Isomorphism and the N-Queens Problem

Paul Cull
Rajeev Pandey

Technical Report 94–20–02

Department of Computer Science
Oregon State University
Corvallis, Oregon
97331–3202
{pc, rpandey}@cs.orst.edu

February 11, 1994

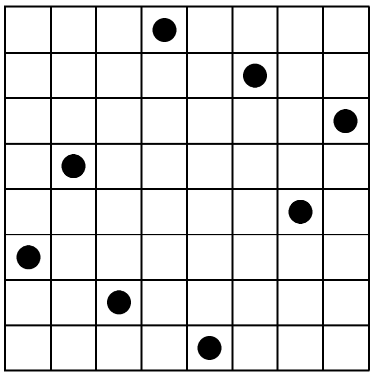
Abstract

The N-Queens problem is commonly used to teach the programming technique of backtrack search. The N-Queens problem may also be used to illustrate the important concept of isomorphism. Here we show how the N-Queens problem can be used as a vehicle to teach the concepts of isomorphism, transformation groups or generators, and equivalence classes. We indicate how these ideas can be used in a programming exercise. We include a bibliography of 29 papers.

1 Introduction

The 8-Queens problem [Wir71] or the more general N-Queens problem is often used to explicate backtracking in computing courses. A recent paper by Gray [Gra93] presents a detailed analysis of the N-Queens problem, and how fully analyzing the problem can lead to better solutions. In this note, we want to point out that N-Queens can also be used as a vehicle for teaching the ideas of isomorphism and transformation groups. We have successfully presented the following material in our junior level algorithms course, as well as in first-year graduate courses.

In the next section we briefly describe the N-Queens problem. In section 3 we describe isomorphism. Section 4 examines the N-Queens problem in the context of isomorphism. Section 5 examine methods of generating nonisomorphic solutions to the N-Queens problem. We append a bibliography of literature on the N-Queens problem as well.


(a)

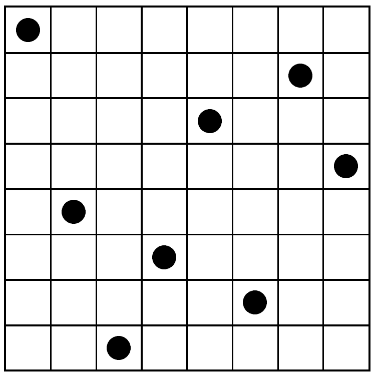

(b)

Figure 1: Two solutions to the 8-Queens Problem.

2 The N-Queens Problem

The objective in the N-Queens problem is to place N queens on an N×N chessboard such that no two queens can attack one another under the normal rules of chess. A queen may attack any other piece lying along the row, column, or diagonal containing the queen. The N-Queens problem is often stated as: Can N queens be placed on an N×N chessboard so that no queen can attack another queen? In this form the answer is easy: if N $\notin \{2,3\}$ say YES. Further, it is easy to find a non-attacking placement for the queens. Several papers in the bibliography detail approaches to finding non-attacking placements.

Two solutions to the 8-Queens problem are shown in Figure 1. Since any solution to N-Queens must have exactly one queen per row (and column) of the chessboard, a solution such as Figure 1(a) can be expressed as the vector (6,4,7,1,8,2,5,3), i.e. the row which the queen in each column occupies. Similarly, solution 1(b) can be notated as (1,5,8,6,3,7,2,4).

The construction of an enumerating backtracking program seems to be necessary to find *all* the non-attacking placements. Even finding the number of such placements seems to be difficult. What does "all" mean in the queens problem statement? For example, when $N = 4$, there seems to be two solutions, (2,4,1,3) and (3,1,4,2), but these solutions are merely mirror images of one another, so there is in some sense only one solution.

The point here is that the N-Queens problem gives us a chance to discuss isomorphic and nonisomorphic solutions. Further, by a small addition to the standard N-Queens program, we can produce a program which outputs only the nonisomorphic solutions. In addition, this exercise will also introduce students to a use of transformation groups.

