Lab 6

Hugo Morvan, Daniele Bozzoli

2023-12-12

Question 1: Genetic algorithm

In this assignment, you will try to solve the n–queen problem using a genetic algorithm. Given an n by n chessboard, the task is to place n queens on it so that no queen is attacked by any other queen. You can read more about the problem at https://en.wikipedia.org/wiki/Eight_queens_puzzle.

1.

An individual in the population is a chessboard with some placement of the n queens on it. The first task is to code an individual. You are to consider three encodings for this question.

- (a) A collection (e.g., a list—but the choice of data structure is up to you) of n pairs denoting the coordinates of each queen, e.g., (5, 6) would mean that a queen is standing in row 5 and column 6.

```
#1.1.a
# Pairs representing the coordinates of each queen, stored in a matrix
# Representation_type = "pairs"
n <- 5
pairs <- matrix(nrow = n, ncol = 2)
for (i in 1:n){
  pairs[i,] <- sample(1:n, 2)
}
typeof(pairs) #integer
```

```
## [1] "integer"
```

```
ncol(pairs) #2
```

```
## [1] 2
```

```
print(pairs)
```

```
##      [,1] [,2]
## [1,]    3    1
## [2,]    3    4
## [3,]    2    5
## [4,]    1    4
## [5,]    1    5
```

- (b) On n numbers, where each number has log2 n binary digits—this number encodes the position of the queen in the given column. Notice that as queens cannot attack each other, in a legal configuration there can be only one queen per column. You can pad your binary representation with 0s if necessary.

```
#1.1.b
# A list of n numbers, where each number is a binary representation of the position of the queen
# in the given column
# Representation_type = "binary"
n <- 5
binary <- list()
for (i in 1:n){
  binary[[i]] <- c(as.integer(intToBits(sample(1:n, 1))))
}
typeof(binary) #list
```

```
## [1] "list"
```

```
ncol(binary) #NULL
```

```
## NULL
```

```
print(binary)
```

```
## [[1]]
##  [1] 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##
## [[2]]
##  [1] 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##
## [[3]]
##  [1] 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##
## [[4]]
##  [1] 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##
## [[5]]
##  [1] 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

*(c) On n numbers, where each number is the row number of the queen in each column. Notice that this encoding differs from the previous one by how the row position is stored. Here it is an integer, in item 1b it was represented through its binary representation. This will induce different ways of crossover and mutating the state.

```
#1.1.c
# A list of n numbers, where each number is the row number of the queen in each column
# Representation_type = "singles"
n <- 5

row <- c()
for (i in 1:n){
```

```
  row[i] <- sample(1:n, 1)
}
typeof(row) #integer
```

```
## [1] "integer"
```

```
ncol(row) #NULL
```

```
## NULL
```

```
print(row)
```

```
## [1] 1 5 3 4 4
```

The tasks below, 2–7 are to be repeated for each of the three encodings above.

```
vizualize_board <- function(layout, n){
  #Given a layout, vizualizes the board
  #cat("Type of layout", typeof(layout), "\n", "Number of columns", ncol(layout), "\n")
  if(typeof(layout) == "list" & is.null(ncol(layout))){
    #Binary
    #print("Viz: Binary rep detected")
    #Idea: create a matrix of 0s, then replace the 0s with 1s where the queens are
    board <- matrix(nrow = n, ncol = n)
    #fill the board with 0s
    for (i in 1:nrow(board)){
      for (j in 1:ncol(board)){
        board[i,j] <- 0
      }
    }
    for (i in 1:n){
      for (j in 1:length(layout[[i]])){
        if(layout[[i]][j] == 1){
          board[i,j] <- 1
        }
      }
    }
    board <- t(board)
    #print(board)
  }else if(typeof(layout) == "integer" & is.null(ncol(layout))){
    #Singles
    #print("Viz: Singles rep detected")
    #Idea: create a matrix of 0s, then replace the 0s with 1s where the queens are
    board <- matrix(nrow = n, ncol = n)
    #fill the board with 0s
    for (i in 1:nrow(board)){
      for (j in 1:ncol(board)){
        board[i,j] <- 0
      }
    }
    for (i in 1:n){
```

```
      board[i,layout[i]] <- 1
    }
    board <- t(board)
    #print(board)
  }else if(typeof(layout) == "integer" & ncol(layout) == 2){
    #Pairs
    #print("Viz: Pairs rep detected")
    #IDea: create a matrix of 0s, then replace the 0s with 1s where the queens are
    board <- matrix(nrow = n, ncol = n)
    #fill the board with 0s
    for (i in 1:nrow(board)){
      for (j in 1:ncol(board)){
        board[i,j] <- 0
      }
    }
    for (i in 1:nrow(layout)){
      board[layout[i,1], layout[i,2]] <- 1
    }
    #print(board)
  }
  #Print the board in the console:
  for (i in 1:nrow(board)){
    cat("|")
    for (j in 1:ncol(board)){
      if(board[i,j] == 0){
        cat(" |")
      }else{
        cat("Q|")
      }
    }
    cat("\n")
  }
}
# pairs
# vizualize_board(pairs, 5)
# binary
# vizualize_board(binary, 5)
# row
# vizualize_board(row ,5)
```

2.

Define the function crossover(): for two chessboard layouts it creates a kid by taking columns 1,. . . , p from the first individual and columns p + 1,. . . , n from the second. Obviously, 0 < p <= n/2, and p in N. Experiment with different values of p.

