LAB6

Greeshma Jeev Koothuparambil(greko370), Sangeeth Sankunny Menon(sansa237)

2023-12-21

Question 1: Genetic Algorithm

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.

```
library(R.utils)
library(tuple)

#1.a Collection

list_individual <- function(n){

  i <-c()
  j <-c()
  iter <-1
  while (iter<=n) {


    j[iter] <- round(runif(1, min = 1, max = n), 0)
    iter <- iter+1

  }

  listdat <- data.frame("row" = 1:n, "col" = j)

  return(listdat)
}

individual1 <- list_individual(4)
individual2 <- list_individual(4)
```

(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.

```
#1.b Binary


binary_individual <- function(n){
  bincol <- c()

  for (iter in 1:n) {

    i <- round(runif(1, min = 1, max = n), 0)

    tempbin <- intToBin(c(i, n))
      bincol[iter] <- tempbin[1]

    }
  return(bincol)
}

individual3 <- binary_individual(4)
individual4 <- binary_individual(4)
```

(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.c Row Position


row_individual <- function(n){
  rowcol <- c()

  for (iter in 1:n) {

    i <- round(runif(1, min = 1, max = n), 0)

    rowcol[iter] <- i

  }
  return(rowcol)
}

individual5 <- row_individual(4)
individual6 <- row_individual(4)
```

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 N. Experiment with different values of p.

```
#2 Crossover


crossover <-function(p, ind1, ind2){
  if(typeof(ind1)=="list"){
```

```
    child <- ind1
    child[c(p:nrow(child)),] <- ind2[c(p:nrow(child)),]

  }
  if(typeof(ind1)=="character"){

    child <- c(ind1[1:p], ind2[(p+1):length(ind2)])
  }
  if(typeof(ind1)=="double"){
    child <- c(ind1[1:p], ind2[(p+1):length(ind2)])
  }
  return(child)
}

child1 <- crossover(2, individual1, individual2)
child2 <- crossover(2, individual3, individual4)
child3 <- crossover(2, individual5, individual6)
```

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

```
#3 Mutant

mutate <-function(ind){
  mutant <- ind
  if(typeof(ind)=="list"){
    rowmutant <- round(runif(1, min = 1, max = nrow(ind)), 0)
    colmutant <- ind$col[rowmutant]
    while (colmutant == ind$col[rowmutant]) {
      colmutant <- round(runif(1, min = 1, max = nrow(ind)), 0)
    }

    mutant$col[rowmutant] <- colmutant

  }
  if(typeof(ind)=="character"){
    colmutant <- round(runif(1, min = 1, max = length(ind)), 0)
    rowmutant <- ind[colmutant]
    while (rowmutant == ind[colmutant]) {
      i <- round(runif(1, min = 1, max = length(ind)), 0)
      tempbin <- intToBin(c(i, length(ind)))
      rowmutant <- tempbin[1]
    }
    mutant[colmutant] <- rowmutant
  }
  if(typeof(ind)=="double"){
    colmutant <- round(runif(1, min = 1, max = length(ind)), 0)
    rowmutant <- ind[colmutant]
    while (rowmutant == ind[colmutant]) {
      rowmutant <- round(runif(1, min = 1, max = length(ind)), 0)
    }
    mutant[colmutant] <- rowmutant
  }
  return(mutant)
```

```
}

mutant1 <- mutate(individual1)
mutant2 <- mutate(individual3)
mutant3 <- mutate(individual5)
```

4. Define a fitness function for a given configuration. Experiment with three: binary—is a solution or not;number of queens not attacked; (n 2 -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.

```
#4 Fitness

#LIST Fitness

fitnesslist <- function(individual){
  attacklist <- 0

  n <- nrow(individual)
  ind <- individual
  noattack <- round(n*(n-1)/2,0)

  for(row in 1:n){
    col <- ind[row,2]
    i <- row
    j <- col
    while(i>1 && j<n){
      if(ind[i-1,2]== j+1){
        attacklist <- attacklist+1
      }
      i<- i-1
      j<- j+1
    }
    i <- row
    j <- col
    while(i<n && j<n){
      if(ind[i+1,2]== j+1){
        attacklist <- attacklist+1
      }
      i<- i+1
      j<- j+1
    }
    i <- row
    j <- col
    while(i<n && j>1){
      if(ind[i+1,2]== j-1){
        attacklist <- attacklist+1
      }
      i<- i+1
      j<- j-1
    }
    i <- row
```

```
    j <- col
    while(i>1 && j>1){
      if(ind[i-1,2]== j-1){
        attacklist <- attacklist+1
      }
      i<- i-1
      j<- j-1
    }
  }
  attacklist <- attacklist/2
 for (i in 1:n) {
   colattack <- rep(0,n)
   colattack[which(ind$col ==i)] <- 1
   attack <- sum(colattack)*(sum(colattack)-1)/2
   attacklist <- attacklist+ attack
 }

  noattack <- noattack -attacklist

  returnvals <- list(noattack = noattack, attacknum = attacklist)
  return(returnvals)
}

listfit <- fitnesslist(individual1)

#BINARY fitness

fitnessbinary <- function(individual){
  n <- length(individual)
  attacklist <- 0
  noattack <- 0

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

      if (individual[i] != individual[j]) {
        if(abs(strtoi(individual[i], base = 2) - strtoi(individual[j], base = 2)) != abs(i - j)){
          noattack <- noattack+ 1
        }else{
          attacklist <- attacklist+1
        }
      }else{

          attacklist <- attacklist+sum(individual[i] == individual[j:n])
        }

    }
  }

      returnvals <- list(noattack = noattack, attacknum = attacklist)
      return(returnvals)

}
```

```r
binaryfit <- fitnessbinary(individual3)

#ROW fitness

fitnessrow <- function(individual){

  n <- length(individual)
  attacklist <- 0
  noattack <- 0

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

      if (individual[i] != individual[j] ){
        if(abs(individual[i] - individual[j]) != abs(i - j)){
          noattack <- noattack+ 1
        }else{
          attacklist <- attacklist+1
        }
      }else{

        attacklist <- attacklist+1
      }

    }
  }

  returnvals <- list(noattack = noattack, attacknum = attacklist)
  return(returnvals)
}

rowtfit <- fitnessrow(individual6)
```

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.

6. If found return the legal configuration of queens.

