesian d-Space

3.1 In Two Dimensions

On a 2-dimensional grid, a queen can attack along two axes, and along two diagonals. For a queen located at coordinates (i, j), the axes are given by $x = i$ and $y = j$. The diagonals are $x - i = y - j$ and $x - i = j - y$. Figure 1 shows this. These equations provide the basis for testing a position on the actual board for validity, where a non-valid position is attackable and a valid position is not attackable.

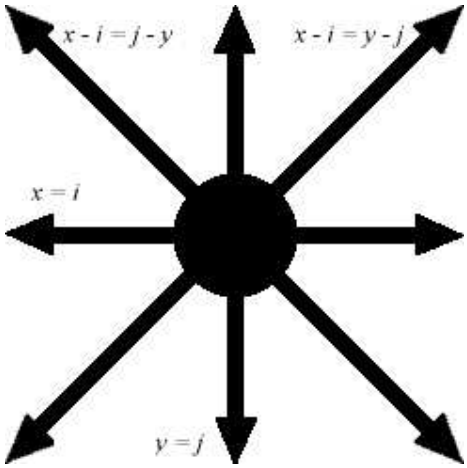

Figure 1: The directions of attack and their equations.

3.2 In Three Dimensions

In a 3-dimensional chess grid, a queen can be thought of as located in three 2-dimensional grids (corresponding to the XY plane, the XZ plane, and the YZ plane). There thus are six 2-dimensional diagonals and three axes. For a queen at (i, j, k), the diagonals are the previous two, as well as $x - i = z - k$, $x - i = k - z$, $y - j = z - k$, and $y - j = k - z$. The axes of course are $x = i$, $y = j$, and $z = k$. There are also four 3-dimensional diagonals. Their equations are, for a queen at (i, j, k):

- $x - i = y - j = z - k$
- $x - i = y - j = k - z$

- $x - i = j - y = z - k$
- $i - x = y - j = z - k$

3.3 In Four Dimensions

Similarly, in a 4-dimensional chess hyperspace, the queen is in four 3-dimensional subspaces (i.e., cubes) that are projections to the lower dimension. For each of these subspaces, equations to attack diagonals in three, two, or one dimensions may be found as previously. There also are eight 4-dimensional diagonals, given for a queen at i, j, k, l in a 4-dimensional space $XYZW$ by:

- $x - i = y - j = z - k = w - l$
- $x - i = y - j = z - k = l - w$
- $x - i = y - j = k - z = w - l$
- $x - i = j - y = z - k = w - l$
- $i - x = y - j = z - k = w - l$
- $i - x = y - j = z - k = l - w$
- $i - x = y - j = k - z = w - l$
- $i - x = j - y = z - k = w - l$

3.4 Counting the Number of d-Dimensional Attack Diagonals

In general, we can consider the problem of enumerating the the number of d-dimensional lines of attack for a queen in d dimensions to be the same as that of counting the number of d-dimensional diagonals in a hypercube. Using a topological approach, we see that for all d, each vertex in a d-dimensional hypercube has exactly one polar reciprocal [8]. A d-dimensional diagonal is a line connecting a vertex and its polar reciprocal. Since there are 2^d vertices in a hypercube, there must be 2^{d-1} d-dimensional diagonals in a d-dimensional hypercube. This tells us that a queen in d dimensions has 2^{d-1} d-dimensional attack lines.

3.5 The Equations for the d-Dimensional Attack Lines

The equations for the d-dimensional attack diagonals for a queen at location $i_1, i_2, \ldots, i_d$ in a d-dimensional chess hyperspace $X_1, X_2, \ldots, X_d$ are of the form:

$$\pm(x_1 - i_1) = \pm(x_2 - i_2) = \ldots = \pm(x_d - i_d)$$

(Compare with Subsections 3.1, ff.)

There are obviously 2^d such equations in all. However, noting that changing the signs on all the terms gives us a new equation with the same meaning as the one changed, there are 2^{d-1} equations that are distinct, which is a nice agreement with our topological reasoning.

