Laboration report in Computational Statistics

Laboration 6

732A90

Duc Tran
William Wiik

Division of Statistics and Machine Learning
Department of Computer Science
Linköping University

14 december 2023

Contents

1 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.

1.1 1.1

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. We are not using chess notation (in this case e5) as we do not want to limit n by 26 and also further work will be easier with a numerical notation.

b. On n numbers, where each number has $\log_2 n$ 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.

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.

```
# A function to make bit to integer
bitToInt <- function(x) {
  num <- strtoi(paste(as.character(rev(x)), collapse = ""), base = 2) + 1
  return(num)
}

# A function that plots the queens in a cheesboard
plot_chessboard <- function(individual, encoding = 1) {

  n <- length(individual)

  if(encoding == 1) {
    # Create a data frame for plotting
    df <- data.frame(
      row = unlist(lapply(individual, function(queen) queen$row)),
      column = unlist(lapply(individual, function(queen) queen$column))
    )
  } else if(encoding == 2) {


    df <- data.frame(
      row = unlist(lapply(individual, bitToInt)),
      column = 1:n
    )
```

```
  } else {

    df <- data.frame(
      row = individual,
      column = 1:n
    )

  }
  # Plot the chessboard
  ggplot(df, aes(x = column, y = row)) +
    geom_point(shape = 1, size = 5, color = "red") +
    scale_x_continuous(breaks = 1:n, labels = 1:n, limits = c(1,n)) +
    scale_y_continuous(breaks = 1:n, labels = 1:n, limits = c(1,n)) +
    coord_fixed() +
    theme_minimal() +
    labs(title = "Queens Placement on Chessboard")
}



# Function to initialize a random individual
initialize_individual <- function(n, encoding = 1) {


  if(encoding == 1) {
    coordinates_row <- sample(1:n, n, replace = TRUE)
    coordinates_column <- 1:n


    # Initialize a list
    individual_list <- lapply(1:n, function(i) {
      list(
        row = coordinates_row[i],
        column = coordinates_column[i]
      )
    })
    return(individual_list)
  } else if(encoding == 2) {

    # index start from 0 instead of 1
    coordinates_row <- lapply(sample(0:(n-1),n,replace = TRUE), function(x)
      as.numeric(intToBits(x))[1:log2(n)])

    return(coordinates_row)

  } else {

    coordinates_row <- sample(1:n, n, replace = TRUE)
```

```
    return(coordinates_row)

  }

}
```

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

1.2 1.2

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, $0 < p \leq n/2$, and $p \in \mathbb{N}$. Experiment with different values of p.

```
crossover <- function(individual1, individual2, p, encoding = 1) {
  n <- length(individual1)

  if(p < 1 | p  > (n/2)) {stop("p does not satisfy 0 < p <= n/2")}

  if(encoding == 1) {
    # Create a new individual
    new_individual <- vector("list", length = n)

    # Copy columns 1 to p from individual1
    for (i in 1:p) {
      new_individual[[i]] <- individual1[[i]]
    }

    # Copy columns p+1 to n from individual2
    for (i in (p+1):n) {
      new_individual[[i]] <- individual2[[i]]
    }

  }  else if(encoding == 2) {

    # Create a new individual
    new_individual <- vector("list", length = n)

    # Mutate all queens from both individuals
    for(queen in 1:n){

      new_pos <- c()

      new_pos <- individual1[[queen]][1:p]

      new_pos <-c(new_pos, individual2[[queen]][-c(1:p)])

      new_individual[[queen]] <- new_pos
```

```
    }

  } else {

    # Create a new individual
    new_individual <- c(individual1[1:p],individual2[(p+1):n])

  }

  return(new_individual)
}
```

We decided to use $p = 2$ for all crossovers.

1.3 1.3

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

```
mutate <- function(individual, encoding = 1) {

  n <- length(individual)

  # Randomly select a queen to mutate
  queen_to_mutate <- sample(1:n, 1)


  if(encoding == 1) {

    possible_rows <- setdiff(1:n,individual[[queen_to_mutate]]$row)
    mutate_row <- sample(possible_rows, 1)

    # Move the selected queen to the new row
    individual[[queen_to_mutate]]$row <- mutate_row


  } else if(encoding == 2) {

    possible_rows <- setdiff(1:n,bitToInt(individual[[queen_to_mutate]]))
    mutate_row <- sample(possible_rows, 1)

    # Move the selected queen to the new row
    individual[[queen_to_mutate]] <- as.numeric(intToBits(mutate_row))[1:log2(n)]

  } else {

    individual[queen_to_mutate] <- sample(setdiff(1:n, individual[queen_to_mutate]), 1)
```

```
  }

  return(individual)
}
```

