Lab 6

Hugo Morvan, Daniele Bozzoli

2024-01-23

Question 1: Genetic algorithm

Pairs Encoding

```
#################### PAIRS REPRESENTATION ####################

# Question 2 : Crossover

crossover <- function(layout1, layout2, p){
  #PAIRS (matrix)
  kid <- matrix(nrow = nrow(layout1), ncol = 2)
  for (i in 1:p){
    kid[i,] <- layout1[i,]
  }
  for (i in (p+1):nrow(layout2)){
    kid[i,] <- layout2[i,]
  }
  return(kid)
}

# Question 3 : Mutate

mutate <- function(layout,n){
  pair <- sample(n,1)
  layout[pair,1] <- sample(n,1)
  layout[pair,2] <- sample(n,1)
  #Disadvantages of this layout : can move to a location where there is already a queen
  # --> Good or bad ? Good: simulate a queen "disappearing", bad: duplicates in the representation
  return(layout)
}

# Question 4 : Fitness

fitness <- function(layout, fitness_function, n){
  board <- matrix(nrow = n, ncol = n)
  #fill the board with 0s
  for (i in 1:n){
    for (j in 1:n){
      board[i,j] <- 0
    }
```

```
 }
 #fill the board with 1s where there are queens
 for (i in 1:n){
   board[layout[i,1], layout[i,2]] <- 1
 }
 #================================================================================
#Convert the board to a list of coordinates
 my_vec <- c()
 for(i in 1:n){
   for(j in 1:n){
     if(board[i,j]==1){
       my_vec <- c(my_vec, c(i,j))
     }
   }
 }
 if(length(my_vec) == 0){
   return(0)
 }
 points<-array( my_vec, dim=c(2,length(my_vec)/2))
 are_attacking_eachother <- function(Q1,Q2){
   #https://stackoverflow.com/questions/57239548/how-to-check-if-a-queen-is-under-attack-in-nqueens
   (Q1[1] == Q2[1] | Q1[2] == Q2[2] | Q1[1]-Q1[2] == Q2[1]-Q2[2] | Q1[1]+Q1[2] == Q2[1]+Q2[2])
 }
 fitness <- 0
 if(length(points)/2 == 1){
   if(fitness_function == "binary"){
     return(0)
   }else if(fitness_function == "num_safe"){
     return(1)
   }else{
     #number of attacking pairs
     return(0)
   }
 }else if(fitness_function == "binary"){
     for(queen in 1:((length(points)/2)-1)){
       for(other_queen in (queen+1):(length(points)/2)){
         Q1 <- c(points[1,queen], points[2,queen])
         Q2 <- c(points[1,other_queen], points[2,other_queen])
         if(are_attacking_eachother(Q1, Q2)){
           fitness <- 1
         }
       }
     }
 }else if(fitness_function == "num_safe"){
   safe <- 0
    for(queen in 1:((length(points)/2)-1)){
       for(other_queen in (queen+1):(length(points)/2)){
         Q1 <- c(points[1,queen], points[2,queen])
         Q2 <- c(points[1,other_queen], points[2,other_queen])
         if(!are_attacking_eachother(Q1, Q2)){
           safe <- safe + 2
         }
       }
```

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

# Question 5 : Genetic Algorithm

generate_random_layout_pairs <- function(n){
  pairs<-array(0, dim=c(n,2))
  for (i in 1:n){
    pairs[i,1] <- sample(n,1)
    pairs[i,2] <- sample(n,1)
  }
  return(pairs)
}

darwin_pairs <- function(n_generations, board_size, n_population, mutprob, fitness_function, crossover_
  ###### Initialization ######
  #generate random population
  population <- list()
  for(i in 1:n_population){
    population[[i]] <- generate_random_layout_pairs(board_size)
  }

  #calculate fitness of population
```

```
  fitnesses <- c()
  for(i in 1:n_population){
    fitnesses[i] <- fitness(population[[i]], fitness_function, board_size)
  }
  n_attacking_queens <- c()
  solutions <- list()
  ###### Genetic Algorithm ######
  for(generation in 1:n_generations){

    #Two individuals are randomly sampled from the current population,
    # they are further used as parents
    parents <- sample(population, 2, replace = FALSE)

    #One individual with the smallest fitness is selected from the current population,
    # this will be the victim
    victim <- population[which.min(fitnesses)]
    victim_idx <- which.min(fitnesses)
    #The two sampled parents are to produce a kid by crossover
    kid <- crossover(parents[[1]], parents[[2]], p=crossover_p)

    #this kid should be mutated with probability mutprob
    if(runif(1) < mutprob){
      kid <- mutate(kid, board_size)
    }

    #The victim is replaced by the kid in the population
    population[[victim_idx]] <- kid

    #Do not forget to update the vector of fitness values of the population
    fitnesses[victim_idx] <- fitness(kid, fitness_function, board_size)

    #Remember the number of pairs of queens attacking each other at the given iteration
    attacking_queens <- 0
    for(i in 1:n_population){
      attacking_queens <- attacking_queens + fitness(population[[i]], "num_attacking", board_size)
      if(fitness(population[[i]], "num_attacking", board_size) == 0){
        solutions[[length(solutions)+1]] <- population[[i]]
      }
    }
    n_attacking_queens <- c(n_attacking_queens, attacking_queens)

  }
  return(list(n_attacking_queens = n_attacking_queens, solutions = solutions))
}
```

6.

If found, return the legal configuration of queens.

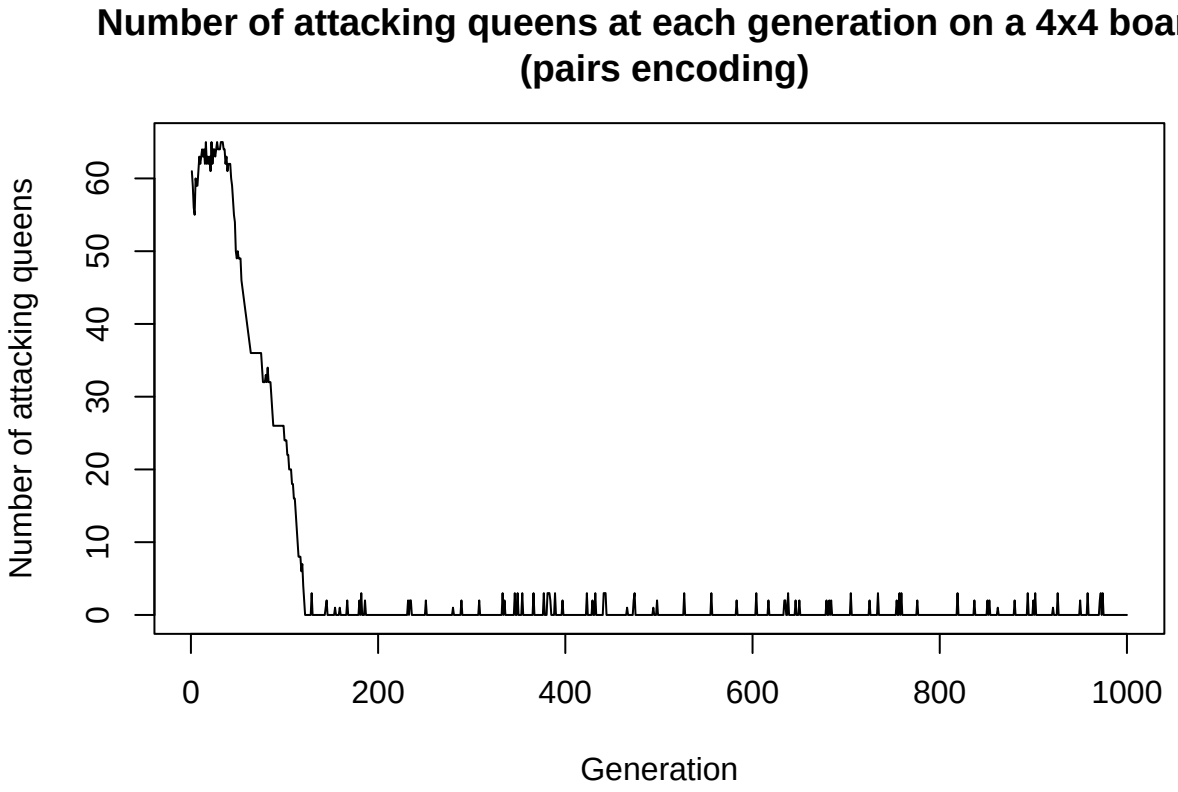
```
## found 17967 legal configurations
```

```
## Example of a legal configuration :
```

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

7.

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



Binary Encoding

```
#################### Binary Representation ####################

# Question 2 : Crossover

crossover <- function(layout1, layout2, p){
  #p < log2(n)/2
  kid <- list()
  for (i in 1:p){
    kid[[i]] <- layout1[[i]]
  }
  for (i in (p+1):length(layout2)){
    kid[[i]] <- layout2[[i]]
```

```
  }
  return(kid)
}

# Question 3 : Mutate

mutate <- function(layout,n){
    #randomly select a binary string
    binary_string <- sample(1:length(layout), 1)
    #randomly select a bit in the binary string
    bit <- sample(1:length(layout[[binary_string]]), 1)
    #flip the bit
    layout[[binary_string]][bit] <- abs(layout[[binary_string]][bit] - 1)
    return(layout)
}

# Question 4 : Fitness

fitness <- function(layout, fitness_function, n){
  #Very twisted and inefficient way of doing things, to be modified
  board <- matrix(nrow = n, ncol = n)
  #fill the board with 0s
  for (i in 1:nrow(board)){
    for (j in 1:ncol(board)){
      board[i,j] <- 0
    }
  }
  for (i in 1:length(layout)){
    board[i, BinToDec(layout[[i]])] <- 1
  }
  board <- t(board)

  #================================================================================
  #At this point, the board var should contain a matrix on the state of the board, with 1s for queens a
  #Convert the board to a list of coordinates
  my_vec <- c()
  for(i in 1:n){
    for(j in 1:n){
      if(board[i,j]==1){
        my_vec <- c(my_vec, c(i,j))
      }
    }
  }
  if(length(my_vec) == 0){
    return(0)
  }
  points<-array( my_vec, dim=c(2,length(my_vec)/2))
  are_attacking_eachother <- function(Q1,Q2){
    #https://stackoverflow.com/questions/57239548/how-to-check-if-a-queen-is-under-attack-in-nqueens
    (Q1[1] == Q2[1] | Q1[2] == Q2[2] | Q1[1]-Q1[2] == Q2[1]-Q2[2] | Q1[1]+Q1[2] == Q2[1]+Q2[2])
  }
  fitness <- 0
  #If only one queen on the board:
```

```
if(length(points)/2 <= 1){
  if(fitness_function == "binary"){
    return(0)
  }else if(fitness_function == "num_safe"){
    return(length(points)/2)
  }else{ #number of attacking pairs
    return(0)
  }

}else if(fitness_function == "binary"){
    for(queen in 1:((length(points)/2)-1)){
      for(other_queen in (queen+1):(length(points)/2)){
        Q1 <- c(points[1,queen], points[2,queen])
        Q2 <- c(points[1,other_queen], points[2,other_queen])
        if(are_attacking_eachother(Q1, Q2)){
          fitness <- 1
        }
      }
    }
}else if(fitness_function == "num_safe"){
  safe <- 0
   for(queen in 1:((length(points)/2)-1)){
      for(other_queen in (queen+1):(length(points)/2)){
        Q1 <- c(points[1,queen], points[2,queen])
        Q2 <- c(points[1,other_queen], points[2,other_queen])
        if(!are_attacking_eachother(Q1, Q2)){
          safe <- safe + 2
        }
      }
   }
  #to scale to [0,1], we can divide by the max number of queens that can be safe
  fitness <- safe