```r
## 5 Genetic Algorithm

GA <- function(n, encode, mutprob,fitness){
  iterations <- 100
  popsize <- 50
  numfit <- 0
  population <- data.frame("individualid" = integer(),
                           "fitness" = integer(),
                           "unfitness" = integer())


  individuallist <- list()
  for(i in 1:popsize){
    individuallist[[i]] <- encode(n)
    id<- i
    fitnessval <- fitness(individuallist[[i]])
    fit <- fitnessval$noattack
    unfit <- fitnessval$attacknum
    data <- data.frame("individualid" = id,
                       "fitness" = fit,
                       "unfitness" = unfit)
    population <- rbind(population, data)

  }
  generationattack <- c()
  fittime <- 0
  cat("\n", "Working on ", as.character(substitute(encode)), "encoding and mutprobability ", mutprob, "

  for (i in 1:iterations) {

    population <- population[order(population$fitness,decreasing = T),]

    #Best  and worst individual
    #best  <- individuallist[[population$individualid[1]]]
    #print("The best individual has an orientation as follows:\n")
    #print(best)
    #cat("The best individual has a fitness value of ", population$fitness[1])

    #victim  <- individuallist[[population$individualid[nrow(population)]]]
    #print("The victim has an orientation as follows:\n")
    #print(victim)
    #cat("The victim has a fitness value of ", population$fitness[nrow(population)])



    #Evolution
    parents <- sample(individuallist, 2, replace = F)
    pvalue <- sample(1: floor(n/2), 1)
    kid <- crossover(pvalue, parents[[1]], parents[[2]])
    #kid <-kid[[1]]
    if(runif(1)>mutprob){
      kid <- mutate(kid)
    }
```

```
    individuallist[[population$individualid[nrow(population)]]] <- kid

    kidfitness <- fitness(kid)
      population$fitness[nrow(population)] <- kidfitness$noattack
    population$unfitness[nrow(population)] <- kidfitness$attacknum

    #Attacking Capability
    generationattack[i] <-  mean(population$unfitness)

    if(any(population$fitness ==6)&& fittime==0){
      print( "One of the Legal Configuration is :")
      index <-population$individualid[population$fitness ==6]
      print(individuallist[[ index[1]]])
      print( "The generation to which the configuration belongs is: ")
      print(i)
      fittime <-1
    }

    tempfit <- length(which(population$fitness ==6))
    numfit <- numfit+tempfit
  }

  avgfit <- numfit/iterations
  cat("\n Average Fitness over generations: ", avgfit)

  return(generationattack)

}
```

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

```
GAlist41 <- GA(n=4, encode = list_individual,mutprob = 0.1, fitness = fitnesslist)
```

```
##
##  Working on  list_individual encoding and mutprobability  0.1
##
##  Average Fitness over generations:  0
```

```
GAbinary41 <- GA(n=4, encode = binary_individual,mutprob = 0.1, fitness = fitnessbinary)
```

```
##
##  Working on  binary_individual encoding and mutprobability  0.1
##
##  Average Fitness over generations:  0
```

```
GArow41 <- GA(n=4, encode = row_individual,mutprob = 0.1, fitness = fitnessrow)
```

```
##
##  Working on  row_individual encoding and mutprobability  0.1
## [1] "One of the Legal Configuration is :"
## [1] 3 1 4 2
```

```
## [1] "The generation to which the configuration belongs is: "
## [1] 1
##
##  Average Fitness over generations:  1
```

```
#7 Plot on Attacking queens
library(ggplot2)
library(gridExtra)
set.seed(1234567890)


df<-data.frame(x=1:100,GAlist41, GAbinary41, GArow41)
p41<-ggplot(df,aes(x=x))+
  geom_line(aes(y=GAlist41, colour = "GAlist41"))+
  geom_line(aes(y=GAbinary41, colour = "GAbinary41"))+
  geom_line(aes(y=GArow41, colour = "GArow41"))+
  labs(x = "Number of iteration", y = "Number of getting attack ")
```

8. Run your code for n = 4, 8, 16 (if n = 16 requires too much computational time take a different n {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?

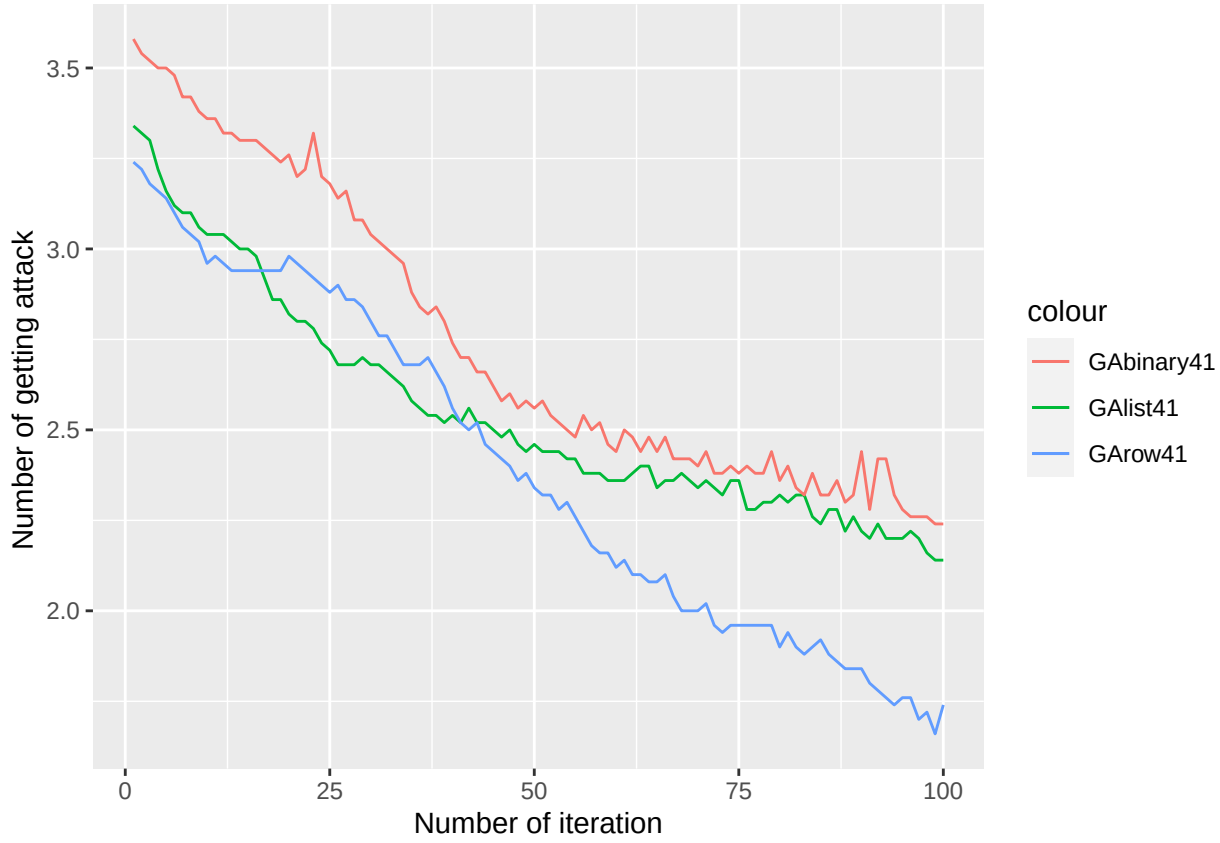
```
##
##  Working on  list_individual encoding and mutprobability  0.5
## [1] "One of the Legal Configuration is :"
##   row col
## 1   1   3
## 2   2   1
## 3   3   4
## 4   4   2
## [1] "The generation to which the configuration belongs is: "
## [1] 1
##
##  Average Fitness over generations:  3.38

##
##  Working on  binary_individual encoding and mutprobability  0.5
## [1] "One of the Legal Configuration is :"
## [1] "010" "100" "001" "011"
## [1] "The generation to which the configuration belongs is: "
## [1] 1
##
##  Average Fitness over generations:  2.39