The distinct equations for the d-dimensional attack diagonals may thus be given by:

- $x_1 - i_1 = x_2 - i_2 = \ldots = x_d - i_d$
- $x_1 - i_1 = x_2 - i_2 = \ldots = i_d - x_d$
  $\vdots$
- $i_1 - x_1 = i_2 - x_2 = \ldots = x_d - i_d$

4 The Algorithm for Finding Solutions to the n-Queens Problem

The traditional method of finding solutions for the n-queens problem is called backtracking, which is another name for the well-known technique of *depth-first search* (DFS) [9]. Goodrich and Tamassia [3] describe backtracking in terms of graph theory, where vertices and their incident edges within a graph are traversed via this method. As described by Goodrich and Tamassia, the algorithm works by starting on a source vertex v_1 of some graph G, which contains the n vertex set (v_1, v_2,...,v_n) and all incident edges. First, the algorithm traverses an arbitrary incident edge e that is unexplored, arriving at vertex v_i, where i is of any value of the set (2, 3,...,n). If v_i has not been visited, the previous step is repeated of traversing an unexplored incident edge of v_i to the next vertex. However, if v_i has been visited, e is traversed back to v_1, and a different unexplored incident edge is traversed. This backwards traversal is the actual backtrack.

In the context of the queens problem, the search space of possible queen positions on the board is representable as a graph.

4.1 In Two Dimensions

The way this algorithm is used for the n queens problem is by first placing a queen at the position of the board where the first column and first row intersect. In the 2-dimensional case, the algorithm traverses to the next row, first column, and attempts to place a queen at this position. Since both queens are in the same column, the test fails, and so the algorithm traverses to the next column and tries again, and continues to do so until a valid position is found in the row. Once one is found, the algorithm traverses to the next row and repeats the process of searching for a valid column within that row to place a queen. However, if no valid columns are found within a given row, except for the first row, the algorithm must traverse to the previous row and attempt to place the queen for that row in the next valid column.

This is the backtrack move, and it is repeated until either

- a previously placed queen is replaced into a valid column for its row, or
- all queens have been placed at their farthest valid column position in their respective row.

A solution has been found when the n^{th} queen is placed on the n^{th} row. All solutions have been found when the first queen is replaced in the n^{th} column of the first row, and all subsequent queens are in their farthest valid column positions. Note that the orientation of board traversal and the row-by-row queen placement are not the only ways to find solutions for the 2-dimensional case; queens can be placed column-by-column as well. The same solutions will be found, as well as the same number of solutions.

4.2 In Multiple Dimensions

Definition 4.1. *$depth_i \triangleq$ the label of a specific i-dimensional hyperplane, where i belongs to the set $0, 1, \ldots, d$. For example, $depth_3$ is the label for a 3-dimensional hyperplane.*

In three dimensions, instead of attempting to place queens by traversing to each successive row, the algorithm tests for placing queens by traversing to each successive $depth_i$. A row ($depth_1$) corresponds to the X axis, a column ($depth_2$) corresponds to the Y axis, and $depth_3$ corresponds to the Z axis. Once the algorithm traverses a $depth_3$, instead of attempting to place queens at positions on columns, it tests positions by columns and rows. Essentially

each $depth_3$ contains a 2-dimensional $n \times n$ board, which is iterated through in a column-by-column, row-by-row fashion. The manner of traversing forward or backward to locate a valid position is the same as the 2-dimensional case. The first queen is placed at the first row, column, and $depth_3$. Next, the algorithm traverses to the next $depth_3$ and attempts to place the second queen at the first row and column. Since this position is on the $z = k$ axis of the previously placed queen, it is not valid. Therefore, the algorithm traverses to the next column, and then the next column, and so on until the n^{th} row and column are reached, or a valid position is located. If a valid position is found, a queen is placed there, and the algorithm traverses to the next $depth_3$, performing the same column-by-column, row-by-row traversal and testing for that $depth_3$, just as before. If no valid positions are found at any $depth_3$ except for the first one, the algorithm backtracks to the previous $depth_3$ and moves the queen for that $depth_3$ to the next valid position. A solution is found when the n^{th} queen is placed in the n^{th} $depth_3$, and all solutions are found when the queen at the first $depth_3$ is at the n^{th} row and column, and all subsequent queens are at the farthest row and column positions possible. Similarly to the 2-dimensional case, the orientation of the board can be changed to where queens are placed in each successive row or column instead of each successive $depth_3$.