}else if(fitness_function == "num_attacking"){
  num_attacking <- 0
  for(queen in 1:((length(points)/2)-1)){
    for(other_queen in (queen+1):(length(points)/2)){
      Q1 <- c(points[1,queen], points[2,queen])
      Q2 <- c(points[1,other_queen], points[2,other_queen])
      if(are_attacking_eachother(Q1, Q2)){
        num_attacking <- num_attacking + 1
      }
    }
  }
  fitness <- num_attacking
}else{
    #(nC2 - number) of pairs of queens attacking each other.
    nc2 <- (n*(n-1)/2)
    num_attacking <- 0
    for(queen in 1:((length(points)/2)-1)){
      for(other_queen in (queen+1):(length(points)/2)){
        Q1 <- c(points[1,queen], points[2,queen])
        Q2 <- c(points[1,other_queen], points[2,other_queen])
```

```
          if(are_attacking_eachother(Q1, Q2)){
            fitness <- 1
          }
        }
      }
      fitness <- nc2 - num_attacking
  }
  return(fitness)
}

# Question 5 : Genetic Algorithm

generate_random_layout_binary <- function(n){
  #Only works for n powers of 2, otherwise it can produces queens outside of the board
  layout <- list()
  for (i in 1:n){
    layout[[i]] <- sample(0:1, ceiling(log2(n)), replace = TRUE)
  }
  return(layout)
}

darwin_bin <- function(n_generations, board_size, n_population, mutprob, fitness_function, crossover_p)

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

  #calculate fitness of population
  fitnesses <- c()
  for(i in 1:n_population){
    fitnesses[i] <- fitness(population[[i]], fitness_function, board_size)
  }
  n_attacking_queens <- c()
  solutions <- list()
  ###### Genetic Algorithm ######
  for(generation in 1:n_generations){
    #Two individuals are randomly sampled from the current population, they are further used as parents
    parents <- sample(population, 2, replace = FALSE)

    #One individual with the smallest fitness is selected from the current population, this will be the
    victim <- population[which.min(fitnesses)][[1]]
    victim_idx <- which.min(fitnesses)

    #The two sampled parents are to produce a kid by crossover
    kid <- crossover(parents[[1]], parents[[2]], p=crossover_p)
    #this kid should be mutated with probability mutprob
    if(runif(1) < mutprob){
      kid <- mutate(kid, board_size)
    }
    #The victim is replaced by the kid in the population
```

```
    population[[victim_idx]] <- kid

    #Do not forget to update the vector of fitness values of the population
    fitnesses[victim_idx] <- fitness(kid, fitness_function, board_size)

  #Remember the number of pairs of queens attacking each other at the given iteration
    attacking_queens <- 0
    for(i in 1:n_population){
      attacking_queens <- attacking_queens + fitness(population[[i]], "num_attacking", board_size)
      if(fitness(population[[i]], "num_attacking", board_size) == 0){
        solutions[[length(solutions)+1]] <- population[[i]]
      }
    }
    n_attacking_queens <- c(n_attacking_queens, attacking_queens)

  }
  return(list(n_attacking_queens = n_attacking_queens, solutions = solutions))
}
```

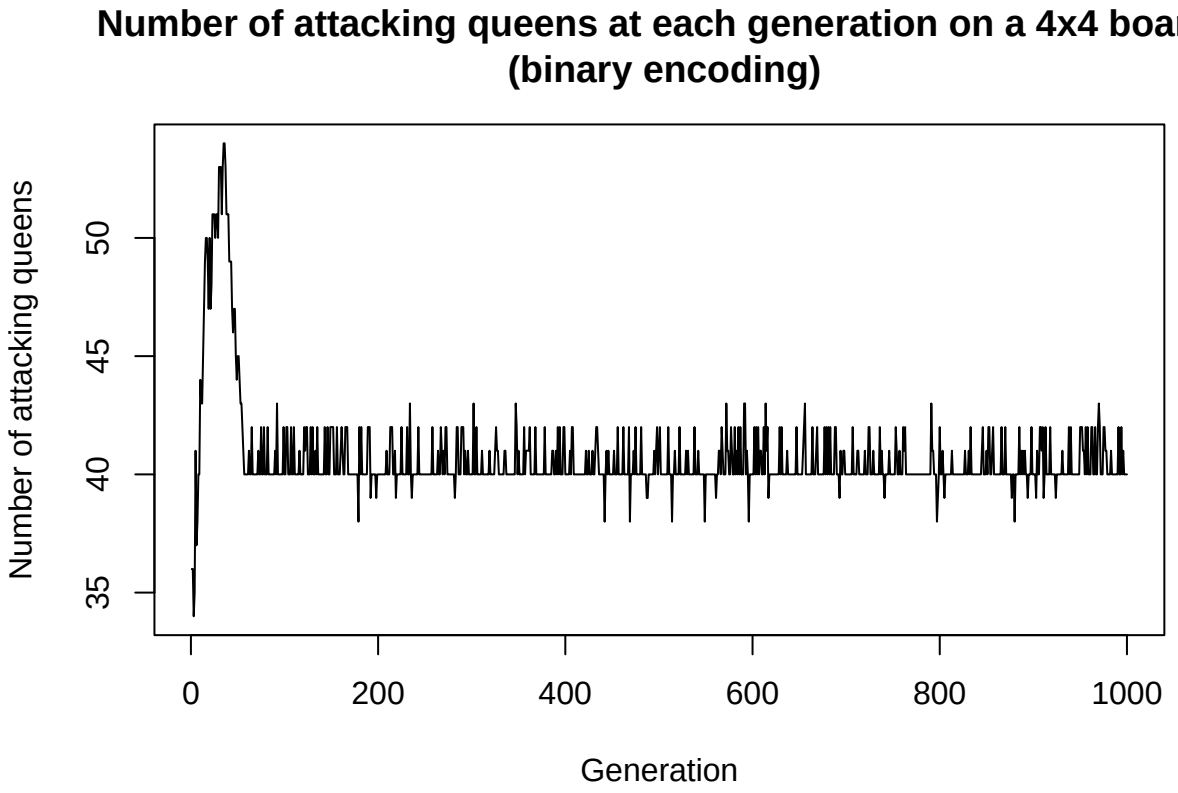
6.

If found, return the legal configuration of queens.

```
## found 39 legal configurations

## Example of a legal configuration :

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

7.

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



Singles Encoding

```
################### Singles Representation ###################

# Question 2 : Crossover

crossover <- function(layout1, layout2, p){
  kid <- c()
    for (i in 1:p){
      kid <- c(kid , layout1[i])
    }
    for (i in (p+1):length(layout2)){
      kid <- c(kid , layout2[i])
    }
    return(kid)
}

# Question 3 : Mutate

mutate <- function(layout,n){
    row <- sample(n,1)
    layout[row] <- sample(n,1)
    #Disadvantages of this method : a queen can only be moved within a column, and it can also not be m
    return(layout)
```

```
}

# Question 4 : Fitness

fitness <- function(layout, fitness_function, n){
  #Very twisted and inefficient way of doing things, to be modified
  board <- matrix(nrow = n, ncol = n)
  #fill the board with 0s
  for (i in 1:n){
    for (j in 1:n){
      board[i,j] <- 0
    }
  }
  for (i in 1:n){
    board[i,layout[i]] <- 1
  }
  board <- t(board)
  #================================================================================
  #At this point, the board var should contain a matrix on the state of the board, with 1s for queens a
  #Convert the board to a list of coordinates
  my_vec <- c()
  for(i in 1:n){
    for(j in 1:n){
      if(board[i,j]==1){
        my_vec <- c(my_vec, c(i,j))
      }
    }
  }
  if(length(my_vec) == 0){
    return(0)
  }
  points<-array( my_vec, dim=c(2,length(my_vec)/2))
  are_attacking_eachother <- function(Q1,Q2){
    #https://stackoverflow.com/questions/57239548/how-to-check-if-a-queen-is-under-attack-in-nqueens
    (Q1[1] == Q2[1] | Q1[2] == Q2[2] | Q1[1]-Q1[2] == Q2[1]-Q2[2] | Q1[1]+Q1[2] == Q2[1]+Q2[2])
  }
  fitness <- 0
  #If only one queen on the board:
  if(length(points)/2 <= 1){
    if(fitness_function == "binary"){
      return(0)
    }else if(fitness_function == "num_safe"){
      return(length(points)/2)
    }else{ #number of attacking pairs
      return(0)
    }
  }else if(fitness_function == "binary"){
      for(queen in 1:((length(points)/2)-1)){
        for(other_queen in (queen+1):(length(points)/2)){
          Q1 <- c(points[1,queen], points[2,queen])
          Q2 <- c(points[1,other_queen], points[2,other_queen])
          if(are_attacking_eachother(Q1, Q2)){
            fitness <- 1
```

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

# Question 5 : Genetic Algorithm

generate_random_layout_singles <- function(n){
  layout <- c()
  for (i in 1:n){
    layout[i] <- sample(1:n,1)
  }
```

```
  return(layout)
}

darwin_singles <- function(n_generations, board_size, n_population, mutprob, fitness_function, crossover

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

  #calculate fitness of population
  fitnesses <- c()
  for(i in 1:n_population){
    fitnesses[i] <- fitness(population[[i]], fitness_function, board_size)
  }
  n_attacking_queens <- c()
  solutions <- list()
  ###### Genetic Algorithm ######
  for(generation in 1:n_generations){
    #Two individuals are randomly sampled from the current population, they are further used as parents
    parents <- sample(population, 2, replace = FALSE)

    #One individual with the smallest fitness is selected from the current population, this will be the
    victim <- population[which.min(fitnesses)][[1]]
    victim_idx <- which.min(fitnesses)

    #The two sampled parents are to produce a kid by crossover
    kid <- crossover(parents[[1]], parents[[2]], p=crossover_p)
    #this kid should be mutated with probability mutprob
    if(runif(1) < mutprob){
      kid <- mutate(kid, board_size)
    }
    #The victim is replaced by the kid in the population

    population[[victim_idx]] <- kid

    #Do not forget to update the vector of fitness values of the population
    fitnesses[victim_idx] <- fitness(kid, fitness_function, board_size)