```
#1.2
crossover <- function(layout1, layout2, p){
  #Given 2 chessboard layouts, returns a kid by taking columns 1,..., p from the first individual and c
  if(typeof(layout1) == "list" & is.null(ncol(layout1))){
    #BINARY (one binary rep = one column)
    #print("Cross: Binary rep detected")
    #Idea: take the first p columns from layout 1 and adds them to the kid, then take the last n-p colm
```

```
    kid <- list()
    for (i in 1:p){
      kid[[i]] <- layout1[[i]]
    }
    for (i in (p+1):length(layout2)){
      kid[[i]] <- layout2[[i]]
    }

    return(kid)
  }
  else if(typeof(layout1) == "integer" & is.null(ncol(layout1))){
    #SINGLES
    #print("Cross: Singles rep detected")
    #Take the first p number from layout 1 and adds them to the kid, then take the last n-p numbers fro
    kid <- c()
    for (i in 1:p){
      kid <- c(kid , layout1[i])
    }
    for (i in (p+1):length(layout2)){
      kid <- c(kid , layout2[i])
    }
    return(kid)
  }else if(typeof(layout1) == "integer" & ncol(layout1) == 2){
    #PAIRS (matrix)
    #print("Cross: Pairs rep detected")
    #Idea : for layout1 and layout 2, take the first p pairs from layout1 and the last n-p pairs from l
    kid <- matrix(nrow = nrow(layout1), ncol = 2)
    for (i in 1:p){
      kid[i,] <- layout1[i,]
    }
    for (i in (p+1):nrow(layout2)){
      kid[i,] <- layout2[i,]
    }
    return(kid)
  }

}
```

```
#Testing the crossover function for n = 4, layout 1 is full of 1s, layout 2 is full of 0s, and p = 2

#binary rep:
layout1 <- list()
layout2 <- list()
for (i in 1:4){
  layout1[[i]] <- c(1,1,1,1)
  layout2[[i]] <- 0
}
print("layout1")
```

```
## [1] "layout1"
```

```
vizualize_board(layout1, 4)
```

```
## |Q|Q|Q|Q|
## |Q|Q|Q|Q|
## |Q|Q|Q|Q|
## |Q|Q|Q|Q|
```

```
print("layout2")
```

```
## [1] "layout2"
```

```
vizualize_board(layout2, 4)
```

```
## | | | | |
## | | | | |
## | | | | |
## | | | | |
```

```
cross1 <- crossover(layout1, layout2, 2)
print("Crossover of layout1 and layout2 on p = 2")
```

```
## [1] "Crossover of layout1 and layout2 on p = 2"
```

```
vizualize_board(cross1, 4)
```

```
## |Q|Q| | |
## |Q|Q| | |
## |Q|Q| | |
## |Q|Q| | |
```

```
print("______________________")
```

```
## [1] "______________________"
```

```
#singles rep:
layout3 <- as.integer(c(1,2,3,4))
layout4 <- as.integer(c(4,3,2,1))
print("layout3")
```

```
## [1] "layout3"
```

```
vizualize_board(layout3, 4)
```

```
## |Q| | | |
## | |Q| | |
## | | |Q| |
## | | | |Q|
```

```r
print("layout4")
```

```
## [1] "layout4"
```

```r
vizualize_board(layout4, 4)
```

```
## | | | |Q|
## | | |Q| |
## | |Q| | |
## |Q| | | |
```

```r
cross2 <- crossover(layout3, layout4, 2)
print("Crossover of layout3 and layout4 on p = 2")
```

```
## [1] "Crossover of layout3 and layout4 on p = 2"
```

```r
vizualize_board(cross2, 4)
```

```
## |Q| | |Q|
## | |Q|Q| |
## | | | | |
## | | | | |
```

```r
print("_____________________")
```

```
## [1] "_____________________"
```

```r
#pairs rep:
layout5 <- matrix(as.integer(c(1,1,1,1,1,2,3,4)), nrow = 4, ncol = 2)
layout6 <- matrix(as.integer(c(4,4,4,4,1,2,3,4)), nrow = 4, ncol = 2)
print("layout5")
```

```
## [1] "layout5"
```

```r
vizualize_board(layout5, 4)
```

```
## |Q|Q|Q|Q|
## | | | | |
## | | | | |
## | | | | |
```

```r
print("layout6")
```

```
## [1] "layout6"
```

```
vizualize_board(layout6, 4)
```

```
## | | | | |
## | | | | |
## | | | | |
## |Q|Q|Q|Q|
```

```
cross3 <- crossover(layout5, layout6, 2)
print("Crossover of layout5 and layout6 on p = 2")
```

```
## [1] "Crossover of layout5 and layout6 on p = 2"
```

```
vizualize_board(cross3, 4)
```

```
## |Q|Q| | |
## | | | | |
## | | | | |
## | | |Q|Q|
```

3.

Define the function mutate() that randomly moves a queen to a new position.

```
#1.3
mutate <- function(layout,n){

  #Given a chessboard layout, returns a layout with a randomly moved queen

  if(typeof(layout) == "list" & is.null(ncol(layout))){
    #BINARY
    #print("Mut: Binary rep detected")
    #randomly select a 0 location
    zero_coords <- c()
    for (i in 1:n){
      for (j in 1:n){
        if(layout[[i]][j] == 0){
          zero_coords <- c(zero_coords, i)
          zero_coords <- c(zero_coords, j)
        }
      }
    }
    rand_idx0 <- sample(length(zero_coords)/2, 1)
    y0 <- zero_coords[rand_idx0*2]
    x0 <- zero_coords[rand_idx0*2 - 1]

    #randomly select a 1 location
    one_coords <- c()
    for (i in 1:n){
      for (j in 1:n){
        if(layout[[i]][j] == 1){
          one_coords <- c(one_coords, i)
```

```
          one_coords <- c(one_coords, j)
        }
      }
    }
    rand_idx1 <- sample(length(one_coords)/2, 1)
    y1 <- one_coords[rand_idx1*2]
    x1 <- one_coords[rand_idx1*2 - 1]

    #set the 0 to a 1 and the 1 to a 0

    layout[[x0]][y0] <- 1
    layout[[x1]][y1] <- 0
    return(layout)

  }else if(typeof(layout) == "integer" & is.null(ncol(layout))){
    #SINGLES
    #print("Mut: Singles rep detected")
    #randomly select a row and change the value of the number at this position, and check if the new qu
    row <- sample(n,1)
    layout[row] <- sample(n,1)
    #Disadvantages of this method : a queen can only be moved within a column, and it can also not be m
    return(layout)