1.4 1.4

Define a fitness function for a given configuration. Experiment with three: binary—is a solution or not; number of queens not attacked; 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.

```
fitness_1 <- function(individual, encoding = 1) {

  n <- length(individual)

  if(encoding == 1) {
    for(i in 1:(n-1))  {
      for(j in (i+1):n) {
        # if row is equal then 0
        if(unlist(individual[[i]][1]) == unlist(individual[[j]][1])) {

          return(0)

          # if the queens have same diag then 0
        } else {

          rowdiff <- abs((unlist(individual[[i]][1]) - unlist(individual[[j]][1])))
          coldiff <- abs((unlist(individual[[i]][2]) - unlist(individual[[j]][2])))

          if(rowdiff == coldiff) return(0)
        }

      }
    }
    return(1)
  } else if(encoding == 2) {
    for(i in 1:(n-1))  {
      for(j in (i+1):n) {
        # if row is equal then 0
        if(all(unlist(individual[[i]]) == unlist(individual[[j]]))) {

          return(0)

          # if the queens have same diag then 0
        } else {

          rowdiff <- abs(bitToInt(unlist(individual[[i]])) - bitToInt(unlist(individual[[j]])))
```

```
          coldiff <-   abs(i-j)

          if(rowdiff == coldiff) return(0)
        }

      }
    }
    return(1)

  } else {

    if(length(individual) != length(unique(individual))) return(0)

    for(i in 1:(n-1)) {
      for(j in (i+1):n) {

        if(abs(individual[i] - individual[j]) == (j-i)) return(0)

      }
    }
    return(1)
  }
}


fitness_2 <- function(individual, encoding = 1) {

  n <- length(individual)

  queens_attacked <- vector(length = n)

  if(encoding == 1) {
    for(i in 1:(n-1))  {

      # all queens are attacked
      if(all(queens_attacked)) return(0)

      for(j in (i+1):n) {
        # all queens are attacked
        if(all(queens_attacked)) return(0)


        # rows
        if(unlist(individual[[i]][1]) == unlist(individual[[j]][1])) {

          queens_attacked[i] <- TRUE
          queens_attacked[j] <- TRUE

          # diag
```

```
      } else {

        rowdiff <- abs((unlist(individual[[i]][1]) - unlist(individual[[j]][1])))
        coldiff <- abs((unlist(individual[[i]][2]) - unlist(individual[[j]][2])))

        if(rowdiff == coldiff) {

          queens_attacked[i] <- TRUE
          queens_attacked[j] <- TRUE
        }

      }
    }
  }

  # numbers of queens that are not attacked divided by n to scale it
  return((n-sum(queens_attacked))/n)

} else if (encoding == 2) {

  # make the binary encoding to numeric
  individual <- unlist(lapply(individual, bitToInt))

  for(i in 1:(n-1))  {

    # all queens are attacked
    if(all(queens_attacked)) return(0)

    for(j in (i+1):n) {
      # all queens are attacked
      if(all(queens_attacked)) return(0)

      if(individual[i] == individual[j]) {

        queens_attacked[i] <- TRUE
        queens_attacked[j] <- TRUE

      }else {

        rowdiff <- abs(individual[i] - individual[j])
        coldiff <- abs(i - j)

        if(rowdiff == coldiff) {

          queens_attacked[i] <- TRUE
          queens_attacked[j] <- TRUE
        }

      }
```

```
      }
    }
    return((n-sum(queens_attacked))/n)


  } else {

    for(i in 1:(n-1))  {

      # all queens are attacked
      if(all(queens_attacked)) return(0)

      for(j in (i+1):n) {
        # all queens are attacked
        if(all(queens_attacked)) return(0)

        if(individual[i] == individual[j]) {

          queens_attacked[i] <- TRUE
          queens_attacked[j] <- TRUE

        }else {

          rowdiff <- abs(individual[i] - individual[j])
          coldiff <- abs(i - j)

          if(rowdiff == coldiff) {

            queens_attacked[i] <- TRUE
            queens_attacked[j] <- TRUE
          }

        }
      }
    }
    return((n-sum(queens_attacked))/n)
  }
}


fitness_3 <- function(individual, encoding = 1) {

  n <- length(individual)

  # Number of pairs
  count <- 0
  if(encoding == 1) {
    for(i in 1:(n-1))  {
```

```
    for(j in (i+1):n) {

      # row
      if(unlist(individual[[i]][1]) == unlist(individual[[j]][1])) {

        count <- count + 1

        # diag
      } else {

        rowdiff <- abs((unlist(individual[[i]][1]) - unlist(individual[[j]][1])))
        coldiff <- abs((unlist(individual[[i]][2]) - unlist(individual[[j]][2])))

        if(rowdiff == coldiff) {

          count <- count + 1

        }

      }
    }
  }

  # numbers of pairs divided my scale
  scale <- sum(1:(n-1))
  fitness <- 1 - (count / scale)
  return(list(count, fitness))

} else if(encoding == 2) {

  # make the binary encoding to numeric
  individual <- unlist(lapply(individual, bitToInt))


  for(i in 1:(n-1))  {

    for(j in (i+1):n) {

      if(individual[i] == individual[j]) {

        count <- count + 1

      } else {

        rowdiff <- abs(individual[i] - individual[j])
        coldiff <- abs(i - j)

        if(rowdiff == coldiff) {
```

```
            count <- count + 1

          }

        }
      }
    }

    scale <- sum(1:(n-1))
    fitness <- 1 - (count / scale)
    return(list(count, fitness))


  } else {

    for(i in 1:(n-1))  {

      for(j in (i+1):n) {

        if(individual[i] == individual[j]) {

          count <- count + 1

        } else {

          rowdiff <- abs(individual[i] - individual[j])
          coldiff <- abs(i - j)

          if(rowdiff == coldiff) {

            count <- count + 1

          }

        }
      }
    }

    scale <- sum(1:(n-1))
    fitness <- 1 - (count / scale)
    return(list(count, fitness))

  }

}
```

The binary fitness only says if we have a solution or not, which means that when we replace the worst individual with a new child, the worst individual is random. The other fitness functions tells us how close we are to a possible solution.

1.5 1.5 - 1.6

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.

If found return the legal configuration of queens.

```
# In the implemented genetic algorithm we assign four individuals.
solution <- function(size, mutprob = 0.1, fitness = 1, encoding = 1, iter_size = 1000) {
  mutprob <- mutprob

  ind1 <- initialize_individual(size, encoding = encoding)
  ind2 <- initialize_individual(size, encoding = encoding)
  ind3 <- initialize_individual(size, encoding = encoding)
  ind4 <- initialize_individual(size, encoding = encoding)

  list_ind <- list(ind1, ind2, ind3, ind4)
  n <- length(list_ind)

  fitness_value <- c()
  max_fitness <- c()
  queens_attacked <- c() # Individual with min of queens attacked

  queens_attacked_per_ind <- c() # Number of queens attacked per individual

  for(iter in 1:iter_size) {

    for(i in 1:n) {

      queens_attacked_per_ind[i] <- fitness_3(list_ind[[i]], encoding = encoding)[[1]]
```

```
    if(fitness == 1){

      fitness_value[i] <- fitness_1(list_ind[[i]], encoding = encoding)

    }   else if(fitness == 2)  {

      fitness_value[i] <- fitness_2(list_ind[[i]], encoding = encoding)

    }   else {

      fitness_value[i] <- fitness_3(list_ind[[i]], encoding = encoding)[[2]]

    }

  }
  queens_attacked[iter] <- queens_attacked_per_ind[which.min(queens_attacked_per_ind)]

  # legal config. of queens found
  if(any(fitness_value == 1)) {

    index <- which(fitness_value == 1)[1]
    max_fitness[iter] <- max(fitness_value) # Incase the a randomized individual is a solution
    return(list(list_ind, max_fitness, fitness_value, queens_attacked))

  }

  index_ind <- sample(1:n,2, replace = FALSE)

  child <- crossover(list_ind[[index_ind[1]]],list_ind[[index_ind[2]]], p = 2, encoding = encoding)

  if(mutprob > runif(1)) {

    child <- mutate(child, encoding = encoding)

  }

  replace_index <- which.min(fitness_value)

  if(fitness == 1) fitness_value[replace_index]  <- fitness_1(child, encoding = encoding)
  else if(fitness == 2) fitness_value[replace_index]  <- fitness_2(child, encoding = encoding)
  else  fitness_value[replace_index]  <- fitness_3(child, encoding = encoding)[[2]]

  list_ind[[replace_index]] <- child

  max_fitness[iter] <- max(fitness_value)
  # cat(iter, " max fitness is: ", max(fitness_value),"\n")
}

return(list(list_ind, max_fitness, fitness_value, queens_attacked))
```

```
}
```

1.6 1.7

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

Answer:

In the code below we run the algorithm for $n = 4$, the first encoding, fitness function number 3 and mutprob = 0.5 with maximum iteration of 1000. We also include a plot of a chessboard where the queens are placed according to the last iteration for the individual with the maximum fitness value.

In figure 1, we see that number of pairs queens attacking each other will always be the same or decreasing when we run the algorithm. The second plot confirms that the last iteration was a legal state.