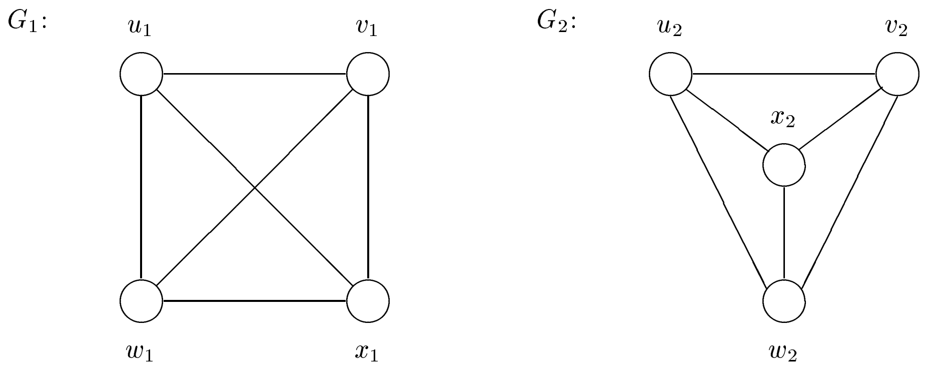


Figure 2: Two Isomorphic Graphs.

3 Isomorphism

Two objects are *isomorphic* when they have equal forms, that is, in some sense the objects are identical. The sense is, of course, important. For example, a metal paper clip and a plastic paper clip are isomorphic when I want to clip papers together, but they are not isomorphic when I want to play with my magnetic paper clip holder. In more complicated cases, the objects are made up of parts and the sense of sameness includes some of the interrelationships between the parts. For example, I might consider bees and mosquitos as isomorphic in producing red welts on humans, but when I consider the body parts, the bee and the mosquito are not isomorphic because their business ends are different.

For mathematics students, the idea of isomorphism is usually introduced in algebra. For example, the field of real numbers is isomorphic to the field of complex numbers with zero imaginary part, or two semigroups (S, +) and (G, $*$) are isomorphic when there is an invertible function $h : S \to G$ so that $h(s_1 + s_2) = h(s_1) * h(s_2)$ for all s_1 and s_2. Computer science students are usually introduced to isomorphism in the context of graphs. Two graphs G_1 and G_2 are isomorphic if there is an invertable function h which maps each vertex of G_1 to a vertex of G_2 so that adjacent vertices of G_1 map to adjacent vertices of G_2 and vice versa, as in Figure 2.

4 Isomorphism and the N-Queens Problem

For solutions to the N-Queens we want the renaming to preserve the relationships between the queens. What transformations should be allowed? Clearly mirror image, mentioned above, should be included, but it seems that several mirror images are possible. The obvious ones place a mirror parallel to one side of the board, but what about reflecting across one of the diagonals of the board? Other allowed transformations should include rotations by multiples of 90^o ($\pi/2$ radians). Obviously one could combine rotations and mirror images to get other transformations. This combining is *function composition* since each transformation takes a solution as input and gives a solution as output. Function composition is *associative*. Associativity means that if T_1, T_2, and T_3 are three transformations and $T_1 \circ T_2$ is the composite transformation obtained by applying T_2 and then T_1, then the following equation holds: $(T_1 \circ T_2) \circ T_3 = T_1 \circ (T_2 \circ T_3)$.

Further, the composition of any allowed transformations always gives an allowed transformation, so the