  }else if(typeof(layout) == "integer" & ncol(layout) == 2){
    #PAIRS (matrix)
    #print("Mut: Pairs rep detected")
    #randomly select a row and a column and change the value of the pair at this position
    #check if the new queen is not on another queen , and if it is not off the board
    pair <- sample(n,1)
    layout[pair,1] <- sample(n,1)
    layout[pair,2] <- sample(n,1)
    #Disadvantages of this layout : can move to a location where there is already a queen
    # --> Good or bad ? Good: simulate a queen "disappearing", bad: duplicates in the representation
    return(layout)
  }
}
```

```
#Testing the mutate function for each representation
#binary rep:
layout1 <- list()
layout1[[1]] <- c(0,0,0,0)
layout1[[2]] <- c(1,1,0,0)
layout1[[3]] <- c(0,0,0,0)
layout1[[4]] <- c(0,0,0,0)
vizualize_board(layout1, 4)
```

```
## | |Q| | |
## | |Q| | |
## | | | | |
## | | | | |
```

```
mutated1 <- mutate(layout1, 4)
vizualize_board(mutated1, 4)
```

```
## | |Q| | |
## | | | | |
## | | | |Q|
## | | | | |
```

```
print("_______________________________")
```

```
## [1] "_______________________________"
```

```
#singles rep:
layout2 <- as.integer(c(1,2,1,3))
vizualize_board(layout2, 4)
```

```
## |Q| |Q| |
## | |Q| | |
## | | | |Q|
## | | | | |
```

```
mutated2 <- mutate(layout2, 4)
vizualize_board(mutated2, 4)
```

```
## |Q|Q|Q| |
## | | | | |
## | | | |Q|
## | | | | |
```

```
print("_______________________________")
```

```
## [1] "_______________________________"
```

```
#pairs rep:
layout3 <- matrix(as.integer(c(1,1,1,1,1,2,3,4)), nrow = 4, ncol = 2)
vizualize_board(layout3, 4)
```

```
## |Q|Q|Q|Q|
## | | | | |
## | | | | |
## | | | | |
```

```
mutated3 <- mutate(layout3, 4)
vizualize_board(mutated3, 4)
```

```
## |Q|Q| |Q|
## |Q| | | |
## | | | | |
## | | | | |
```

4.

Define a fitness function for a given configuration. Experiment with three: 1) binary — is a solution or not; "binary" 2) number of queens not attacked; "num 3) (nC2 - number) of pairs of queens attacking each other. If needed scale the value of the fitness function to [0, 1]. Experiment which could be the best one. Try each fitness function for each encoding method. You should not expect the binary fitness function to work well, explain why this is so.

```
#1.1.4
fitness <- function(layout, fitness_function, n){
  #Given a chessboard layout, returns the fitness of the layout
  if(typeof(layout) == "list" & is.null(ncol(layout))){

    #Binary
    #print("Viz: Binary rep detected")
    #Idea: create a matrix of 0s, then replace the 0s with 1s where the queens are
    board <- matrix(nrow = n, ncol = n)
    #fill the board with 0s
    for (i in 1:nrow(board)){
      for (j in 1:ncol(board)){
        board[i,j] <- 0
      }
    }
    for (i in 1:n){
      for (j in 1:length(layout[[i]])){
        if(layout[[i]][j] == 1){
          board[i,j] <- 1
        }
      }
    }
    board <- t(board)
  }else if(typeof(layout) == "integer" & is.null(ncol(layout))){
    #print("Fit: Singles rep detected")
    #Convert to a matrix
    board <- matrix(nrow = n, ncol = n)
    #fill the board with 0s
    for (i in 1:nrow(board)){
      for (j in 1:ncol(board)){
        board[i,j] <- 0
      }
    }
    for (i in 1:n){
      board[i,layout[i]] <- 1
    }
    board <- t(board)
  }else if(typeof(layout) == "integer" & ncol(layout) == 2){
    #print("Fit: Pairs rep detected")
    #Convert to a matrix
    board <- matrix(nrow = n, ncol = n)
    #fill the board with 0s
    for (i in 1:nrow(board)){
      for (j in 1:ncol(board)){
        board[i,j] <- 0
      }
```

```
  }
  for (i in 1:nrow(layout)){
    board[layout[i,1], layout[i,2]] <- 1
  }
}
#================================================================================
#At this point, the board var should contain a matrix on the state of the board, with 1s for queens a
#Convert the board to a list of coordinates
my_vec <- c()
for(i in 1:n){
  for(j in 1:n){
    if(board[i,j]==1){
      my_vec <- c(my_vec, c(i,j))
    }
  }
}
if(length(my_vec) == 0){
  return(0)
}
points<-array( my_vec, dim=c(2,length(my_vec)/2))
are_attacking_eachother <- function(Q1,Q2){
  #https://stackoverflow.com/questions/57239548/how-to-check-if-a-queen-is-under-attack-in-nqueens
  #"A queen can attack another if :
  #they are on the same line : X1 = X2
  #they are on the same column : Y1 = Y2
  #they are on the same diagonal : X1-Y1 = X2-Y2 or X1+Y1 = X2+Y2"
  (Q1[1] == Q2[1] | Q1[2] == Q2[2] | Q1[1]-Q1[2] == Q2[1]-Q2[2] | Q1[1]+Q1[2] == Q2[1]+Q2[2])
}
fitness <- 0
if(length(points)/2 == 1){
  if(fitness_function == "binary"){
    return(0)
  }else if(fitness_function == "num_safe"){
    return(1)
  }else{
    #number of attacking pairs
    return(0)
  }
}else if(fitness_function == "binary"){
    for(queen in 1:((length(points)/2)-1)){
      for(other_queen in (queen+1):(length(points)/2)){
        Q1 <- c(points[1,queen], points[2,queen])
        Q2 <- c(points[1,other_queen], points[2,other_queen])
        if(are_attacking_eachother(Q1, Q2)){
          fitness <- 1
        }
      }
    }
}else if(fitness_function == "num_safe"){
  safe <- 0
   for(queen in 1:((length(points)/2)-1)){
      for(other_queen in (queen+1):(length(points)/2)){
        Q1 <- c(points[1,queen], points[2,queen])
```