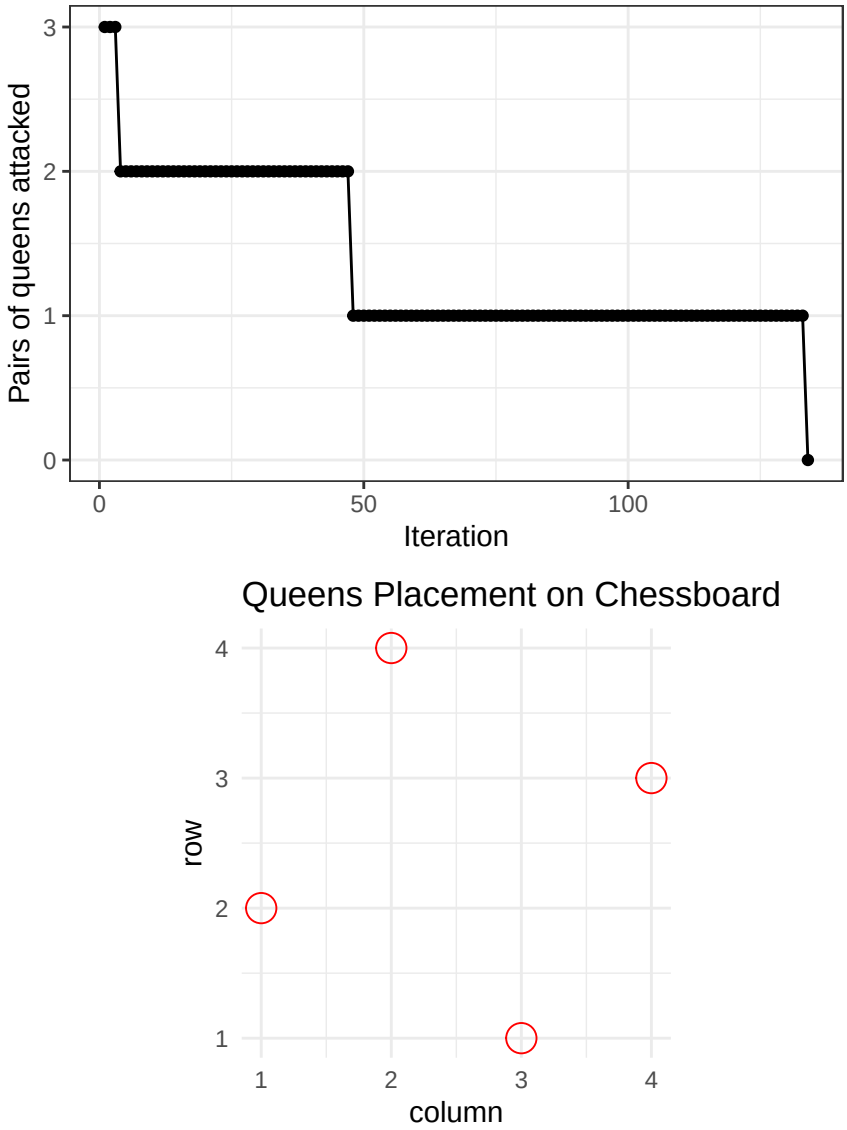

Figure 1: Pairs of the number of queens attacking each other & Queens placemnet on chessboard.

1.7 1.8

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?

```
# Run the algorithm for the parameters speciefied in 1.8 and print outs if the
# algorithm find a legal state
set.seed(5)
for(n in c(4,8,16)){
  for(mutprob in c(0.1, 0.5, 0.9)){
    for(fitness in 1:3){
      for(encoding in 1:3){
        run <- solution(n, mutprob = mutprob, fitness = fitness, encoding = encoding, iter_size = 1000)
        if(max(run[[2]]) == 1){
          print(paste0("Solution found for n = ", n,
                       ", mutprob = ", mutprob,
                       ", fitness function = ", fitness,
                       ", encoding = ", encoding))
        }
      }
    }
  }
}
```

```
## [1] "Solution found for n = 4, mutprob = 0.1, fitness function = 1, encoding = 2"
## [1] "Solution found for n = 4, mutprob = 0.1, fitness function = 2, encoding = 3"
## [1] "Solution found for n = 4, mutprob = 0.1, fitness function = 3, encoding = 1"
## [1] "Solution found for n = 4, mutprob = 0.1, fitness function = 3, encoding = 2"
## [1] "Solution found for n = 4, mutprob = 0.5, fitness function = 1, encoding = 3"
## [1] "Solution found for n = 4, mutprob = 0.5, fitness function = 2, encoding = 3"
## [1] "Solution found for n = 4, mutprob = 0.5, fitness function = 3, encoding = 1"
## [1] "Solution found for n = 4, mutprob = 0.9, fitness function = 1, encoding = 1"
## [1] "Solution found for n = 4, mutprob = 0.9, fitness function = 1, encoding = 2"
## [1] "Solution found for n = 4, mutprob = 0.9, fitness function = 1, encoding = 3"
## [1] "Solution found for n = 4, mutprob = 0.9, fitness function = 2, encoding = 1"
## [1] "Solution found for n = 4, mutprob = 0.9, fitness function = 2, encoding = 2"
## [1] "Solution found for n = 4, mutprob = 0.9, fitness function = 2, encoding = 3"
## [1] "Solution found for n = 4, mutprob = 0.9, fitness function = 3, encoding = 1"
## [1] "Solution found for n = 4, mutprob = 0.9, fitness function = 3, encoding = 2"
## [1] "Solution found for n = 4, mutprob = 0.9, fitness function = 3, encoding = 3"
## [1] "Solution found for n = 8, mutprob = 0.1, fitness function = 3, encoding = 3"
## [1] "Solution found for n = 8, mutprob = 0.5, fitness function = 2, encoding = 3"
## [1] "Solution found for n = 8, mutprob = 0.5, fitness function = 3, encoding = 3"
## [1] "Solution found for n = 8, mutprob = 0.9, fitness function = 2, encoding = 2"
## [1] "Solution found for n = 16, mutprob = 0.9, fitness function = 2, encoding = 2"
```

1.8 1.9

Question: Discuss which encoding and objective function worked best

Answer:

As we can see in the output above in 1.8, we found 21 legal state. It is easier to found a legal state when

- n is lower
- When mutprob is higher
- When fitness function is not the first fitness function (confirms conclusion in 1.4).

When it comes to which encoding that worked best we can see that encoding 1 had 5 legal state, encoding 2 had 7 legal state and encoding 3 had 9. Encoding 3 seem to be best, but it can be random since that they are evaluated on randomized individuals.

2 Question 2: EM algorithm

The data in this question consists of 1000 observation times for when a certain product fails. In total 811 are observed at the time of failure and 189 were observed when the product had already failed, meaning these 189 observations are left-censored.

2.1 Assignment 2.1

Question: 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?

Answer: In figure 2, a histogram over all data are presented and in figure 3, a histogram with only observed data are presented (censored observations were removed).