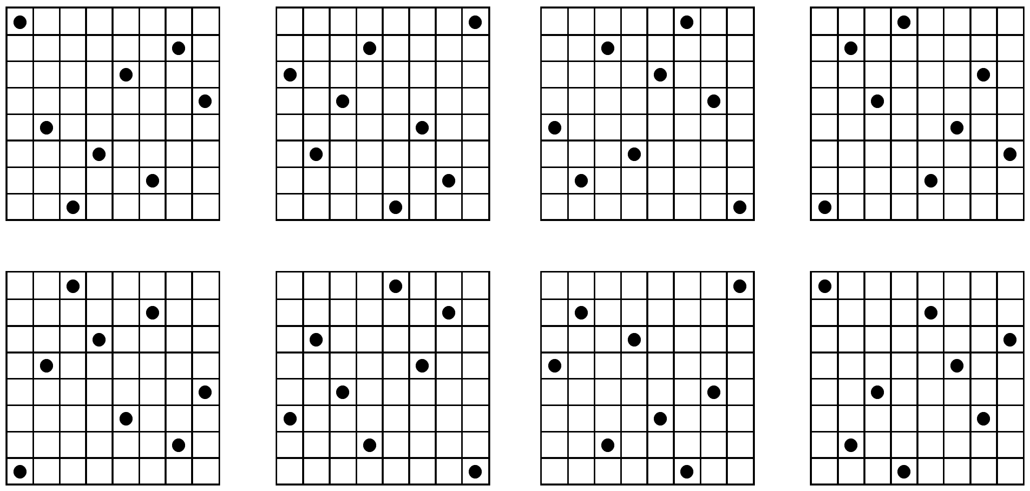


Figure 3: Generating Isomorphic Solutions to 8-Queens.

set of transformations is *closed* under composition. There is a special transformation, 'do nothing to the board', which is the *identity* transformation. Finally, each transformation can be undone, that is, each transformation has an *inverse*. Each mirror image is its own inverse. For any rotation by a multiple of 90^o, applying the rotation 3 times will give the inverse of the rotation. So for any sequence of rotations and mirror images, there is an inverse which can be formed by taking the inverses of the transformations in reverse order.

A set with an operation that is associative, closed, and has an identity and inverses is, of course, a *group*[1]. In fact, the allowed transformations for the N-Queens is the group of transformations which transform a square into itself. Consider a board with corners labelled A, B, C, and D in clockwise order. When a transformation is applied, the corner labelled A can be mapped to any of 4 positions, and the next corner clockwise after A must be either B or D. So there are $4 * 2 = 8$ different transformations.

The set of transformations for N-Queens is a *dihedral group*. A dihedral group Δ_n is defined as the group of symmetries of a regular polygon P_n of n sides. Elements of Δ_n can be obtained by the operations of rotation R through $360°/n$, and the operation reflection M about some side:

$$\begin{tikzcd}
I \arrow[r, Leftrightarrow] \arrow[d, Leftrightarrow] & R \arrow[r, Leftrightarrow] \arrow[d, Leftrightarrow] & R^2 \arrow[r, Leftrightarrow] \arrow[d, Leftrightarrow] & R^3 \arrow[d, Leftrightarrow] & \cdots & R^n \arrow[d, Leftrightarrow] \\
M \arrow[r, Leftrightarrow] & RM \arrow[r, Leftrightarrow] & R^2M \arrow[r, Leftrightarrow] & R^3M & \cdots & R^nM
\end{tikzcd}$$

For N-Queens the appropriate group is Δ_4. Using the operations of reflection (M), and 90° rotation (R), one solution can be transformed into seven other solutions. The solution to 8-Queens from Figure 1(b) is transformed as follows:

[1]You mean that abstract algebra has some use?

$$\begin{tikzcd}
15863724 \arrow[r, Leftrightarrow, "R"] \arrow[d, Leftrightarrow, "M"'] & 36428571 \arrow[r, Leftrightarrow, "R"] & 57263148 \arrow[r, Leftrightarrow, "R"] & 82417536 \arrow[d, Leftrightarrow, "M"'] \\
84136275 \arrow[r, Leftrightarrow, "R"] & 63571428 \arrow[r, Leftrightarrow, "R"] & 42736851 \arrow[r, Leftrightarrow, "R"] & 17582463
\end{tikzcd}$$

Figure 3 shows the boards corresponding to the vectors above.

