Computational Statistics Lab 6

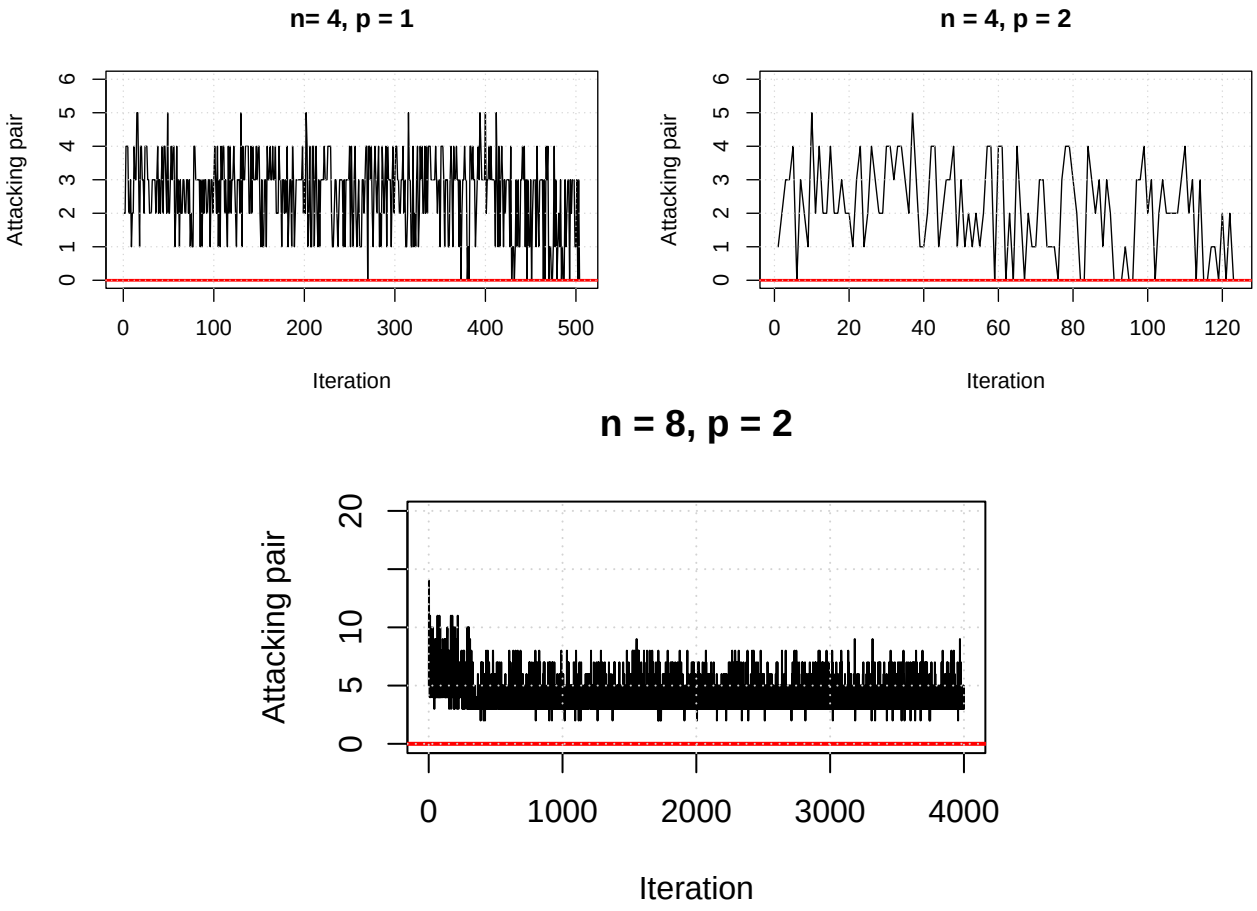
Simge Cinar & Ronald Yamashita

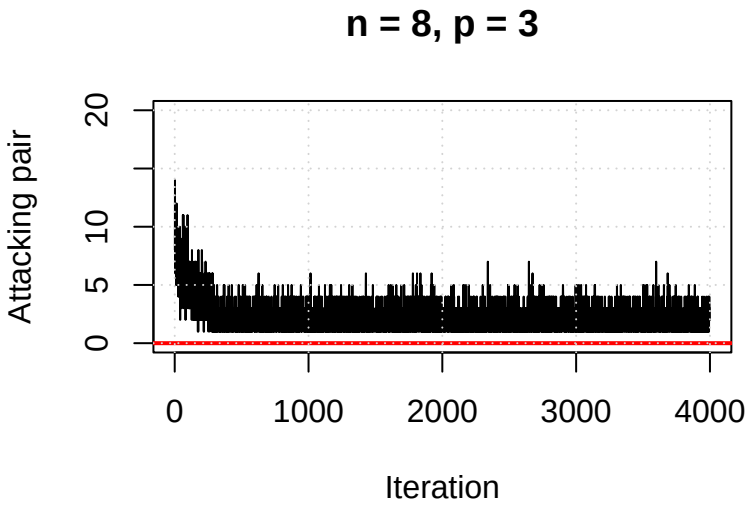
2023-12-22

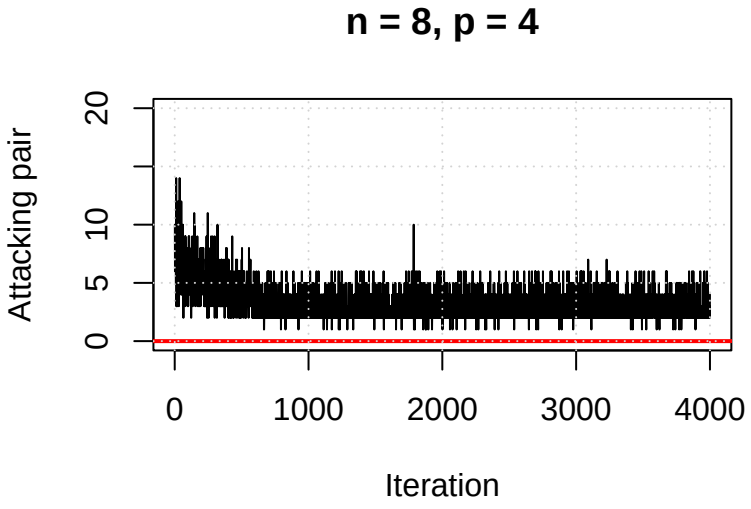
Question 1: Genetic Algorithm

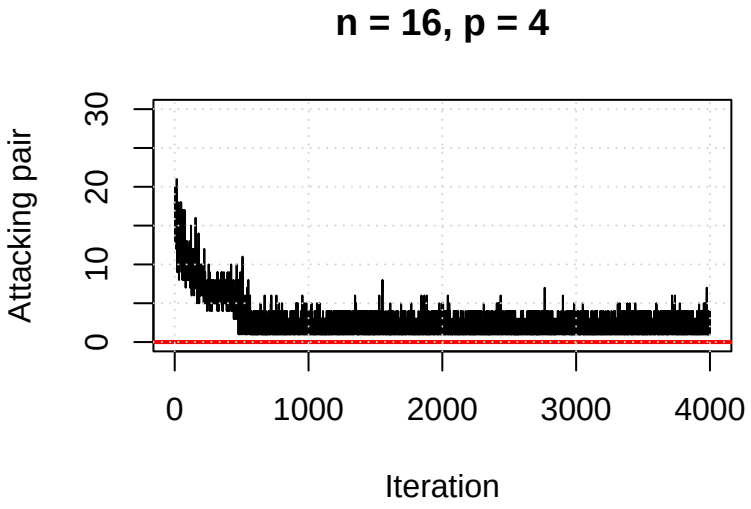