For d dimensions, generalize column as $depth_1$, and row as $depth_2$. The first queen is placed at the first $depth_1, depth_2, \ldots, depth_d$. The algorithm traverses to the next $depth_d$, testing for validity at each position of that $depth_d$'s $(d-1)$-dimensional hyperplane, backtracking to the queen at the previous $depth_d$ if no valid positions are found. However, if a valid position is found, the queen is placed there and the algorithm traverses to the next $depth_d$ to place the next queen. This process is repeated for queens 2 through n. A solution is found when the n^{th} queen is placed at the n^{th} $depth_d$. All solutions are found when the first queen is at the n^{th} $depth_1, depth_2, \ldots, depth_{d-1}$, and queens 2 through n are at the farthest possible $depth_1, depth_2, \ldots, depth_{d-1}$ in their respective $depth_d$. As mentioned in each of the previous cases, the orientation of board traversal and queen placement can be changed to where queens are placed by row, column, ..., or $depth_{d-1}$ instead of by $depth_d$.

5 The Solutions to the n-Queens Problem in d Dimensions

The numbers of n-queens solutions in d dimensions						
d	$n=3$	$n=4$	$n=5$	$n=6$	$n=7$	$n=8$
2	0	2	10	4	40	92
3	72	7196	98106	205444488	60754055080	–
4	4632	5313008	11353276978	–	–	–
5	198096	2268218096	–	–	–	–
6	14348907	–	–	–	–	–

Table 1: n-queens solutions—dashes indicate where the count is too large.

The solutions enumerated are not unique, in the sense that rotations and mirror-images are considered different solutions. (In this sense, the standard 2-dimensional problem has 92 solutions, of which 12 are unique.)

In Table 1, the general tendency of the number of solutions for a board of a given size n and dimension d is that it is greater than the number of solutions for a board of size m and dimension d, where m is less than n.

Also of note is the tendency for the number of solutions for a board of a given size n and dimension d to be greater than the number of solutions for a board of size d and dimension n.

Given the following:

Definition 5.1. $Q(n,d) \triangleq$ *the number of solutions for an n-queens problem in d dimensions.*

We can express our observations as follows:

Conjecture 5.2. $Q(n,d) > Q(m,d), \forall n > m,\ d > 2.$

Conjecture 5.3. $Q(n,d) > Q(d,n), \forall n > d.$

6 The Algorithmic Implementation of the n-Queens Problem in d Dimensions

The implementation of the algorithm used to find the solution counts in Table 1 follows the traditional algorithm described in Section 4 closely.

6.1 Representing a Location in d-Dimensional Space in a Single-Dimensional Array

Our implementation works by "unrolling" the d-dimensional chess space onto a long 1-dimensional array of size n^d.

For the standard 2-dimensional case, the relationship between the row and column position and the location in the 1-dimensional array may be given as follows.

Remark 6.1. *Given that the coordinates of a position on a 2-dimensional board are i_2 (denoting the row), and i_1 (denoting the column), the equation for calculating the position's single coordinate p in the array is:*

$p = i_1 + (i_2 \times n)$

In a generalized d-dimensional situation,

Remark 6.2. *If i_j, where $j \in \{1, 2, \ldots, d\}$, are the coordinates of a location in d-dimensional space, the equation for the single coordinate in the 1-dimensional array is:*

$$p = \sum_{j=1}^{d} \left(i_j \times n^{j-1} \right)$$

6.2 The Implementation Structures

The implementation makes use of the following structures to represent a given problem of n queens in d dimensions:

```
typedef int cPos;

struct cQueen {
    cPos pos;
};

struct cBoard {
    int size, dimensions;
    list < cQueen > queens;
    list < cPos > *lists;
    ...
};
```

These three—`cPos`, `cQueen`, and `cBoard`—respectively represent:

- individual positions on the board;
- queens placed at a certain position; and
- the board with its size n, d dimensions, placed queens, and each queen position's attack list.

6.3 The `cPos` Structure

Several things need to be noted about the use of these structures within the implementation. First, `cPos` is only a single integer value. The reason for this is that the possible queen positions are stored as though they were 1-dimensional, but they are used in a d-dimensional space. This usage is described more below (see Subsection 6.4). Thus new implementations of the positions for each dimension are not necessary.

