Solving the N-Queens Problem with Local Search

Enrico Schumann
es@enricoschumann.net

This vignette provides example code for a combinatorial problem: the *N-Queens Problem*.

1 The problem

The goal is to place N queens on a chess-board of size $N \times N$ in such a way that no queen is attacked. A queen may move vertically, horizontally and on a diagonal. So whenever there is more than one queen on any row, column or diagonal, the position is invalid. To solve the problem with an Local Search (LS), we need three components:

1. a way to represent a solution (i.e. a position on the chessboard);
2. a way to evaluate such a solution;
3. and, since we use a LS, a method to modify a solution.

We start by attaching the package and fixing a seed.

```
> library("NMOF")
> set.seed(134577)
```

2 Representing a solution

Since on any row there cannot be more than one queen, we may store a position as a vector of columns on which the queens are placed. (In chess, rows would be called ranks and columns would be files, but we prefer matrix terminology.) Thus, a candidate solution `p` (p for position) could look as follows:

```
> N <- 8              ## board size
> p <- sample.int(N)  ## a random solution
> data.frame(row = 1:N, column = p)
```

```
  row column
1   1      8
2   2      7
3   3      3
4   4      1
5   5      6
6   6      2
7   7      5
8   8      4
```

Or (a very bad solution):

```
> p <- rep(1, N)
> data.frame(row = 1:N, column = p)
```

```
  row column
1   1      1
2   2      1
3   3      1
4   4      1
5   5      1
6   6      1
7   7      1
8   8      1
```

We may also want to visualise a position, for which may use the function print_board.

```
> print_board <- function(p, q.char = "Q", sep = " ") {
      n <- length(p)
      row <- rep("-", n)
      for (i in seq_len(n)) {
          row_i <- row
          row_i[p[i]] <- q.char

          cat(paste(row_i, collapse = sep))
          cat("\n")
      }
  }
> print_board(p)
```

```
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
```

3 Evaluating a solution

We need to compute on what row, column, diagonal (top left to bottom right) or reverse diagonal (top right to bottom left) a queen stands. Rows and columns are simple; we label the diagonals as follows.

```
> mat <- array(NA, dim = c(N,N))  ## diagonals
> for (r in 1:N)
      for (c in 1:N)
          mat[r,c] <- c - r
> mat
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    0    1    2    3    4    5    6    7
[2,]   -1    0    1    2    3    4    5    6
[3,]   -2   -1    0    1    2    3    4    5
[4,]   -3   -2   -1    0    1    2    3    4
[5,]   -4   -3   -2   -1    0    1    2    3
[6,]   -5   -4   -3   -2   -1    0    1    2
[7,]   -6   -5   -4   -3   -2   -1    0    1
[8,]   -7   -6   -5   -4   -3   -2   -1    0
```

```
> mat <- array(NA, dim = c(N,N))  ## reverse diagonals
> for (r in 1:N)
      for (c in 1:N)
          mat[r,c] <- c + r - (N + 1)
> mat
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]   -7   -6   -5   -4   -3   -2   -1    0
[2,]   -6   -5   -4   -3   -2   -1    0    1
[3,]   -5   -4   -3   -2   -1    0    1    2
[4,]   -4   -3   -2   -1    0    1    2    3
[5,]   -3   -2   -1    0    1    2    3    4
[6,]   -2   -1    0    1    2    3    4    5
[7,]   -1    0    1    2    3    4    5    6
[8,]    0    1    2    3    4    5    6    7
```

Note that for reverse diagonals, the `N + 1` would not be necessary; it serves only to shift the diagonal labels so that the main diagonal is zero.

Thus for a given solution `p`, we know the row, column, diagonal and reverse diagonal for each queen. We define the quality of a solution by the number of attacks that happen: for a valid solution, that number should be zero.

```
> n_attacks <- function(p) {
      ## more than one Q on a column?
      sum(duplicated(p)) +

      ## more than one Q on a diagonal?
      sum(duplicated(p - seq_along(p))) +

      ## more than one Q on a reverse diagonal?
      sum(duplicated(p + seq_along(p)))
  }
> n_attacks(p)
```

```
[1] 7
```

4 Changing a solution

A given position may be modified by picking one row randomly and then moving the queen there to the left or right. We allow for moves up to `step` squares, which we set to `3` in the example.

```
> neighbour <- function(p) {
      step <- 3
      i <- sample.int(N, 1)
      p[i] <- p[i] + sample(c(1:step, -(1:step)), 1)

      if (p[i] > N)
          p[i] <- 1
      else if (p[i] < 1)
          p[i] <- N
      p
  }
```

```
> print_board(p)
```

```
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
```

```
> print_board(p <- neighbour(p))
```

```
Q - - - - - - -
Q - - - - - - -
- - - Q - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
```

```
> print_board(p <- neighbour(p))
```

```
Q - - - - - - -
Q - - - - - - -
- - - Q - - - -
Q - - - - - - -
Q - - - - - - -
- - - - - - - Q
Q - - - - - - -
Q - - - - - - -
```

5 Solving the model

We use three different LS methods: a 'classical' Stochastic Local Search (`LSopt`), Threshold Accepting (`TAopt`) and Simulated Annealing (`SAopt`).

```
> p0 <- rep(1, N)  ## or a random initial solution: p0 <- sample.int(N)
> print_board(p0)
```

```
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
Q - - - - - - -
```

```
> sol <- LSopt(n_attacks, list(x0 = p0,
                               neighbour = neighbour,
                               printBar = FALSE,
                               nS = 10000))
```

```
Local Search.
Initial solution:  7
Finished.
Best solution overall: 0
```

```
> print_board(sol$xbest)
```

```
- Q - - - - - -
- - - - - - Q -
- - Q - - - - -
- - - - - Q - -
- - - - - - - Q
- - - - Q - - -
Q - - - - - - -
- - - Q - - - -
```

```
> sol <- TAopt(n_attacks, list(x0 = p0,
                               neighbour = neighbour,
                               printBar = FALSE,
                               nS = 1000))
```

```
Threshold Accepting

  Computing thresholds ...   OK
  Estimated remaining running time: 0.23 secs

  Running Threshold Accepting ...
  Initial solution: 7
```

```
  Finished.
  Best solution overall: 0
```

```
> print_board(sol$xbest)
```

```
- - - - Q - - -
- - - - - - Q -
- Q - - - - - -
- - - Q - - - -
- - - - - - - Q
Q - - - - - - -
- - Q - - - - -
- - - - - Q - -
```

```
> sol <- SAopt(n_attacks, list(x0 = p0,
                               neighbour = neighbour,
                               printBar = FALSE,
                               nS = 1000))
```

```
Simulated Annealing.

Calibrating acceptance criterion ... OK
Estimated remaining running time: 0.225 secs.

Running Simulated Annealing ...
Initial solution:  7
Finished.
Best solution overall: 0
```

```
> print_board(sol$xbest)
```

```
- - - - Q - - -
- - - - - - Q -
- Q - - - - - -
- - - Q - - - -
- - - - - - - Q
Q - - - - - - -
- - Q - - - - -
- - - - - Q - -
```

References

Manfred Gilli, Dietmar Maringer, and Enrico Schumann. *Numerical Methods and Optimization in Finance*. Elsevier, 2011.