  #Remember the number of pairs of queens attacking each other at the given iteration
    attacking_queens <- 0
    for(i in 1:n_population){
      attacking_queens <- attacking_queens + fitness(population[[i]], "num_attacking", board_size)
      if(fitness(population[[i]], "num_attacking", board_size) == 0){
        solutions[[length(solutions)+1]] <- population[[i]]
      }
    }
    n_attacking_queens <- c(n_attacking_queens, attacking_queens)
  }
  return(list(n_attacking_queens = n_attacking_queens, solutions = solutions))
}
```

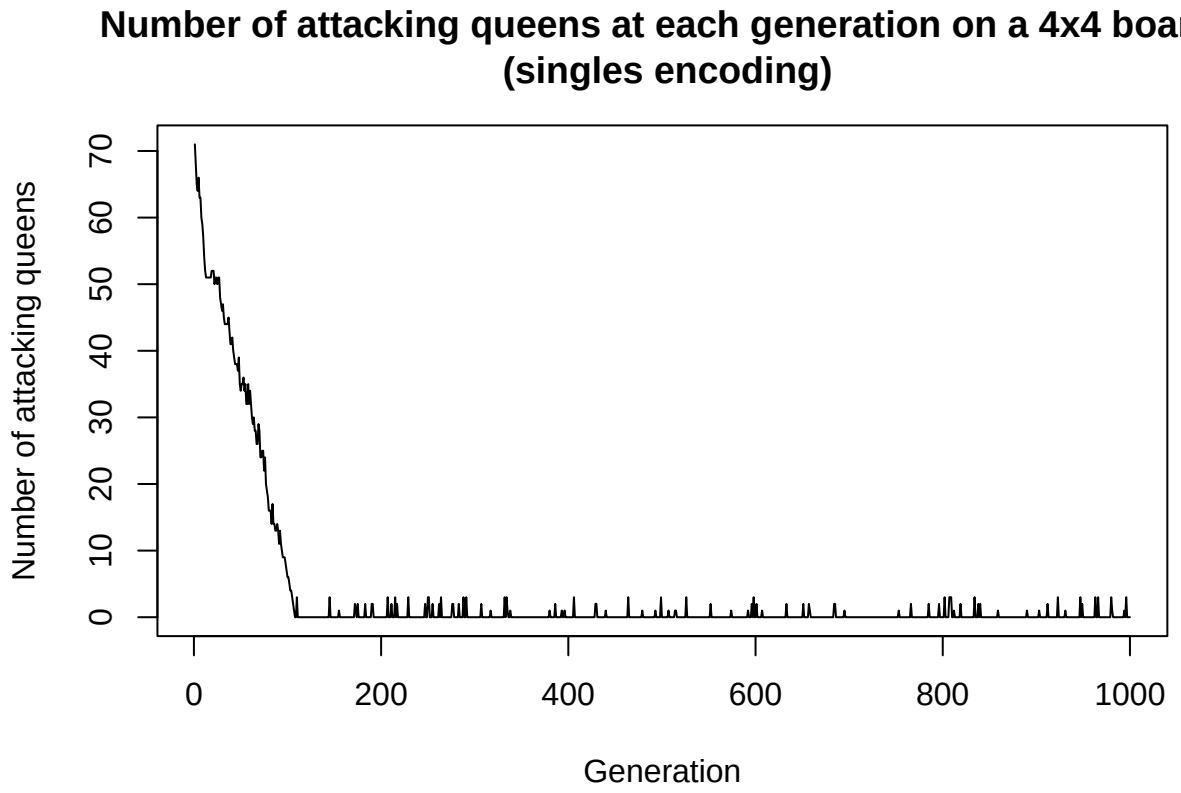
6.

If found, return the legal configuration of queens.

```
## found 18153 legal configurations

## Example of a legal configuration :

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

7.

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



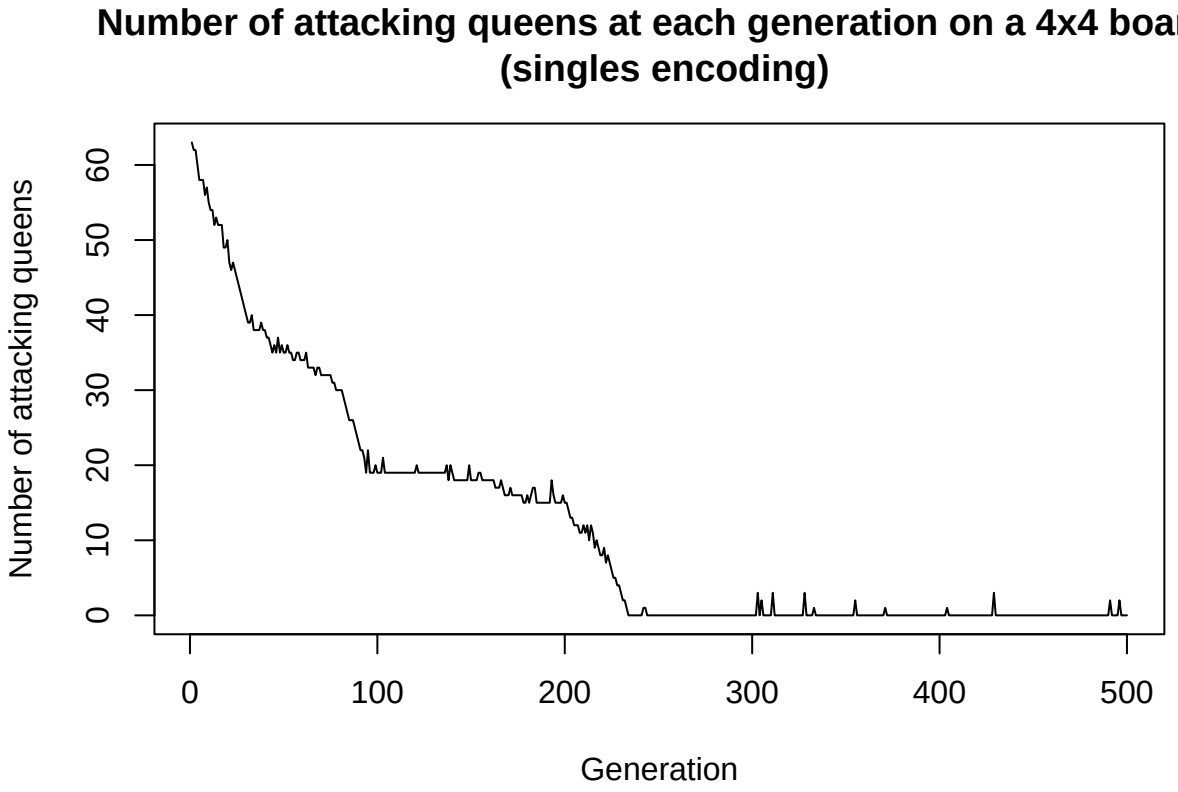
8.

Run your code for n = 4, 8, 16 , the different encodings, objective functions, and mutprob= 0.1, 0.5, 0.9. Did you find a legal state?

Testing Different board sizes :

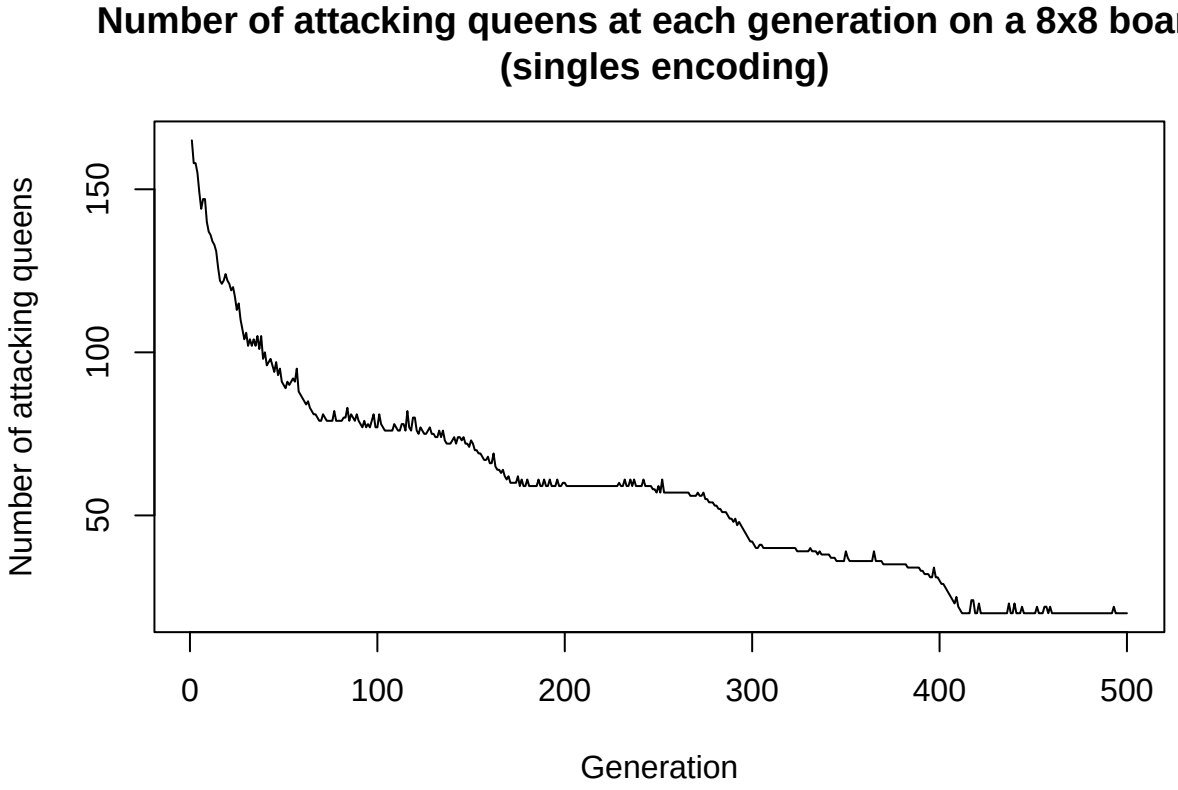
```
## N = 4 : found 5971 legal configurations
```

```
## time taken: 0.38592 seconds
```

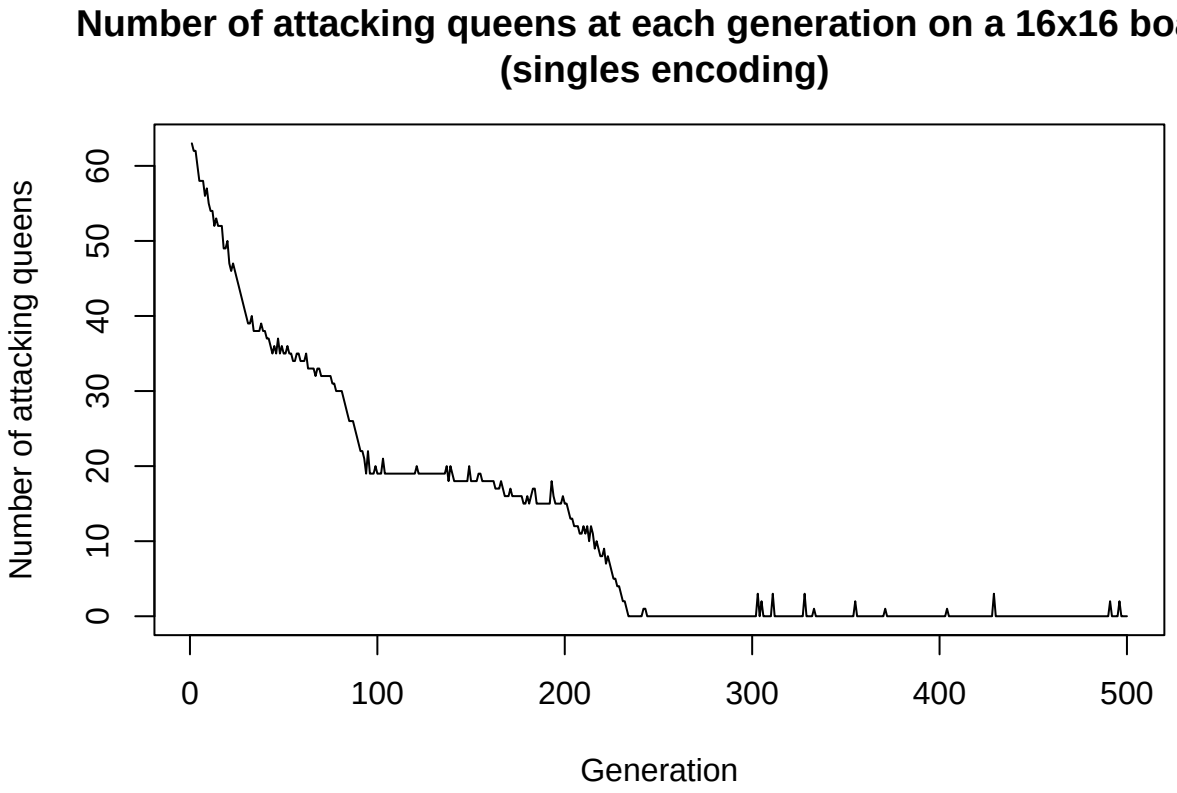



```
## N = 8 : found 0 legal configurations
```

```
## time taken: 0.9965189 seconds
```