##
##  Working on  row_individual encoding and mutprobability  0.5
##
##  Average Fitness over generations:  0

##
##  Working on  list_individual encoding and mutprobability  0.9
```

```
## [1] "One of the Legal Configuration is :"
##   row col
## 1   1   3
## 2   2   1
## 3   3   4
## 4   4   2
## [1] "The generation to which the configuration belongs is: "
## [1] 1
##
##  Average Fitness over generations:  1

##
##  Working on  binary_individual encoding and mutprobability  0.9
## [1] "One of the Legal Configuration is :"
## [1] "010" "100" "001" "011"
## [1] "The generation to which the configuration belongs is: "
## [1] 28
##
##  Average Fitness over generations:  1.39

##
##  Working on  row_individual encoding and mutprobability  0.9
## [1] "One of the Legal Configuration is :"
## [1] 3 1 4 2
## [1] "The generation to which the configuration belongs is: "
## [1] 1
##
##  Average Fitness over generations:  2.95

##
##  Working on  list_individual encoding and mutprobability  0.1
##
##  Average Fitness over generations:  0

##
##  Working on  binary_individual encoding and mutprobability  0.1
##
##  Average Fitness over generations:  0

##
##  Working on  row_individual encoding and mutprobability  0.1
##
##  Average Fitness over generations:  0

##
##  Working on  list_individual encoding and mutprobability  0.5
##
##  Average Fitness over generations:  0

##
##  Working on  binary_individual encoding and mutprobability  0.5
##
##  Average Fitness over generations:  0
```

```
##
##  Working on  row_individual encoding and mutprobability  0.5
##
##  Average Fitness over generations:  0

##
##  Working on  list_individual encoding and mutprobability  0.9
##
##  Average Fitness over generations:  0

##
##  Working on  binary_individual encoding and mutprobability  0.9
##
##  Average Fitness over generations:  0

##
##  Working on  row_individual encoding and mutprobability  0.9
##
##  Average Fitness over generations:  0

##
##  Working on  list_individual encoding and mutprobability  0.1
##
##  Average Fitness over generations:  0

##
##  Working on  binary_individual encoding and mutprobability  0.1
##
##  Average Fitness over generations:  0

##
##  Working on  row_individual encoding and mutprobability  0.1
##
##  Average Fitness over generations:  0

##
##  Working on  list_individual encoding and mutprobability  0.5
##
##  Average Fitness over generations:  0

##
##  Working on  binary_individual encoding and mutprobability  0.5
##
##  Average Fitness over generations:  0

##
##  Working on  row_individual encoding and mutprobability  0.5
##
##  Average Fitness over generations:  0

##
##  Working on  list_individual encoding and mutprobability  0.9
##
##  Average Fitness over generations:  0
```

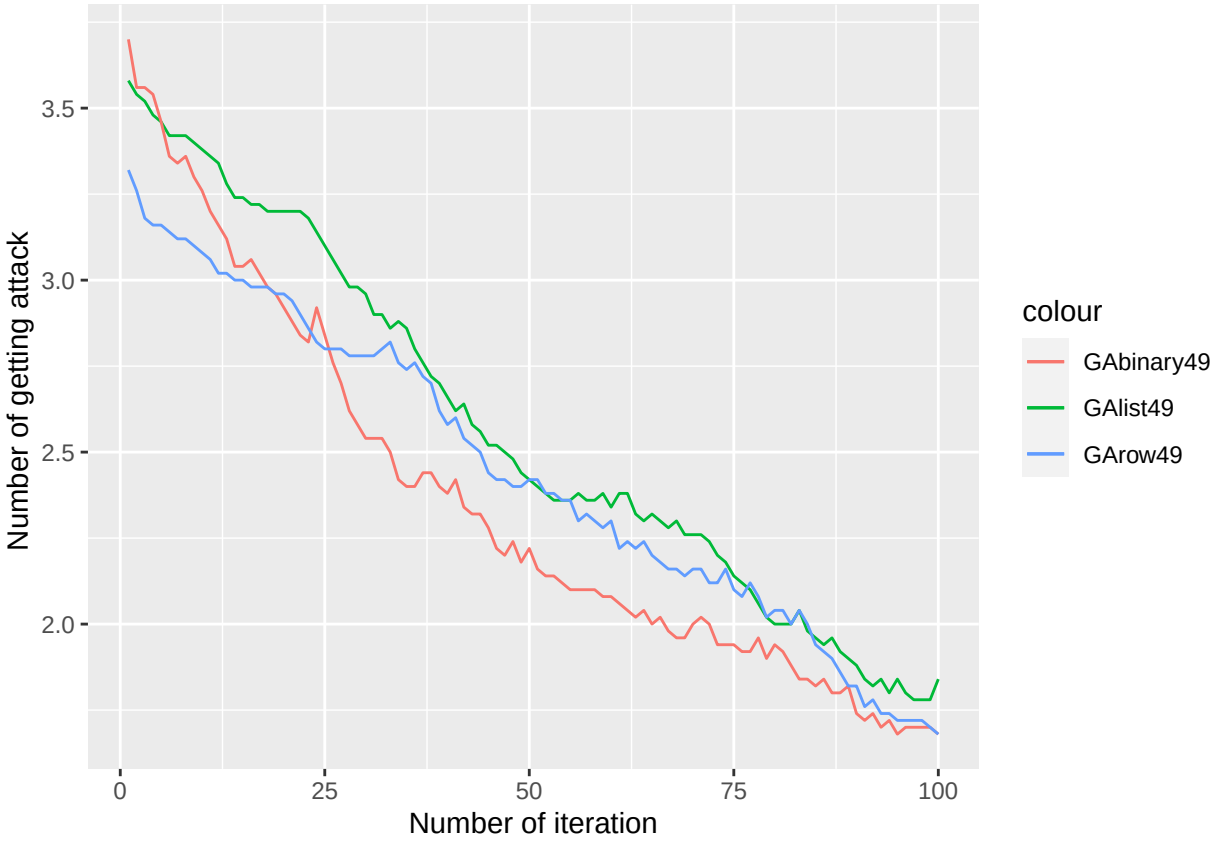
```
##
##  Working on  binary_individual encoding and mutprobability  0.9
##
##  Average Fitness over generations:  0