```
          Q2 <- c(points[1,other_queen], points[2,other_queen])
          if(!are_attacking_eachother(Q1, Q2)){
            safe <- safe + 2
          }
        }
     }
    #to scale to [0,1], we can divide by the max number of queens that can be safe
    fitness <- safe
  }else{
      #(nC2 - number) of pairs of queens attacking each other.
      nc2 <- (n*(n-1)/2)
      num_attacking <- 0
      for(queen in 1:((length(points)/2)-1)){
        for(other_queen in (queen+1):(length(points)/2)){
          Q1 <- c(points[1,queen], points[2,queen])
          Q2 <- c(points[1,other_queen], points[2,other_queen])
          if(are_attacking_eachother(Q1, Q2)){
            fitness <- 1
          }
        }
      }
      fitness <- nc2 - num_attacking
  }
  return(fitness)
}
```

```
layout1 <- list()
layout2 <- list()
layout3 <- list()
for (i in 1:4){
  layout1[[i]] <- 0
  layout2[[i]] <- 0
  layout3[[i]] <- 0
}

layout1[[1]] <- c(1,0,0,0)
layout1[[4]] <- c(0,0,0,1)

layout2[[1]] <- c(1,0,0,0)

layout3[[1]] <- c(1,0,0,0)
layout3[[3]] <- c(0,0,0,1)

vizualize_board(layout1, 4)
```

```
## |Q| | | |
## | | | | |
## | | | | |
## | | | |Q|
```

```
print("binary fitness of layout 1")
```

```
## [1] "binary fitness of layout 1"
```

```r
fitness(layout = layout1, fitness_function = "binary", 4)
```

```
## [1] 1
```

```r
vizualize_board(layout2, 4)
```

```
## |Q| | | |
## | | | | |
## | | | | |
## | | | | |
```

```r
print("binary fitness of layout 2")
```

```
## [1] "binary fitness of layout 2"
```

```r
fitness(layout = layout2, fitness_function = "binary", 4)
```

```
## [1] 0
```

```r
vizualize_board(layout3, 4)
```

```
## |Q| | | |
## | | | | |
## | | | | |
## | | |Q| |
```

```r
print("binary fitness of layout 3")
```

```
## [1] "binary fitness of layout 3"
```

```r
fitness(layout = layout3, fitness_function = "binary", 4)
```

```
## [1] 0
```

```r
print("_____________________________________________________")
```

```
## [1] "_____________________________________________________"
```

```r
vizualize_board(layout1, 4)
```

```
## |Q| | | |
## | | | | |
## | | | | |
## | | | |Q|
```

```
print("num_safe fitness of layout 1")
```

```
## [1] "num_safe fitness of layout 1"
```

```
fitness(layout = layout1, fitness_function = "num_safe", 4)
```

```
## [1] 0
```

```
vizualize_board(layout2, 4)
```

```
## |Q| | | |
## | | | | |
## | | | | |
## | | | | |
```

```
print("num_safe fitness of layout 2")
```

```
## [1] "num_safe fitness of layout 2"
```

```
fitness(layout = layout2, fitness_function = "num_safe", 4)
```

```
## [1] 1
```

```
vizualize_board(layout3, 4)
```

```
## |Q| | | |
## | | | | |
## | | | | |
## | | |Q| |
```

```
print("num_safe fitness of layout 3")
```

```
## [1] "num_safe fitness of layout 3"
```

```
fitness(layout = layout3, fitness_function = "num_safe", 4)
```

```
## [1] 2
```

```
print("_____________________________________________________")
```

```
## [1] "_____________________________________________________"
```

5.

Implement a genetic algorithm that takes the choice of encoding, mutation probability, and fitness function as parameters. Your implementation should start with a random initial configuration. Each element of the population should have its fitness calculated. Do not forget to have in your code a limit for the number of iterations (but this limit should not be lower than 100, unless this causes running time issues, which should be clearly presented then), so that your code does not run forever. Count the number of pairs of queens attacking each other. At each iteration :

- (a) Two individuals are randomly sampled from the current population, they are further used as parents (use sample()).
- (b) One individual with the smallest fitness is selected from the current population, this will be the victim (use order()).
- (c) The two sampled parents are to produce a kid by crossover, and this kid should be mutated with probability mutprob (use crossover(), mutate()).
- (d) The victim is replaced by the kid in the population.
- (e) Do not forget to update the vector of fitness values of the population.
- (f) Remember the number of pairs of queens attacking each other at the given iteration.

```
#1.5
generate_random_layout <- function(n){
  layout <- list()
  for (i in 1:n){
    layout[[i]] <- c(sample(1:n, 1), sample(1:n, 1), sample(1:n, 1), sample(1:n, 1))
  }
  return(layout)
}
vizualize_board(generate_random_layout(4),4)
n_generations <- 100
n_queens <- 4
n_population <- 10
mutprob <- 0.1
fitness_function <- "binary"

#generate random population
population <- list()
for(i in 1:n_population){
  population[[i]] <- generate_random_layout(n_queens)
}

#calculate fitness of population
fitnesses <- c()
for(i in 1:n_population){
  fitnesses[i] <- fitness(population[[i]], fitness_function, n_queens)
}
n_attacking_queens <- c()
#iterate through generations
for(generation in 1:n_generations){
  cat("generation: ", generation, "\r")
  #Two individuals are randomly sampled from the current population, they are further used as parents
```

```
  parents <- sample(population, 2, replace = FALSE)
  #One individual with the smallest fitness is selected from the current population, this will be the v
  victim <- population[which.min(fitnesses)]
  #The two sampled parents are to produce a kid by crossover
  kid <- crossover(parents[[1]], parents[[2]], p=2)
  #this kid should be mutated with probability mutprob
  kid <- mutate(kid, mutprob)
  #The victim is replaced by the kid in the population
  population[which(population == victim)] <- kid
  #Do not forget to update the vector of fitness values of the population
  fitnesses[which(population == victim)] <- fitness(kid, fitness_function, n_queens)
  #Remember the number of pairs of queens attacking each other at the given iteration
  n_attacking_queens <- c(n_attacking_queens, sum(fitnesses))
}
```

6.

If found, return the legal configuration of queens.

```
#1.6
```

7.

Provide a plot of the number of pairs queens attacking each other at each iteration of the algorithm.