5 Generating Nonisomorphic N-Queens Solutions

An interesting problem is finding the number of nonisomorphic solutions for each value of N, as well as a method of quickly generating these nonisomorphic solutions. It might seem easy to find the number of nonisomorphic solutions to N-Queens by finding the total number of solutions and dividing by 8. Unfortunately this doesn't work because there are solutions which can be mapped to themselves by some of the transformations. For example, the solution (6,4,7,1,8,2,5,3) to 8-Queens (Figure 1(a)), is transformed to itself after two rotations:

$$\begin{tikzcd}
64718253 \arrow[r, Leftrightarrow, "R"] \arrow[d, Leftrightarrow, "M"'] & 53172864 \arrow[r, Leftrightarrow, "R"] & 64718253 \arrow[r, Leftrightarrow, "R"] & 53172864 \arrow[d, Leftrightarrow, "M"'] \\
35281746 \arrow[r, Leftrightarrow, "R"] & 46827135 \arrow[r, Leftrightarrow, "R"] & 35281746 \arrow[r, Leftrightarrow, "R"] & 46827135
\end{tikzcd}$$

Below is a table showing the total number of solutions to N-Queens, as well as the number of nonisomorphic solutions to the N-Queens problem for various values of N:

N	**total number of solutions:**	**number of nonisomorphic solutions:**
0	0	0
1	1	1
2	0	0
3	0	0
4	2	1
5	10	2
6	4	1
7	40	6
8	92	12
9	352	46
10	724	92
11	2680	341
12	14200	1787
13	73712	9233
14	365596	45752
15	2279184	285053

While we want to be able to count the number of nonisomorphic solutions, we also want to find one solution for each isomorphism class. Two solutions are in the same isomorphism class exactly when some allowed transformation transforms one solution to the other. It is probably worth mentioning at this point that being isomorphic is an *equivalence relation*, and reminding students that an equivalence relation is *reflexive*, *symmetric*, and *transitive*. As an easy exercise you can ask the students to find which group property implies reflexive, which group property implies symmetric, and which group property implies transitive.

Since there are only 8 transformations to consider, a simple way to find nonisomorphic solutions is to maintain a set of the nonisomorphic solutions found so far, generate all solutions, and then determine if applying any of the 8 transformations gives a solution already in the set. If each of the 8 transformations gives a solution *not* in the set, then the new solution should be added to the set of nonisomorphic solutions.

Unfortunately this method has the drawback that there may be a very large (more than exponential) number of nonisomorphic solutions in the set. So comparing a new solution with the nonisomorphic solutions can take a very long time. Luckily, there is a shortcut. In the backtracking algorithm, the first queen is placed in the first allowed square in the first column, then the second queen is placed in the first allowed position in the second column, and so forth. This means that the solutions will be generated in order if we consider each solution of N-Queens as a base N+1 number. Hence a solution is isomorphic to a previously found solution if and only if one of the 8 transformations produces a solution which is *less* than the present solution.

The less than relation may be stated as:

Let $S_1 = (a_1, a_2, ..., a_n)$ and $S_2 = (b_1, b_2, ..., b_n)$,
which are two solutions to the n-Queens, so that each $a_i, b_i \in \{1, 2, ..., n\}$.
Then $S_1 < S_2$ iff $(a_1 a_2 ... a_n) < (b_1 b_2 ... b_n)$, where:
$(a_1 a_2 ... a_n) < (b_1 b_2 ... b_n)$ iff $a_1 < b_1$ in the usual ordering $1 < 2 < ... < n$

$$\text{or, } a_1 = b_1 \text{ and } (a_2 ... a_n) < (b_2 ... b_n)$$

$$\ddots$$

$$\text{to bottom out the recursion, we have } (a_n) < (b_n) \text{ iff } a_n < b_n \text{ in the usual ordering.}$$