First let's try different values for p in the crossover function for each encoding. We set mutation probability to 0.5 and used "not attacking queen" fitness function.

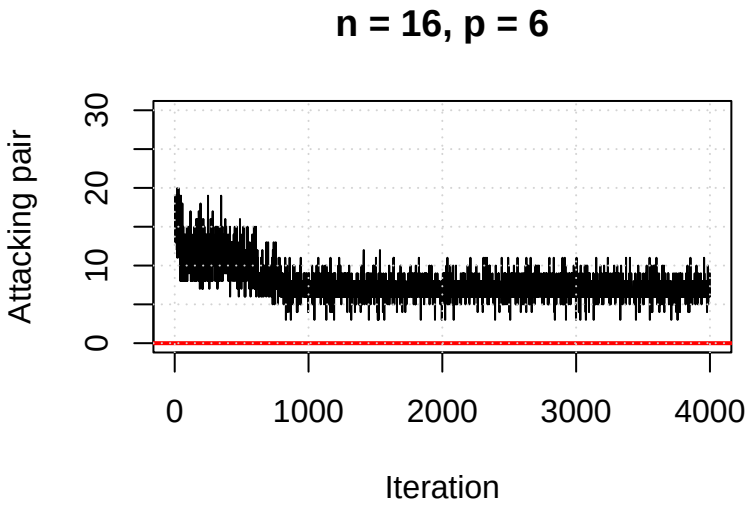
Encoding a crossover p value graphs:

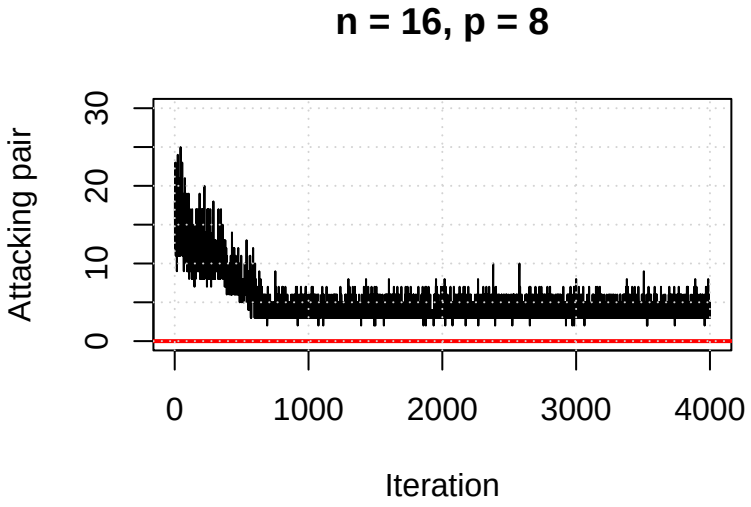



n = 8, p = 3
Attacking pair
0
5
10
20
0
1000
2000
3000
4000
Iteration

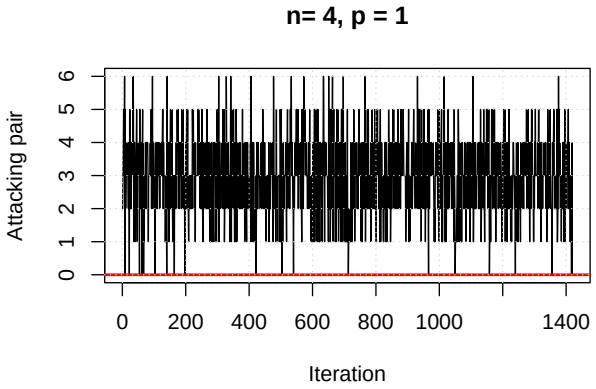

n = 8, p = 4
Attacking pair
0
5
10
20
0
1000
2000
3000
4000
Iteration

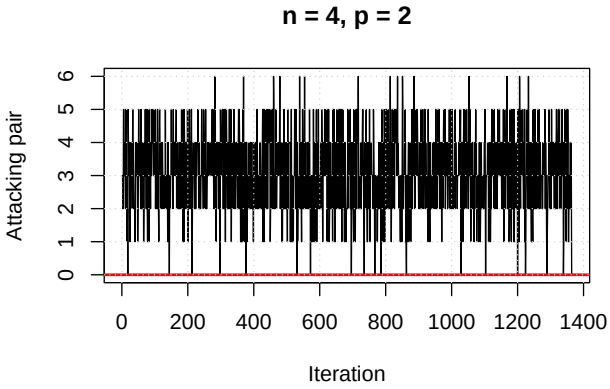