```
#1.7
```

8.

Run your code for n = 4, 8, 16 (if n = 16 requires too much computational time take a different n in {10, 11, 12, 13, 14, 15}, but do not forget that this is not a power of 2 and more care is needed in the second encoding), the different encodings, objective functions, and mutprob= 0.1, 0.5, 0.9. Did you find a legal state?

```
#1.8
```

9.

Discuss which encoding and objective function worked best.

```
#1.9
```

Question 2: EM algorithm

The data file censoredproc.csv contains the time after which a certain product fails. Some of these measurements are left-censored (cens=2)—i.e., we did not observe the time of failure, only that the product had already failed when checked upon. Status cens=1 means that the exact time of failure was observed.

1.

Plot a histogram of the values. Do it for all of the data, and also when the censored observations are removed. Do the histograms remind of an exponential distribution?

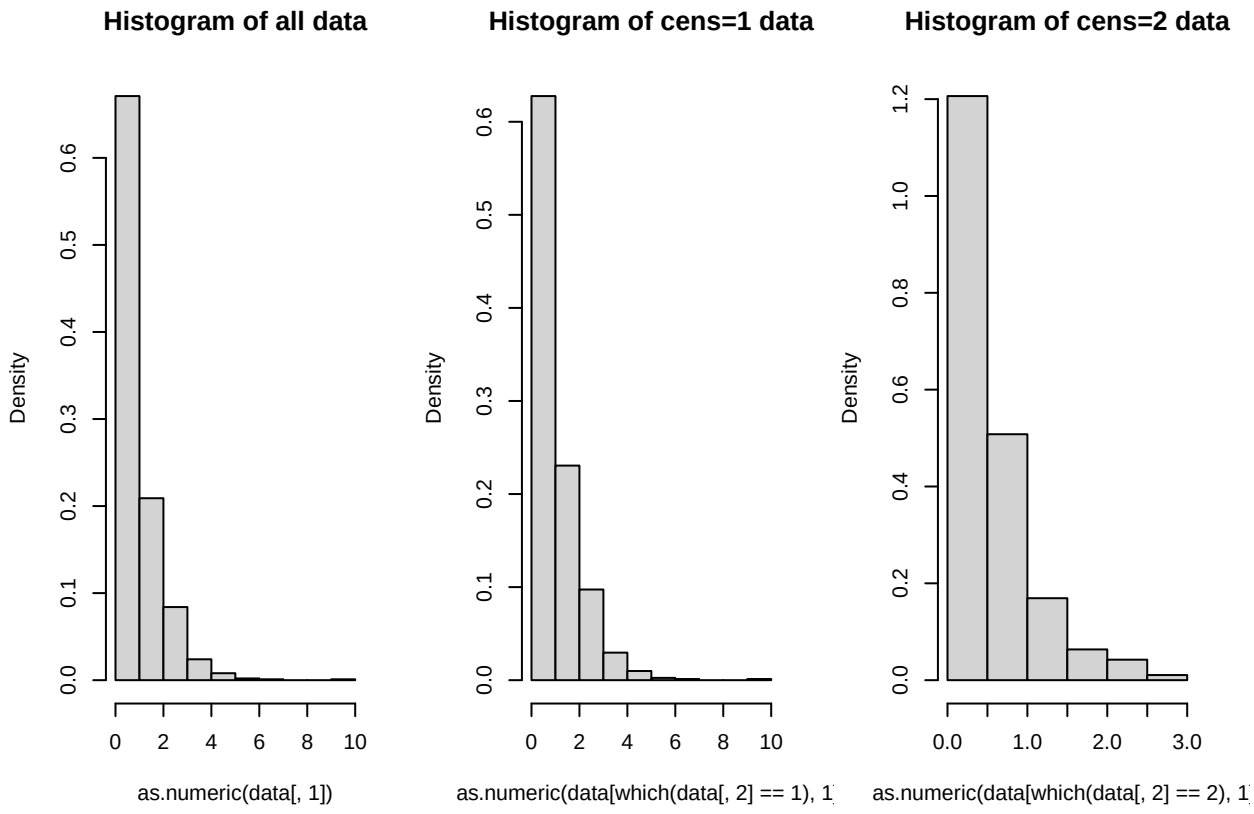
```
# 2.1
data <- read.csv2("censoredproc.csv")

par(mfrow=c(1,3))

# histogram of all data
hist(as.numeric(data[,1]), freq=F, main="Histogram of all data")

# histogram of data with cens=1
hist(as.numeric(data[which(data[,2]==1),1]), freq=F, main="Histogram of cens=1 data")

# histogram of data with cens=2
hist(as.numeric(data[which(data[,2]==2),1]), freq=F, main="Histogram of cens=2 data")
```




```
min(as.numeric(data[which(data[,2]==1),1]))
```

```
## [1] 0.0006415918
```

```r
min(as.numeric(data[which(data[,2]==2),1]))
```

```
## [1] 0.0002742931
```

Yes, they both look like exponential distributions.

2.

Assume that the underlying data comes from an exponential distribution with parameter λ. This means that observed values come from the exponential λ distribution, while censored from a truncated exponential distribution. Write down the likelihood function.

```r
# 2.2
```

3.

The goal now is to derive an EM algorithm that estimates λ. Based on the above found likelihood function, derive the EM algorithm for estimating λ. The formula in the M –step can be differentiated, but the derivative is non–linear in terms of λ so its zero might need to be found numerically.

```r
#2.3
```

4.

Implement the above in R. Take $\lambda_0 = 100$ as the starting value for the algorithm and stopping condition if the change in the estimate is less than 0.001. At what $\hat{\lambda}$ did the EM algorithm stop at? How many iterations were required?

```r
#2.4
```

5.

Plot the density curve of the $exp(\hat{\lambda})$ distribution over your histograms in task 1.