So given a solution S, one should add S to the set of nonisomorphic solutions when for each of the seven non-identity transformations $T_1, T_2, ..., T_7$,

$$S \leq T_i(S), \text{ for } i = 1, 2, ..., 7.$$

For example, when the solution (1,5,8,6,3,7,2,4) for the 8-Queens is generated (Figure 1(b)), each of the transformations applied to (1,5,8,6,3,7,2,4) yields (3,6,4,2,8,5,7,1), (5,7,2,6,3,1,4,8), (8,2,4,1,7,5,3,6), (8,4,1,3,6,2,7,5), (6,3,5,7,1,4,2,8), (4,2,7,3,6,8,5,1), and (1,7,5,8,2,4,6,3). Since each of these transformed solutions is greater than or equal to (1,5,8,6,3,7,2,4), the solution (1,5,8,6,3,7,2,4) should be added to the nonisomorphic solutions to the 8-Queens. On the other hand, when (6,4,7,1,8,2,5,3) is generated (Figure 1(a)), a rotation R applied to (6,4,7,1,8,2,5,3) yields (5,3,1,7,2,8,6,4) which is less than (6,4,7,1,8,2,5,3), and so (6,4,7,1,8,2,5,3) should not be added to the set of nonisomorphic solutions.

The 8 transformations can be expressed as a sequence of the simple mirror image and rotate by 90^o transformations. These simple transformations are called *generators* because every element in the group can be generated by using these simple transformations. In programming the isomorphism test, one could write separate code for each of the transformations, but one could instead write code for the generators and use the code several times. If S is a solution and M stands for mirror image, R for rotate by 90^o, then the 8 solutions isomorphic to S are S, R(S), RR(S), RRR(S), MRRR(S), RMRRR(S), RRMRRR(S), RRRMRRR(S). So by initializing the variable `transformed` with the original solution vector `original`, and iteratively applying the appropriate transformation (either `Rotate` or `Mirror_image`), a program can easily generate the 7 solutions which have to be tested against the original solution. The following C code will determine if a solution is isomorphic to any previous solution generated by the backtracking algorithm:

```
/*
  check_if_isomorph: If the solution is an isomorph to a previously
  generated solution, return TRUE. Return FALSE if solution is new.
*/

BOOLEAN check_if_isomorph(int original[MAX])
{
  int i;
  int transformed[MAX];
  BOOLEAN iso_flag;

  for (i = 0; i < size; i++)          /* make a copy of the solution vector */
    transformed[i] = original[i];

  for (i = 0; i < 7; i++)             /* generate the 7 transformations */
    {
      if (i != 3)
```

```
        Rotate(transformed);
      else
        Mirror_image(transformed);
      iso_flag = compare_vector(original, transformed);
      if (! iso_flag) return (TRUE);
    }
  return (FALSE);
}
```

The generators R and M are not the only possible generators. Let F be the transformation which flips the square across its counter-diagonal. Then F and M are a set of generators for Δ_4, and FMFMFMF is a sequence of these generators which will generate the whole transformation group. Showing that the previous statement is true would be a reasonable exercise to see if your students have followed the development. You also might want them to decide which set of generators is easier to program.

In several texts, the fact that the queen in the first column never has to be placed in the second half of the column if one is only interested in nonisomorphic solutions is mentioned, for example Horowitz and Sahni, p. 363 [HoS78]:

"Observe that for finding inequivalent solutions the algorithm need only set $X(I) = 2, 3, ..., \lceil n/2 \rceil$."

Unfortunately this has been widely misinterpreted by students to mean that this restriction alone is sufficient to generate only nonisomorphic solutions. We hope that the above description has been sufficient to help you explain the actual situation to your students.

6 Conclusion

The major point of this note is that N-Queens is a good example which is typical of combinatorial enumeration problems. The typical features are:

- solutions are generated by backtracking algorithms
- ordering on solutions is imposed by the generating algorithm
- desire for nonisomorphic solutions
- group of transformations indicating which solutions are isomorphic
- group expressible by simple generators
- "on-the-fly" nonisomorphism test by applying a sequence of generators to a solution and checking that all the transformed solutions are $\geq$ the solution in question.

Another problem with similar characteristics is the Knight's Tour Problem (where the transformation group consists of rotation, reflection, and 'take a path backwards') [CuD78].

References

[AbY89] Abramson, Bruce, and Yung, Moti, "Divide and Conquer under Global Constraints: A Solution to the N-Queens Problem," *Journal of Parallel and Distributed Computing*, 6:649–662 (1989).

