Computational Statistics Computer Lab 6 (Group 7)

Qinyuan Qi(qinqi464) Satya Sai Naga Jaya Koushik Pilla (satpi345)

2023-12-22

Question 1: Genetic algorithm (Solved by Qinyuan Qi)

Answer:

(1) 3 Encodings :

We define 3 encodings accordingly, the following codes are the init codes to generate the init configs.

Also we wrote a function to print out the board layout.

```
########################## Config Encoding Function 1.1 ############################
# First encoding: n pairs encoding
init_configuration_1 <- function(board_size = 8) {
  configuration <- data.frame(x = c(), y = c())
  queen_count <- 0
  for (i in 1:board_size) {
    for(j in 1:board_size) {
      isQueen <- sample(c(0, 1), 1)
      if (queen_count < board_size) {
        if (isQueen == 1){
          configuration <- rbind(configuration, c(i, j))
          queen_count <- queen_count + 1
        }
      }
    }
  }
  return(configuration)
}

# Second Encoding: binary encoding
# if board_size = 8, then the max n is 2^8 - 1 = 255
init_configuration_2 <- function(board_size = 8,max_try_count = 1000) {
  max_value <- 2^board_size - 1
  queen_count <- 0
  index <- 1
  configuration <- rep(0, board_size)
  max_try_count <- 1000
  while(queen_count < board_size && max_try_count > 0) {
    number <- sample(0:max_value, 1)

    new_queen <- sum(as.numeric(intToBits(number)))
    if (queen_count + new_queen <= board_size && new_queen == 1) {
      configuration[index] <- number
      index <- index + 1
```

```
      queen_count <- queen_count + new_queen
    }
    max_try_count <- max_try_count - 1
  }

  if(max_try_count == 0) {
    print("Error: cannot generate a configuration")
  }

  return(configuration)

}

# Third Encoding: vector position encoding
init_configuration_3 <- function(board_size = 8) {
  configuration <- sample(1:board_size, board_size)
  return(configuration)
}

# print board common function
print_board <- function(configuration, method = 1) {
  if(method == 1) {
    n <- dim(configuration)[1]
    config <- matrix(0, nrow = n, ncol = n)
    for (i in 1:n) {
      config[configuration[i, 1], configuration[i, 2]] <- 1
    }
  } else if (method == 2) {
    # convert the number vector to a matrix(binary encoding)
    n <- length(configuration)
    config <- matrix(0, nrow = n, ncol = n)
    for (i in 1:n) {
      number <- configuration[i]
      for (j in n:1) {
        config[i, j] <- number %% 2
        number <- number %/% 2
      }
    }
  } else {
    # method 3
    # convert a vector to a matrix(column encoding)
    n <- length(configuration)
    config <- matrix(0, nrow = n, ncol = n)
    for (i in 1:n) {
      config[i, configuration[i]] <- 1
    }
  }

  n <- dim(config)[1]
  for(i in 1:n) {
    for(j in 1:n) {
      if(config[i, j] == 1) {
        if (j == 1) {
```

```
          cat("|Q|")
        }else if (j == n){
          cat("Q|", "\n")
        }else{
          cat("Q|")
        }
      } else {
        if (j == 1) {
          cat("|-|")
        }else if (j == n){
          cat("-|", "\n")
        }else{
          cat("-|")
        }
      }
    }
  }
}
```

```
##########################Sample Function Call ##############################
board_size <- 8

# this p value shuld less equal than board_size/2
p_val <- 4

configuration <- init_configuration_1(board_size)
configuration
```

```
##   X1L X1L.1
## 1   1     1
## 2   1     3
## 3   1     4
## 4   1     5
## 5   1     6
## 6   1     7
## 7   2     2
## 8   2     3
```

```
print_board(configuration, 1)
```

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

```
configuration <- init_configuration_2(board_size)
configuration
```

```
## [1]   4   2 128  64  16  32   8   1
```

```
print_board(configuration, 2)
```

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

```
configuration <- init_configuration_3(board_size)
configuration
```

```
## [1] 5 2 1 4 7 3 8 6
```

```
print_board(configuration, 3)
```

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

(2) Crossover function:

Crossover function defined as follows, all 3 encodings share the same crossover function.

Regarding P parameter in this function, we will test this in later steps.

```
########################## Crossover Function ############################
crossover <- function(config1, config2, p = 4){
  ncol <- length(config1)
  child_config <- config1
  for(i in 1:(ncol-p)){
    child_config[p+i] = config2[i]
  }
  return(child_config)
}
```

(3) Mutate functions:

We mutate the 3 encodings using the following functions.

```
########################## Mutate functions ############################

# (x,y) encoding, change y only
mutate1 <- function(config) {
  ncol <- length(config)
  mutated_config <- config
  queen_to_mutate <- sample(1:ncol, 1)
  new_position <- mutated_config[queen_to_mutate]
  new_y <- sample(1:ncol, 1)
  new_position[2] <- new_y
  mutated_config[queen_to_mutate] <- new_position
  return(mutated_config)
```

```
}

# (binary) encoding, change one bit randomly
mutate2 <- function(config) {
  ncol <- length(config)
  mutated_config <- config
  queen_to_mutate <- sample(1:ncol, 1)
  queen_integer <- mutated_config[queen_to_mutate]

  num_bits <- log2(queen_integer) %/% 1 + 1

  # Randomly select a bit position to flip
  bit_position <- sample(0:(num_bits - 1), 1)

  # Use bitwise XOR to flip the selected bit
  flipped_integer <- bitwXor(queen_integer, 2^bit_position)

  return(flipped_integer)
}


# (col location) encoding, change col number only
mutate3 <- function(config) {
  ncol <- length(config)
  mutated_config <- config
  queen_to_mutate <- sample(1:ncol, 1)
  mutated_config[queen_to_mutate] <- sample(1:ncol, 1)
  return(mutated_config)
}
```

(4):

To calc the fitness value of each encoding, we need to implement 3 different functions.