```r
#2.5
```

6.

Study how good your EM algorithm is compared to usual maximum likelihood estimation with data reduced to only the uncensored observations. To this end we will use a parametric bootstrap. Repeat 1000 (reduce if computational time is too long—but carefully report the running times) times the following procedure:

- (a) Simulate the same number of data points as in the original data, from the exponential $\hat{\lambda}$ distribution.
- (b) Randomly select the same number of points as in the original data for censoring. For each observation for censoring—sample a new time from the uniform distribution on [0, true time]. Remember that the observation was censored.

- (c) Estimate λ both by your EM-algorithm, and maximum likelihood based on the uncensored observations. Compare the distributions of the estimates of λ from the two methods. Plot the histograms, report whether they both seem unbiased, and what is the variance of the estimators.

```
#2.6
```

Appendix

```
knitr::opts_chunk$set(echo = TRUE)
#1.1.a
# Pairs representing the coordinates of each queen, stored in a matrix
# Representation_type = "pairs"
n <- 5
pairs <- matrix(nrow = n, ncol = 2)
for (i in 1:n){
  pairs[i,] <- sample(1:n, 2)
}
typeof(pairs) #integer
ncol(pairs) #2
print(pairs)
#1.1.b
# A list of n numbers, where each number is a binary representation of the position of the queen
# in the given column
# Representation_type = "binary"
n <- 5
binary <- list()
for (i in 1:n){
  binary[[i]] <- c(as.integer(intToBits(sample(1:n, 1))))
}
typeof(binary) #list
ncol(binary) #NULL
print(binary)

#1.1.c
# A list of n numbers, where each number is the row number of the queen in each column
# Representation_type = "singles"
n <- 5

row <- c()
for (i in 1:n){
  row[i] <- sample(1:n, 1)
}
typeof(row) #integer
ncol(row) #NULL
print(row)
vizualize_board <- function(layout, n){
  #Given a layout, vizualizes the board
  #cat("Type of layout", typeof(layout), "\n", "Number of columns", ncol(layout), "\n")
  if(typeof(layout) == "list" & is.null(ncol(layout))){
    #Binary
    #print("Viz: Binary rep detected")
```

```
  #Idea: create a matrix of 0s, then replace the 0s with 1s where the queens are
  board <- matrix(nrow = n, ncol = n)
  #fill the board with 0s
  for (i in 1:nrow(board)){
    for (j in 1:ncol(board)){
      board[i,j] <- 0
    }
  }
  for (i in 1:n){
    for (j in 1:length(layout[[i]])){
      if(layout[[i]][j] == 1){
        board[i,j] <- 1
      }
    }
  }
  board <- t(board)
  #print(board)
}else if(typeof(layout) == "integer" & is.null(ncol(layout))){
  #Singles
  #print("Viz: Singles rep detected")
  #Idea: create a matrix of 0s, then replace the 0s with 1s where the queens are
  board <- matrix(nrow = n, ncol = n)
  #fill the board with 0s
  for (i in 1:nrow(board)){
    for (j in 1:ncol(board)){
      board[i,j] <- 0
    }
  }
  for (i in 1:n){
    board[i,layout[i]] <- 1
  }
  board <- t(board)
  #print(board)
}else if(typeof(layout) == "integer" & ncol(layout) == 2){
  #Pairs
  #print("Viz: Pairs rep detected")
  #IDea: create a matrix of 0s, then replace the 0s with 1s where the queens are
  board <- matrix(nrow = n, ncol = n)
  #fill the board with 0s
  for (i in 1:nrow(board)){
    for (j in 1:ncol(board)){
      board[i,j] <- 0
    }
  }
  for (i in 1:nrow(layout)){
    board[layout[i,1], layout[i,2]] <- 1
  }
  #print(board)
}
#Print the board in the console:
for (i in 1:nrow(board)){
  cat("|")
  for (j in 1:ncol(board)){
```

```
      if(board[i,j] == 0){
        cat(" |")
      }else{
        cat("Q|")
      }
    }
    cat("\n")
  }
}
# pairs
# vizualize_board(pairs, 5)
# binary
# vizualize_board(binary, 5)
# row
# vizualize_board(row ,5)
#1.2
crossover <- function(layout1, layout2, p){
  #Given 2 chessboard layouts, returns a kid by taking columns 1,..., p from the first individual and c
  if(typeof(layout1) == "list" & is.null(ncol(layout1))){
    #BINARY (one binary rep = one column)
    #print("Cross: Binary rep detected")
    #Idea: take the first p columns from layout 1 and adds them to the kid, then take the last n-p colm
    kid <- list()
    for (i in 1:p){
      kid[[i]] <- layout1[[i]]
    }
    for (i in (p+1):length(layout2)){
      kid[[i]] <- layout2[[i]]
    }

    return(kid)
  }
  else if(typeof(layout1) == "integer" & is.null(ncol(layout1))){
    #SINGLES
    #print("Cross: Singles rep detected")
    #Take the first p number from layout 1 and adds them to the kid, then take the last n-p numbers fro
    kid <- c()
    for (i in 1:p){
      kid <- c(kid , layout1[i])
    }
    for (i in (p+1):length(layout2)){
      kid <- c(kid , layout2[i])
    }
    return(kid)
  }else if(typeof(layout1) == "integer" & ncol(layout1) == 2){
    #PAIRS (matrix)
    #print("Cross: Pairs rep detected")
    #Idea : for layout1 and layout 2, take the first p pairs from layout1 and the last n-p pairs from l
    kid <- matrix(nrow = nrow(layout1), ncol = 2)
    for (i in 1:p){
      kid[i,] <- layout1[i,]
    }
    for (i in (p+1):nrow(layout2)){
```

```
      kid[i,] <- layout2[i,]
    }
    return(kid)
  }

}

#Testing the crossover function for n = 4, layout 1 is full of 1s, layout 2 is full of 0s, and p = 2

#binary rep:
layout1 <- list()
layout2 <- list()
for (i in 1:4){
  layout1[[i]] <- c(1,1,1,1)
  layout2[[i]] <- 0
}
print("layout1")
vizualize_board(layout1, 4)
print("layout2")
vizualize_board(layout2, 4)
cross1 <- crossover(layout1, layout2, 2)
print("Crossover of layout1 and layout2 on p = 2")
vizualize_board(cross1, 4)
print("______________________")