[Bie93] Biernat, Martin J., "Teaching Tools for Data Structures and Algorithms," *SIGCSE Bulletin*, 25(4):9–12 (December 1993).

[BiR75] Bitner, James R., and Reingold, Edward M., "Backtrack Programming Techniques," *Communications of the ACM*, 18(11):651–656 (November 1975).

[BrD75] Bruen, A., and Dixon, R., "The n-Queens Problem," *Discrete Mathematics*, 12:393–395 (1975).

[Cha74] Chandra, A. K., "Independent Permutations as Related to a Problem of Moser and a Theorem of Pólya," *J. Combin. Theory*, 16(1):111–120 (January 1974).

[CMV86] Clapp, Russell M., Mudge, Trevor N., and Volz, Richard A., "Solutions to the n Queens Problem Using Tasking in Ada," *SIGPLAN Notices*, 21(12):99–110 (Decemeber 1986).

[Cla86] Clay, C., "A New Solution to the N $\leq$ 8 Queens Problem," *SIGPLAN Notices*, 21(12):28–30 (August 1986).

[CoH86] Cockayne, E. J., and Hedetniemi, S. T., "On the Diagonal Queens Domination Problem," *Journal of Combinatorial Theory*, Series A 42, 137–139 (1986).

[CuD78] Cull, Paul, and De Curtins, J., "Knight's Tour Revisited," *Fibonacci Quarterly*, 16, 276–285 (1978).

[DeT91] Demirors, O., and Tanik, M. M., "Peaceful Queens and Magic Squares," Technical Report 91–CSE–7, Department of Computer Science and Engineering, Southern Methodist University, Dallas, TX 75275-0122 (February 1991).

[ErT91] Erbas, Cengiz, and Tanik, Murat M., "N–Queens Problem and its Algorithms," Technical Report 91–CSE–8, Department of Computer Science and Engineering, Southern Methodist University, Dallas, TX 75275-0122 (February 1991).

[ETA92] Erbas, Cengiz, Tanik, Murat M. and Aliyazicioglu, Zekeriya, "A Note on Falkowski's N-Queens Solutions," Technical Report 92–CSE–14, Department of Computer Science and Engineering, Southern Methodist University, Dallas, TX 75275-0122 (May 1992).

[FaS90] Falkowski, Bernd-Jürgen and Schmitz, Lothar, "A Note on the Queens' Problem," *Information Processing Letters*, 23:39–46 (July 1986).

[Gol87] Goldsby, Michael E., "Solving the "N $\leq$ 8 Queens" Problem with CSP and Modula-2,"*SIGPLAN Notices*, 22(2):43–52 (February 1987).

[Gra93] Gray, John S., "Is Eight Enough? - The Eight Queens Problem Re-examined," *SIGCSE Bulletin*, 25(3):39–44,51 (September 1993).

[HaV73] Hansche, Brian, and Vucenic, Wayne, "On the n-queens problem," (abstract) *Notices of the American Mathematical Society*, 20:A-568 (1973).

[HLM69] Hoffman, E.J., Loessi, J.C., and Moore, R.C., "Constructions for the Solution of the m Queens Problem," *National Mathematics Magazine*, 66–72 (March–April 1969).

[HoS78] Horowitz, E., and Sahni, S., *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, Maryland (1978).

[Kal90] Kalé, L.V., "An Almost Perfect Heuristic for the N Nonattacking Queens Problem," *Information Processing Letters*, 34:173–178 (April 1990).

[Kea93] Keating, John G., "Hopfield Networks, Neural Data Structures and the Nine Flies Problem: Neural Network Programming Projects for Undergraduates," *SIGCSE Bulletin*, 25(4):33–37,40,60 (December 1993).

[Kla67] Klarner, D. A., "The Problem of Reflecting Queens," *American Mathematical Monthly*, 74:953–955 (1967).