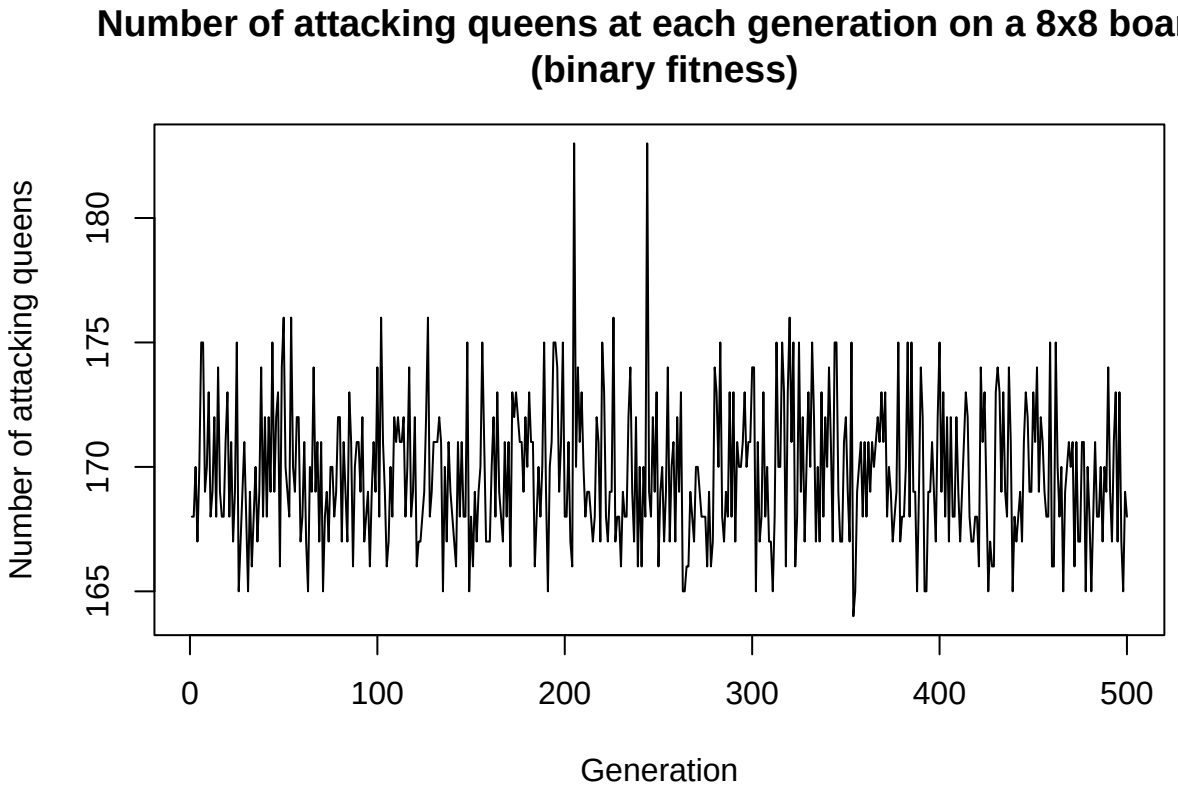



```
## N = 16 : found 0 legal configurations

## time taken: 3.333057 seconds
```

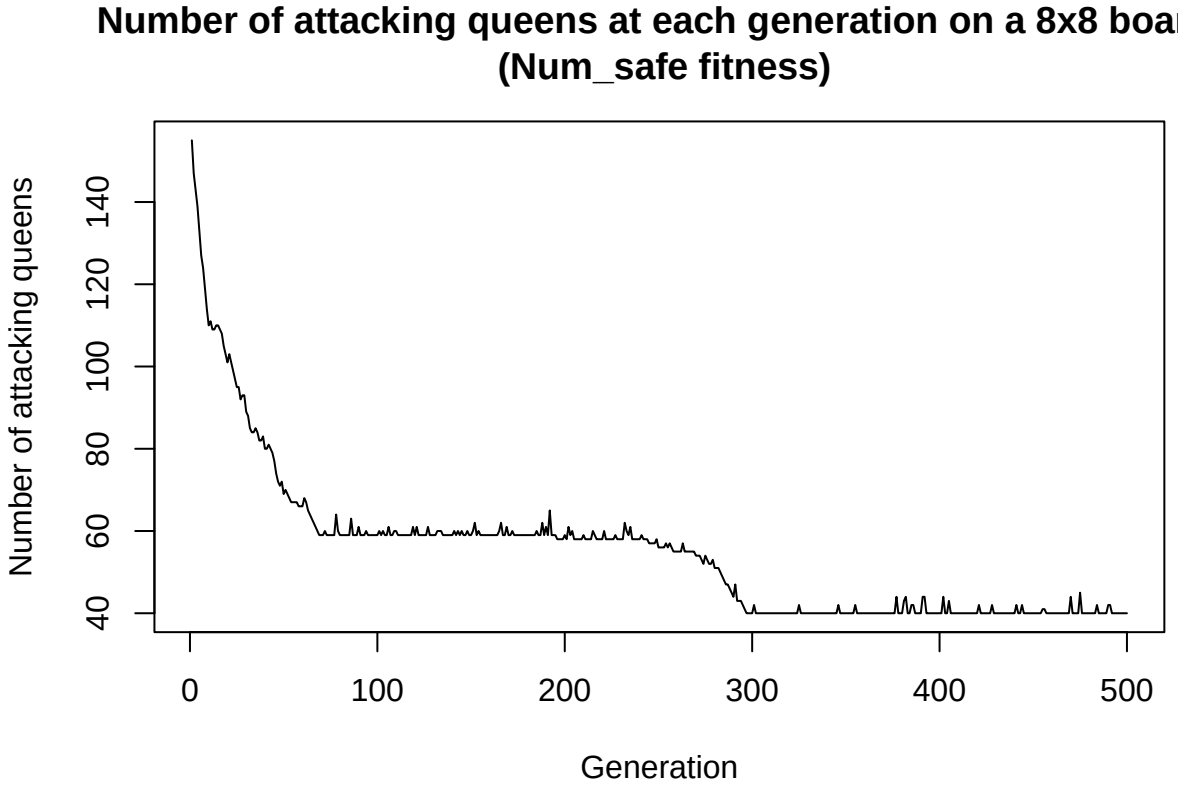


Testing Different fitness functions :

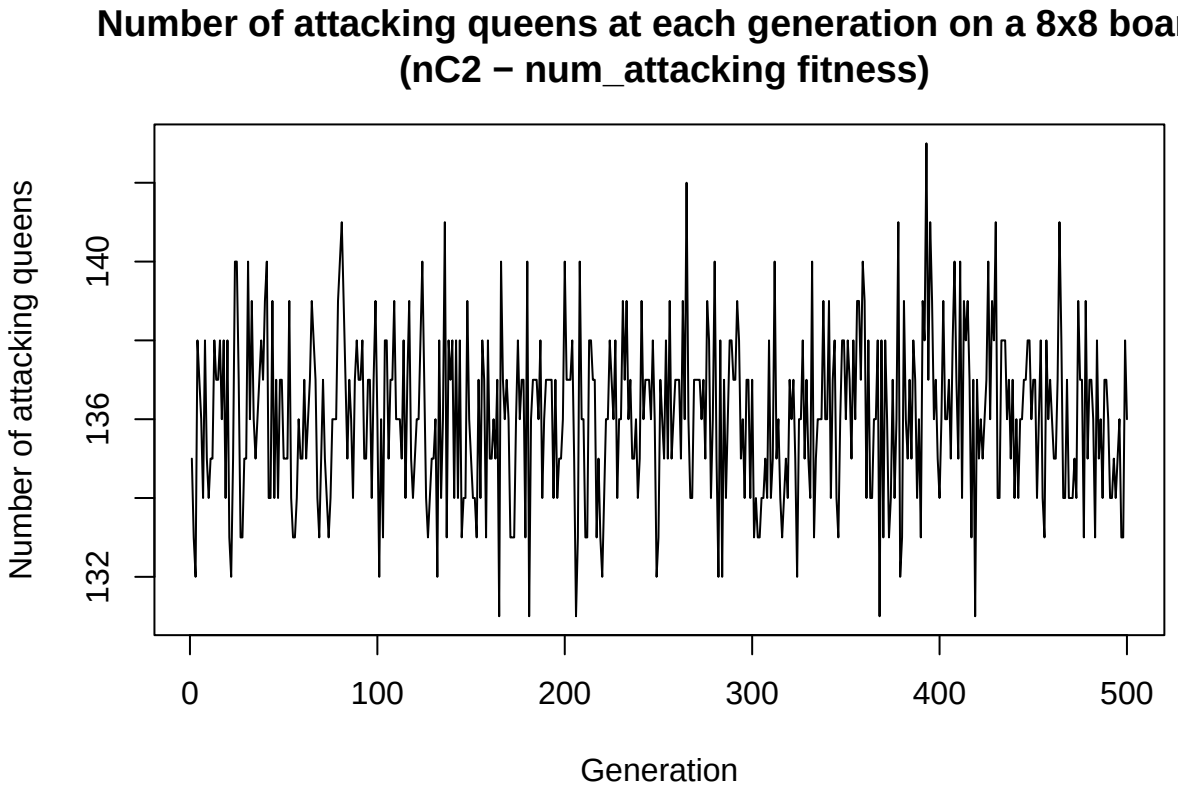
```
## Binary: found 0 legal configurations

## time taken: 0.9566491 seconds
```




```
## Num_safe : found 0 legal configurations

## time taken: 0.9103661 seconds
```

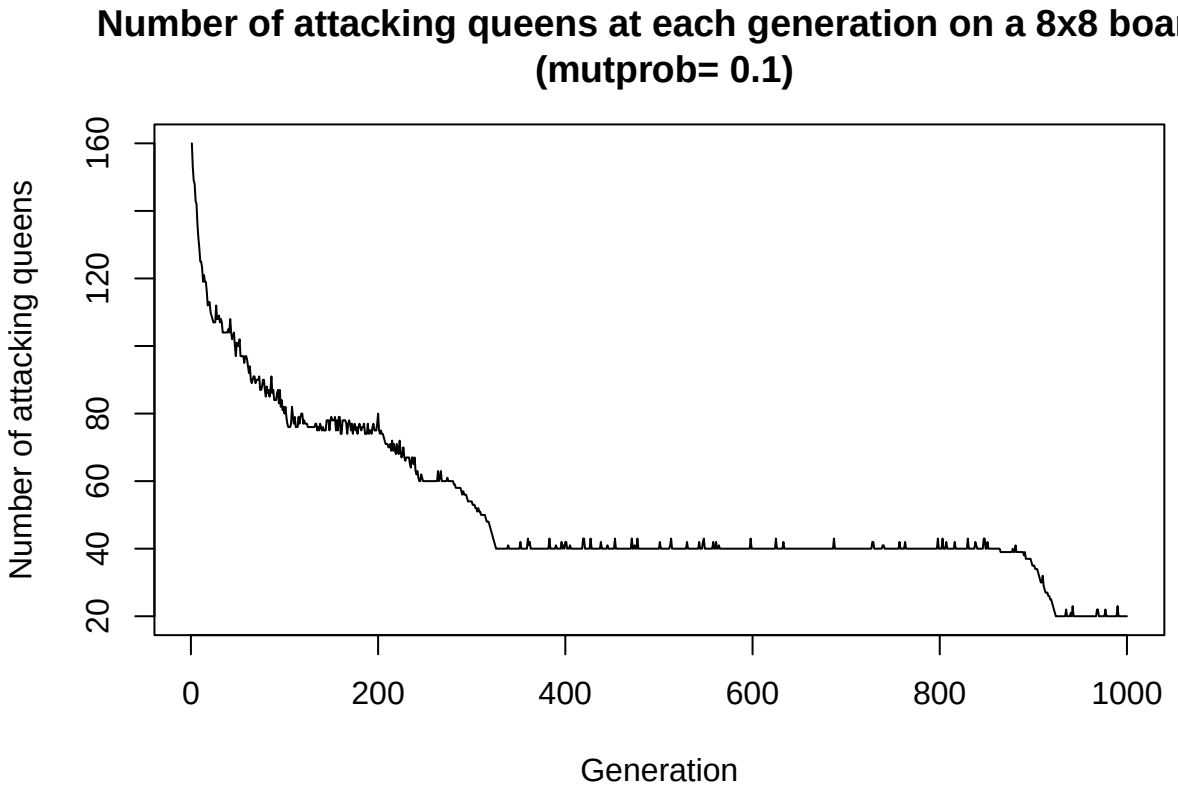



```
## nC2 - num_attacking : found 0 legal configurations

## time taken: 0.9128129 seconds
```