n = 16, p = 4
Attacking pair
0
10
20
30
0
1000
2000
3000
4000
Iteration

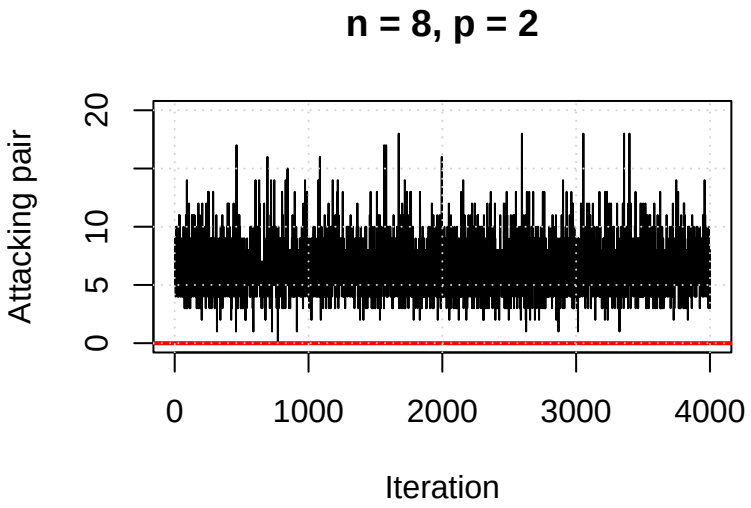


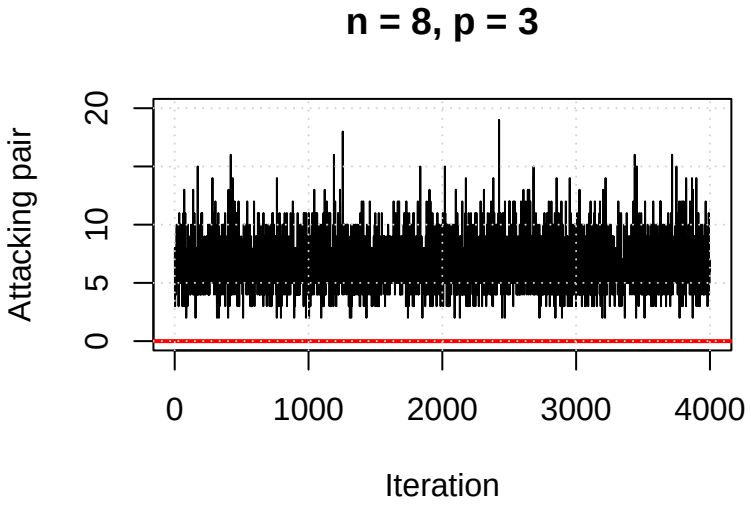


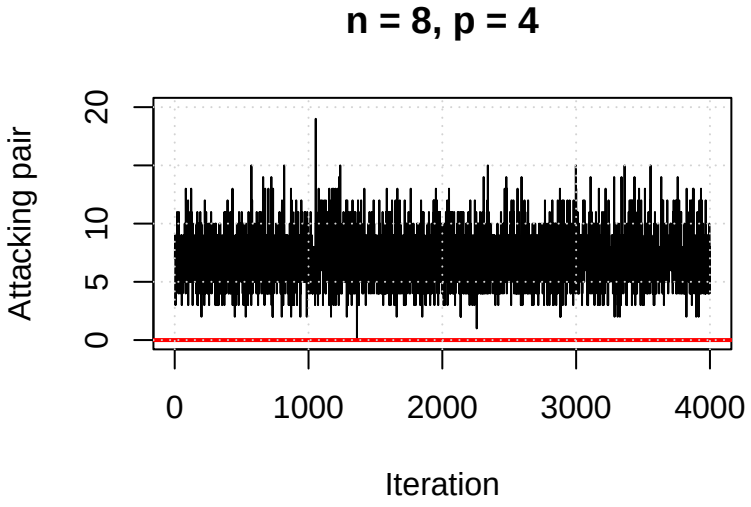
Encoding b crossover p value graphs:

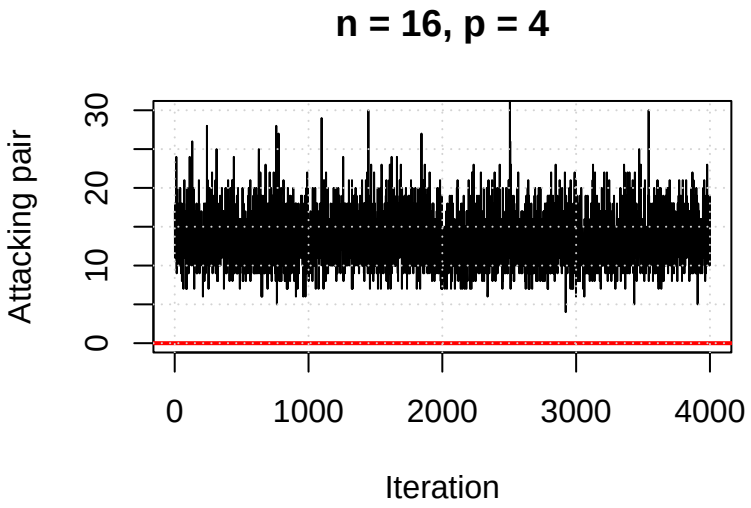


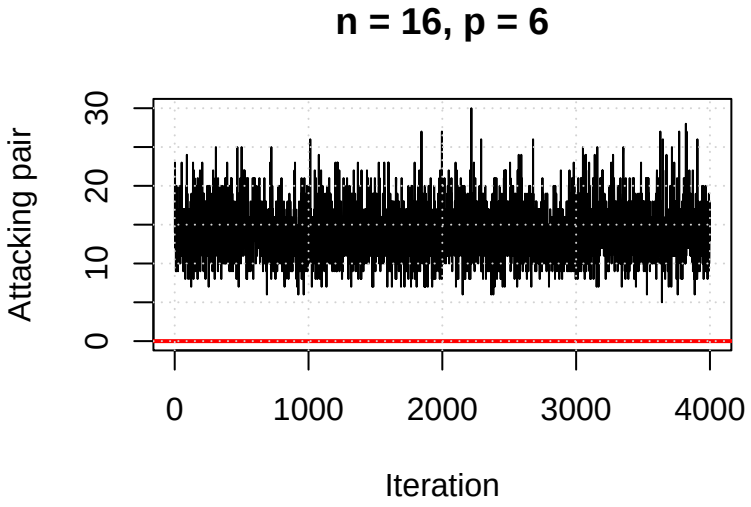


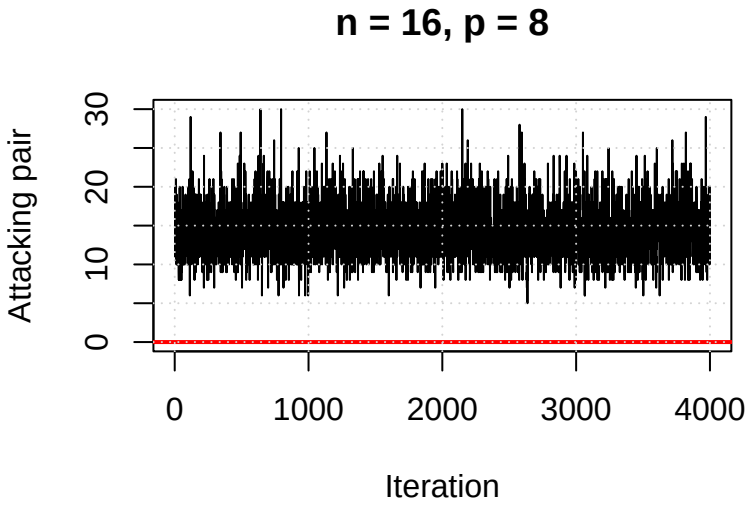

n = 8, p = 2
Attacking pair
0
5
10
20
0
1000
2000
3000
4000
Iteration


n = 8, p = 3
Attacking pair
0
5
10
20
0
1000
2000
3000
4000
Iteration

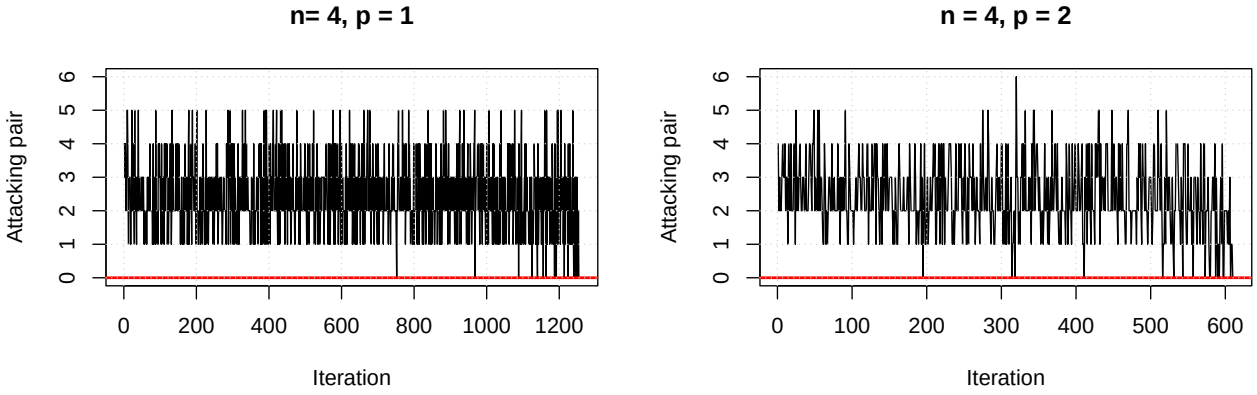

n = 8, p = 4
Attacking pair
0
5
10
20
0
1000
2000