[MaM92] Mańdziuk, Jacek, and Macukow, Bohdan, "A neural network designed to solve the N-Queens Problem,"*Biological Cybernetics*, 66:375–379 (1992).

[Pol18] Pólya, G., "Uber die 'doppelt-periodischen' Losungen des n-Damen-Problems," In Ahrens, W. (Ed.), *Mathematische Unterhaltungen und Spiele.* Teubner, Leipzig, pp. 364–374 (1918).

[Rei87] Reichling, Matthias, "A Simplified Solution of the N Queens' Problem," *Information Processing Letters*, 25:253–255 (June 1987).

[Sal90] Saletore, Vikram A., "Machine Independent Parallel Execution of Speculative Computation," *PhD Thesis*, Department of Computer Science, University of Illinois, Urbana-Champaign (September 1990).

[Seb69] Sebastian, J. D., "Some Computer Solutions to the Reflecting Queens Problem," *American Mathematical Monthly*, 76:399–400 (1969).

[SoG90] Sosic, Rok, and Gu, Jun, "A Polynomial Time Algorithm for the N-Queens Problem," *SIGART Bulletin*, 1(3):7–11 (October 1990).

[StS87] Stone, Harold S., and Stone, Janice M., "Efficient Search techniques–An empirical study of the N-Queens Problem," *IBM Journal of Research and Development*, 31(4):464-474 (July 1987).

[Wir71] Wirth, Niklaus, "Program Development by Stepwise Refinement," *Communications of the ACM*, 14(4):221–227 (April 1971).