For the binary encoding, we restricted that every row must have a queen and only one queen to make check function easier to implement.

```
############################ Common Function ##############################
# common functions for fitness function
# check attack between queen position
is_attack <- function(queen1, queen2) {
  return(
    queen1[1] == queen2[1] || queen1[2] == queen2[2] ||
    abs(queen1[1] - queen2[1]) == abs(queen1[2] - queen2[2])
  )
}

# find the queen position in a binary string
find_set_bits_indices <- function(n,board_size) {
  bit_vector <- rev(intToBits(n)[1:board_size])
  bit_indice <- 0
  for(i in 1:length(bit_vector)){
    if (bit_vector[i] == 1){
      bit_indice <- i
      break
```

```
    }
  }
  return(bit_indice)
}


########################## Fitness Function ############################
# encoding (X,Y) represent the position of the queen
fitness1 <- function(config) {
   return(fitness(config))
}

# encoding (X) represent a number whose binary string shows
# the position of the queen in the current row

fitness2 <- function(config) {
  new_config <- data.frame(x = c(), y = c())
  row_number <- length(config)
  for(i in 1:row_number){
    number <- config[i]
    queen_pos <- find_set_bits_indices(number,row_number)
    new_config <- rbind(new_config, c(i,config[i]))
  }
  return(fitness(new_config))
}

fitness3 <- function(config) {
  new_config <- data.frame(x = c(), y = c())
  queen_num <- length(config)
  for(i in 1:queen_num){
    new_config <- rbind(new_config, c(i,config[i]))
  }
  return(fitness(new_config))
}

# our fitness function to handle all the encodings, other encodings will be
# converted to this encoding.(X,Y) Location Encoding

# check whole config is valid or not and return
# attacked queens number and unattacked queens number
fitness <- function(config) {
  queen_num <- nrow(config)
  attacked_queens <- c()

  for (i in 1:(queen_num - 1)) {
    for (j in (i + 1):queen_num) {
      if (is_attack(config[i,], config[j,])) {
        attacked_queens <- c(attacked_queens, i, j)
      }
    }
  }

  unattacked_queens <- setdiff(1:queen_num, unique(attacked_queens))
```

```
  num_unattacked_queens <- length(unattacked_queens)
  valid <- (length(unique(attacked_queens)) == 0)

  return(list(valid = valid,
              num_unattacked_queens = num_unattacked_queens,
              num_attacked_queens = queen_num - num_unattacked_queens))
}

# (x,y) pair encoding
init_configuration_1 <- function(board_size = 8) {
  configuration <- data.frame(x = c(), y = c())
  queen_count <- 0
  for (i in 1:board_size) {
    for(j in 1:board_size) {
      isQueen <- sample(c(0, 1), 1)
      if (queen_count < board_size) {
        if (isQueen == 1){
          configuration <- rbind(configuration, c(i, j))
          queen_count <- queen_count + 1
        }
      }
    }
  }
  return(configuration)
}

# Sample Call

#board_size <- 8

# this p value shuld less equal than board_size/2
#p_val <- 4

#configuration <- init_configuration_1(board_size)
#configuration
#print_board(configuration, 1)
#fitness1(configuration)

#configuration <- init_configuration_2(board_size)
#configuration
#print_board(configuration, 2)
#fitness2(configuration)

#configuration <- init_configuration_3(board_size)
#configuration
#print_board(configuration, 3)
#fitness3(configuration)
```

(5)(6)(7):

The genetic_algorithm implemented as follows

```
#####################   genetic_algorithm  ###################################
genetic_algorithm <- function(method = 1,board_size=8){
```