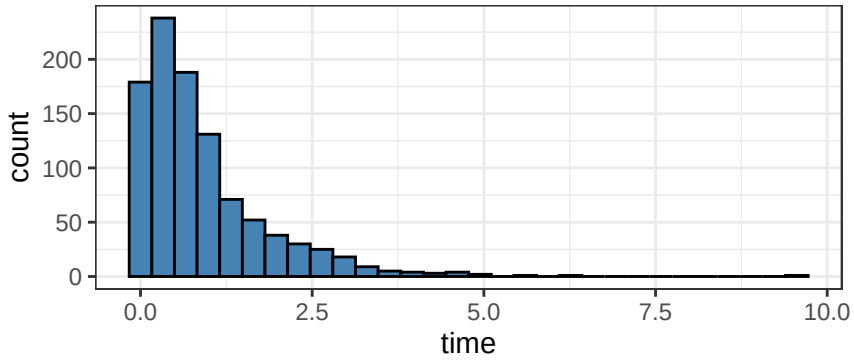

Figure 2: Histogram of all values.

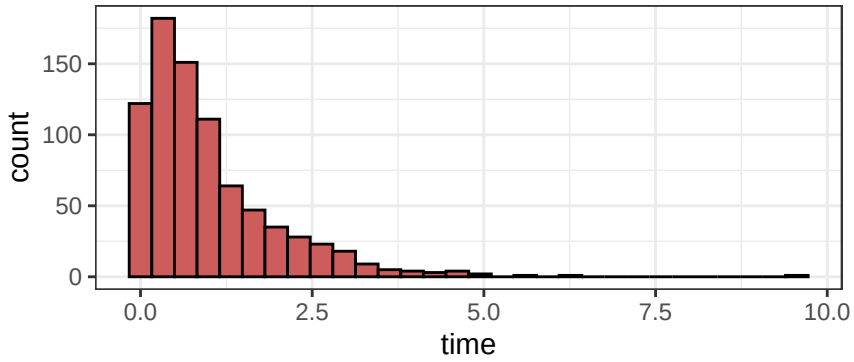

Figure 3: Histogram of all non-censored values.

From figure 2 and 3, both distributions of values appear to come from an exponential distribution.

2.2 Assignment 2.2

Question: 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.

Answer: In this question the number of non-censored observed data are denoted as n and the observations as x_i. The number of censored data are denoted as m, the observed truncated value as c_j with the real unknown values as z_j.

The probability density function for exponential distribution is:

$$f(x|\lambda) = \lambda e^{-\lambda x} \ , \ x \geq 0 \tag{1}$$

The probability density function for left truncated exponential distribution is:

$$f(z|Z \leq y, \lambda) = \frac{g(z)}{F(y)} \tag{2}$$

where $g(z) = f(z)$ for $z \leq y$ and $g(z) = 0$ everywhere else. y is the observed truncated value c, where we know that for each observation j, the condition $z_j \leq c_j$ holds true.

The cumulative distribution function for exponential distribution is:

$$F(x|\lambda) = 1 - e^{-\lambda x} \tag{3}$$

The likelihood function for non-censored exponential variables is:

$$\mathcal{L}(\lambda, x_1, ..., x_n) = \prod_{i=1}^{n} f(x_i, \lambda) = \prod_{i=1}^{n} \lambda e^{-\lambda x_i} \tag{4}$$

The log-likelihood function for non-censored exponential variables is:

$$log\mathcal{L}(\lambda, x_1, ..., x_n) = n \cdot ln(\lambda) - \lambda \sum_{i=1}^{n} x_i \tag{5}$$

The likelihood function for censored exponential variables is:

$$\mathcal{L}(\lambda, z_1, ..., z_m) = \prod_{j=1}^{m} \frac{g(z)}{F(y)} = \prod_{j=1}^{m} \frac{\lambda e^{-\lambda z_j}}{1 - e^{-\lambda c_j}} \tag{6}$$

The log-likelihood function for censored exponential variables is:

$$log\mathcal{L}(\lambda, z_1, ..., z_m) = \sum_{j=1}^{m} \Big(log(\lambda e^{-\lambda z_j}) - log(1 - e^{-\lambda c_j}) \Big) \tag{7}$$

$$= \sum_{j=1}^{m} \Big(log(\lambda) - \lambda z_j - log(1 - e^{-\lambda c_j}) \Big) \tag{8}$$

$$= m \cdot log(\lambda) - \lambda \sum_{j=1}^{m} z_j - \sum_{j=1}^{m} log(1 - e^{-\lambda c_j}) \tag{9}$$

This leads to the likelihood for **all** data is:

$$\mathcal{L}(\lambda, x_1, ..., x_n, z_1, ..., z_m) = \prod_{i=1}^{n} \lambda e^{-\lambda x_i} \prod_{j=1}^{m} \frac{\lambda e^{-\lambda z_j}}{1 - e^{-\lambda c_j}} \tag{10}$$

The log-likelihood for **all** data is:

$$log\mathcal{L}(\lambda, x_1, ..., x_n, z_1, ..., z_m) = n \cdot ln(\lambda) - \lambda \sum_{i=1}^{n} x_i + m \cdot log(\lambda) - \lambda \sum_{j=1}^{m} z_j - \sum_{j=1}^{m} log(1 - e^{-\lambda c_j}) \tag{11}$$

2.3 Assignment 2.3

Question: 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.

Answer: The EM algorithm is as follows:

Let

$Q(\lambda, \lambda^k) = E[loglik(\mathbf{X}, \mathbf{C}, \mathbf{Z}, \lambda | \mathbf{X}, \mathbf{C}, \lambda)]$

1: $k = 0, \lambda^0 = \lambda^0$
2: **while** Convergence not attained **and** $k < k_{max} + 1$ **do**
3: **E-step:** Derive $Q(\lambda, \lambda^k)$
4: **M-step:** $\lambda^{k+1} = argmax_\lambda \; Q(\lambda, \lambda^k)$
5: k++
6: **end while**

The expected value for the random variable z_j (value of the censored observation) is as follows:

$$E[z_j | z_j < c_j, \lambda] = \frac{\int_0^{c_j} z_j g(z_j) dz_j}{F(c_j)} \tag{12}$$

$$= \frac{\int_0^{c_j} z_j \cdot \lambda e^{-\lambda z_j} dz_j}{1 - e^{-\lambda c_j}} \tag{13}$$

$$= \frac{\lambda}{1 - e^{-\lambda c_j}} \cdot \int_0^{c_j} z_j \cdot e^{-\lambda z_j} dz_j \tag{14}$$

$$= [\text{integration by parts, } f = z_j, g' = e^{-\lambda z_j}] \tag{15}$$

$$= \frac{\lambda}{1 - e^{-\lambda c_j}} \cdot \left(\Big[z_j \cdot \frac{e^{-\lambda z_j}}{-\lambda} \Big]_{z_j=0}^{z_j=c_j} - \int_0^{c_j} \cdot \frac{e^{-\lambda z_j}}{-\lambda} dz_j \right) \tag{16}$$

$$= \frac{\lambda}{1 - e^{-\lambda c_j}} \cdot \left(c_j \cdot \frac{e^{-\lambda c_j}}{-\lambda} + 0 + \frac{1}{\lambda} \Big[\frac{e^{-\lambda z_j}}{-\lambda} \Big]_{z_j=0}^{z_j=c_j} \right) \tag{17}$$

$$= \frac{1}{1 - e^{-\lambda c_j}} \cdot \left(- c_j \cdot e^{-\lambda c_j} + \Big(\frac{e^{-\lambda c_j}}{-\lambda} - \frac{1}{\lambda} \Big) \right) \tag{18}$$

$$= \frac{1}{1 - e^{-\lambda c_j}} \cdot \left(\frac{1}{\lambda} - \Big(c_j + \frac{1}{\lambda} \Big) e^{-\lambda c_j} \right) \tag{19}$$

Taking the expected value of equation (11) we can see that the only part with random variables is $\sum_{j=1}^{m} z_j$ and all the other values are assumed to be known in the EM-algorithm. Furthermore we note that:

$$E\left[\sum_{j=1}^{m} Z_j\right] = E[Z_1] + E[Z_2] + ... + E[Z_m] \tag{20}$$

2.4 Assignment 2.4

Question: 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?