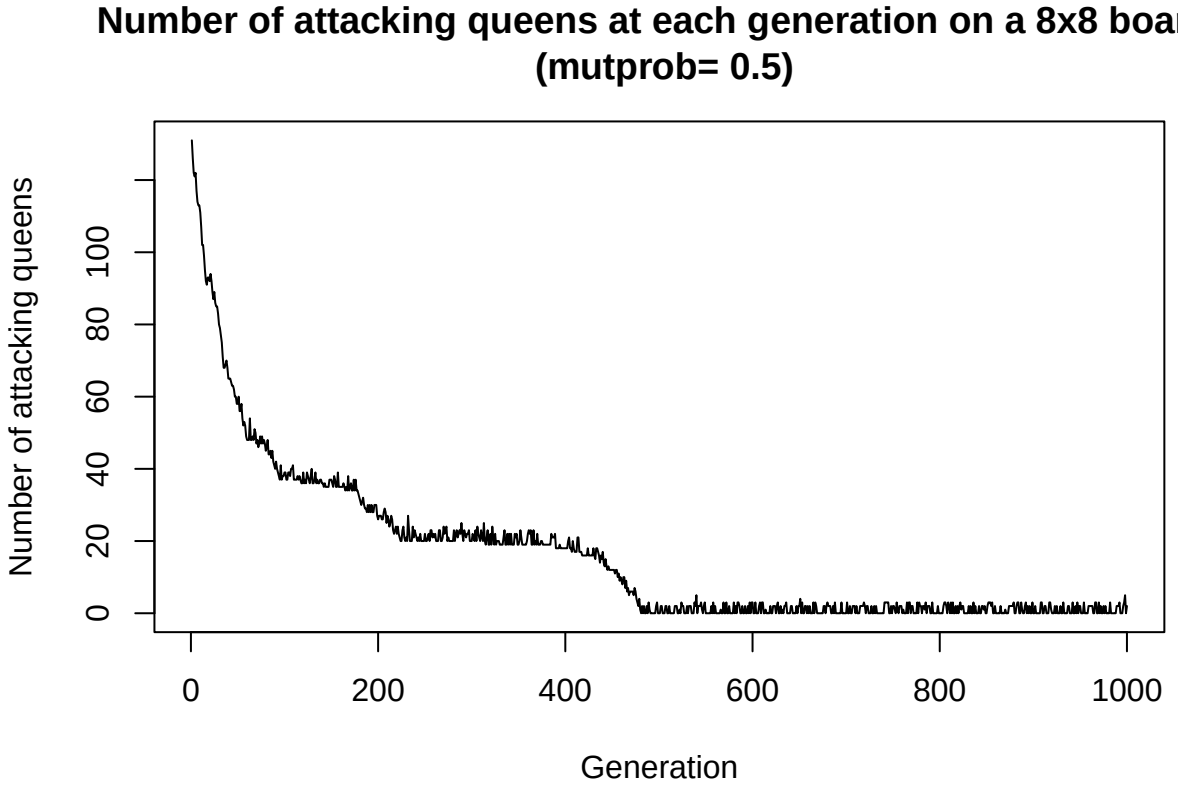


Testing Different mutation probabilities :

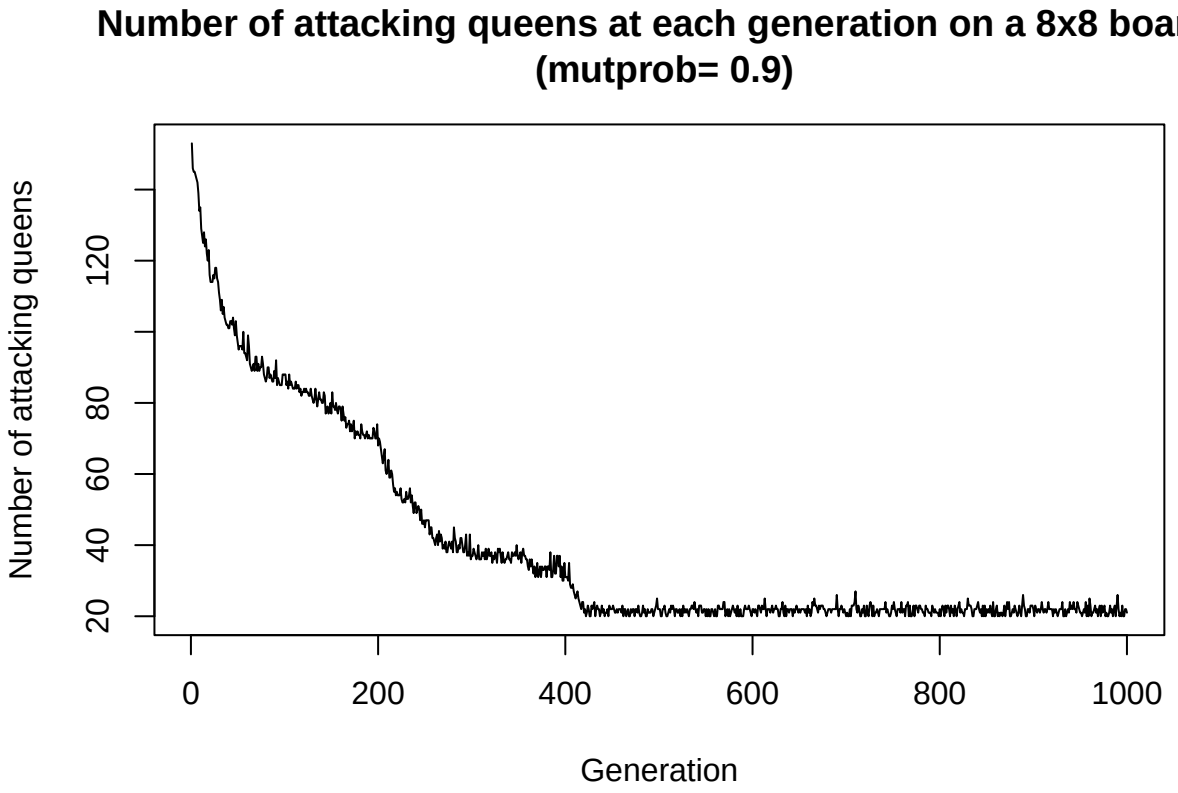
```
## mutprob= 0.1: found 0 legal configurations

## time taken: 1.910705 seconds
```




```
## mutprob= 0.5: found 10885 legal configurations

## time taken: 1.80575 seconds
```




```
## mutprob= 0.9: found 0 legal configurations

## time taken: 1.844177 seconds
```



The mutation probability seems the affect the "smoothness" of the graph. This is probably due to the fact that improvement is made when a new (good) layout is introduced into the population, and this is more likely to happen when the mutation probability is higher.

9.

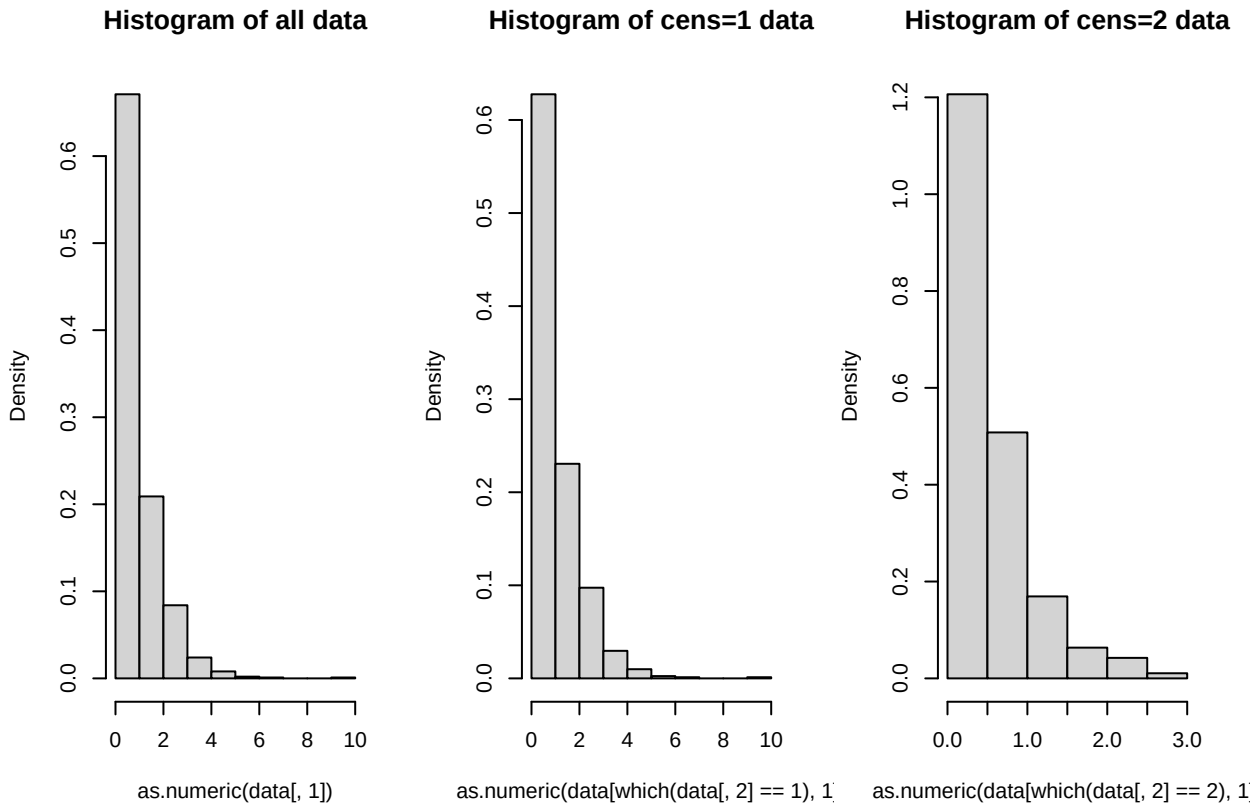
Discuss which encoding and objective function worked best.

The way we coded this assignment, the singles encoding seems to be working the best as it shows the most consistent improvement over the generations. The binary encoding, although computationally friendlier, shows a more erratic behavior. The pairs encoding is flawed by nature because it introduces unwanted behaviors. The number of safe queens fitness function seems to be the best one in our case. The binary fitness function is not good at all because it does not give any information about the quality of a solution, which is necessary for the genetic algorithm to perform well. The nC2 - num_attacking fitness function did not seem to work well, but it could be due to a design problem in our fitness function.

Question 2: EM algorithm

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?




```
## [1] "min"
```

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

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

```
## [1] "max"
```

```
## [1] 9.557361
```

```
## [1] 2.55981
```

Yes, they both look like exponential distributions.

2.

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

The Exponential Distribution Function has the following density function:

$$f(x, \lambda) = \lambda e^{-\lambda x}$$

for $x \geq 0$ and $\lambda > 0$.

and has the following cumulative distribution function:

$$F(x, \lambda) = P(X \leq x) = 1 - e^{-\lambda x}$$

for $x \geq 0$ and $\lambda > 0$.