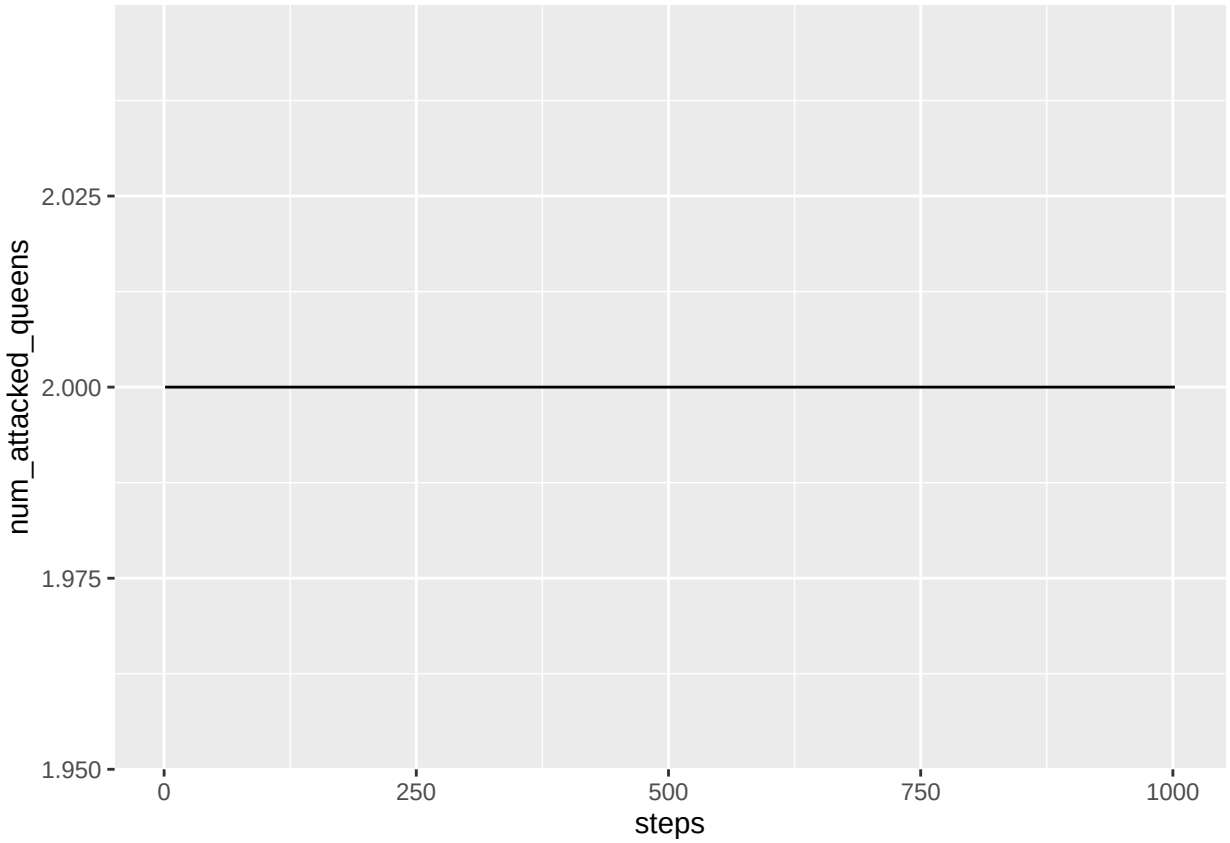
```
if (method == 1){
  configuration_1 <- init_configuration_1(board_size)
  val_1 <- fitness1(configuration_1)
} else if (method == 2){
  configuration_1 <- init_configuration_2(board_size)
  val_1 <- fitness2(configuration_1)
}else{
  configuration_1 <- init_configuration_3(board_size)
  val_1 <- fitness3(configuration_1)
}

configuration_2 <- NULL

max_steps <- 1000
steps <- 0

num_attacked_queens_vector <- c(val_1$num_attacked_queens)

while(val_1$num_attacked_queens != 0 && steps <= max_steps ) {

  if (is.null(configuration_2)){
    if (method == 1){
      configuration_2 <- init_configuration_1(board_size)
      val_2 <- fitness1(configuration_2)
    }else if(method == 2){
      configuration_2 <- init_configuration_2(board_size)
      val_2 <- fitness2(configuration_2)
    }else{
      configuration_2 <- init_configuration_3(board_size)
      val_2 <- fitness3(configuration_2)
    }
  }

  # cross over
  child_config <- crossover(configuration_1,
                            configuration_2,
                            p = p_val)

  # mutate
  if (method == 1){
    mutated_config <- mutate1(child_config)
    val_child <- fitness1(mutated_config)

  } else if (method == 2){
    mutated_config <- mutate2(child_config)
    val_child <- fitness2(mutated_config)
  } else {
    mutated_config <- mutate3(child_config)
    val_child <- fitness3(mutated_config)
  }

  configs <- c(1,2,3)
  num_attacked_queens <- c(val_1$num_attacked_queens,
```

```
                      val_2$num_attacked_queens,
                      val_child$num_attacked_queens)
    df <- data.frame(config = configs,
                     num_attacked_queens = num_attacked_queens)

    custom_order <- order(df$num_attacked_queens)
    sorted_df <- df[custom_order, ]

    # choose minial 2 values
    if (sorted_df$config[1] == 1){
      configuration_1 <- configuration_1
      val_1 <- val_1
    }else if(sorted_df$config[1] == 2){
      configuration_1 <- configuration_2
      val_1 <- val_2
    }else {
      configuration_1 <- mutated_config
      val_1 <- val_child
    }

    if (sorted_df$config[2] == 1){
      configuration_2 <- configuration_1
      val_2 <- val_1
    }else if(sorted_df$config[2] == 2){
      configuration_2 <- configuration_2
      val_2 <- val_2
    }else {
      configuration_2 <- mutated_config
      val_2 <- val_child
    }

    num_attacked_queens_vector <- c(num_attacked_queens_vector,val_1$num_attacked_queens)
    steps <- steps + 1

  }

  # print the queen position if found the solution
  if (val_1$num_attacked_queens == 0){
    print_board(configuration_1, 3)
  }else{
    print("not found the solution")
  }

  df <- data.frame(steps = 1:length(num_attacked_queens_vector),
                   num_attacked_queens = num_attacked_queens_vector)
  ggplot(data=df, aes(x = steps, y = num_attacked_queens)) + geom_line()

}
```

(8):

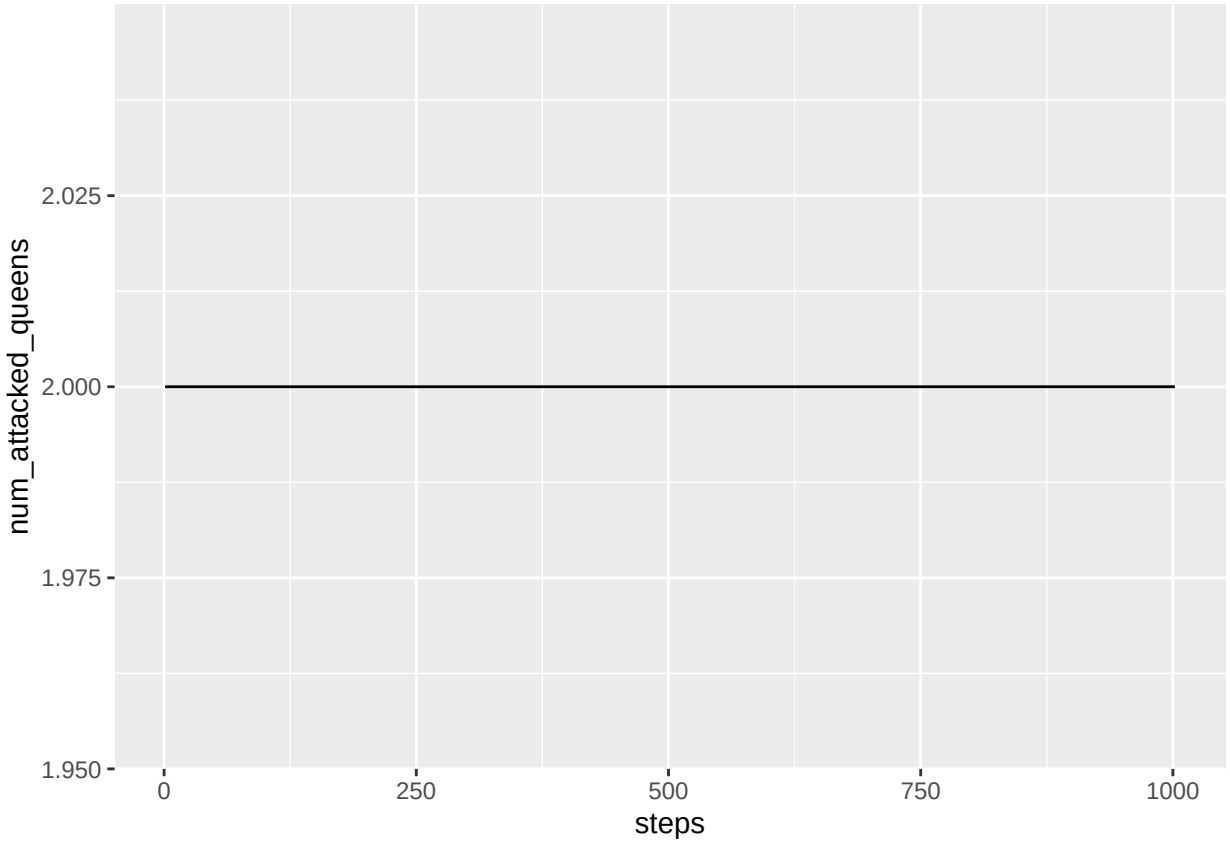
```
p_val <- 2
genetic_algorithm(method=3,board_size=4)
```

```
## [1] "not found the solution"
```