Answer: The code as follows was implemented as the EM-algorithm.

```
# Divides the data to censored and non-censored.
cens_data <- data[data$cens==2, ]
no_cens_data <- data[data$cens==1, ]
no_cens <- data$cens == 1

# Number of censored and non-censored data
n <- sum(no_cens)
m <- 1000-n

# Parameters from the question
max_it <- 100
lambda <- 100
min_change <- 0.001

# Calculates expected value of the truncated random variable (z_j) for given lambda
exp_val_cens <- function(lambda, ci){
  expected_value <- 1/(1-exp(-lambda*ci)) * ((1/lambda) - (ci+(1/lambda))*exp(-lambda*ci) )
  return(expected_value)
}


# EM-algorithm
llik <- c()
set.seed(13)

# Function that calculates the E-step
E_llik <- function(lambda){
  m*log(lambda) - lambda*exp_val - sum(log(1-exp(-lambda*cens_data$time))) +
    n*log(lambda) - lambda*sum(no_cens_data$time)
}
for(it in 1:max_it) {
  # E-step:
  # Calculates the expected value of sum of censored data (z_j)
  exp_val <- 0
  for(xi in 1:189){
```

```
    exp_val <- exp_val + exp_val_cens(lambda, cens_data$time[xi])
  }
  llik[it] <- E_llik(lambda)

  # M-step
  lambda <- optim(100, fn=E_llik, control = list(fnscale=-1))$par
  cat("iteration: ", it, "log likelihood: ", llik[it], "\n")


  # Stop if the log likelihood has not changed significantly
  if (it>1){
    if (abs(llik[it] - llik[it-1]) < min_change){
      # Exits the EM-algorithm
      break
    }
  }
}
```

```
## iteration:  1 log likelihood:  -76283.23
## iteration:  2 log likelihood:  -560.2008
## iteration:  3 log likelihood:  -559.6081
## iteration:  4 log likelihood:  -559.6081
```

```
lambda
```

```
## [1] 1.005859
```

From the output, the EM-algorithm converged after 4 iterations at $\lambda = 1.005859$

2.5 Assignment 2.5

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

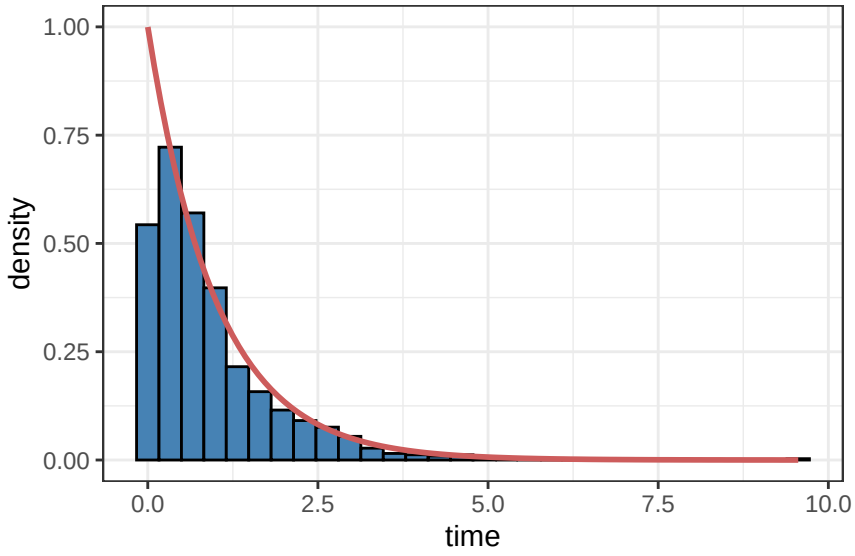

Figure 4: Histogram of all data with the density function of $\lambda = 1.005859$.

From figure 4, it is plausible that the distribution of the data comes from an exponential distribution with $\lambda = 1.005859$.

2.6 Assignment 2.6

Question: 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 $\hat{\lambda}$ 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.

Solution:

```
set.seed(13)
# Number of uncensored data
n <- 811
# Number of censored data
m <- 189

# Parameters for EM-algorithm
max_it <- 100
min_change <- 0.001

# Saved values for the assignment
llik <- c()
sample_lambda <- c()
ml_lambda <- c()


for(sample in 1:1000){
  # a
  lambda <- 1.005859 # Found from 1.4
  data <- rexp(1000, rate=lambda)

  # b
  # Remember which data that are censored
  cens_index <- sample(1:1000, 189)
  # Divided the data into censored and uncensored
  cens_data <- data[cens_index]
  no_cens_data <- data[-cens_index]

  # Sample censored data from same distribution but from [true time, infinity) since cens_data[i] is tru
  for(i in 1:189){
    cens_data[i] <- rexp(1, rate=lambda) + cens_data[i]
  }

  # EM-algorithm
  for(it in 1:max_it) {
    # E-step:
    # Calculates the expected value of sum of censored data
    exp_val <- 0
    for(xi in 1:189){
      exp_val <- exp_val + exp_val_cens(lambda, cens_data[xi])
    }

    E_llik <- function(lambda){
      m*log(lambda) - lambda*exp_val - sum(log(1-exp(-lambda*cens_data))) +
        n*log(lambda) - lambda*sum(no_cens_data)
    }
    llik[it] <- E_llik(lambda)
```

```
    # M-step
    lambda <- optim(100, fn=E_llik, control = list(fnscale=-1))$par

    # Stop if the log likelihood has not changed significantly
    if (it>1){
      if (abs(llik[it] - llik[it-1]) < min_change){
        # Exits the EM-algorithm
        break
      }
    }
  }
  # Lambda value found from the EM-algorithm
  sample_lambda[sample] <- lambda
  # ML-estimation of lambda on uncensored data
  ml_lambda[sample] <- n / sum(no_cens_data)
}
```

In figure 5, histogram of $\lambda-$values found by the EM-algorithm is presented. In figure 6, histogram of $\lambda-$values calculated on the uncensored data with ML-estimation is presented.