6.4 The `cBoard` Structure

Also of note are the `queens` list and `*lists` variables stored in the `cBoard` structure. The `queens` list is merely a bi-directional linked list that is used to store the queens that have been placed on the board. When a valid position is found for a queen on a hyperplane, it is added to this list. The `queens` list is essentially the permutation of a given solution. As shown in [2], solutions for the 2-dimensional problem can be represented by a permutation $(a_1, a_2, \ldots, a_i, \ldots, a_n)$, where $i \in \{1, 2, \ldots, n\}$. i corresponds to the row number of a queen, and a_i corresponds to the column number of a queen. This permutation representation is extended to represent more than two dimensions in the implementation, where i still belongs to the same set but corresponds to the $depth_d$ of a queen, and a_i corresponds to the queen's position within the board.

The `*lists` variable is a slightly more complicated member of `cBoard`. It is a pointer to a sequence of lists, and so it is used as an array. This array contains n^d elements, where a single element corresponds to a single position on the board. Each element is a list that stores the hypothetical positions that a queen could attack if it were placed on that element's corresponding position.

Shown below is an example of how `*lists` elements are populated for the 2-dimensional case:

```
int row1, col1, row2, col2;
cPos tempPos;
```

```
cBoard board;
for (row1 = 0; row1 < board.size; row1++)
    for (col1 = 0; col1 < board.size; col1++)
        for (row2 = 0; row2 < board.size; row2++)
             for (col2 = 0; col2 < board.size; col2++)
                  if ((row1 == row2) || (col2 == col1) ||
                     (abs(row1 - row2) == abs(col1 - col2)))
                     {
                         tempPos = row2*board.size + col2;
                         board.lists[row1*board.size
                                     + col1].push_back(tempPos);
                     }
```

The list population works by iterating through each position on the board for both the position corresponding to the current list element and the positions that a queen located there can attack. The if-statement is the implementation of the queen attack position equations shown in Section 3. It tests if the positions lie on the same X-axis, Y-axis, and diagonals. Notice the use of the absolute value function, `abs`, in the diagonal equality test. The reason for using the absolute value is to cover both 2-dimensional diagonals in a single test instead of two. Next, given that the `if`-statement returns true to indicate that a queen at `row1`, `col1` can attack the position at `row2`, `col2`, the position is stored in temporary position variable `tempPos` and is then added to the current element in the board's `lists` array's list.

6.5 The Placement of Queens in the Implementation

The actual method for board traversal and queen placement in the implementation is via a recursive function. In [3], a recursive algorithm for traversing every vertex in a graph G is discussed. Although the purpose of traversing the queens board is different from that given there, recursion is the method used by this implementation of the backtracking algorithm as well. The code for the function is:

```
void placeQueens(cBoard *board, int depth, unsigned int* count)
{
    int boardPos;
    static int maxPos = (int)pow((float)board->size,
                                 (float)board->dimensions - 1.0f);
    cQueen tempQueen;
    for(boardPos = 0; boardPos < maxPos; boardPos++)
    {
```

```
        tempQueen.pos = depth * maxPos + boardPos;
        if (valid(board, tempQueen) == 1)
        {
                board->queens.push_back(tempQueen);
                if (depth == (board->size - 1))
                {
                    *count = *count + 1;
                   ...
                }else
                   placeQueens(board, depth + 1,  count);
                board->queens.pop_back();
        };
    };
};
```

For parameters, this function takes a pointer to the `cBoard` structure being used, the current $depth_d$, and a pointer to a counter that keeps track of the number of solutions found, respectively. The variable *boardPos* is used to iterate through positions of the current hyperplane at `depth`, and `maxPos` is the $n^{(d-1)th}$ position on that hyperplane. `tempQueen` is the queen to be placed. As the function iterates through the positions of `depth`'s $(d-1)$-dimensional hyperplane, it assign's `tempQueen`'s `pos` member to the current position on the board. Next, valid is called to test if `tempQueen` is in an attacked position. If it is not, `tempQueen` is added to the list of placed queens, and if depth is the final $depth_d$, the solution counter is incremented by one. Otherwise `placeQueens` is called again recursively for the next $depth_d$. Finally, once the algorithm is done with the queen for this position on depth, it is removed from the list.

