Table 2. Algorithms for the *n* queens problem

n	Wirth	Rohl	*lexqueens*
6	0.92	0.35	0.33
7	3.60	1.30	1.25
8	15.4	4.9	4.8
9	69	21.4	20.3
10	328	93	91
11	1681	460	457
12	9201	2424	2409

Received October 1982

ROBERT W. IRVING
Department of Mathematics
University of Salford
Salford M5 4WT
UK

References

1. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs (1976).
2. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Pitman, Potomac (1978).
3. J. S. Rohl, Generating Permutations by choosing. *The Computer Journal* **21**, 302-305 (1978).
4. J. S. Rohl, Letter to the Editor. *The Computer Journal* **22**, 191 (1979).
5. R. Sedgewick, Permutation generation methods. *Computing Surveys* **9**, 137-165 (1977).
6. J. M. Wilson, Interrupted permutations in lexicographic order. Algorithm 114, *The Computer Journal* **26**, 92 (1983).
7. R. J. Ord-Smith, Generation of permutations in lexicographic order. Algorithm 323, *Comm. ACM* **11**, 117 (1968).

Consistency in Point-in-Polygon Tests

In many applications it is necessary to be able to determine if a point is contained in a polygon. If co-ordinate values are approximate, we may be happy to have borderline cases dealt with in an arbitrary manner. However, if two regions have a common border section, decisions should be consistent. A simple algorithm for determining containment consistently is given.

In many applications it is necessary to be able to determine if a point is contained in a polygon. For example, in a geographical information system we may wish to determine if a point representing a house is inside a region with a polygonal boundary. Commonly, an area will be partitioned into a number of regions. In this case different regions have common boundary sections. If co-ordinate values are approximate, we may be happy to have borderline cases dealt with in an arbitrary but consistent manner. For example, if a point is on a borderline, between two regions, it may be treated as being in one region or the other, but should not be treated as being in neither or both regions. A similar case arises when one region is contained in another. If a point is considered to be in the inner region, it should also be considered to be in the outer region.

The problem of determining whether or not a point is inside a polygon has been widely considered.[1,4] We develop an algorithm proposed in Ref. 5 as corrected in Ref. 3 with some further modification to ensure that points on or near a boundary are treated consistently. To determine if a point is inside a polygon, we can draw a horizontal line (called a ray) from the point to the right. Each time the ray crosses the polygon, it goes from inside to outside or vice versa. Therefore, the point is inside the polygon if and only if the ray crosses the polygon an odd number of times. The following FORTRAN 77 logical function solves this problem, assuming that the co-ordinates of the point are given by PX and PY and that the X and Y co-ordinates of the vertices of the polygon are in the first N elements of the arrays X and Y, respectively.

```
   LOGICAL FUNCTION INSIDE (PX,
                         PY, N, X, Y)
   DIMENSION X(N), Y(N)
   LOGICAL CROSS
   INSIDE = .FALSE.
   DO 10 I = 1, N - 1
10 IF (CROSS (PX, PY, X(I), Y(I), X(I +
   1), Y(I + 1))) INSIDE = .NOT. INSIDE
   IF (CROSS (PX, PY, X(N), Y(N), X(1),
         Y(1))) INSIDE = .NOT. INSIDE
   RETURN
   END
```

A call to CROSS (PX, PY, X1, Y1, X2, Y2) returns true if and only if a ray from (PX, PY) crosses the line segment (i.e. polygon side) joining (X1, Y1) and (X2, Y2). Before considering the implementation of CROSS we should consider how to handle borderline points.