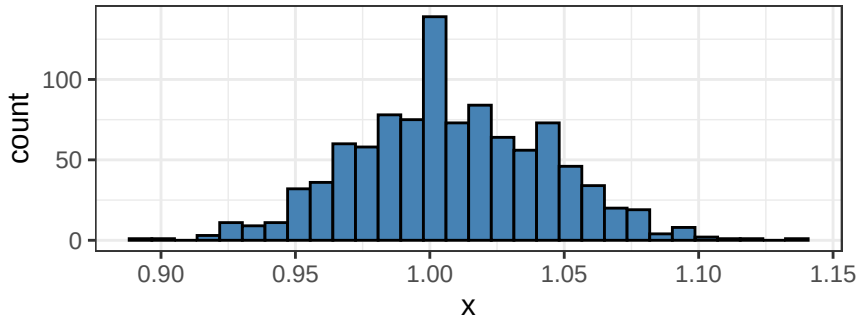

Figure 5: Histogram of λ for 1000 parametric bootstrap samples with the EM-algorithm.

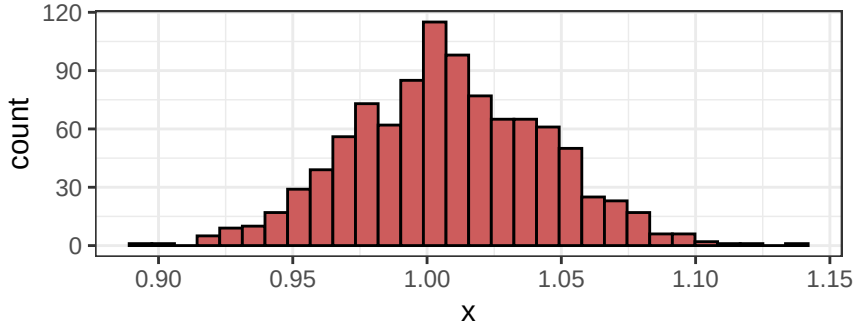

Figure 6: Histogram of λ for 1000 values calculated on the non-censored observations with ML-estimation.

The variance of EM-algorithm and ML-estimation are both around 0.0013.

```
var(sample_lambda)
```

```
## [1] 0.001286399
```

```
var(ml_lambda)
```

```
## [1] 0.001287565
```

```
mean(sample_lambda)
```

```
## [1] 1.00823
```

```
mean(ml_lambda)
```

```
## [1] 1.008237
```

```
median(sample_lambda)
```

```
## [1] 1.005859
```

```
median(ml_lambda)
```

```
## [1] 1.006916
```

Both the algorithm appears to be unbiased.

3 Statement of Contribution

We solved the tasks and wrote the report as a group.