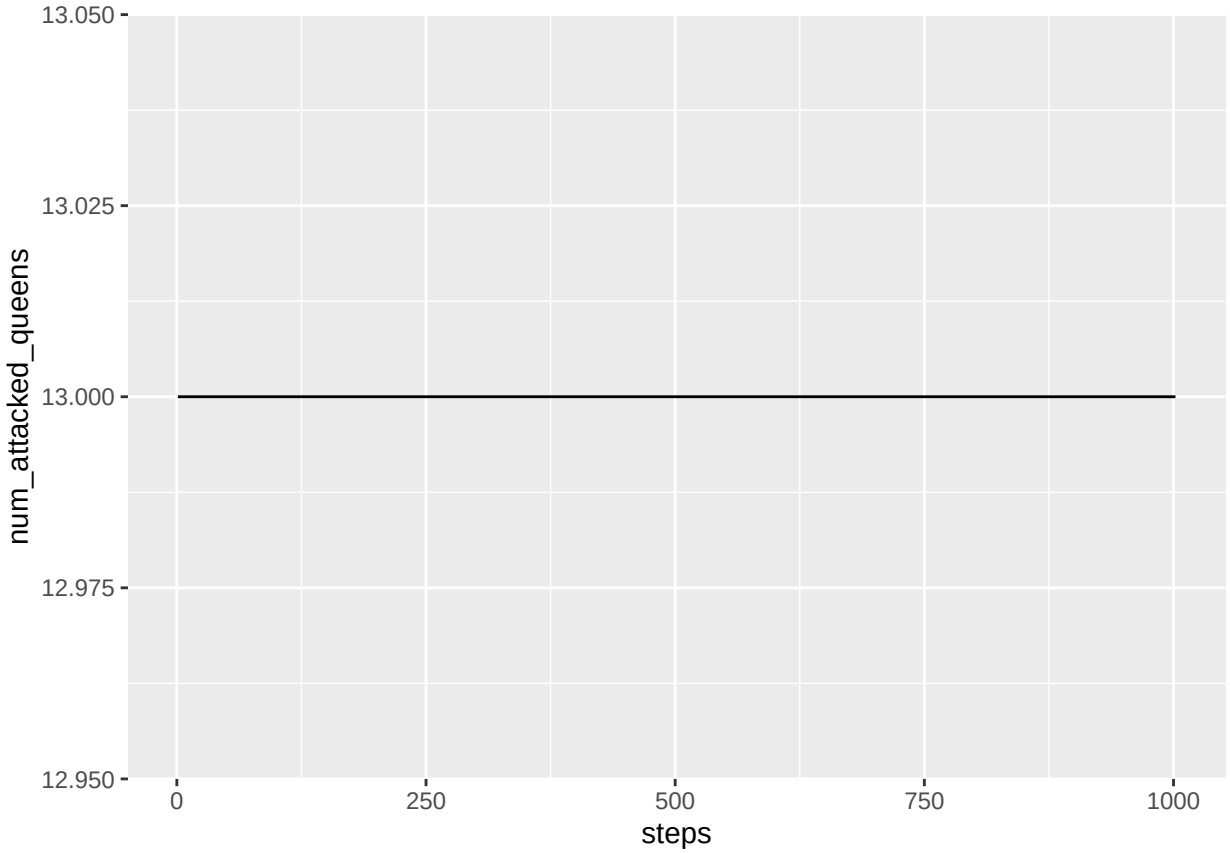

```
p_val <- 4
genetic_algorithm(method=3,board_size=8)
```

```
## [1] "not found the solution"
```




```
p_val <- 8
genetic_algorithm(method=3,board_size=16)
```

```
## [1] "not found the solution"
```



(9):

Since we did not get a reasonable plot, so we don't comment here.

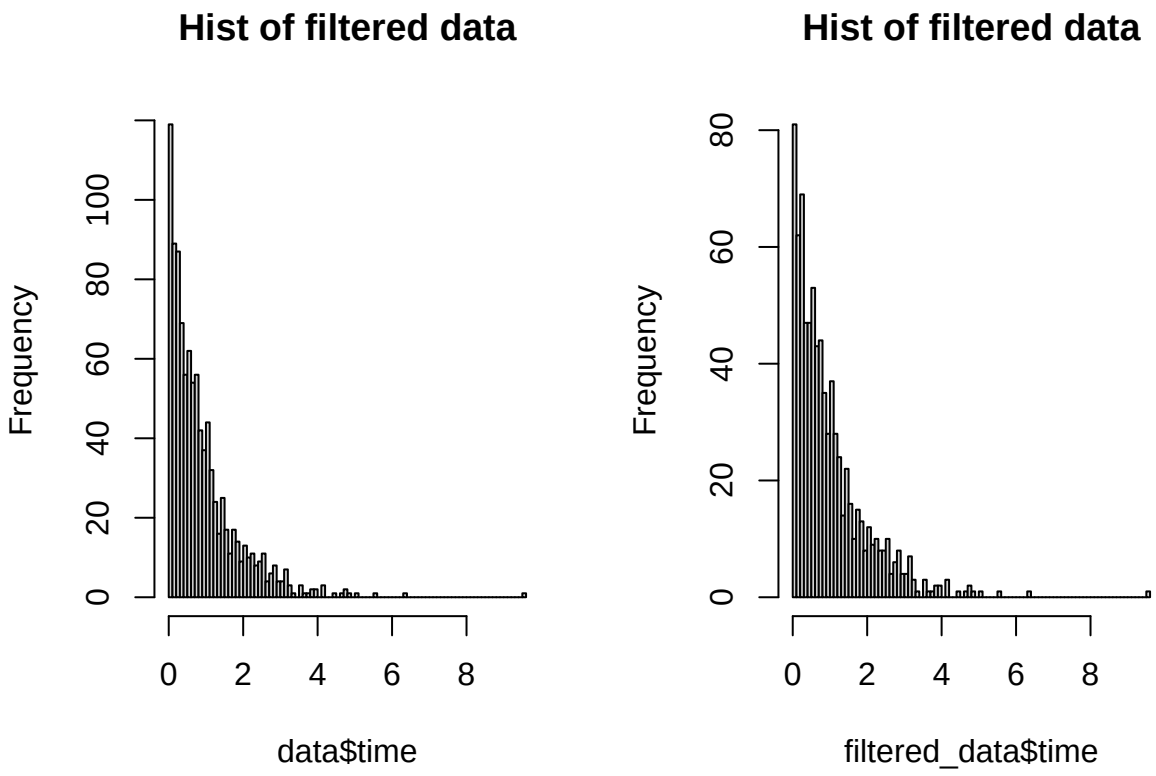
Question 2: EM algorithm (Solved by Satya Sai Naga Jaya Koushik Pilla)