To handle borderline cases consistently, we will shift each polygon to the left by an infinitesimal distance, and downwards by an infinitely smaller distance. (This also eliminates the nasty cases which can arise if a polygon vertex lies on the ray from the point.) The following logical function CROSS tests a point and one side of a polygon.

```
  LOGICAL FUNCTION CROSS (X, Y, X1, Y1, X2, Y2)
  IF (((Y.LT.Y1) .EQV. (Y.LT.Y2)) .OR. (X.GE.X1.AND.X.GE.X2)) THEN
C   SIDE ENTIRELY ABOVE, BELOW OR ENTIRELY TO LEFT OF POINT
    CROSS = .FALSE.
  ELSE IF (X.LT.X1.AND.X.LT.X2) THEN
C   SIDE TO RIGHT OF POINT, BUT NOT ENTIRELY ABOVE OR BELOW
    CROSS = .TRUE.
  ELSE IF (X1.LT.X2) THEN
C   SIDE NOT ENTIRELY ABOVE OR BELOW POINT AND
C   SIDE NOT ENTIRELY TO LEFT OR RIGHT
    CROSS = X.LT. (X1 + (Y - Y1) * (X2 - X1) / (Y2 - Y1))
  ELSE
C   SAME AS ABOVE
    CROSS = X.LT. (X2 + (Y - Y2) * (X1 - X2) / (Y1 - Y2))
  END IF
  RETURN
  END
```

We present now a complete explanation of the above function. Clearly, a ray from (X,Y) cannot cross a ray from (X1, Y1) to (X2, Y2) shifted as described above, if (Y.LT.Y1.AND.Y.LT.Y2) .OR. (Y.GE.Y1.AND.Y.GE.Y2). This condition reduces to (Y.LT.Y1).EQ.(Y.LT.Y2). Furthermore, if (X.GE.X1.AND.X.GE.X2) then the ray starts to the right of the shifted side and so cannot cross the ray. These two cases are allowed for in the first alternative to the IF statement of the function CROSS. In the remaining cases, a horizontal line through the point (X,Y) must cross the shifted line segment (since either Y.LT.Y1.AND.Y.GE.Y2 or Y.GE.Y1.AND.Y.LT.Y2 if the first IF alternative is not selected). If (X.LT.X1.AND.X.LT.X2) then the shifted line segment lies to the right of the point and must cross the ray.

There is really only one remaining case, but we have broken it down into two subcases for reasons that will become clear shortly. (The values assigned to CROSS in the last two IF alternatives are algebraically equivalent but due to rounding errors may not produce the same result on a computer.) If none of the above cases apply then the side crosses the horizontal line but is not entirely to the left or entirely to the right of the point. We must compute the exact point where the horizontal line and side cross, and test to see if this is to the left or right of the point. Each of the last two IF alternatives do this.

The reason we must break the last case into two subcases is that if two polygons have some sides in common, the vertices need not occur in the same order (i.e. the order of the common vertices may be reversed). However, we want the same side and same point always to produce the same result when CROSS is called.

To determine if a ray from (X,Y) crosses a side from (XA, YA) to (XB, YB), given that YA $\neq$ YB, we can compute the X co-ordinate of the point where the side crosses a horizontal line through (X,Y) using the expression XA + (Y − YA) * (XB − YA)/(YB − YA). A crossing occurs if and only if X is less than this value. We know that XA $\neq$ XB, or one of the earlier cases would apply. We make certain that this test always produces the same result by requiring that XA < XB, and reversing the vertices if necessary.

If retrieval of information on the basis of location is rarely performed, or if only a small number of points is to be considered, then the above algorithm may be applied to each point in turn. If a large number of points is to be searched frequently, the points may be organized in a tree structure.[2] A set of points and a rectangle which contains all of the points in the set are associated with each node of the tree. If the rectangle is entirely inside or outside the region of interest then so are all the points in the rectangle. Otherwise, we must consider a lower level node having smaller associate rectangles, and so forth, down to the level of individual points.

This approach easily generalizes to three (or more) dimensions, to regions with other than polygonal boundaries, and to other geometrical operations.

F. W. BURTON†‡
School of Computing Studies
University of East Anglia
Norwich, NR4 7TJ
England

V. J. KOLLIAS
Athens Faculty of Agriculture
Laboratory of Soils and Agricultural Chemistry
Iera Odos, Votanicos
Athens, Greece

J. G. KOLLIAS†
Department of Computer Sciences
National Technical University of Athens
9, Heroon Polytechnion Avenue
Zografou
Athens (621), Greece

†Part of this work was done while visiting Michigan Technological University.
‡Present address: Department of Electrical Engineering and Computer Science, Box 104, University of Colorado at Denver, 1100 Fourteenth Street, Denver, Colorado 80202, USA.