4 Appendix

The code used in this laboration report are summarised in the code as follows:

```
library(knitr)
library(dplyr)
library(ggplot2)


knitr::opts_chunk$set(
  echo = TRUE,
  fig.width = 4.5,
  fig.height = 3)

# A function to make bit to integer
bitToInt <- function(x) {
  num <- strtoi(paste(as.character(rev(x)), collapse = ""), base = 2) + 1
  return(num)
}

# A function that plots the queens in a cheesboard
plot_chessboard <- function(individual, encoding = 1) {

  n <- length(individual)

  if(encoding == 1) {
    # Create a data frame for plotting
    df <- data.frame(
      row = unlist(lapply(individual, function(queen) queen$row)),
      column = unlist(lapply(individual, function(queen) queen$column))
    )
  } else if(encoding == 2) {


    df <- data.frame(
      row = unlist(lapply(individual, bitToInt)),
      column = 1:n
    )

  } else {

    df <- data.frame(
      row = individual,
```

```
      column = 1:n
    )

  }
  # Plot the chessboard
  ggplot(df, aes(x = column, y = row)) +
    geom_point(shape = 1, size = 5, color = "red") +
    scale_x_continuous(breaks = 1:n, labels = 1:n, limits = c(1,n)) +
    scale_y_continuous(breaks = 1:n, labels = 1:n, limits = c(1,n)) +
    coord_fixed() +
    theme_minimal() +
    labs(title = "Queens Placement on Chessboard")
}



# Function to initialize a random individual
initialize_individual <- function(n, encoding = 1) {


  if(encoding == 1) {
    coordinates_row <- sample(1:n, n, replace = TRUE)
    coordinates_column <- 1:n


    # Initialize a list
    individual_list <- lapply(1:n, function(i) {
      list(
        row = coordinates_row[i],
        column = coordinates_column[i]
      )
    })
    return(individual_list)
  } else if(encoding == 2) {

    # index start from 0 instead of 1
    coordinates_row <- lapply(sample(0:(n-1),n,replace = TRUE), function(x)
      as.numeric(intToBits(x))[1:log2(n)])

    return(coordinates_row)

  } else {

    coordinates_row <- sample(1:n, n, replace = TRUE)
    return(coordinates_row)

  }

}
```

```r
crossover <- function(individual1, individual2, p, encoding = 1) {
  n <- length(individual1)

  if(p < 1 | p  > (n/2)) {stop("p does not satisfy 0 < p <= n/2")}

  if(encoding == 1) {
    # Create a new individual
    new_individual <- vector("list", length = n)

    # Copy columns 1 to p from individual1
    for (i in 1:p) {
      new_individual[[i]] <- individual1[[i]]
    }

    # Copy columns p+1 to n from individual2
    for (i in (p+1):n) {
      new_individual[[i]] <- individual2[[i]]
    }

  }  else if(encoding == 2) {

    # Create a new individual
    new_individual <- vector("list", length = n)

    # Mutate all queens from both individuals
    for(queen in 1:n){

      new_pos <- c()

      new_pos <- individual1[[queen]][1:p]

      new_pos <-c(new_pos, individual2[[queen]][-c(1:p)])

      new_individual[[queen]] <- new_pos

    }

  } else {

    # Create a new individual
    new_individual <- c(individual1[1:p],individual2[(p+1):n])

  }

  return(new_individual)
}
```

```r
mutate <- function(individual, encoding = 1) {

  n <- length(individual)

  # Randomly select a queen to mutate
  queen_to_mutate <- sample(1:n, 1)


  if(encoding == 1) {

    possible_rows <- setdiff(1:n,individual[[queen_to_mutate]]$row)
    mutate_row <- sample(possible_rows, 1)

    # Move the selected queen to the new row
    individual[[queen_to_mutate]]$row <- mutate_row


  } else if(encoding == 2) {

    possible_rows <- setdiff(1:n,bitToInt(individual[[queen_to_mutate]]))
    mutate_row <- sample(possible_rows, 1)

    # Move the selected queen to the new row
    individual[[queen_to_mutate]] <- as.numeric(intToBits(mutate_row))[1:log2(n)]

  } else {

    individual[queen_to_mutate] <- sample(setdiff(1:n, individual[queen_to_mutate]), 1)

  }

  return(individual)
}


fitness_1 <- function(individual, encoding = 1) {

  n <- length(individual)

  if(encoding == 1) {
    for(i in 1:(n-1))  {
      for(j in (i+1):n) {
        # if row is equal then 0
        if(unlist(individual[[i]][1]) == unlist(individual[[j]][1])) {
```

```
          return(0)

          # if the queens have same diag then 0
        } else {

          rowdiff <- abs((unlist(individual[[i]][1]) - unlist(individual[[j]][1])))
          coldiff <- abs((unlist(individual[[i]][2]) - unlist(individual[[j]][2])))

          if(rowdiff == coldiff) return(0)
        }

      }
    }
    return(1)
  } else if(encoding == 2) {
    for(i in 1:(n-1))  {
      for(j in (i+1):n) {
        # if row is equal then 0
        if(all(unlist(individual[[i]]) == unlist(individual[[j]]))) {

          return(0)

          # if the queens have same diag then 0
        } else {

          rowdiff <- abs(bitToInt(unlist(individual[[i]])) - bitToInt(unlist(individual[[j]])))
          coldiff <-   abs(i-j)

          if(rowdiff == coldiff) return(0)
        }

      }
    }
    return(1)

  } else {

    if(length(individual) != length(unique(individual))) return(0)

    for(i in 1:(n-1)) {
      for(j in (i+1):n) {

        if(abs(individual[i] - individual[j]) == (j-i)) return(0)

      }
    }
    return(1)
  }
}
```

```r
fitness_2 <- function(individual, encoding = 1) {

  n <- length(individual)

  queens_attacked <- vector(length = n)

  if(encoding == 1) {
    for(i in 1:(n-1))  {

      # all queens are attacked
      if(all(queens_attacked)) return(0)

      for(j in (i+1):n) {
        # all queens are attacked
        if(all(queens_attacked)) return(0)


        # rows
        if(unlist(individual[[i]][1]) == unlist(individual[[j]][1])) {

          queens_attacked[i] <- TRUE
          queens_attacked[j] <- TRUE

          # diag
        } else {

          rowdiff <- abs((unlist(individual[[i]][1]) - unlist(individual[[j]][1])))
          coldiff <- abs((unlist(individual[[i]][2]) - unlist(individual[[j]][2])))

          if(rowdiff == coldiff) {

            queens_attacked[i] <- TRUE
            queens_attacked[j] <- TRUE
          }

        }
      }
    }

    # numbers of queens that are not attacked divided by n to scale it
    return((n-sum(queens_attacked))/n)

  } else if (encoding == 2) {

    # make the binary encoding to numeric
    individual <- unlist(lapply(individual, bitToInt))
```

```
  for(i in 1:(n-1))  {

    # all queens are attacked
    if(all(queens_attacked)) return(0)

    for(j in (i+1):n) {
      # all queens are attacked
      if(all(queens_attacked)) return(0)

      if(individual[i] == individual[j]) {

        queens_attacked[i] <- TRUE
        queens_attacked[j] <- TRUE

      }else {

        rowdiff <- abs(individual[i] - individual[j])
        coldiff <- abs(i - j)

        if(rowdiff == coldiff) {

          queens_attacked[i] <- TRUE
          queens_attacked[j] <- TRUE
        }

      }
    }
  }
  return((n-sum(queens_attacked))/n)


} else {

  for(i in 1:(n-1))  {

    # all queens are attacked
    if(all(queens_attacked)) return(0)

    for(j in (i+1):n) {
      # all queens are attacked
      if(all(queens_attacked)) return(0)

      if(individual[i] == individual[j]) {

        queens_attacked[i] <- TRUE
        queens_attacked[j] <- TRUE

      }else {
```

```
          rowdiff <- abs(individual[i] - individual[j])
          coldiff <- abs(i - j)

          if(rowdiff == coldiff) {

            queens_attacked[i] <- TRUE
            queens_attacked[j] <- TRUE
          }

        }
      }
    }
    return((n-sum(queens_attacked))/n)
  }
}


fitness_3 <- function(individual, encoding = 1) {

  n <- length(individual)

  # Number of pairs
  count <- 0
  if(encoding == 1) {
    for(i in 1:(n-1))  {

      for(j in (i+1):n) {

        # row
        if(unlist(individual[[i]][1]) == unlist(individual[[j]][1])) {

          count <- count + 1

          # diag
        } else {

          rowdiff <- abs((unlist(individual[[i]][1]) - unlist(individual[[j]][1])))
          coldiff <- abs((unlist(individual[[i]][2]) - unlist(individual[[j]][2])))

          if(rowdiff == coldiff) {

            count <- count + 1

          }

        }
      }
    }
```

```
  # numbers of pairs divided my scale
  scale <- sum(1:(n-1))