Answer:

(1) Plot 2 histograms:

According to the plots generated, we found that the plot seems follow exponential distribution.

```
################################## (2.1) ########################################
# Load data
data <- read.csv("censoredproc.csv",
                 sep = ";", header = TRUE)
# We will filter out the left-censored data which cens=2
filtered_data <- data[data$cens ==1,]
layout(matrix(c(1:2), 1, 2))
# plot the data
hist(data$time, breaks = 100, main="Hist of filtered data")
# plot the filtered data
hist(filtered_data$time, breaks = 100, main="Hist of filtered data")
```



(2):

The general CDF form of an exponential distribution is:

$$F(x,\lambda) = \begin{cases} 1 - e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

So PDF of an exponential distribution is derivative of F on x:

$$f(x,\lambda) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Likelihood function for the exponential distribution is as follows.

$$L(\lambda; x_1, x_2 ... x_n) = \prod f(x,\lambda) = \lambda^n exp(-\lambda \sum_{j=1}^{n} x_j)$$

PDF for the truncated exponential distribution is derived as follows.

$$P(X \leq x | X \leq c) = \frac{P(X \leq x, X \leq c)}{P(X \leq c)} = \frac{P(X \leq \lambda)}{P(X \leq c)} = \frac{\lambda e^{-\lambda x}}{c e^{-cx}} = \frac{\lambda}{c} e^{-\lambda x}$$

So likelihood function for the truncated exponential distribution is as follows.

$$L(\lambda|X \leq c; x_1, x_2...x_n) = \prod P(X \leq x|X \leq c) = (\frac{\lambda^n}{c^n})exp(-\lambda \sum_{j=1}^{n} x_j)$$

(3):

Since it's relative straight forward , we will use the likelihood function directly.

So we will derive the EM function using the likelihood function in 2.2.

E-Step:

Let's compute the expectation of likelihood as follows.

$$Q(\lambda, \lambda^t) = E(L(\lambda|X \leq c; x_1, x_2...x_n), \lambda^t)$$

M-Step:

In M Step , we need to maximum Q with respect to λ.

$$\lambda^{t+1} = argmax_\lambda Q(\lambda, \lambda^t)$$

$$Q(\lambda, \lambda^t) = E(L(\lambda|X \leq c; x_1, x_2...x_n), \lambda^t)$$

(4):

According to the output, we know that estimated lambda: 7.80785 and Number of iterations: 6

```
estep <- function(lambda, x, c) {
  return(lambda / c * exp(-lambda * x))
}

# Function to compute the M-step
mstep <- function(lambda, x, c) {
  return(sum(x) / sum(c * exp(-lambda * x)))
}

# EM algorithm
em_algorithm <- function(initial_lambda, observed_data, truncation_point, max_iter = 100, tol = 0.001) {
  lambda_current <- initial_lambda

  for (iter in 1:max_iter) {
    # E-step
    expected_values <- estep(lambda_current, observed_data, truncation_point)

    # M-step
    lambda_next <- mstep(lambda_current, observed_data, expected_values)

    # Check for convergence
    if (abs(lambda_next - lambda_current) < tol) {
      break
    }

    # Update lambda for the next iteration
    lambda_current <- lambda_next
```

```
  }

  return(list(lambda = lambda_current, iterations = iter))
}

set.seed(12345)
truncation_point <- 2

# Initial guess for lambda
initial_lambda <- 100

# Run EM algorithm
result <- em_algorithm(initial_lambda, filtered_data, truncation_point)

# Print the result
cat("Estimated lambda:", result$lambda, "\n")
```

```
## Estimated lambda: 7.807858
```

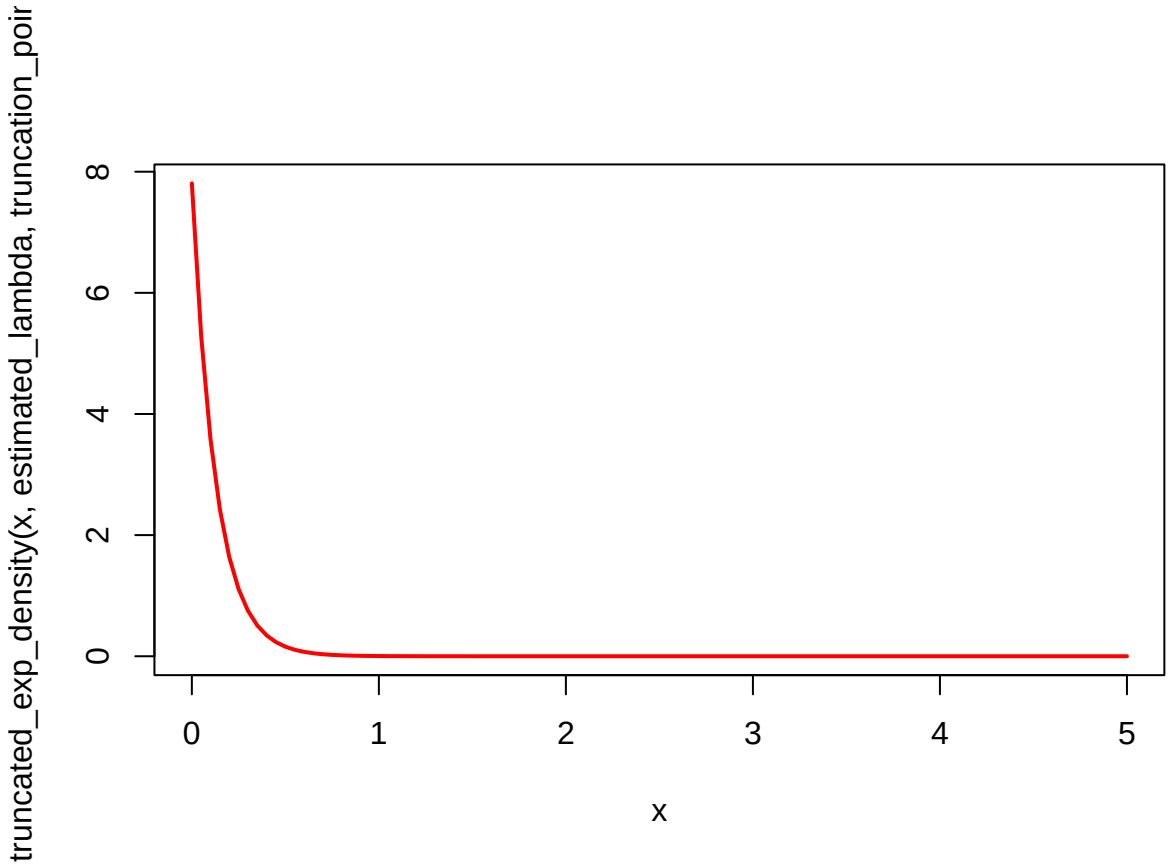
```
cat("Number of iterations:", result$iterations, "\n")
```

```
## Number of iterations: 6
```