#singles rep:
layout3 <- as.integer(c(1,2,3,4))
layout4 <- as.integer(c(4,3,2,1))
print("layout3")
vizualize_board(layout3, 4)
print("layout4")
vizualize_board(layout4, 4)
cross2 <- crossover(layout3, layout4, 2)
print("Crossover of layout3 and layout4 on p = 2")
vizualize_board(cross2, 4)
print("______________________")

#pairs rep:
layout5 <- matrix(as.integer(c(1,1,1,1,1,2,3,4)), nrow = 4, ncol = 2)
layout6 <- matrix(as.integer(c(4,4,4,4,1,2,3,4)), nrow = 4, ncol = 2)
print("layout5")
vizualize_board(layout5, 4)
print("layout6")
vizualize_board(layout6, 4)
cross3 <- crossover(layout5, layout6, 2)
print("Crossover of layout5 and layout6 on p = 2")
vizualize_board(cross3, 4)
#1.3
mutate <- function(layout,n){

  #Given a chessboard layout, returns a layout with a randomly moved queen

  if(typeof(layout) == "list" & is.null(ncol(layout))){
```

```
  #BINARY
  #print("Mut: Binary rep detected")
  #randomly select a 0 location
  zero_coords <- c()
  for (i in 1:n){
    for (j in 1:n){
      if(layout[[i]][j] == 0){
        zero_coords <- c(zero_coords, i)
        zero_coords <- c(zero_coords, j)
      }
    }
  }
  rand_idx0 <- sample(length(zero_coords)/2, 1)
  y0 <- zero_coords[rand_idx0*2]
  x0 <- zero_coords[rand_idx0*2 - 1]

  #randomly select a 1 location
  one_coords <- c()
  for (i in 1:n){
    for (j in 1:n){
      if(layout[[i]][j] == 1){
        one_coords <- c(one_coords, i)
        one_coords <- c(one_coords, j)
      }
    }
  }
  rand_idx1 <- sample(length(one_coords)/2, 1)
  y1 <- one_coords[rand_idx1*2]
  x1 <- one_coords[rand_idx1*2 - 1]

  #set the 0 to a 1 and the 1 to a 0

  layout[[x0]][y0] <- 1
  layout[[x1]][y1] <- 0
  return(layout)

}else if(typeof(layout) == "integer" & is.null(ncol(layout))){
  #SINGLES
  #print("Mut: Singles rep detected")
  #randomly select a row and change the value of the number at this position, and check if the new qu
  row <- sample(n,1)
  layout[row] <- sample(n,1)
  #Disadvantages of this method : a queen can only be moved within a column, and it can also not be m
  return(layout)

}else if(typeof(layout) == "integer" & ncol(layout) == 2){
  #PAIRS (matrix)
  #print("Mut: Pairs rep detected")
  #randomly select a row and a column and change the value of the pair at this position
  #check if the new queen is not on another queen , and if it is not off the board
  pair <- sample(n,1)
  layout[pair,1] <- sample(n,1)
  layout[pair,2] <- sample(n,1)
```

```
    #Disadvantages of this layout : can move to a location where there is already a queen
    # --> Good or bad ? Good: simulate a queen "disappearing", bad: duplicates in the representation
    return(layout)
  }
}
#Testing the mutate function for each representation
#binary rep:
layout1 <- list()
layout1[[1]] <- c(0,0,0,0)
layout1[[2]] <- c(1,1,0,0)
layout1[[3]] <- c(0,0,0,0)
layout1[[4]] <- c(0,0,0,0)
vizualize_board(layout1, 4)
mutated1 <- mutate(layout1, 4)
vizualize_board(mutated1, 4)
print("_______________________________")

#singles rep:
layout2 <- as.integer(c(1,2,1,3))
vizualize_board(layout2, 4)
mutated2 <- mutate(layout2, 4)
vizualize_board(mutated2, 4)
print("_______________________________")

#pairs rep:
layout3 <- matrix(as.integer(c(1,1,1,1,1,2,3,4)), nrow = 4, ncol = 2)
vizualize_board(layout3, 4)
mutated3 <- mutate(layout3, 4)
vizualize_board(mutated3, 4)
#1.1.4
fitness <- function(layout, fitness_function, n){
  #Given a chessboard layout, returns the fitness of the layout
  if(typeof(layout) == "list" & is.null(ncol(layout))){

    #Binary
    #print("Viz: Binary rep detected")
    #Idea: create a matrix of 0s, then replace the 0s with 1s where the queens are
    board <- matrix(nrow = n, ncol = n)
    #fill the board with 0s
    for (i in 1:nrow(board)){
      for (j in 1:ncol(board)){
        board[i,j] <- 0
      }
    }
    for (i in 1:n){
      for (j in 1:length(layout[[i]])){
        if(layout[[i]][j] == 1){
          board[i,j] <- 1
        }
      }
    }
    board <- t(board)
  }else if(typeof(layout) == "integer" & is.null(ncol(layout))){
```

```
  #print("Fit: Singles rep detected")
  #Convert to a matrix
  board <- matrix(nrow = n, ncol = n)
  #fill the board with 0s
  for (i in 1:nrow(board)){
    for (j in 1:ncol(board)){
      board[i,j] <- 0
    }
  }
  for (i in 1:n){
    board[i,layout[i]] <- 1
  }
  board <- t(board)
}else if(typeof(layout) == "integer" & ncol(layout) == 2){
  #print("Fit: Pairs rep detected")
  #Convert to a matrix
  board <- matrix(nrow = n, ncol = n)
  #fill the board with 0s
  for (i in 1:nrow(board)){
    for (j in 1:ncol(board)){
      board[i,j] <- 0
    }
  }
  for (i in 1:nrow(layout)){
    board[layout[i,1], layout[i,2]] <- 1
  }
}
#================================================================================
#At this point, the board var should contain a matrix on the state of the board, with 1s for queens a
#Convert the board to a list of coordinates
my_vec <- c()
for(i in 1:n){
  for(j in 1:n){
    if(board[i,j]==1){
      my_vec <- c(my_vec, c(i,j))
    }
  }
}
if(length(my_vec) == 0){
  return(0)
}
points<-array( my_vec, dim=c(2,length(my_vec)/2))
are_attacking_eachother <- function(Q1,Q2){
  #https://stackoverflow.com/questions/57239548/how-to-check-if-a-queen-is-under-attack-in-nqueens
  #"A queen can attack another if :
  #they are on the same line : X1 = X2
  #they are on the same column : Y1 = Y2
  #they are on the same diagonal : X1-Y1 = X2-Y2 or X1+Y1 = X2+Y2"
  (Q1[1] == Q2[1] | Q1[2] == Q2[2] | Q1[1]-Q1[2] == Q2[1]-Q2[2] | Q1[1]+Q1[2] == Q2[1]+Q2[2])
}
fitness <- 0
if(length(points)/2 == 1){
  if(fitness_function == "binary"){
```