##
##  Working on  row_individual encoding and mutprobability  0.9
##
##  Average Fitness over generations:  0
```

The plot for attacking queen Vs Iteration for n=4 and mutprob =0.1



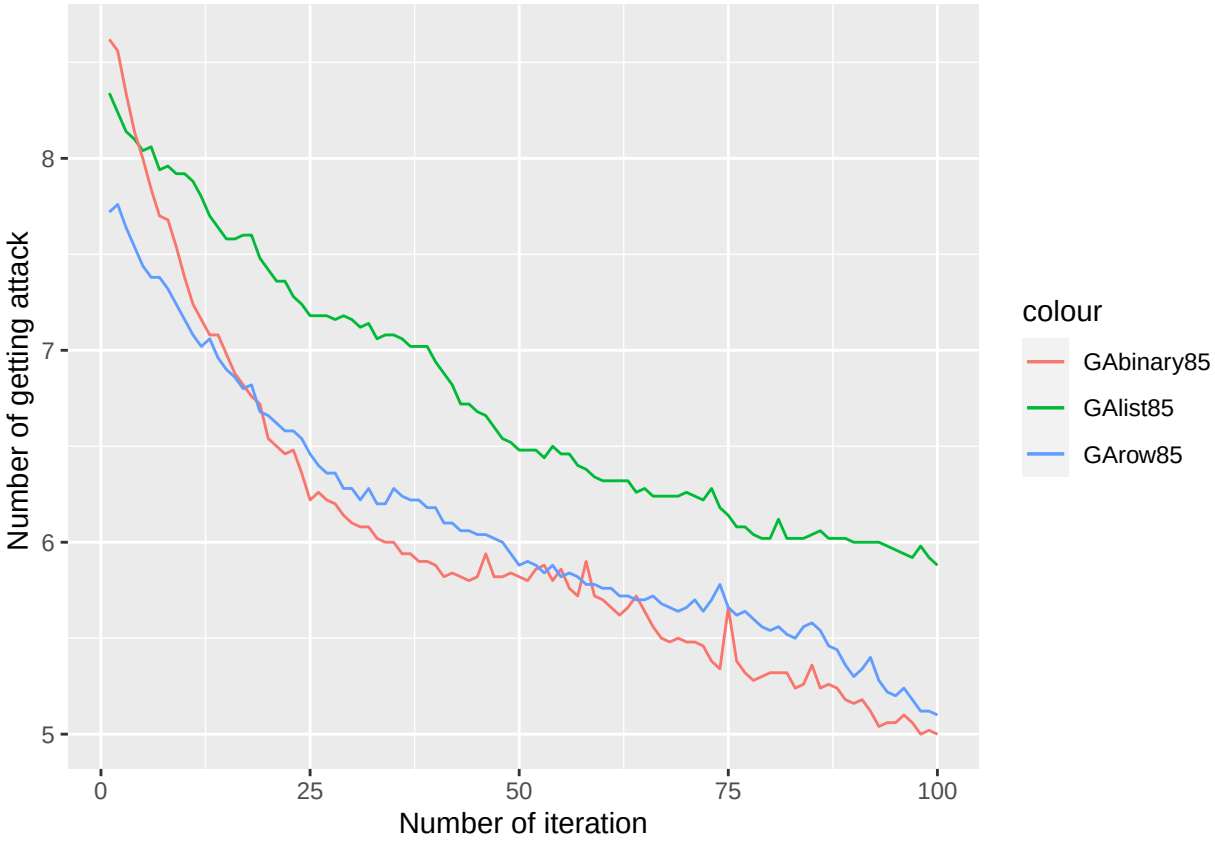
The plot for attacking queen Vs Iteration for n=4 and mutprob =0.5

The plot for attacking queen Vs Iteration for n=4 and mutprob =0.9

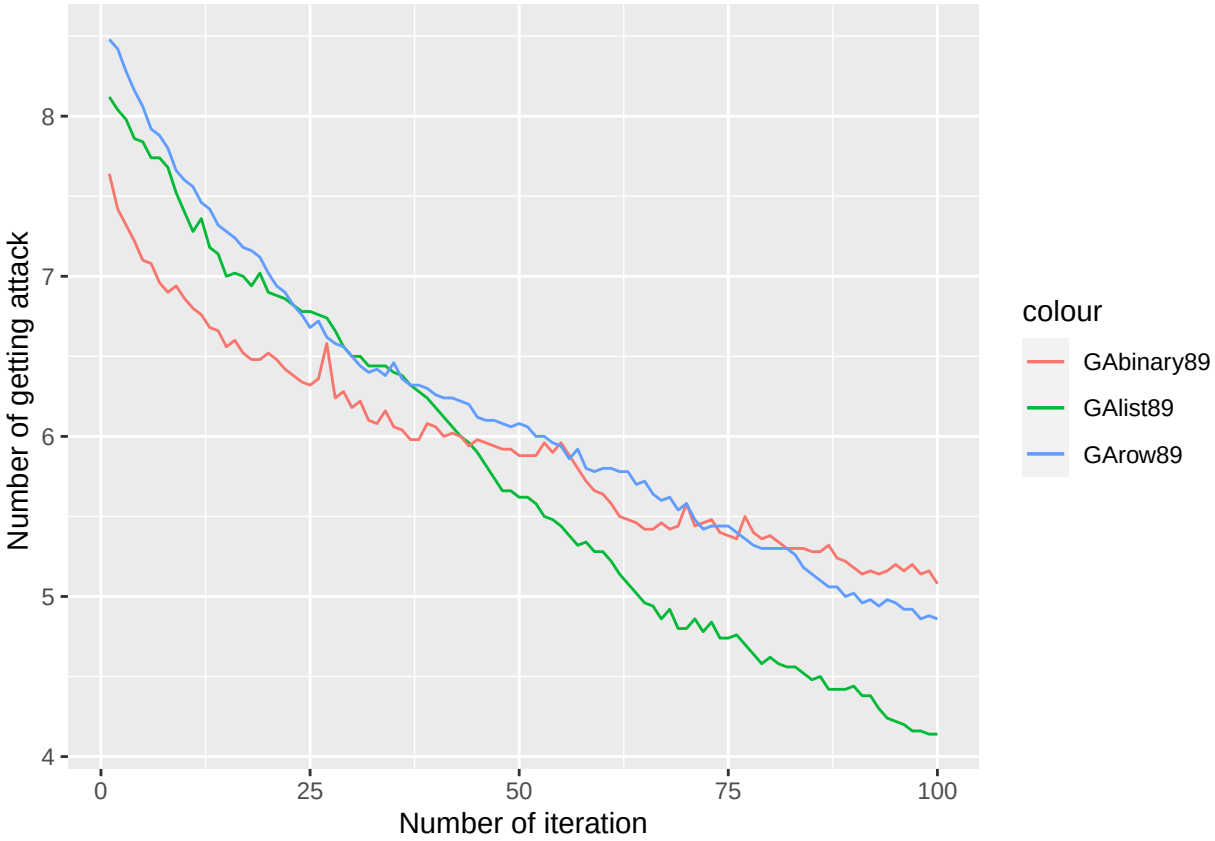


The plot for attacking queen Vs Iteration for n=8 and mutprob =0.1

The plot for attacking queen Vs Iteration for n=8 and mutprob =0.5



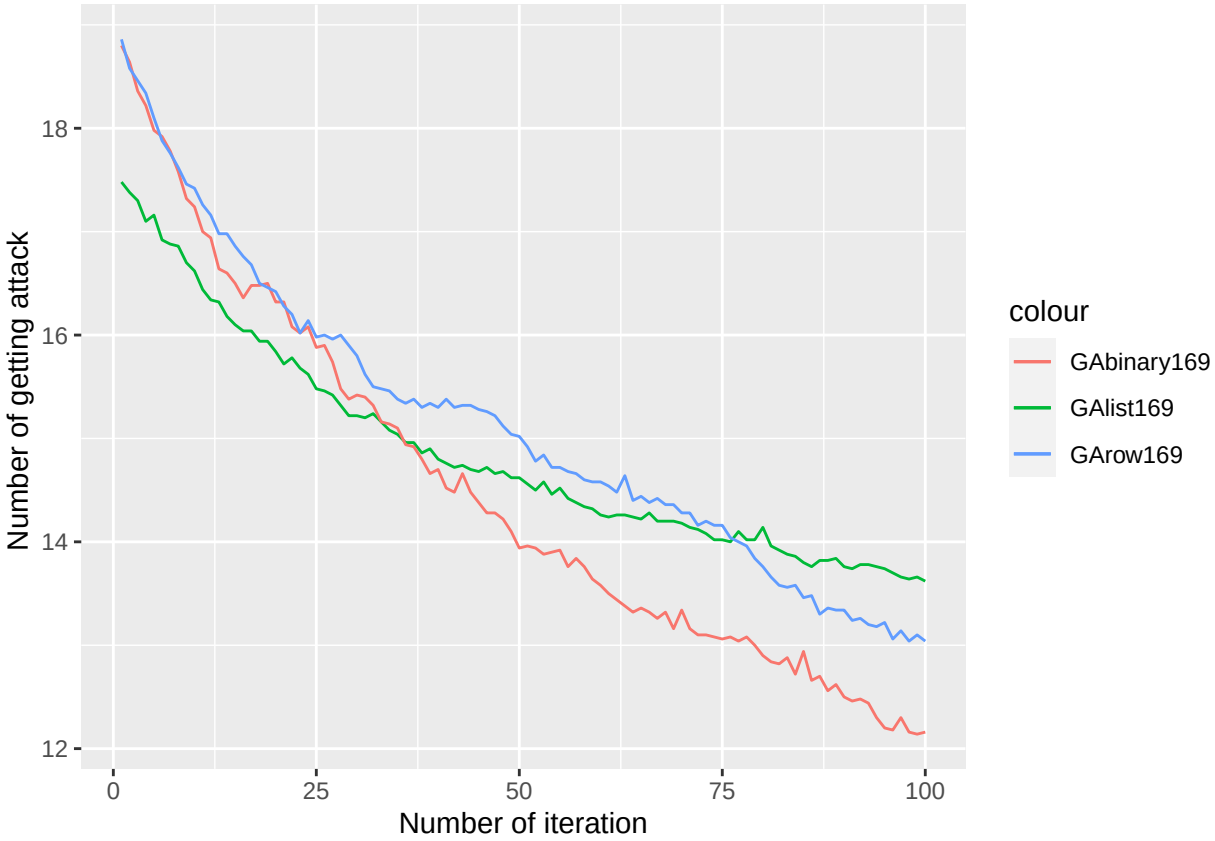
The plot for attacking queen Vs Iteration for n=8 and mutprob =0.9



The plot for attacking queen Vs Iteration for n=16 and mutprob =0.1

The plot for attacking queen Vs Iteration for n=16 and mutprob =0.5

The plot for attacking queen Vs Iteration for n=16 and mutprob =0.9