6.6 Implementation Optimizations

A major performance optimization for this implementation occurs during the population of attacked positions in the `cBoard` structure's `lists` array's population stage, discussed in Subsection 6.4. Since the recursive algorithm places queens on a $depth_d$ by $depth_d$ basis, the positions of attack that exist on the current $(d-1)$-dimensional hyperplane of a given position do not need to be computed and stored. Therefore, these positions of attack are not stored for a given position. This shrinks the list, saving large amounts of memory, and also computation time when the `valid` function iterates through the previously placed queens' positions' attack lists.

7 Potential changes to the algorithmic implementation of the n-Queens problem in d dimensions

A number of performance oriented changes could be made to the algorithmic implementation described above. The authors of this paper treat the problem of finding attack positions for a queen in 3 or more dimensions in a manner quite similar to that in the well-known 2-dimensional board. If there are better methods, it is possible that much computational time would be saved, resulting in a greater range of values for which $Q(n, d)$ (see Definition 5.1) could be computed.

Also of note is the single-threaded nature of the current implementation. If the implementation called the `placeQueens` (see Section 6.5) function for each position in the $(d-1)$-dimensional hyperplane for the first queen, and the algorithm only found the solutions for when the first queen was in a single position and did not move it to the next position, all solutions could be found by calls to the function. If each function call was in a single thread, n^{d-1} threads—each finding solutions for the first queen at a specific starting position—could find the total number of solutions. Similarly, the threads could be distributed and ran amongst multiple machines on a network. The only issue would be keeping the solution counter in sync between threads.

Obviously, if another algorithm besides backtracking is discovered that can find all possible solutions for a given n and d, and it happens to be faster, this would be changed. However, the thoroughness of the backtracking algorithm has yet to be matched by any other method.

8 Conclusions and Suggested Work

We have generalized the standard n-queens problem on a two-dimensional board into a problem of placing n queens in a d-dimensional chess grid. It is trivial to observe that such solutions must always exist for all $n \geq 4, d \geq 2$, since we know [1] that solutions always exist for $n \geq 4, d = 2$, and since the existence of a solution for a given n in r dimensions always implies the existence of solutions in all dimensions $t > r$. (An r-dimensional solution may just be placed in one suitable fragment of a t-dimensional chess space.)

However, it is far from obvious that non-trivial solutions (those that cannot be projected onto lower-dimensional subspaces) always exist for all $n \geq 4, d \geq 2$. A natural extension of the known result would be to prove that they do. It is interesting in this regard to note that whilst there is no solution for $n = 3, d = 2$, there is one for $n = 3, d = 3$.

We also have stated two conjectures earlier (see Conjectures 5.2, 5.3) suggested by our counts for the number of solutions for various small n, d values. Proofs or counter-examples, and similar results, would be interesting.

More powerful computers, better optimized code (e.g., using distributed computation or multi-threading), or faster algorithms might conceivably fill more of Table 1. It is known that the complexity of finding all solutions for $d = 2$ is $O(n!)$. It would be interesting to find out if this result can be matched or even exceeded in higher dimensions.

Acknowledgement

The authors would like to thank K. R. Knopp for useful discussions on this topic.

References

[1] Ahrens, W. Mathematische Unterhaltungen und Spiele (Berlin, 1910).

[2] Demirors, O., Rafraf, N., and Tanik, M. M. Obtaining N-queens solutions from Magic Squares and Constructing Magic Squares from N-Queens solutions. *J. Recreational Mathematics* 24 (4), 1992, 272–280.

[3] Goodrich, M. T., and Tamassia, R. Algorithm Design: Foundations, Analysis, and Internet Examples. Wiley, 2002.

[4] Nudelman, S. P. The modular n-queens problem in higher dimensions. *Discrete Mathematics* 146, 1995, 159–167.

[5] Bernhardsson, B. Explicit Solutions to the N-Queens Problem for all N. *SIGART bulletin* 2 (2), 1991, 7.

[6] Heden, O. On the modular n-queen problem. *Discrete Mathematics* 102, 1992, 155-161.

[7] Polya, G. Über die "doppelt-periodishen" Lösungen des n-Damen-Problems. *Mathematische Unterhaltungen und Spiele* 2 (2), 1921.

[8] Diaconis, P., and Keller, J. B. Fair Dice. *American Mathematical Monthly* 96 (4), April 1989, 337-339.

[9] Cormen, T., Leiserson, C. E., Rivest, R., and Stein, C. Introduction to Algorithms. Second Edition, McGraw-Hill, 2001.