```
      return(0)
    }else if(fitness_function == "num_safe"){
      return(1)
    }else{
      #number of attacking pairs
      return(0)
    }
  }else if(fitness_function == "binary"){
      for(queen in 1:((length(points)/2)-1)){
        for(other_queen in (queen+1):(length(points)/2)){
          Q1 <- c(points[1,queen], points[2,queen])
          Q2 <- c(points[1,other_queen], points[2,other_queen])
          if(are_attacking_eachother(Q1, Q2)){
            fitness <- 1
          }
        }
      }
  }else if(fitness_function == "num_safe"){
    safe <- 0
     for(queen in 1:((length(points)/2)-1)){
        for(other_queen in (queen+1):(length(points)/2)){
          Q1 <- c(points[1,queen], points[2,queen])
          Q2 <- c(points[1,other_queen], points[2,other_queen])
          if(!are_attacking_eachother(Q1, Q2)){
            safe <- safe + 2
          }
        }
     }
    #to scale to [0,1], we can divide by the max number of queens that can be safe
    fitness <- safe
  }else{
      #(nC2 - number) of pairs of queens attacking each other.
      nc2 <- (n*(n-1)/2)
      num_attacking <- 0
      for(queen in 1:((length(points)/2)-1)){
        for(other_queen in (queen+1):(length(points)/2)){
          Q1 <- c(points[1,queen], points[2,queen])
          Q2 <- c(points[1,other_queen], points[2,other_queen])
          if(are_attacking_eachother(Q1, Q2)){
            fitness <- 1
          }
        }
      }
      fitness <- nc2 - num_attacking
  }
  return(fitness)
}
layout1 <- list()
layout2 <- list()
layout3 <- list()
for (i in 1:4){
  layout1[[i]] <- 0
  layout2[[i]] <- 0
```

```
  layout3[[i]] <- 0
}

layout1[[1]] <- c(1,0,0,0)
layout1[[4]] <- c(0,0,0,1)

layout2[[1]] <- c(1,0,0,0)

layout3[[1]] <- c(1,0,0,0)
layout3[[3]] <- c(0,0,0,1)

vizualize_board(layout1, 4)
print("binary fitness of layout 1")
fitness(layout = layout1, fitness_function = "binary", 4)

vizualize_board(layout2, 4)
print("binary fitness of layout 2")
fitness(layout = layout2, fitness_function = "binary", 4)

vizualize_board(layout3, 4)
print("binary fitness of layout 3")
fitness(layout = layout3, fitness_function = "binary", 4)

print("------------------------------------------------------------")

vizualize_board(layout1, 4)
print("num_safe fitness of layout 1")
fitness(layout = layout1, fitness_function = "num_safe", 4)

vizualize_board(layout2, 4)
print("num_safe fitness of layout 2")
fitness(layout = layout2, fitness_function = "num_safe", 4)

vizualize_board(layout3, 4)
print("num_safe fitness of layout 3")
fitness(layout = layout3, fitness_function = "num_safe", 4)

print("------------------------------------------------------------")

#1.5
generate_random_layout <- function(n){
  layout <- list()
  for (i in 1:n){
    layout[[i]] <- c(sample(1:n, 1), sample(1:n, 1), sample(1:n, 1), sample(1:n, 1))
  }
  return(layout)
}
vizualize_board(generate_random_layout(4),4)
n_generations <- 100
n_queens <- 4
n_population <- 10
mutprob <- 0.1
fitness_function <- "binary"
```

```r
#generate random population
population <- list()
for(i in 1:n_population){
  population[[i]] <- generate_random_layout(n_queens)
}

#calculate fitness of population
fitnesses <- c()
for(i in 1:n_population){
  fitnesses[i] <- fitness(population[[i]], fitness_function, n_queens)
}
n_attacking_queens <- c()
#iterate through generations
for(generation in 1:n_generations){
  cat("generation: ", generation, "\r")
  #Two individuals are randomly sampled from the current population, they are further used as parents
  parents <- sample(population, 2, replace = FALSE)
  #One individual with the smallest fitness is selected from the current population, this will be the v
  victim <- population[which.min(fitnesses)]
  #The two sampled parents are to produce a kid by crossover
  kid <- crossover(parents[[1]], parents[[2]], p=2)
  #this kid should be mutated with probability mutprob
  kid <- mutate(kid, mutprob)
  #The victim is replaced by the kid in the population
  population[which(population == victim)] <- kid
  #Do not forget to update the vector of fitness values of the population
  fitnesses[which(population == victim)] <- fitness(kid, fitness_function, n_queens)
  #Remember the number of pairs of queens attacking each other at the given iteration
  n_attacking_queens <- c(n_attacking_queens, sum(fitnesses))
}
#1.6
#1.7
#1.8
#1.9
# 2.1
data <- read.csv2("censoredproc.csv")

par(mfrow=c(1,3))

# histogram of all data
hist(as.numeric(data[,1]), freq=F, main="Histogram of all data")

# histogram of data with cens=1
hist(as.numeric(data[which(data[,2]==1),1]), freq=F, main="Histogram of cens=1 data")

# histogram of data with cens=2
hist(as.numeric(data[which(data[,2]==2),1]), freq=F, main="Histogram of cens=2 data")

min(as.numeric(data[which(data[,2]==1),1]))
min(as.numeric(data[which(data[,2]==2),1]))

# 2.2
#2.3
```

```
#2.4
#2.5
#2.6
```