9. Discuss which encoding and objective function worked best.

From the above graphs, majority of the graphs exhibit pair encoding giving better results. Also, the average ideal solutions achieved for each encoding is printed by the algorithm and row encoding exhibits higher values.

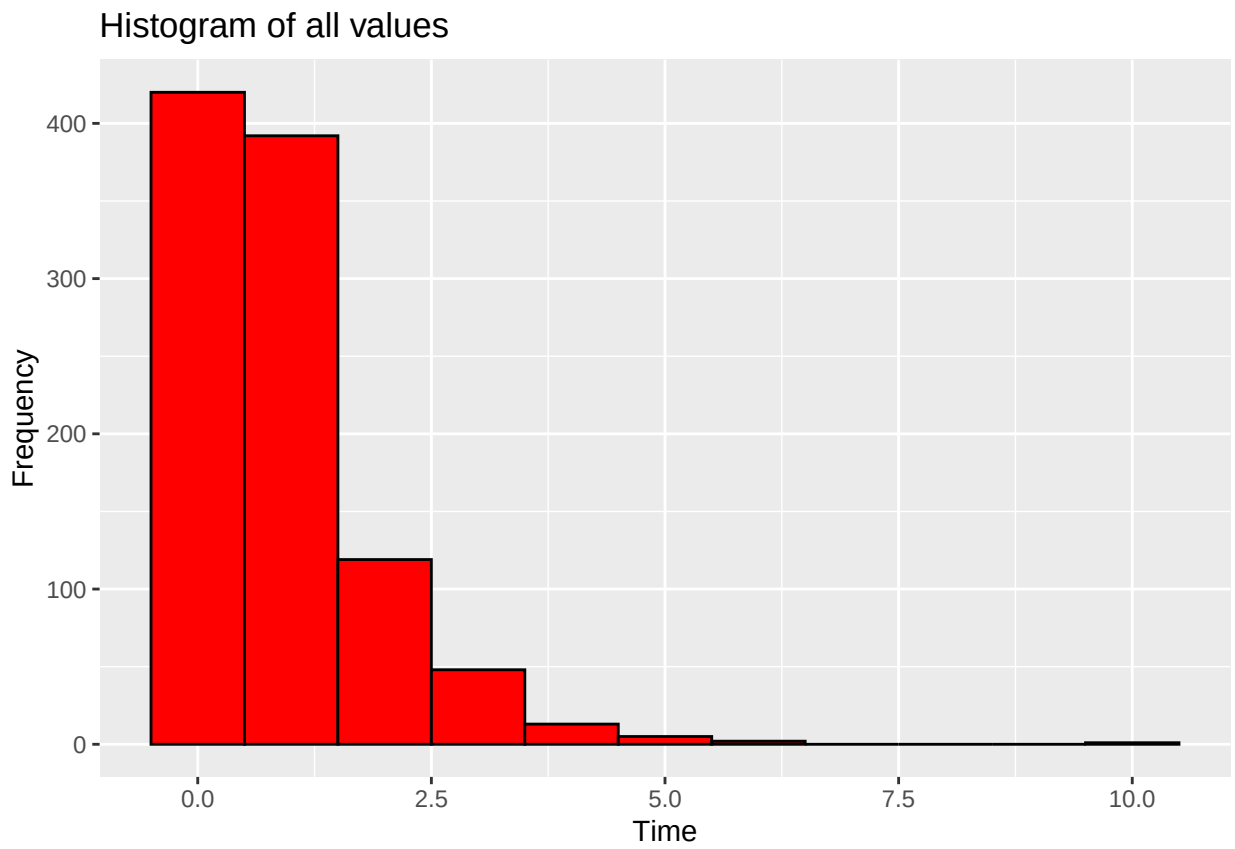
The coding for fitness for pair is computationally slow. The binary function uses only 1s and 0s for calculation which wont be giving better results.

Question 2: EM algorithm

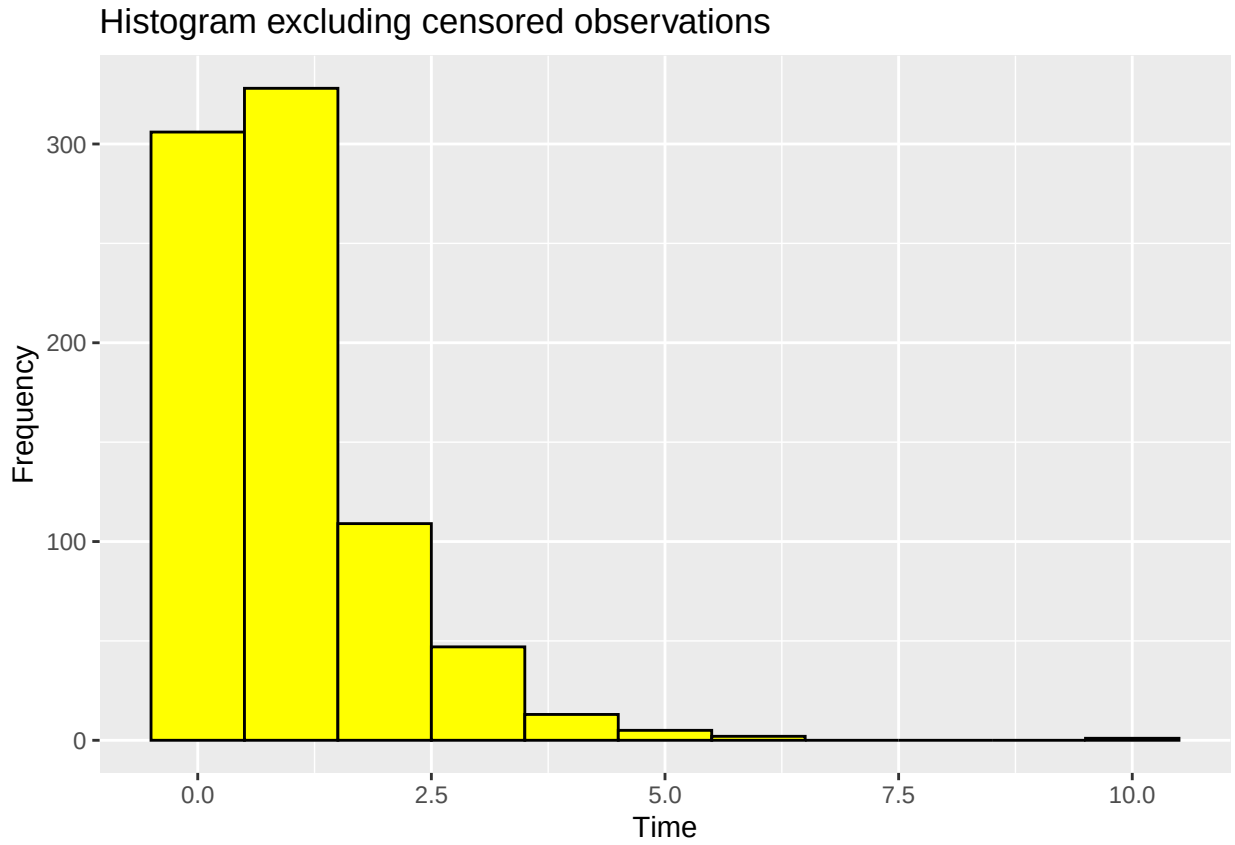
The data file censoredproc.csv contains the time after which a certain product fails. Some of these measurements are left-censored (cens=2)|i.e., we did not observe the time of failure, only that the product had already failed when checked upon. Status cens=1 means that the exact time of failure was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