A Source Code

```
/*
    Queen Solution Program
    Originally written by:
    John S. Gray      1993
    (9/93 SIGCSE Bulletin)

    Modified by Rajeev Pandey 01/94 to compute nonisomorphic solutions.
*/
/**************************************************************************/

typedef enum {FALSE, TRUE} BOOLEAN;

#define MAX 15                       /* largest board size          */
                                     /*      GLOBALS                */
int a[MAX],                          /* vector holding solution     */
    size = 0,                        /* board size selected by user */
    dim  = 0,                        /* array size selected by user */
    type = 0,                        /* display type                */
    iso  = 0,                        /* isomorphic solutions only?  */
    numb = 0;                        /* total number of solutions   */

/**************************************************************************/
void main ( )
{

  void find_sol (int, int);

  printf ("\n N-Queens demonstration \n");
  do                                        /* obtain board size */
    {
      printf ("\n\n\nEnter size of the board 1-%d > ", MAX);
      scanf ("%d", &size);
    }
  while (size < 0 || size > MAX);
  dim = size - 1;
  do                                        /* obtain isomorphic or not */
    {
      printf ("Enter solution type (1 = all solutions,\
2 = only nonisomorphic solutions) > ");
      scanf ("%d", &iso);
    }
  while (iso < 0 || iso > 2);

  do                                         /* obtain output format */
    {
      printf ("Enter output display type (1 = diagrammatic,\
2 = numeric) > ");
      scanf ("%d", &type);
    }
  while (type < 0 || type > 2);

  printf ("\n\n");
  if (iso == 1)
    printf (" For a board of size %d, solutions are: \n", size);
  else
    printf (" For a board of size %d, nonisomorphic solutions are: \n", size);
  find_sol(0,0);          /* initial call */
  if (! numb)
    printf ("\n\n ZERO!");
  printf ("\n\n\n");
}

/***********************************************************************/
/*

   Function find_sol: Passed row and column position.
                      Generates locations for "size" number of
                      queen. Uses recursion to backtrack.

   Non recursive version--a few more lines of code, but less system
                          intensive.
*/

void find_sol (int row, int col)
{

  void print_it ();
  BOOLEAN no_conflict(int, int);

  do
    {
      if (no_conflict(row,col))
        {                          /* Check for conflicts, if none  */
          a[col] = row;            /* save the location in vector   */
          if (col == size - 1)     /* if full solution, display it  */
```

```
            print_it();
          else
            {                         /* otherwise, reset the row and  */
              row = 0; ++col;         /* check the next column         */
              continue;
            }
        }
      if (row < size - 1)             /* if more rows are available    */
        ++row;                        /* try next row, same column     */
      else                            /* otherwise, backtrack until a  */
        {
          while (col && a[col-1]+1 > size-1)
            --col;                    /* column is found with a row    */
          row = a[--col]+1;           /* value that can be incremented */
        }                             /* to the next row               */
    }
  while (row < size);                 /* loop until beyond last row in */
}                                     /* the first column              */

/***********************************************************************/
/*
   Function no_conflict: Passed row and column position. Returns
                         TRUE if no conflict with previous
                         locations else returns FALSE.
*/

BOOLEAN no_conflict (int row, int col)
{
  register i;
  int d;                    /* temporary diagonal offset value */
  BOOLEAN ok = TRUE;        /* assume no conflict at the start */

/*
   Step backward and check for conflicts with
   previous selections.
*/

  for (i = col-1; i>=0 && ok; --i)
    {
      d = col - i;
      if (a[i] == row   ||   /* check for conflict in: same row,            */
          a[i]-d == row ||   /*                        same major diagonal  */
          a[i]+d == row)     /*                        same minor diagonal. */
        ok = FALSE;
    }
  return (ok);
}

/************************************************************************/
/*
   Function print_it: Displays output in user selected format.
*/

void print_it () {
  register r, c;
  int i;
  int tmp[MAX];
  static char line[] = "+---+---+---+---+---+---+---+---+";
  line [size*4+1] = '\0';           /* cut string to size */

/*
  check to make sure solution is not isomorphic to some previously generated
  solution
*/
  if (iso == 2)
    if (check_if_isomorph(a))
      return;

  printf ("\n%06d : ", ++numb);

  switch (type)
    {
    case 1:
      printf ("\n\n%s\n", line);
      for (r = 0; r < size; ++r)
        {
          for (c=0; c < size; ++c)
            printf ("| %c ", a[r] == c ? 'Q' : (r+c) % 2 ? '#':' ');
          printf ("|\n%s\n", line);
        }
      break;
    case 2:
    default:
      for (c=0; c < size; ++c)
        printf ("%3d", a[c] +1);
    }
}
```

```
/**********************************************************************/

void mirror_image (int vector[MAX])
{
 int i;

 for (i=0; i < size; i++)
   {
     vector[i] = dim - vector[i];
   }
 return;
}

/**********************************************************************/

void rotate (int vector[MAX])
{
 int i;
 int tmp_vector[MAX];

 for (i=0; i < size; i++)
   {
     tmp_vector[vector[i]] = i;
   }
 for (i=0; i < size; i++)
   {
     vector[i] = tmp_vector[dim - i];
   }
 return;
}

/**********************************************************************/

/* compare_vector returns true if vector v1 <= vector v2, else returns FALSE*/

BOOLEAN compare_vector(int v1[MAX], int v2[MAX])
{
  int pos;
  BOOLEAN tmp;

  for (pos=0; pos < size; ++pos)
    {
      if (v1[pos] > v2[pos])
        {
          return (FALSE);
        }
      if (v1[pos] < v2[pos])
        {
          return (TRUE);
         }
    }
  for (pos=0; pos < size; ++pos)
    {
      if (v1[pos] != v2[pos])
        {
          return (FALSE);
        }
    }
  return (TRUE);
}

/**************************************************************************/
/*
  If the solution is an isomorph to a previously generated solution,
  return TRUE, else return FALSE if solution is new.
*/

BOOLEAN check_if_isomorph(int original[MAX])
{
  int i;
  int transformed[MAX];
  BOOLEAN iso_flag;

  for (i = 0; i < size; i++)          /* make a copy of the solution vector */
    transformed[i] = original[i];

  for (i = 0; i < 7; i++)             /* generate the 7 transformations */
    {
      if (i != 3)
        rotate(transformed);
      else
        mirror_image(transformed);
      iso_flag = compare_vector(original, transformed);
      if (! iso_flag) return (TRUE);
    }
  return (FALSE);
}
/**************************************************************************/
```