The Truncated Exponential Distribution Function truncated on the left at c has the following density function:

$$f(y|c, \lambda) = \frac{\lambda e^{-\lambda y}}{1 - \lambda e^{-\lambda c}}$$

for $\lambda > 0$ and $0 < y \leq c$. (Source Al-Athari 2008)

To find the Cumulative distribution function, we use the following property derived in class:

$$F_{y \leq c}(Y) = P(Y \leq y | X \leq c) = \frac{P(Y \leq y, Y \leq c)}{P(Y \leq c)} = \frac{P(Y \leq y)}{P(Y \leq c)}$$

Therefore the Truncated Exponential Distribution Function truncated on the left at c has the following cumulative distribution function:

$$F_{y \leq c}(Y) = P(Y \leq y | Y \leq c) = \frac{1 - e^{-\lambda y}}{1 - e^{-\lambda c}}$$

Since $F_{XY}(x, y) = xy$, the combined CDF of the exponential distribution and the truncated exponential distribution is:

$$F_{x, y \leq c}(X, Y \leq c, \lambda) = P(X \leq x) * P(Y \leq y | Y \leq c) = (1 - e^{-\lambda x}) * \frac{1 - e^{-\lambda y}}{1 - e^{-\lambda c}}$$

It follows that the likelihood function is:

$$L_{x, y \leq c}(\lambda) = \prod_{i=1}^{n} \lambda e^{-\lambda x_i} * \prod_{j=1}^{m} \frac{\lambda e^{-\lambda y_j}}{1 - e^{-\lambda c_j}}$$

and the log-likelihood function is:

$$logL_{x, y \leq c}(\lambda) = \sum_{i=1}^{n} log(\lambda e^{-\lambda x_i}) + \sum_{j=1}^{m} log(\frac{\lambda e^{-\lambda y_j}}{1 - e^{-\lambda c_j}})$$

$$= n \log \lambda - \lambda \sum_{i=1}^{n} x_i + m \log \lambda - \lambda \sum_{j=1}^{m} y_j - \sum_{j=1}^{m} \log(1 - e^{-\lambda c_j})$$

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.

The formula for the Q function in the E-step is:

$$Q(\lambda, \lambda^k) = E[logL_{x, y \leq c}(\lambda | X, Y) | \lambda^k, X]$$

$$= E[n \log \lambda - \lambda \sum_{i=1}^{n} x_i + m \log \lambda - \lambda \sum_{j=1}^{m} y_j - \sum_{j=1}^{m} \log(1 - e^{-\lambda c_j})]$$

$$= n\log\lambda - \lambda E[\sum_{i=1}^{n} x_i] + m\log\lambda - \lambda E[\sum_{j=1}^{m} y_j] - \sum_{j=1}^{m}\log(1-e^{-\lambda c_j})$$

.

Since $E[\sum_{i=1}^{n} x_i] = n\bar{x}$ for the exponential distribution and $E[\sum_{j=1}^{m} y_j] = m*(\frac{1}{\lambda} - \frac{c}{e^{\lambda c}-1})$ for the truncated exponential distribution (Source : Al-Athari 2008); we obtain the final equation :

$$Q(\lambda,\lambda^k) = n\log\lambda - \lambda n\bar{x} + m\log\lambda - \lambda\sum_{j=1}^{m}(\frac{1}{\lambda^{(k)}} - \frac{c_j}{e^{\lambda^{(k)}c_j}-1}) - \sum_{j=1}^{m}\log(1-e^{-\lambda c_j})$$

where λ^k is the current estimate of λ at the iteration k of the EM algorithm.

4.

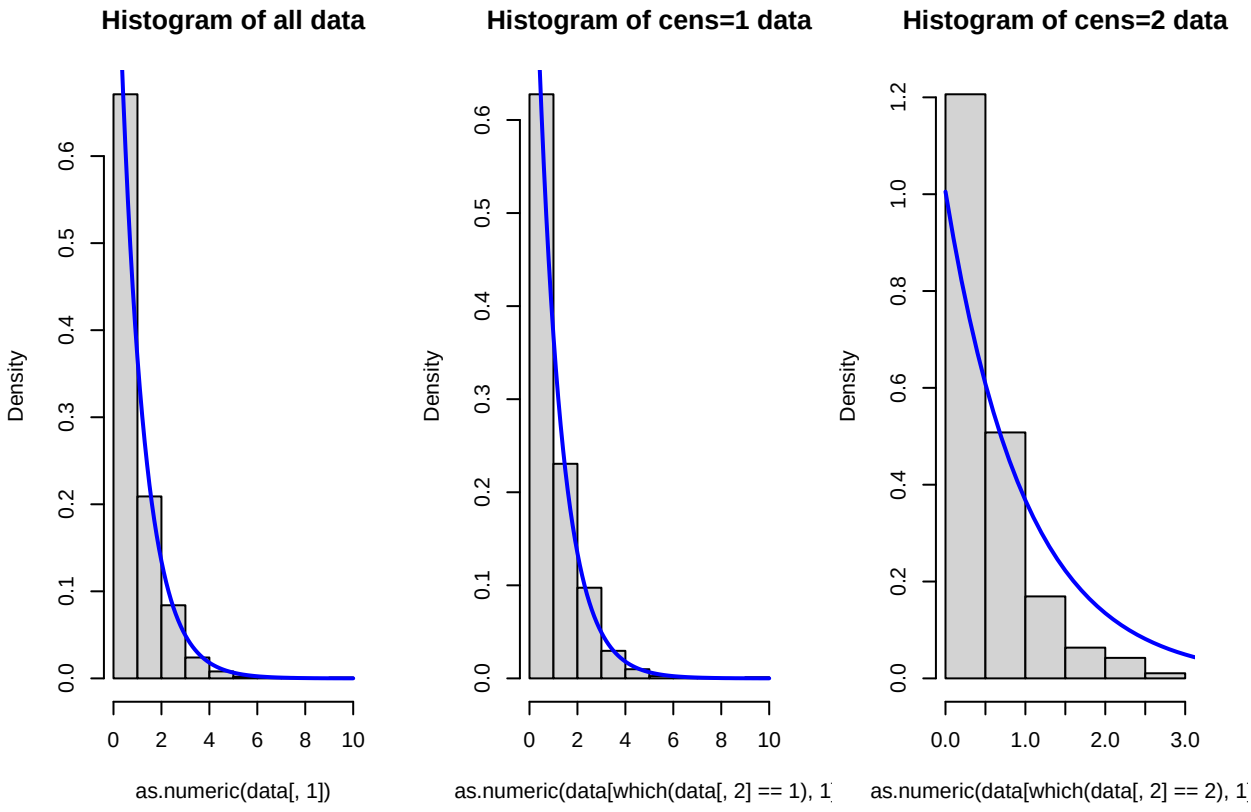
Implement the above in R. Take $\lambda_0 = 100$ as the starting value for the algorithm and stopping condition if the change in the estimate is less than 0.001. At what $\hat{\lambda}$ did the EM algorithm stop at? How many iterations were required?

```
## [1] "EM algorithm stopped at iteration 3"
```

```
## [1] 100.000000   1.055640   1.005230   1.004765
```

5.

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



6.

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

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

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

Appendix

```
knitr::opts_chunk$set(echo = FALSE, cache = TRUE)
BinToDec <- function(x){
  #https://stackoverflow.com/questions/12892348/convert-binary-string-to-binary-or-decimal-value
  sum(2^(which(rev(unlist(strsplit(as.character(x), ""))) == 1))-1))
}

vizualize_board <- function(layout, n){
  #Given a layout, vizualizes the board
  if(typeof(layout) == "list" & is.null(ncol(layout))){
    #print("Binary")
    #Binary
    board <- matrix(nrow = n, ncol = n)
    #fill the board with 0s
    for (i in 1:nrow(board)){
      for (j in 1:ncol(board)){
        board[i,j] <- 0
      }
    }
    for (i in 1:n){
      board[i,BinToDec(layout[[i]])] <- 1
    }
    board <- t(board)
    #print(board)
  }else if(typeof(layout) == "integer" & is.null(ncol(layout))){
    #print("Singles")
    #Singles
    #fill the board with 0s
    board <- matrix(nrow = n, ncol = n)
    for (i in 1:n){
      for (j in 1:n){
        board[i,j] <- 0
      }
```

```
    }
    for (i in 1:n){
      board[i,layout[i]] <- 1
    }
    board <- t(board)
    #print(board)
  }else{
    #Pairs
    #print("Pairs")
    board <- matrix(nrow = n, ncol = n)
    #fill the board with 0s
    for (i in 1:nrow(board)){
      for (j in 1:ncol(board)){
        board[i,j] <- 0
      }
    }
    for (i in 1:nrow(layout)){
      board[layout[i,1], layout[i,2]] <- 1
    }
  }
  #Print the board in the console:
  for (i in 1:(nrow(board)*2)){
    cat("_")
  }
  cat("\n")
  for (i in 1:nrow(board)){
    cat("|")
    for (j in 1:ncol(board)){
      if(board[i,j] == 0){
        cat(" |")
      }else{
        cat("Q|")
      }
    }
    cat("\n")
  }
}
#################### PAIRS REPRESENTATION ####################

# Question 2 : Crossover

crossover <- function(layout1, layout2, p){
  #PAIRS (matrix)
  kid <- matrix(nrow = nrow(layout1), ncol = 2)
  for (i in 1:p){
    kid[i,] <- layout1[i,]
  }
  for (i in (p+1):nrow(layout2)){
    kid[i,] <- layout2[i,]
  }
  return(kid)
}
```

```r
# Question 3 : Mutate

mutate <- function(layout,n){
  pair <- sample(n,1)
  layout[pair,1] <- sample(n,1)
  layout[pair,2] <- sample(n,1)
  #Disadvantages of this layout : can move to a location where there is already a queen
  # --> Good or bad ? Good: simulate a queen "disappearing", bad: duplicates in the representation
  return(layout)
}

# Question 4 : Fitness

fitness <- function(layout, fitness_function, n){
  board <- matrix(nrow = n, ncol = n)
  #fill the board with 0s
  for (i in 1:n){
    for (j in 1:n){
      board[i,j] <- 0
    }
  }
  #fill the board with 1s where there are queens
  for (i in 1:n){
    board[layout[i,1], layout[i,2]] <- 1
  }
  #==============================================================================
 #Convert the board to a list of coordinates
  my_vec <- c()
  for(i in 1:n){
    for(j in 1:n){
      if(board[i,j]==1){
        my_vec <- c(my_vec, c(i,j))
      }
    }
  }
  if(length(my_vec) == 0){
    return(0)
  }
  points<-array( my_vec, dim=c(2,length(my_vec)/2))
  are_attacking_eachother <- function(Q1,Q2){
    #https://stackoverflow.com/questions/57239548/how-to-check-if-a-queen-is-under-attack-in-nqueens
    (Q1[1] == Q2[1] | Q1[2] == Q2[2] | Q1[1]-Q1[2] == Q2[1]-Q2[2] | Q1[1]+Q1[2] == Q2[1]+Q2[2])
  }
  fitness <- 0
  if(length(points)/2 == 1){
    if(fitness_function == "binary"){
      return(0)
    }else if(fitness_function == "num_safe"){
      return(1)
    }else{
      #number of attacking pairs
      return(0)
    }
```