```
# #q1
library(ggplot2)
library(dplyr)

data <- read.csv("C:/Users/sange/OneDrive/Documents/censoredproc.csv")

ggplot(data, aes(x = time)) +
  geom_histogram(binwidth = 1, fill = "red", color = "black") +
  labs(title = "Histogram of all values", x = "Time", y = "Frequency")
```




```
observed_data <- data %>% filter(cens == 1)

ggplot(observed_data, aes(x = time)) +
  geom_histogram(binwidth = 1, fill = "yellow", color = "black") +
  labs(title = "Histogram excluding censored observations", x = "Time", y = "Frequency")
```



Yes, the histograms remind of an exponential distribution.

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.

Let T_1,T_2,..T_n represent the observed failure times, and let C_1,C_2,..C_m represent the censored times, where n is the number of observed failures and m is the number of censored observations.
The likelihood function for this mixed scenario involving both observed and censored data can be expressed as follows:

For observed failures:

$$L(\lambda|T_1, T_2, ..T_n) = \prod_{i=1}^{n} \lambda e^{-\lambda T_i} = \lambda^n e^{-\lambda \sum_{i=1}^{n} T_i}$$

For censored data: Since the censored data is truncated, meaning it is observed to be greater than a certain value, say C_{min}, the likelihood for censored data would involve the cumulative distribution function (CDF) of the exponential distribution truncated at C_{min}:

$$L_c(\lambda|C_1, C_2, ..C_n) = \prod_{j=1}^{m} (1 - e^{-\lambda C_{min}}) = (1 - e^{-\lambda C_{min}})^m$$

The combined likelihood function, considering both observed and censored data, would then be the product of these likelihoods:

$$L(\lambda \mid observed, censored) = L(\lambda|T_1, T_2, ..T_n).L_c(\lambda|C_1, C_2, ..C_n)$$

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.

Deriving the EM Algorithm for Estimating λ
Assumptions:

Observed failure times $T_1,T_2,..T_n $ follow an exponential distribution with parameter λ

Censored failure times $C_1, C_2, ..C_n$ are from a truncated exponential distribution.

EM Algorithm Steps:

1. Initialization: Initialize $\lambda(\lambda_0)$.

2. E-step (Expectation): Calculate the expected value of the log-likelihood function considering the unobserved or missing data (censored data).

3. M-step (Maximization): Update λ to maximize the expected log-likelihood function. Utilize numerical optimization methods (e.g., Newton-Raphson, gradient descent) for maximizing the likelihood function due to its nonlinearity.

4. Iteration: Iterate between the E-step and M-step until convergence. Convergence criteria include a change in λ smaller than a predefined threshold (ε) or reaching a maximum number of iterations.

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
# Load required libraries
library(dplyr)
library(readr)

# Read the CSV file
data <- read_csv("C:/Users/sange/OneDrive/Documents/censoredproc.csv")

# Extract observed and censored data
observed_data <- data %>%
  filter(cens == 1) %>%
  pull(time)

censored_data <- data %>%
  filter(cens == 2) %>%
  pull(time)

# Function to calculate the expected log-likelihood
expected_log_likelihood <- function(lambda, observed, censored) {
  log_likelihood_observed <- length(observed) * log(lambda) - lambda * sum(observed)

  log_likelihood_censored <- sum(log(lambda) - lambda * censored - log(1 - exp(-lambda * censored)))
  final_log<- log_likelihood_observed + log_likelihood_censored
  returnVal<-c(lambda,final_log)
  return(final_log)
}
```

```r
lambda_initial<-0
# Updated EM algorithm function to estimate lambda
EM_algorithm <- function(observed, censored, lambda_initial, epsilon = 0.001, max_iter = 1000) {
  lambda_old <- lambda_initial
  iteration <- 0
  converged <- FALSE
  observed<- observed_data
  censored<-censored_data
  while (!converged && iteration < max_iter) {
    lambda_new <- lambda_old

    # E-step: Calculate the expected log-likelihood
    expected_ll <- expected_log_likelihood(lambda_old, observed, censored)

    # M-step: Update lambda using numerical optimization (optimize function)
    objective_function <- function(x) {
      lambda<-x[1]
      observed<-x[2]
      censored<-x[3]
      return(expected_log_likelihood(lambda, observed, censored))
    }

    # Find the optimal lambda using optimization
    lambda_new <- nrow(data) / (sum(observed) + 1 + sum(censored * exp(-lambda_old * censored))/sum(1-la



    # Stopping condition
    if (abs(lambda_new - lambda_old) < epsilon) {
      converged <- TRUE
    }

    lambda_old <- lambda_new
    iteration <- iteration + 1
  }

  return(list(lambda_estimate = lambda_old, iterations = iteration))
}



# Applying EM algorithm to the observed and censored data
result <- EM_algorithm(observed_data, censored_data, lambda_initial = 100, epsilon = 0.001)

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