References

1. B. K. Aldred, Points in polygon algorithms. UKSC-0025, UK Scientific Centre, IBM United Kingdom Ltd., County Durham, England (1972).
2. J. L. Bentley, Multidimensional binary search trees used for associative searching. *Commun. of the ACM* **18**, 509–517 (1975).
3. R. Hacker, Certification of algorithm 112: Position of a point relative to polygon. *Common. of the ACM* **5**, 606 (1962).
4. S. Nordbeck and B. Rystedt, Computer cartography point in polygon points. *BIT* **7**, 30–64 (1967).
5. M. Shimrat, Algorithm 112: Position of point relative to polygon. *Commun. of the ACM* **5**, 434 (1962).

Received April 1983

The Halting Problem Does Not Matter

Suppose that *P* is a program whose set of acceptable data *accept(P)* is recursive. Then there is another program *P′* with the same acceptable data, which cannot loop, and which behaves like *P* for all input data in *accept(P)*.

A key measure of program quality is robustness. A program is said to be robust if it will behave sensibly, whatever its input. In particular, it should never under any circumstances go into an infinite loop. There are certain conceivable kinds of data which can only be distinguished and accepted by programs which might loop. Such a set of possible data is called *recursively enumerable* but *not recursive*. (This is the terminology of Ref. 1.) If anybody can construct a set of conceivable data which is not even recursively enumerable, then he has discovered a counterexample to Church's thesis. In practice, all useful programs accept a set of data which is not only recursively enumerable, but also actually recursive.

Proposition

Suppose that *P* is a program whose set of acceptable data *accept(P)* is recursive. Then there is another program *P′* with the same acceptable data:

$$accept(P') = accept(P)$$

and which cannot loop:

$$loop(P') \text{ is empty}$$

and which behaves like *P* for all input data in *accept(P)*: if *P* and *P′* are both given the same word *w* from *accept(P)* as initial datum, then their outputs will be the same.

Proof

Throughout what follows, all programs will be regarded as Turing machines. The output of a program is the content of its tape on termination. The key assumption is that *accept(P)* is recursive. This means that there is a program *Q*, and

$$accept(Q) = accept(P)$$

and *Q* cannot loop. The program *P′* is constructed from *P* and *Q*.

Let *P′* have some input datum *w*. This datum is a sequence of symbols stored at the left hand end of the machine's tape. First, *P′* copies this word one position to the right on the tape, and inserts a special character (say *E*, not recognized by *P*) before it. Secondly, it inserts another special symbol (say *F*, not recognized by *P* or *Q*) after the word, and makes a new copy of *w* to the right of *F*. At the far end of this copy, *P′* inserts a third special symbol (say *G*, not recognized by *Q*). If *w* is the sequence

$$w1, w2, w3, \ldots wn$$

then by this stage the tape appears as

$$E\ w1\ w2 \ldots wn\ F\ w1\ w2 \ldots wn\ G$$

Thirdly, the tape head is moved to the tape cell to the right of the one holding *F*.

Next, *P′* performs all the operations of *Q*. Just one modification is required in *Q*. Whenever the tape head is situated over a cell containing *G*, *P′* will

(a) over-write this cell with a blank character, and move the head to the right
(b) write *G*, and move the head back left.

During this stage, the input will be rejected if and only if the original datum *w* would be rejected by *Q*.

Where *Q* would halt, *P′* contains a subroutine which restores the tape to its initial state. This is possible because *E* and *F* still delimit a copy of *w*, and *G* marks the extent of tape which has been over-written. At the end of the subroutine, which cannot fail or loop, *P′* enters the original program *P*.

Q.E.D.

In practice, programmers worthy of the name always write their code on these lines. The halting problem does not concern them. It should be seen for what it is: a profound feature of mathematics, and a curiosity in the history of computing.

ALAN HUTCHINSON
Department of Computation
U.M.I.S.T.
Sackville Street
Manchester
UK

Reference

1. Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill Kogakusha, International Student Edition (1974).