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

```r
# Question 5 : Genetic Algorithm

generate_random_layout_pairs <- function(n){
  pairs<-array(0, dim=c(n,2))
  for (i in 1:n){
    pairs[i,1] <- sample(n,1)
    pairs[i,2] <- sample(n,1)
  }
  return(pairs)
}

darwin_pairs <- function(n_generations, board_size, n_population, mutprob, fitness_function, crossover_
  ###### Initialization ######
  #generate random population
  population <- list()
  for(i in 1:n_population){
    population[[i]] <- generate_random_layout_pairs(board_size)
  }

  #calculate fitness of population
  fitnesses <- c()
  for(i in 1:n_population){
    fitnesses[i] <- fitness(population[[i]], fitness_function, board_size)
  }
  n_attacking_queens <- c()
  solutions <- list()
  ###### Genetic Algorithm ######
  for(generation in 1:n_generations){

    #Two individuals are randomly sampled from the current population,
    # they are further used as parents
    parents <- sample(population, 2, replace = FALSE)

    #One individual with the smallest fitness is selected from the current population,
    # this will be the victim
    victim <- population[which.min(fitnesses)]
    victim_idx <- which.min(fitnesses)
    #The two sampled parents are to produce a kid by crossover
    kid <- crossover(parents[[1]], parents[[2]], p=crossover_p)

    #this kid should be mutated with probability mutprob
    if(runif(1) < mutprob){
      kid <- mutate(kid, board_size)
    }

    #The victim is replaced by the kid in the population
    population[[victim_idx]] <- kid

    #Do not forget to update the vector of fitness values of the population
    fitnesses[victim_idx] <- fitness(kid, fitness_function, board_size)

    #Remember the number of pairs of queens attacking each other at the given iteration
    attacking_queens <- 0
```

```
    for(i in 1:n_population){
      attacking_queens <- attacking_queens + fitness(population[[i]], "num_attacking", board_size)
      if(fitness(population[[i]], "num_attacking", board_size) == 0){
        solutions[[length(solutions)+1]] <- population[[i]]
      }
    }
    n_attacking_queens <- c(n_attacking_queens, attacking_queens)

  }
  return(list(n_attacking_queens = n_attacking_queens, solutions = solutions))
}

# Question 6 : Legal Configuration
res_pairs <- darwin_pairs(n_generations = 1000,
                          board_size =  4,
                          n_population = 20,
                          mutprob = 0.1,
                          fitness_function = "num_safe",
                          crossover_p = 2)

cat("found", length(res_pairs$solutions), "legal configurations \n")
cat("Example of a legal configuration : \n")
if(length(res_pairs$solutions) > 0){
  vizualize_board(res_pairs$solutions[[1]], 4)
}
# Question 7 : Plot
plot(res_pairs$n_attacking_queens, type="l", xlab="Generation", ylab="Number of attacking queens")
title("Number of attacking queens at each generation on a 4x4 board \n (pairs encoding)")

################### Binary Representation ###################

# Question 2 : Crossover

crossover <- function(layout1, layout2, p){
  #p < log2(n)/2
  kid <- list()
  for (i in 1:p){
    kid[[i]] <- layout1[[i]]
  }
  for (i in (p+1):length(layout2)){
    kid[[i]] <- layout2[[i]]
  }
  return(kid)
}

# Question 3 : Mutate

mutate <- function(layout,n){
    #randomly select a binary string
    binary_string <- sample(1:length(layout), 1)
    #randomly select a bit in the binary string
    bit <- sample(1:length(layout[[binary_string]]), 1)
    #flip the bit
```

```
    layout[[binary_string]][bit] <- abs(layout[[binary_string]][bit] - 1)
    return(layout)
}

# Question 4 : Fitness

fitness <- function(layout, fitness_function, n){
  #Very twisted and inefficient way of doing things, to be modified
  board <- matrix(nrow = n, ncol = n)
  #fill the board with 0s
  for (i in 1:nrow(board)){
    for (j in 1:ncol(board)){
      board[i,j] <- 0
    }
  }
  for (i in 1:length(layout)){
    board[i, BinToDec(layout[[i]])] <- 1
  }
  board <- t(board)

  #============================================================================
  #At this point, the board var should contain a matrix on the state of the board, with 1s for queens a
  #Convert the board to a list of coordinates
  my_vec <- c()
  for(i in 1:n){
    for(j in 1:n){
      if(board[i,j]==1){
        my_vec <- c(my_vec, c(i,j))
      }
    }
  }
  if(length(my_vec) == 0){
    return(0)
  }
  points<-array( my_vec, dim=c(2,length(my_vec)/2))
  are_attacking_eachother <- function(Q1,Q2){
    #https://stackoverflow.com/questions/57239548/how-to-check-if-a-queen-is-under-attack-in-nqueens
    (Q1[1] == Q2[1] | Q1[2] == Q2[2] | Q1[1]-Q1[2] == Q2[1]-Q2[2] | Q1[1]+Q1[2] == Q2[1]+Q2[2])
  }
  fitness <- 0
  #If only one queen on the board:
  if(length(points)/2 <= 1){
    if(fitness_function == "binary"){
      return(0)
    }else if(fitness_function == "num_safe"){
      return(length(points)/2)
    }else{ #number of attacking pairs
      return(0)
    }

  }else if(fitness_function == "binary"){
      for(queen in 1:((length(points)/2)-1)){
        for(other_queen in (queen+1):(length(points)/2)){
```

```
          Q1 <- c(points[1,queen], points[2,queen])
          Q2 <- c(points[1,other_queen], points[2,other_queen])
          if(are_attacking_eachother(Q1, Q2)){
            fitness <- 1
          }
        }
      }
  }else if(fitness_function == "num_safe"){
    safe <- 0
     for(queen in 1:((length(points)/2)-1)){
        for(other_queen in (queen+1):(length(points)/2)){
          Q1 <- c(points[1,queen], points[2,queen])
          Q2 <- c(points[1,other_queen], points[2,other_queen])
          if(!are_attacking_eachother(Q1, Q2)){
            safe <- safe + 2
          }
        }
     }
    #to scale to [0,1], we can divide by the max number of queens that can be safe
    fitness <- safe

  }else if(fitness_function == "num_attacking"){
    num_attacking <- 0
    for(queen in 1:((length(points)/2)-1)){
      for(other_queen in (queen+1):(length(points)/2)){
        Q1 <- c(points[1,queen], points[2,queen])
        Q2 <- c(points[1,other_queen], points[2,other_queen])
        if(are_attacking_eachother(Q1, Q2)){
          num_attacking <- num_attacking + 1
        }
      }
    }
    fitness <- num_attacking
  }else{
      #(nC2 - number) of pairs of queens attacking each other.
      nc2 <- (n*(n-1)/2)
      num_attacking <- 0
      for(queen in 1:((length(points)/2)-1)){
        for(other_queen in (queen+1):(length(points)/2)){
          Q1 <- c(points[1,queen], points[2,queen])
          Q2 <- c(points[1,other_queen], points[2,other_queen])
          if(are_attacking_eachother(Q1, Q2)){
            fitness <- 1
          }
        }
      }
      fitness <- nc2 - num_attacking
  }
  return(fitness)
}

# Question 5 : Genetic Algorithm
```

```r
generate_random_layout_binary <- function(n){
  #Only works for n powers of 2, otherwise it can produces queens outside of the board
  layout <- list()
  for (i in 1:n){
    layout[[i]] <- sample(0:1, ceiling(log2(n)), replace = TRUE)
  }
  return(layout)
}

darwin_bin <- function(n_generations, board_size, n_population, mutprob, fitness_function, crossover_p)

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

  #calculate fitness of population
  fitnesses <- c()
  for(i in 1:n_population){
    fitnesses[i] <- fitness(population[[i]], fitness_function, board_size)
  }
  n_attacking_queens <- c()
  solutions <- list()
  ###### Genetic Algorithm ######
  for(generation in 1:n_generations){
    #Two individuals are randomly sampled from the current population, they are further used as parents
    parents <- sample(population, 2, replace = FALSE)

    #One individual with the smallest fitness is selected from the current population, this will be the
    victim <- population[which.min(fitnesses)][[1]]
    victim_idx <- which.min(fitnesses)

    #The two sampled parents are to produce a kid by crossover
    kid <- crossover(parents[[1]], parents[[2]], p=crossover_p)
    #this kid should be mutated with probability mutprob
    if(runif(1) < mutprob){
      kid <- mutate(kid, board_size)
    }
    #The victim is replaced by the kid in the population

    population[[victim_idx]] <- kid

    #Do not forget to update the vector of fitness values of the population
    fitnesses[victim_idx] <- fitness(kid, fitness_function, board_size)

  #Remember the number of pairs of queens attacking each other at the given iteration
    attacking_queens <- 0
    for(i in 1:n_population){
      attacking_queens <- attacking_queens + fitness(population[[i]], "num_attacking", board_size)
      if(fitness(population[[i]], "num_attacking", board_size) == 0){
        solutions[[length(solutions)+1]] <- population[[i]]
      }
```

```
    }
    n_attacking_queens <- c(n_attacking_queens, attacking_queens)

  }
  return(list(n_attacking_queens = n_attacking_queens, solutions = solutions))
}

# Question 6 : Legal Configuration
res_bin <- darwin_bin(n_generations = 1000,
                          board_size =  4,
                          n_population = 20,
                          mutprob = 0.1,
                          fitness_function = "num_safe",
                          crossover_p = 1) #Bin rep so 2bits only

cat("found", length(res_bin$solutions), "legal configurations \n")
cat("Example of a legal configuration : \n")
if(length(res_bin$solutions) > 0){
  vizualize_board(res_bin$solutions[[1]], 4)
}
# Question 7 : Plot
plot(res_bin$n_attacking_queens, type="l", xlab="Generation", ylab="Number of attacking queens")
title("Number of attacking queens at each generation on a 4x4 board \n (binary encoding)")

#################### Singles Representation ###################

# Question 2 : Crossover

crossover <- function(layout1, layout2, p){
  kid <- c()
    for (i in 1:p){
      kid <- c(kid , layout1[i])
    }
    for (i in (p+1):length(layout2)){
      kid <- c(kid , layout2[i])
    }
    return(kid)
}

# Question 3 : Mutate

mutate <- function(layout,n){
    row <- sample(n,1)
    layout[row] <- sample(n,1)
    #Disadvantages of this method : a queen can only be moved within a column, and it can also not be m
    return(layout)
}

# Question 4 : Fitness

fitness <- function(layout, fitness_function, n){
  #Very twisted and inefficient way of doing things, to be modified
  board <- matrix(nrow = n, ncol = n)
```

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

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

# Question 5 : Genetic Algorithm

generate_random_layout_singles <- function(n){
  layout <- c()
  for (i in 1:n){
    layout[i] <- sample(1:n,1)
  }
  return(layout)
}

darwin_singles <- function(n_generations, board_size, n_population, mutprob, fitness_function, crossove

  #generate random population
  population <- list()
```

```
  for(i in 1:n_population){
    population[[i]] <- generate_random_layout_singles(board_size)
  }

  #calculate fitness of population
  fitnesses <- c()
  for(i in 1:n_population){
    fitnesses[i] <- fitness(population[[i]], fitness_function, board_size)
  }
  n_attacking_queens <- c()
  solutions <- list()
  ###### Genetic Algorithm ######
  for(generation in 1:n_generations){
    #Two individuals are randomly sampled from the current population, they are further used as parents
    parents <- sample(population, 2, replace = FALSE)

    #One individual with the smallest fitness is selected from the current population, this will be the
    victim <- population[which.min(fitnesses)][[1]]
    victim_idx <- which.min(fitnesses)

    #The two sampled parents are to produce a kid by crossover
    kid <- crossover(parents[[1]], parents[[2]], p=crossover_p)
    #this kid should be mutated with probability mutprob
    if(runif(1) < mutprob){
      kid <- mutate(kid, board_size)
    }
    #The victim is replaced by the kid in the population

    population[[victim_idx]] <- kid

    #Do not forget to update the vector of fitness values of the population
    fitnesses[victim_idx] <- fitness(kid, fitness_function, board_size)