```
## Estimated lambda: 1.236563
```

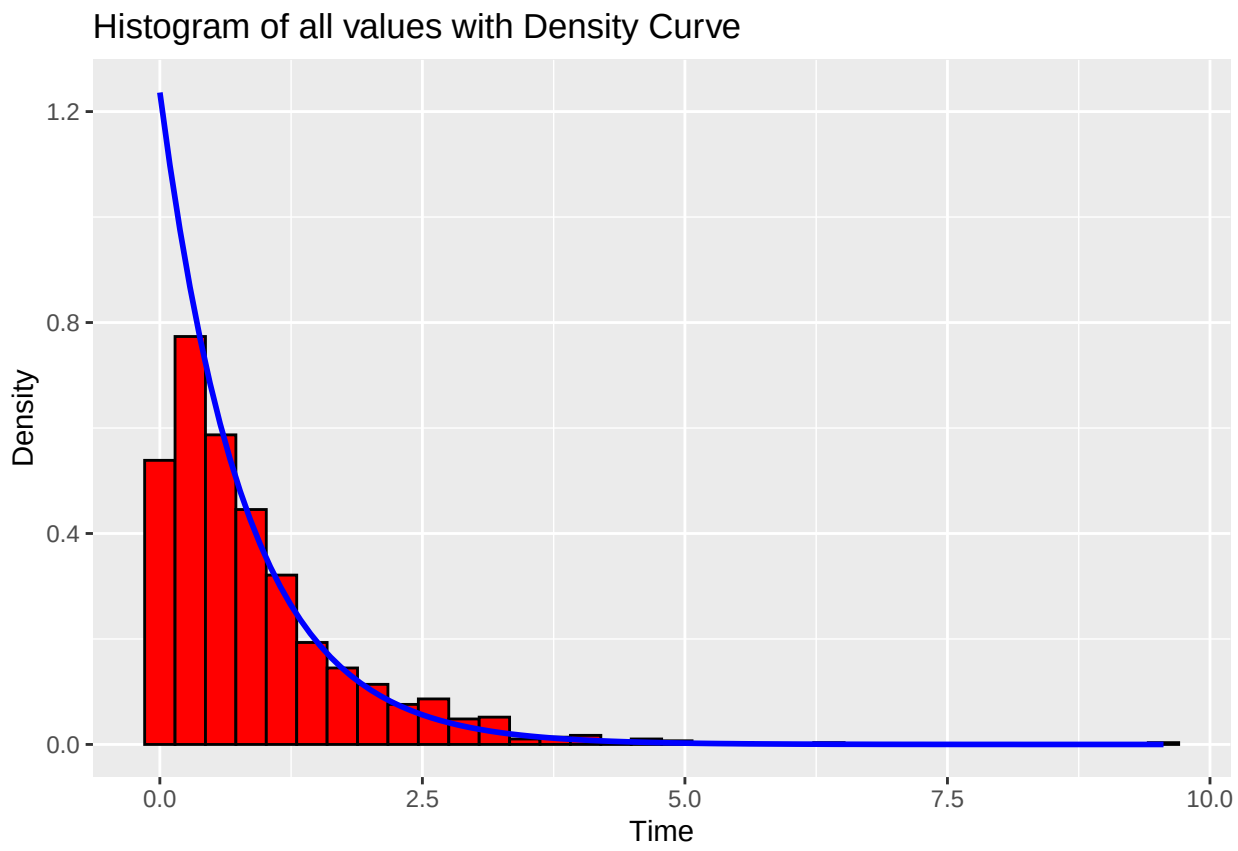
```r
cat("Number of iterations:", result$iterations, "\n")
```

```
## Number of iterations: 2
```

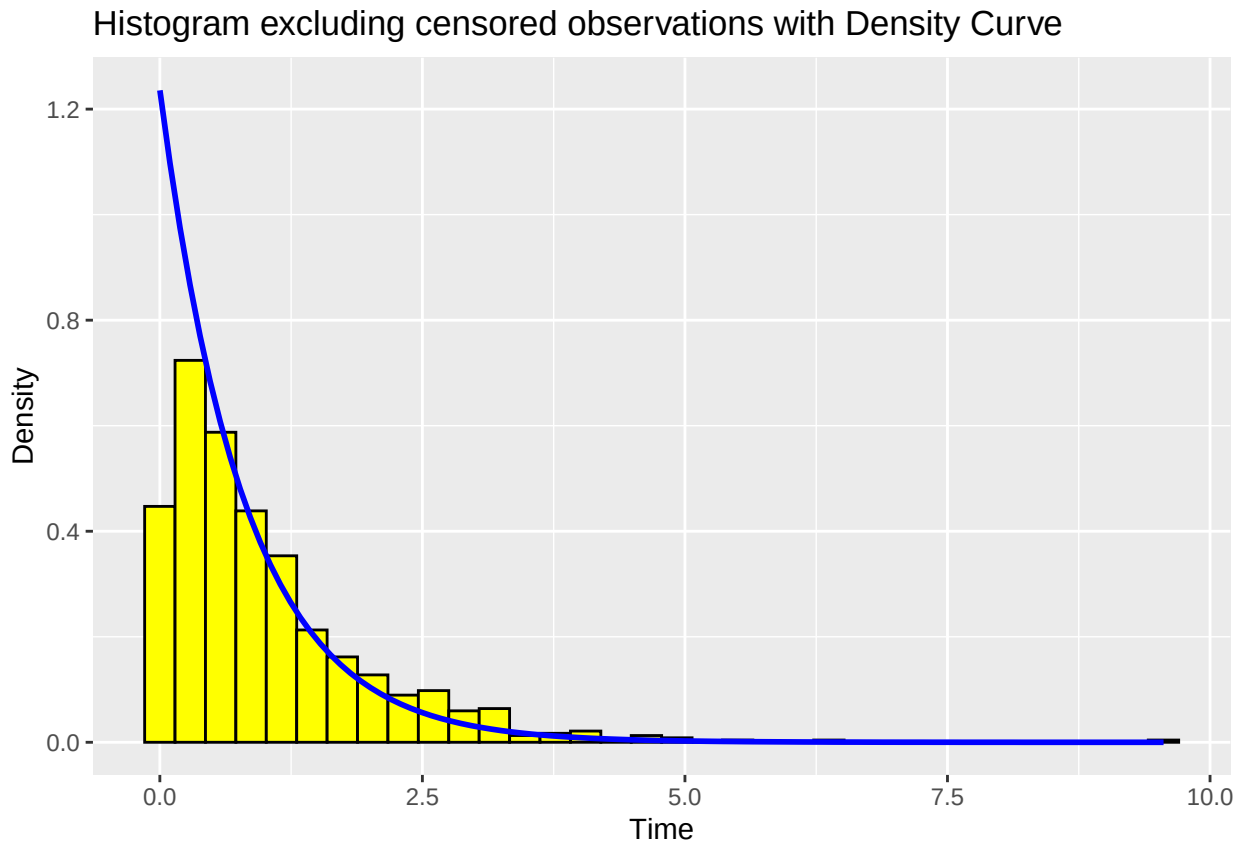
```r
estimated_lambda <- result$lambda_estimate
```

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