(5):

Density curve of truncated exp as follows.

```
estimated_lambda <- result$lambda

truncated_exp_density <- function(x, lambda, c) {
  return(lambda * exp(-lambda * x) / (1 - exp(-lambda * c)))
}

x_values <- seq(0, 5, length.out = 100)
curve(truncated_exp_density(x, estimated_lambda, truncation_point), col = "red", lwd = 2, add = FALSE, 
```



(6):

The code as follows.

```r
# Function to calculate MLE for truncated exponential distribution
mle_truncated_exp <- function(data, c) {
  n <- length(data)
  lambda_hat <- n / sum(data)
  return(lambda_hat)
}

# Function to generate bootstrap samples
generate_bootstrap_samples <- function(data, num_bootstrap) {
  bootstrap_samples <- vector("list", length = num_bootstrap)
  for (i in 1:num_bootstrap) {
    bootstrap_samples[[i]] <- sample(data, replace = TRUE)
  }
  return(bootstrap_samples)
}

# Function to perform MLE and parametric bootstrap
perform_bootstrap <- function(observed_data, truncation_point, num_bootstrap) {
  # MLE for original data
  lambda_hat_original <- mle_truncated_exp(observed_data, truncation_point)

  # Generate bootstrap samples
```

```r
  bootstrap_samples <- generate_bootstrap_samples(observed_data, num_bootstrap)

  # Perform MLE for each bootstrap sample
  lambda_hat_bootstrap <- sapply(bootstrap_samples, function(bootstrap_sample) {
    mle_truncated_exp(bootstrap_sample, truncation_point)
  })

  return(list(lambda_hat_original = lambda_hat_original, lambda_hat_bootstrap = lambda_hat_bootstrap))
}

# Set the number of bootstrap samples
num_bootstrap <- 1000

# Perform MLE and parametric bootstrap
bootstrap_results <- perform_bootstrap(filtered_data, truncation_point, num_bootstrap)

# Print the results
cat("Original MLE Estimate:", bootstrap_results$lambda_hat_original, "\n")
```

```
## Original MLE Estimate: 0.001235972
```

```r
cat("Mean of Bootstrap Estimates:", mean(bootstrap_results$lambda_hat_bootstrap), "\n")
```

```
## Mean of Bootstrap Estimates: 0.001236046
```