  fitness <- 1 - (count / scale)
  return(list(count, fitness))

} else if(encoding == 2) {

  # make the binary encoding to numeric
  individual <- unlist(lapply(individual, bitToInt))


  for(i in 1:(n-1))  {

    for(j in (i+1):n) {

      if(individual[i] == individual[j]) {

        count <- count + 1

      } else {

        rowdiff <- abs(individual[i] - individual[j])
        coldiff <- abs(i - j)

        if(rowdiff == coldiff) {

          count <- count + 1

        }

      }
    }
  }

  scale <- sum(1:(n-1))
  fitness <- 1 - (count / scale)
  return(list(count, fitness))


} else {

  for(i in 1:(n-1))  {

    for(j in (i+1):n) {

      if(individual[i] == individual[j]) {

        count <- count + 1
```

```
        } else {

          rowdiff <- abs(individual[i] - individual[j])
          coldiff <- abs(i - j)

          if(rowdiff == coldiff) {

            count <- count + 1

          }

        }
      }
    }

    scale <- sum(1:(n-1))
    fitness <- 1 - (count / scale)
    return(list(count, fitness))

  }

}



# In the implemented genetic algorithm we assign four individuals.
solution <- function(size, mutprob = 0.1, fitness = 1, encoding = 1, iter_size = 1000) {
  mutprob <- mutprob

  ind1 <- initialize_individual(size, encoding = encoding)
  ind2 <- initialize_individual(size, encoding = encoding)
  ind3 <- initialize_individual(size, encoding = encoding)
  ind4 <- initialize_individual(size, encoding = encoding)

  list_ind <- list(ind1, ind2, ind3, ind4)
  n <- length(list_ind)

  fitness_value <- c()
  max_fitness <- c()
  queens_attacked <- c() # Individual with min of queens attacked

  queens_attacked_per_ind <- c() # Number of queens attacked per individual

  for(iter in 1:iter_size) {

    for(i in 1:n) {

      queens_attacked_per_ind[i] <- fitness_3(list_ind[[i]], encoding = encoding)[[1]]
```

```
    if(fitness == 1){

      fitness_value[i] <- fitness_1(list_ind[[i]], encoding = encoding)

    }   else if(fitness == 2)  {

      fitness_value[i] <- fitness_2(list_ind[[i]], encoding = encoding)

    }   else {

      fitness_value[i] <- fitness_3(list_ind[[i]], encoding = encoding)[[2]]

    }

  }
  queens_attacked[iter] <- queens_attacked_per_ind[which.min(queens_attacked_per_ind)]

  # legal config. of queens found
  if(any(fitness_value == 1)) {

    index <- which(fitness_value == 1)[1]
    max_fitness[iter] <- max(fitness_value) # Incase the a randomized individual is a solution
    return(list(list_ind, max_fitness, fitness_value, queens_attacked))

  }

  index_ind <- sample(1:n,2, replace = FALSE)

  child <- crossover(list_ind[[index_ind[1]]],list_ind[[index_ind[2]]], p = 2, encoding = encoding)

  if(mutprob > runif(1)) {

    child <- mutate(child, encoding = encoding)

  }

  replace_index <- which.min(fitness_value)

  if(fitness == 1) fitness_value[replace_index]  <- fitness_1(child, encoding = encoding)
  else if(fitness == 2) fitness_value[replace_index]  <- fitness_2(child, encoding = encoding)
  else  fitness_value[replace_index]  <- fitness_3(child, encoding = encoding)[[2]]

  list_ind[[replace_index]] <- child

  max_fitness[iter] <- max(fitness_value)
  # cat(iter, " max fitness is: ", max(fitness_value),"\n")
}

return(list(list_ind, max_fitness, fitness_value, queens_attacked))
```

```
}


set.seed(6)
run <- solution(4, mutprob = 0.5, fitness = 4, encoding = 1, iter_size = 1000)

plot1 <- ggplot(data.frame(y = run[[4]]), aes(x = 1:length(run[[4]]), y = y)) +
  geom_line() + geom_point() +
  theme_bw() + labs(x = "Iteration", y = "Pairs of queens attacked")

max_index <- which.max(run[[3]])
plot2 <-plot_chessboard(run[[1]][[max_index]])

cowplot::plot_grid(plot1, plot2, nrow = 2)


# Run the algorithm for the parameters speciefied in 1.8 and print outs if the
# algorithm find a legal state
set.seed(5)
for(n in c(4,8,16)){
  for(mutprob in c(0.1, 0.5, 0.9)){
    for(fitness in 1:3){
      for(encoding in 1:3){
        run <- solution(n, mutprob = mutprob, fitness = fitness, encoding = encoding, iter_size = 1000)
        if(max(run[[2]]) == 1){
          print(paste0("Solution found for n = ", n,
                       ", mutprob = ", mutprob,
                       ", fitness function = ", fitness,
                       ", encoding = ", encoding))
        }
      }
    }
  }
}

data <- read.csv2("censoredproc.csv")
data$time <- as.numeric(data$time)

ggplot(data) +
  geom_histogram(aes(x=time), colour="black", fill="steelblue") +
  theme_bw()
filter <- data$cens == 1
ggplot(data[filter, ]) +
  geom_histogram(aes(x=time), colour="black", fill="indianred") +
  theme_bw()
# Divides the data to censored and non-censored.
cens_data <- data[data$cens==2, ]
no_cens_data <- data[data$cens==1, ]
no_cens <- data$cens == 1
```

```r
# Number of censored and non-censored data
n <- sum(no_cens)
m <- 1000-n

# Parameters from the question
max_it <- 100
lambda <- 100
min_change <- 0.001

# Calculates expected value of the truncated random variable (z_j) for given lambda
exp_val_cens <- function(lambda, ci){
  expected_value <- 1/(1-exp(-lambda*ci)) * ((1/lambda) - (ci+(1/lambda))*exp(-lambda*ci) )
  return(expected_value)
}


# EM-algorithm
llik <- c()
set.seed(13)

# Function that calculates the E-step
E_llik <- function(lambda){
  m*log(lambda) - lambda*exp_val - sum(log(1-exp(-lambda*cens_data$time))) +
    n*log(lambda) - lambda*sum(no_cens_data$time)
}
for(it in 1:max_it) {
  # E-step:
  # Calculates the expected value of sum of censored data (z_j)
  exp_val <- 0
  for(xi in 1:189){
    exp_val <- exp_val + exp_val_cens(lambda, cens_data$time[xi])
  }
  llik[it] <- E_llik(lambda)

  # M-step
  lambda <- optim(100, fn=E_llik, control = list(fnscale=-1))$par
  cat("iteration: ", it, "log likelihood: ", llik[it], "\n")


  # Stop if the log likelihood has not changed significantly
  if (it>1){
    if (abs(llik[it] - llik[it-1]) < min_change){
      # Exits the EM-algorithm
      break
    }
  }
}
lambda
```

```r
exp_curve <- function(lambda){
  lambda*exp(lambda*x)
}
ggplot(data) +
  geom_histogram(aes(x=time, y=..density..), colour="black", fill="steelblue") +
  geom_function(fun=dexp, col="indianred", lwd=1) +
  theme_bw()
set.seed(13)
# Number of uncensored data
n <- 811
# Number of censored data
m <- 189

# Parameters for EM-algorithm
max_it <- 100
min_change <- 0.001

# Saved values for the assignment
llik <- c()
sample_lambda <- c()
ml_lambda <- c()


for(sample in 1:1000){
  # a
  lambda <- 1.005859 # Found from 1.4
  data <- rexp(1000, rate=lambda)

  # b
  # Remember which data that are censored
  cens_index <- sample(1:1000, 189)
  # Divided the data into censored and uncensored
  cens_data <- data[cens_index]
  no_cens_data <- data[-cens_index]

  # Sample censored data from same distribution but from [true time, infinity) since cens_data[i] is tru
  for(i in 1:189){
    cens_data[i] <- rexp(1, rate=lambda) + cens_data[i]
  }

  # EM-algorithm
  for(it in 1:max_it) {
    # E-step:
    # Calculates the expected value of sum of censored data
    exp_val <- 0
    for(xi in 1:189){
      exp_val <- exp_val + exp_val_cens(lambda, cens_data[xi])
    }
```

```
    E_llik <- function(lambda){
      m*log(lambda) - lambda*exp_val - sum(log(1-exp(-lambda*cens_data))) +
        n*log(lambda) - lambda*sum(no_cens_data)
    }
    llik[it] <- E_llik(lambda)

    # M-step
    lambda <- optim(100, fn=E_llik, control = list(fnscale=-1))$par

    # Stop if the log likelihood has not changed significantly
    if (it>1){
      if (abs(llik[it] - llik[it-1]) < min_change){
        # Exits the EM-algorithm
        break
      }
    }
  }
  # Lambda value found from the EM-algorithm
  sample_lambda[sample] <- lambda
  # ML-estimation of lambda on uncensored data
  ml_lambda[sample] <- n / sum(no_cens_data)
}
plot_data <- data.frame(x=sample_lambda)
ggplot(plot_data) +
  geom_histogram(aes(x=x), fill="steelblue", colour="black") +
  theme_bw()

plot_data <- data.frame(x=ml_lambda)
ggplot(plot_data) +
  geom_histogram(aes(x=x), fill="indianred", colour="black") +
  theme_bw()

var(sample_lambda)
var(ml_lambda)

mean(sample_lambda)
mean(ml_lambda)

median(sample_lambda)
median(ml_lambda)
```