  #Remember the number of pairs of queens attacking each other at the given iteration
    attacking_queens <- 0
    for(i in 1:n_population){
      attacking_queens <- attacking_queens + fitness(population[[i]], "num_attacking", board_size)
      if(fitness(population[[i]], "num_attacking", board_size) == 0){
        solutions[[length(solutions)+1]] <- population[[i]]
      }
    }
    n_attacking_queens <- c(n_attacking_queens, attacking_queens)
  }
  return(list(n_attacking_queens = n_attacking_queens, solutions = solutions))
}

# Question 6 : Legal Configuration
res_sin <- darwin_singles(n_generations = 1000,
                          board_size = 4,
                          n_population = 20,
                          mutprob = 0.1,
                          fitness_function ="num_safe",
                          crossover_p = 2)
```

```r
cat("found", length(res_sin$solutions), "legal configurations \n")
cat("Example of a legal configuration : \n")
if(length(res_sin$solutions) > 0){
  vizualize_board(res_sin$solutions[[1]], 4)
}
# Question 7 : Plot
plot(res_sin$n_attacking_queens, type = "l", xlab = "Generation", ylab = "Number of attacking queens")
title("Number of attacking queens at each generation on a 4x4 board \n (singles encoding)")

# n = 4
start_time <- Sys.time()
res_sin4 <- darwin_singles(n_generations = 500,
                           board_size = 4,
                           n_population = 20,
                           mutprob =0.1,
                           fitness_function = "num_safe",
                           crossover_p = 2)
end_time <- Sys.time()

cat("N = 4 : found", length(res_sin4$solutions), "legal configurations")
cat("\n")
cat("time taken:", end_time - start_time, "seconds")
cat("\n")
plot(res_sin4$n_attacking_queens, type = "l", xlab = "Generation", ylab = "Number of attacking queens")
title("Number of attacking queens at each generation on a 4x4 board \n (singles encoding)")

# n = 8
start_time <- Sys.time()
res_sin8 <- darwin_singles(n_generations = 500,
                           board_size = 8,
                           n_population = 20,
                           mutprob =0.1,
                           fitness_function = "num_safe",
                           crossover_p = 4)
end_time <- Sys.time()

cat("N = 8 : found", length(res_sin8$solutions), "legal configurations")
cat("\n")
cat("time taken:", end_time - start_time, "seconds")
cat("\n")
plot(res_sin8$n_attacking_queens, type = "l", xlab = "Generation", ylab = "Number of attacking queens")
title("Number of attacking queens at each generation on a 8x8 board \n (singles encoding)")

# n = 16
start_time <- Sys.time()
res_sin16 <- darwin_singles(n_generations = 500,
                           board_size = 16,
                           n_population = 20,
                           mutprob =0.1,
                           fitness_function = "num_safe",
                           crossover_p = 8)
end_time <- Sys.time()
```

```r
cat("N = 16 : found", length(res_sin16$solutions), "legal configurations")
cat("\n")
cat("time taken:", end_time - start_time, "seconds")
cat("\n")
plot(res_sin4$n_attacking_queens, type = "l", xlab = "Generation", ylab = "Number of attacking queens")
title("Number of attacking queens at each generation on a 16x16 board \n (singles encoding)")


# Binary
start_time <- Sys.time()
res_sinA <- darwin_singles(n_generations = 500,
                           board_size = 8,
                           n_population = 20,
                           mutprob =0.1,
                           fitness_function = "binary",
                           crossover_p = 4)
end_time <- Sys.time()

cat("Binary: found", length(res_sinA$solutions), "legal configurations")
cat("\n")
cat("time taken:", end_time - start_time, "seconds")
cat("\n")
plot(res_sinA$n_attacking_queens, type = "l", xlab = "Generation", ylab = "Number of attacking queens")
title("Number of attacking queens at each generation on a 8x8 board \n (binary fitness)")

# Number of Safe Queens
start_time <- Sys.time()
res_sinB <- darwin_singles(n_generations = 500,
                           board_size = 8,
                           n_population = 20,
                           mutprob =0.1,
                           fitness_function = "num_safe",
                           crossover_p = 4)
end_time <- Sys.time()

cat("Num_safe : found", length(res_sinB$solutions), "legal configurations")
cat("\n")
cat("time taken:", end_time - start_time, "seconds")
cat("\n")
plot(res_sinB$n_attacking_queens, type = "l", xlab = "Generation", ylab = "Number of attacking queens")
title("Number of attacking queens at each generation on a 8x8 board \n (Num_safe fitness)")

# n = 16
start_time <- Sys.time()
res_sinC <- darwin_singles(n_generations = 500,
                           board_size = 8,
                           n_population = 20,
                           mutprob =0.1,
                           fitness_function = "other",
                           crossover_p = 4)
end_time <- Sys.time()

cat("nC2 - num_attacking : found", length(res_sinC$solutions), "legal configurations")
```

```
cat("\n")
cat("time taken:", end_time - start_time, "seconds")
cat("\n")
plot(res_sinC$n_attacking_queens, type = "l", xlab = "Generation", ylab = "Number of attacking queens")
title("Number of attacking queens at each generation on a 8x8 board \n (nC2 - num_attacking fitness)")


# mutprob= 0.1
start_time <- Sys.time()
res_sinA <- darwin_singles(n_generations = 1000,
                           board_size = 8,
                           n_population = 20,
                           mutprob =0.1,
                           fitness_function = "num_safe",
                           crossover_p = 4)
end_time <- Sys.time()

cat("mutprob= 0.1: found", length(res_sinA$solutions), "legal configurations")
cat("\n")
cat("time taken:", end_time - start_time, "seconds")
cat("\n")
plot(res_sinA$n_attacking_queens, type = "l", xlab = "Generation", ylab = "Number of attacking queens")
title("Number of attacking queens at each generation on a 8x8 board \n (mutprob= 0.1)")

# mutprob= 0.5
start_time <- Sys.time()
res_sinB <- darwin_singles(n_generations = 1000,
                           board_size = 8,
                           n_population = 20,
                           mutprob= 0.5,
                           fitness_function = "num_safe",
                           crossover_p = 4)
end_time <- Sys.time()

cat("mutprob= 0.5: found", length(res_sinB$solutions), "legal configurations")
cat("\n")
cat("time taken:", end_time - start_time, "seconds")
cat("\n")
plot(res_sinB$n_attacking_queens, type = "l", xlab = "Generation", ylab = "Number of attacking queens")
title("Number of attacking queens at each generation on a 8x8 board \n (mutprob= 0.5)")

# mutprob= 0.9
start_time <- Sys.time()
res_sinC <- darwin_singles(n_generations = 1000,
                           board_size = 8,
                           n_population = 20,
                           mutprob= 0.9,
                           fitness_function = "num_safe",
                           crossover_p = 4)
end_time <- Sys.time()

cat("mutprob= 0.9: found", length(res_sinC$solutions), "legal configurations")
cat("\n")
```

```r
cat("time taken:", end_time - start_time, "seconds")
cat("\n")
plot(res_sinC$n_attacking_queens, type = "l", xlab = "Generation", ylab = "Number of attacking queens")
title("Number of attacking queens at each generation on a 8x8 board \n (mutprob= 0.9)")

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

par(mfrow=c(1,3))

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

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

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

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


# 2.4

# EM algorithm
get_Q <- function(lambda_k){
  #Given lambda and lambda_k, returns Q(lambda, lambda_k) to be optimised
  Q <- function(lambda){
    n*log(lambda) - lambda*n*mean(x) + m*log(lambda) -lambda*sum((1/lambda_k)-(c/(exp(lambda_k*c)-1)))
  }
  return(Q)
}

###################### Parameters ####################

# uncensored and censored number of observations
n <- length(which(data[,2]==1))
m <- length(which(data[,2]==2))

# uncensored and censored observations
x <- as.numeric(data[which(data[,2]==1),1]) #Uncensored observations
c <- as.numeric(data[which(data[,2]==2),1]) #Censored observations

# starting lambda value and stopping conditions
lambda_0 <- 100 # starting value for lambda_k
epsilon <- 0.001 # stopping condition
max_k <- 100 # maximum number of iterations
```

```r
# initialization of the while loop
k <- 1 # iteration counter
lambda_ks <- lambda_0 # vector of lambda values, starting with lambda_0

######################## EM Algorithm ##################################

while(k < max_k){
  # E-step
  Q <- get_Q(lambda_ks[k])
  # M-step
  lambda_k <- optimise(Q, interval = c(0,100), maximum = TRUE)$maximum
  lambda_ks <- c(lambda_ks, lambda_k)
  # stopping condition
  if(abs(lambda_k - lambda_ks[k]) < epsilon){
    print(paste("EM algorithm stopped at iteration", k))
    break
  }
  # update k
  k <- k +1
}

print(lambda_ks)
#2.5

lambda_graph <- lambda_ks[length(lambda_ks)]

par(mfrow=c(1,3))

hist(as.numeric(data[,1]), freq=F, main="Histogram of all data")
curve(dexp(x, rate = lambda_graph), from=0, to=10, col='blue', add=TRUE, lwd = 2)

hist(as.numeric(data[which(data[,2]==1),1]), freq=F, main="Histogram of cens=1 data")
curve(dexp(x, rate = lambda_graph), from=0, to=10, col='blue', add=TRUE, lwd = 2)

hist(as.numeric(data[which(data[,2]==2),1]), freq=F, main="Histogram of cens=2 data")
curve(dexp(x, rate = lambda_graph), from=0, to=10, col='blue', add=TRUE, lwd = 2)

#2.6

bootstrap <- matrix(NA, ncol = 2, nrow = 1000)
MLE_lambda_hat <- c()

n <- 1000
# MLE
# TO BE REWORKED
for (i in 1:n) {
  bootstrap[,1] <- rexp(n, lambda)
  bootstrap[,2] <- 1
  bootstrap[1:m,1] <- runif(m, 0, max(as.numeric(data[which(data[,2]==2),1])))
  bootstrap[1:m,2] <- 2
  MLE_lambda_hat[i] <- n/sum(bootstrap[(m+1):n,1])
}
```

```r
EM_lambda_hat  <- c()

for(i in 1:1000){

  #Bootstrap data:
  # Draw a 1000 samples from rexp with parameter lambda_hat

  #Sample from this data a number of samples = to the number of censored data un the original data (189

  #Censor them

  ###################### Parameters ######################
  # uncensored and censored number of observations
  n <- length(which(data[,2]==1))
  m <- length(which(data[,2]==2))

  # uncensored and censored observations
  x <- as.numeric(data[which(data[,2]==1),1]) #Uncensored observations
  c <- as.numeric(data[which(data[,2]==2),1]) #Censored observations

  # starting lambda value and stopping conditions
  lambda_0 <- 100 # starting value for lambda_k
  epsilon <- 0.001 # stopping condition
  max_k <- 100 # maximum number of iterations

  # initialization of the while loop
  k <- 1 # iteration counter
  lambda_ks <- lambda_0 # vector of lambda values, starting with lambda_0

  ######################## EM Algorithm ################################

  while(k < max_k){
    # E-step
    Q <- get_Q(lambda_ks[k])
    # M-step
    lambda_k <- optimise(Q, interval = c(0,100), maximum = TRUE)$maximum
    # stopping condition
    if(abs(lambda_k - lambda_ks[k]) < epsilon){
      #print(paste("EM algorithm stopped at iteration", k))
      break
    }
    # update
    lambda_ks <- c(lambda_ks, lambda_k)
    k <- k +1
  }
  EM_lambda_hat[i] <- lambda_ks[length(lambda_ks)]
}

# lambda hat graphs
par(mfrow=c(1,2))
hist(MLE_lambda_hat)
hist(EM_lambda_hat)
```

```r
# variances
var(MLE_lambda_hat)
var(EM_lambda_hat)
```