Appendix: Code for this report

```
########################## Init code for question 1 ############################
rm(list = ls())
library(ggplot2)
library(bitops)
set.seed(12345)

########################## Config Encoding Function 1.1 ############################
# First encoding: n pairs encoding
init_configuration_1 <- function(board_size = 8) {
  configuration <- data.frame(x = c(), y = c())
  queen_count <- 0
  for (i in 1:board_size) {
    for(j in 1:board_size) {
      isQueen <- sample(c(0, 1), 1)
      if (queen_count < board_size) {
        if (isQueen == 1){
          configuration <- rbind(configuration, c(i, j))
          queen_count <- queen_count + 1
        }
      }
    }
  }
  return(configuration)
}

# Second Encoding: binary encoding
# if board_size = 8, then the max n is 2^8 - 1 = 255
init_configuration_2 <- function(board_size = 8,max_try_count = 1000) {
  max_value <- 2^board_size - 1
  queen_count <- 0
  index <- 1
  configuration <- rep(0, board_size)
  max_try_count <- 1000
  while(queen_count < board_size && max_try_count > 0) {
    number <- sample(0:max_value, 1)

    new_queen <- sum(as.numeric(intToBits(number)))
    if (queen_count + new_queen <= board_size && new_queen == 1) {
      configuration[index] <- number
      index <- index + 1
      queen_count <- queen_count + new_queen
    }
    max_try_count <- max_try_count - 1
  }

  if(max_try_count == 0) {
    print("Error: cannot generate a configuration")
  }

  return(configuration)

}
```

```r
# Third Encoding: vector position encoding
init_configuration_3 <- function(board_size = 8) {
  configuration <- sample(1:board_size, board_size)
  return(configuration)
}

# print board common function
print_board <- function(configuration, method = 1) {
  if(method == 1) {
    n <- dim(configuration)[1]
    config <- matrix(0, nrow = n, ncol = n)
    for (i in 1:n) {
      config[configuration[i, 1], configuration[i, 2]] <- 1
    }
  } else if (method == 2) {
    # convert the number vector to a matrix(binary encoding)
    n <- length(configuration)
    config <- matrix(0, nrow = n, ncol = n)
    for (i in 1:n) {
      number <- configuration[i]
      for (j in n:1) {
        config[i, j] <- number %% 2
        number <- number %/% 2
      }
    }
  } else {
    # method 3
    # convert a vector to a matrix(column encoding)
    n <- length(configuration)
    config <- matrix(0, nrow = n, ncol = n)
    for (i in 1:n) {
      config[i, configuration[i]] <- 1
    }
  }

  n <- dim(config)[1]
  for(i in 1:n) {
    for(j in 1:n) {
      if(config[i, j] == 1) {
        if (j == 1) {
          cat("|Q|")
        }else if (j == n){
          cat("Q|", "\n")
        }else{
          cat("Q|")
        }
      } else {
        if (j == 1) {
          cat("|-|")
        }else if (j == n){
          cat("-|", "\n")
        }else{
          cat("-|")
```

```
        }
      }
    }
  }
}
##########################Sample Function Call ##############################
board_size <- 8

# this p value shuld less equal than board_size/2
p_val <- 4

configuration <- init_configuration_1(board_size)
configuration
print_board(configuration, 1)

configuration <- init_configuration_2(board_size)
configuration
print_board(configuration, 2)

configuration <- init_configuration_3(board_size)
configuration
print_board(configuration, 3)
########################## Crossover Function ############################
crossover <- function(config1, config2, p = 4){
  ncol <- length(config1)
  child_config <- config1
  for(i in 1:(ncol-p)){
    child_config[p+i] = config2[i]
  }
  return(child_config)
}
########################## Mutate functions ############################

# (x,y) encoding, change y only
mutate1 <- function(config) {
  ncol <- length(config)
  mutated_config <- config
  queen_to_mutate <- sample(1:ncol, 1)
  new_position <- mutated_config[queen_to_mutate]
  new_y <- sample(1:ncol, 1)
  new_position[2] <- new_y
  mutated_config[queen_to_mutate] <- new_position
  return(mutated_config)
}

# (binary) encoding, change one bit randomly
mutate2 <- function(config) {
  ncol <- length(config)
  mutated_config <- config
  queen_to_mutate <- sample(1:ncol, 1)
  queen_integer <- mutated_config[queen_to_mutate]

  num_bits <- log2(queen_integer) %/% 1 + 1
```

```
  # Randomly select a bit position to flip
  bit_position <- sample(0:(num_bits - 1), 1)

  # Use bitwise XOR to flip the selected bit
  flipped_integer <- bitwXor(queen_integer, 2^bit_position)

  return(flipped_integer)
}


# (col location) encoding, change col number only
mutate3 <- function(config) {
  ncol <- length(config)
  mutated_config <- config
  queen_to_mutate <- sample(1:ncol, 1)
  mutated_config[queen_to_mutate] <- sample(1:ncol, 1)
  return(mutated_config)
}
########################## Common Function ############################
# common functions for fitness function
# check attack between queen position
is_attack <- function(queen1, queen2) {
  return(
    queen1[1] == queen2[1] || queen1[2] == queen2[2] ||
    abs(queen1[1] - queen2[1]) == abs(queen1[2] - queen2[2])
  )
}

# find the queen position in a binary string
find_set_bits_indices <- function(n,board_size) {
  bit_vector <- rev(intToBits(n)[1:board_size])
  bit_indice <- 0
  for(i in 1:length(bit_vector)){
    if (bit_vector[i] == 1){
      bit_indice <- i
      break
    }
  }
  return(bit_indice)
}


########################## Fitness Function ############################
# encoding (X,Y) represent the position of the queen
fitness1 <- function(config) {
   return(fitness(config))
}

# encoding (X) represent a number whose binary string shows
# the position of the queen in the current row

fitness2 <- function(config) {
  new_config <- data.frame(x = c(), y = c())
```

```
  row_number <- length(config)
  for(i in 1:row_number){
    number <- config[i]
    queen_pos <- find_set_bits_indices(number,row_number)
    new_config <- rbind(new_config, c(i,config[i]))
  }
  return(fitness(new_config))
}

fitness3 <- function(config) {
  new_config <- data.frame(x = c(), y = c())
  queen_num <- length(config)
  for(i in 1:queen_num){
    new_config <- rbind(new_config, c(i,config[i]))
  }
  return(fitness(new_config))
}

# our fitness function to handle all the encodings, other encodings will be
# converted to this encoding.(X,Y) Location Encoding

# check whole config is valid or not and return
# attacked queens number and unattacked queens number
fitness <- function(config) {
  queen_num <- nrow(config)
  attacked_queens <- c()

  for (i in 1:(queen_num - 1)) {
    for (j in (i + 1):queen_num) {
      if (is_attack(config[i,], config[j,])) {
        attacked_queens <- c(attacked_queens, i, j)
      }
    }
  }

  unattacked_queens <- setdiff(1:queen_num, unique(attacked_queens))
  num_unattacked_queens <- length(unattacked_queens)
  valid <- (length(unique(attacked_queens)) == 0)

  return(list(valid = valid,
              num_unattacked_queens = num_unattacked_queens,
              num_attacked_queens = queen_num - num_unattacked_queens))
}

# (x,y) pair encoding
init_configuration_1 <- function(board_size = 8) {
  configuration <- data.frame(x = c(), y = c())
  queen_count <- 0
  for (i in 1:board_size) {
    for(j in 1:board_size) {