```r
#q5
# Plot histogram of all values with density curve
ggplot(data, aes(x = time)) +
  geom_histogram(bins = 34, fill = "red", color = "black", aes(y = ..density..)) +
  stat_function(fun = dexp, args = list(rate = estimated_lambda), color = "blue", size = 1) +
  labs(title = "Histogram of all values with Density Curve", x = "Time", y = "Density")
```




```r
# Filter out censored observations (cens = 1 for observed failures)
observed_data <- data[data$cens == 1,]

# Plot histogram excluding censored observations with density curve
ggplot(observed_data, aes(x = time)) +
  geom_histogram(bins = 34, fill = "yellow", color = "black", aes(y = ..density..)) +
  stat_function(fun = dexp, args = list(rate = estimated_lambda), color = "blue", size = 1) +
  labs(title = "Histogram excluding censored observations with Density Curve", x = "Time", y = "Density
```

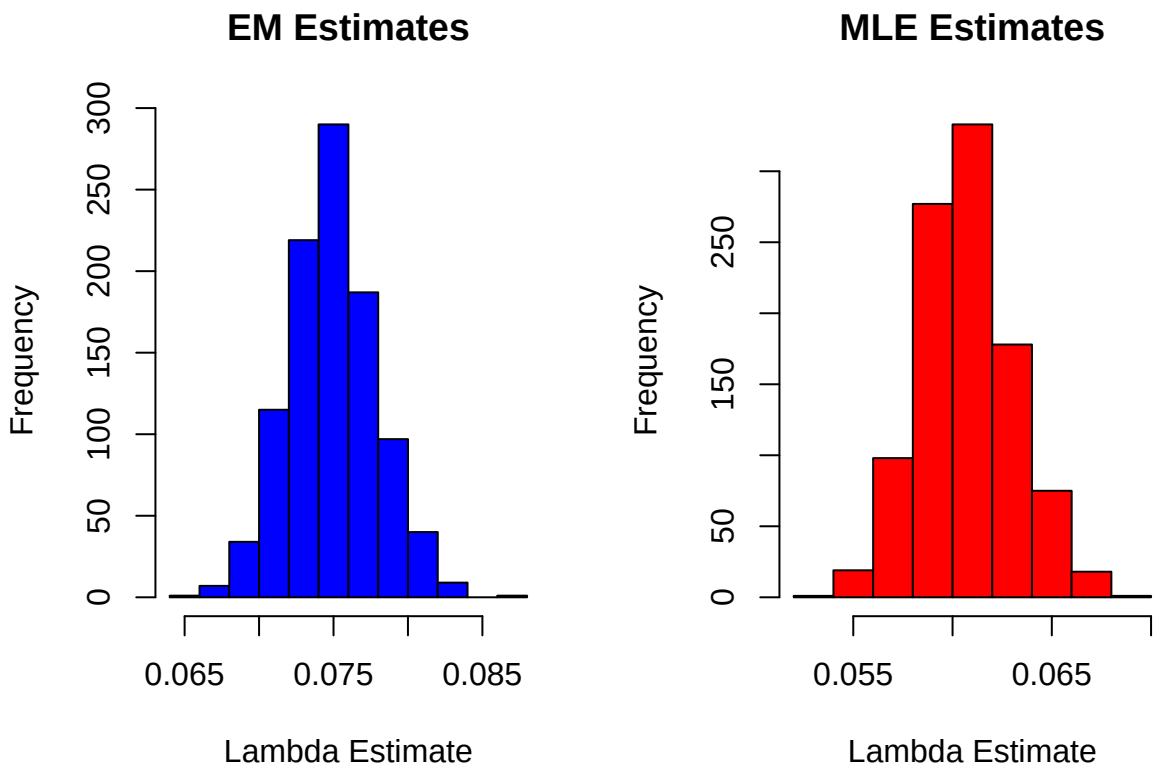


6. Study how good your EM algorithm is compared to usual maximum likelihood estimation with data reduced to only the uncensored observations. To this end we will use a parametric bootstrap. Repeat 1000 (reduce if computational time is too long - but carefully report the running times) times the following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

(a) Simulate the same number of data points as in the original data, from the exponential λ 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 some distribution on [true time,∞). Remember that the observation was censored.
(c) Estimate λ both by your EM-algorithm, and maximum likelihood based on the uncensored observations.

```
library(dplyr)

# Assuming data is loaded and necessary functions for EM estimation are defined

# Set the number of iterations
num_iterations <- 1000

# Initialize empty vectors to store results
results_EM <- numeric(num_iterations)
results_MLE <- numeric(num_iterations)
true_lambda <- 0.05
# Perform iterations
for (i in 1:num_iterations) {
  # Part a) Simulate data from exponential distribution
  simulated_data <- rexp(n = nrow(data), rate = true_lambda)

  # Part b) Censoring
```

```r
  min_observed_time <- min(data$time)  # Calculate minimum observed time

  true_time <- min_observed_time + 0.5
  censored_indices <- sample(1:nrow(data), sum(data$cens == 2), replace = FALSE)
  for (index in censored_indices) {
    # Sample a new time from a distribution on [true_time, infinity)
    simulated_data[index] <- true_time + rexp(1, rate = 1)
  }

  # Part c) Estimate lambda using EM algorithm
  observed_data <- simulated_data[data$cens == 1]
  censored_data <- simulated_data[data$cens == 2]
  result_EM <- EM_algorithm(observed_data, censored_data, lambda_initial = 100, epsilon = 0.001)
  results_EM[i] <- result_EM$lambda_estimate

  # Estimate lambda using Maximum Likelihood Estimation from uncensored data
  lambda_MLE_uncensored <- 1 / mean(observed_data)
  results_MLE[i] <- lambda_MLE_uncensored
}

# Create histograms for EM estimates and MLE estimates
par(mfrow = c(1, 2))  # Set up a grid for side-by-side plots

hist(results_EM, col = "blue", main = "EM Estimates", xlab = "Lambda Estimate", ylab = "Frequency")
hist(results_MLE, col = "red", main = "MLE Estimates", xlab = "Lambda Estimate", ylab = "Frequency")
```



```r
true_lambda <- 0.05  # Replace with the actual true lambda value if known
bias_EM <- mean(results_EM) - true_lambda
bias_MLE <- mean(results_MLE) - true_lambda
# Assuming 'results_EM' and 'results_MLE' contain the estimates

# Calculate variance for EM estimates
variance_EM <- var(results_EM)

# Calculate variance for MLE estimates
variance_MLE <- var(results_MLE)

# Print the variance of the estimators
cat("Variance of EM Estimates:", variance_EM, "\n")
```

```
## Variance of EM Estimates: 8.54874e-06
```

```r
cat("Variance of MLE Estimates:", variance_MLE, "\n")
```

```
## Variance of MLE Estimates: 5.663569e-06
```

If the bias is close to zero, it suggests the estimator is relatively unbiased. Here both bias_EM = 0.024911 and bias_MLE = 0.0106891, are closer to zero. Therefore, both of them seem unbiased.