      isQueen <- sample(c(0, 1), 1)
      if (queen_count < board_size) {
        if (isQueen == 1){
```

```
          configuration <- rbind(configuration, c(i, j))
          queen_count <- queen_count + 1
        }
      }
    }
  }
  return(configuration)
}

# Sample Call

#board_size <- 8

# this p value shuld less equal than board_size/2
#p_val <- 4

#configuration <- init_configuration_1(board_size)
#configuration
#print_board(configuration, 1)
#fitness1(configuration)

#configuration <- init_configuration_2(board_size)
#configuration
#print_board(configuration, 2)
#fitness2(configuration)

#configuration <- init_configuration_3(board_size)
#configuration
#print_board(configuration, 3)
#fitness3(configuration)
#####################   genetic_algorithm  ###################################
genetic_algorithm <- function(method = 1,board_size=8){
  if (method == 1){
    configuration_1 <- init_configuration_1(board_size)
    val_1 <- fitness1(configuration_1)
  } else if (method == 2){
    configuration_1 <- init_configuration_2(board_size)
    val_1 <- fitness2(configuration_1)
  }else{
    configuration_1 <- init_configuration_3(board_size)
    val_1 <- fitness3(configuration_1)
  }

  configuration_2 <- NULL

  max_steps <- 1000
  steps <- 0

  num_attacked_queens_vector <- c(val_1$num_attacked_queens)

  while(val_1$num_attacked_queens != 0 && steps <= max_steps ) {

    if (is.null(configuration_2)){
```

```
  if (method == 1){
    configuration_2 <- init_configuration_1(board_size)
    val_2 <- fitness1(configuration_2)
  }else if(method == 2){
    configuration_2 <- init_configuration_2(board_size)
    val_2 <- fitness2(configuration_2)
  }else{
    configuration_2 <- init_configuration_3(board_size)
    val_2 <- fitness3(configuration_2)
  }
}

# cross over
child_config <- crossover(configuration_1,
                          configuration_2,
                          p = p_val)

# mutate
if (method == 1){
  mutated_config <- mutate1(child_config)
  val_child <- fitness1(mutated_config)

} else if (method == 2){
  mutated_config <- mutate2(child_config)
  val_child <- fitness2(mutated_config)
} else {
  mutated_config <- mutate3(child_config)
  val_child <- fitness3(mutated_config)
}

configs <- c(1,2,3)
num_attacked_queens <- c(val_1$num_attacked_queens,
                  val_2$num_attacked_queens,
                  val_child$num_attacked_queens)
df <- data.frame(config = configs,
                 num_attacked_queens = num_attacked_queens)

custom_order <- order(df$num_attacked_queens)
sorted_df <- df[custom_order, ]

# choose minial 2 values
if (sorted_df$config[1] == 1){
  configuration_1 <- configuration_1
  val_1 <- val_1
}else if(sorted_df$config[1] == 2){
  configuration_1 <- configuration_2
  val_1 <- val_2
}else {
  configuration_1 <- mutated_config
  val_1 <- val_child
}

if (sorted_df$config[2] == 1){
```

```
      configuration_2 <- configuration_1
      val_2 <- val_1
    }else if(sorted_df$config[2] == 2){
      configuration_2 <- configuration_2
      val_2 <- val_2
    }else {
      configuration_2 <- mutated_config
      val_2 <- val_child
    }

    num_attacked_queens_vector <- c(num_attacked_queens_vector,val_1$num_attacked_queens)
    steps <- steps + 1

  }

  # print the queen position if found the solution
  if (val_1$num_attacked_queens == 0){
    print_board(configuration_1, 3)
  }else{
    print("not found the solution")
  }

  df <- data.frame(steps = 1:length(num_attacked_queens_vector),
                   num_attacked_queens = num_attacked_queens_vector)
  ggplot(data=df, aes(x = steps, y = num_attacked_queens)) + geom_line()

}
p_val <- 2
genetic_algorithm(method=3,board_size=4)
p_val <- 4
genetic_algorithm(method=3,board_size=8)
p_val <- 8
genetic_algorithm(method=3,board_size=16)
########################## Init code for question 1 ############################
rm(list = ls())
library(ggplot2)
set.seed(12345)
################################## (2.1) #######################################
# Load data
data <- read.csv("censoredproc.csv",
                 sep = ";", header = TRUE)
# We will filter out the left-censored data which cens=2
filtered_data <- data[data$cens ==1,]
layout(matrix(c(1:2), 1, 2))
# plot the data
hist(data$time, breaks = 100, main="Hist of filtered data")
# plot the filtered data
hist(filtered_data$time, breaks = 100, main="Hist of filtered data")
estep <- function(lambda, x, c) {
  return(lambda / c * exp(-lambda * x))
}

# Function to compute the M-step
```

```r
mstep <- function(lambda, x, c) {
  return(sum(x) / sum(c * exp(-lambda * x)))
}

# EM algorithm
em_algorithm <- function(initial_lambda, observed_data, truncation_point, max_iter = 100, tol = 0.001) {
  lambda_current <- initial_lambda

  for (iter in 1:max_iter) {
    # E-step
    expected_values <- estep(lambda_current, observed_data, truncation_point)

    # M-step
    lambda_next <- mstep(lambda_current, observed_data, expected_values)

    # Check for convergence
    if (abs(lambda_next - lambda_current) < tol) {
      break
    }

    # Update lambda for the next iteration
    lambda_current <- lambda_next
  }

  return(list(lambda = lambda_current, iterations = iter))
}

set.seed(12345)
truncation_point <- 2

# Initial guess for lambda
initial_lambda <- 100

# Run EM algorithm
result <- em_algorithm(initial_lambda, filtered_data, truncation_point)

# Print the result
cat("Estimated lambda:", result$lambda, "\n")
cat("Number of iterations:", result$iterations, "\n")

estimated_lambda <- result$lambda

truncated_exp_density <- function(x, lambda, c) {
  return(lambda * exp(-lambda * x) / (1 - exp(-lambda * c)))
}

x_values <- seq(0, 5, length.out = 100)
curve(truncated_exp_density(x, estimated_lambda, truncation_point), col = "red", lwd = 2, add = FALSE,

# Function to calculate MLE for truncated exponential distribution
mle_truncated_exp <- function(data, c) {
  n <- length(data)
  lambda_hat <- n / sum(data)
```

```r
  return(lambda_hat)
}

# Function to generate bootstrap samples
generate_bootstrap_samples <- function(data, num_bootstrap) {
  bootstrap_samples <- vector("list", length = num_bootstrap)
  for (i in 1:num_bootstrap) {
    bootstrap_samples[[i]] <- sample(data, replace = TRUE)
  }
  return(bootstrap_samples)
}

# Function to perform MLE and parametric bootstrap
perform_bootstrap <- function(observed_data, truncation_point, num_bootstrap) {
  # MLE for original data
  lambda_hat_original <- mle_truncated_exp(observed_data, truncation_point)

  # Generate bootstrap samples
  bootstrap_samples <- generate_bootstrap_samples(observed_data, num_bootstrap)

  # Perform MLE for each bootstrap sample
  lambda_hat_bootstrap <- sapply(bootstrap_samples, function(bootstrap_sample) {
    mle_truncated_exp(bootstrap_sample, truncation_point)
  })

  return(list(lambda_hat_original = lambda_hat_original, lambda_hat_bootstrap = lambda_hat_bootstrap))
}

# Set the number of bootstrap samples
num_bootstrap <- 1000

# Perform MLE and parametric bootstrap
bootstrap_results <- perform_bootstrap(filtered_data, truncation_point, num_bootstrap)

# Print the results
cat("Original MLE Estimate:", bootstrap_results$lambda_hat_original, "\n")
cat("Mean of Bootstrap Estimates:", mean(bootstrap_results$lambda_hat_bootstrap), "\n")
```