3000
4000
Iteration

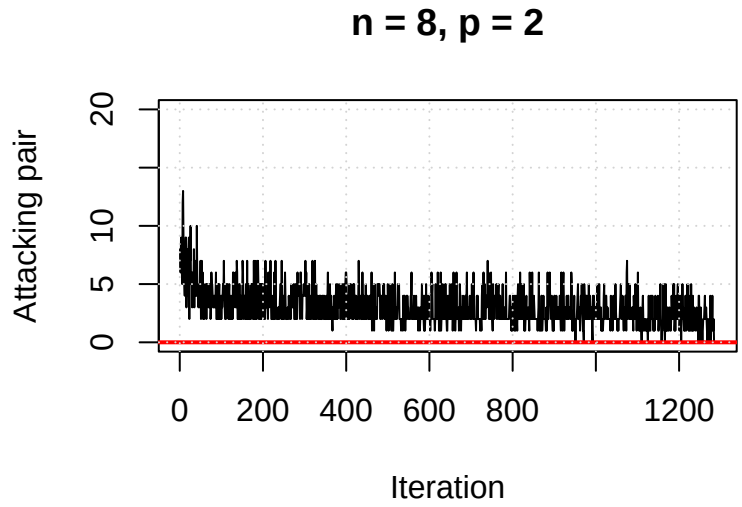

n = 16, p = 4
Attacking pair
0
10
20
30
0
1000
2000
3000
4000
Iteration

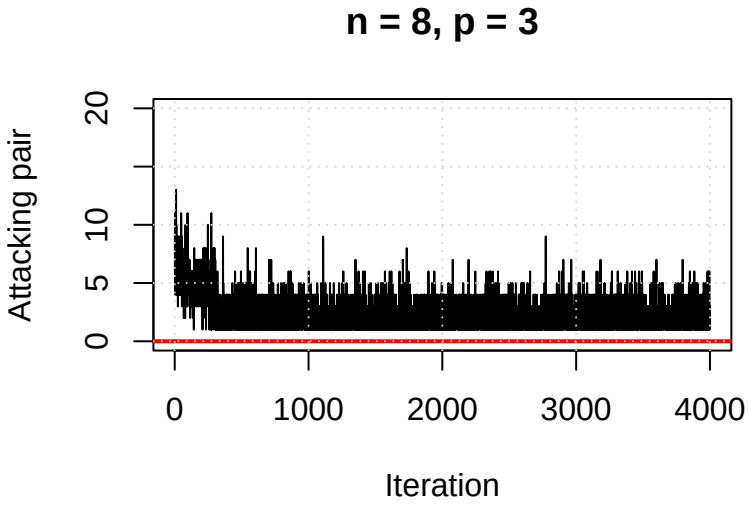

n = 16, p = 6
Attacking pair
0
10
20
30
0
1000
2000
3000
4000
Iteration

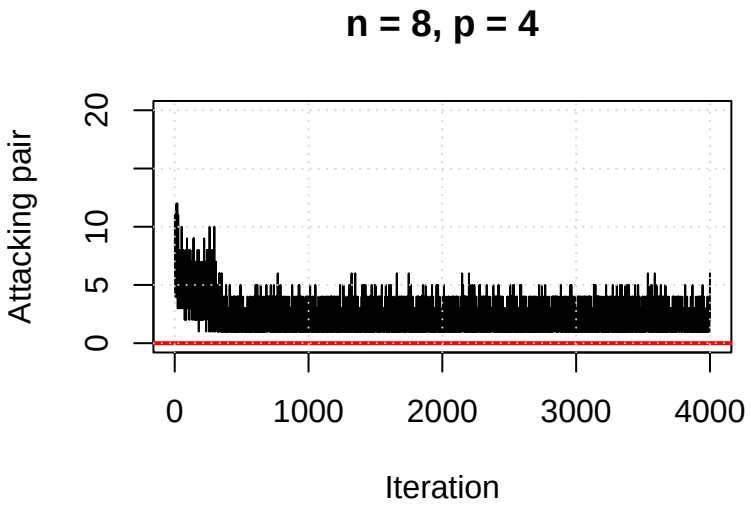

n = 16, p = 8
Attacking pair
0
10
20
30
0
1000
2000
3000
4000
Iteration

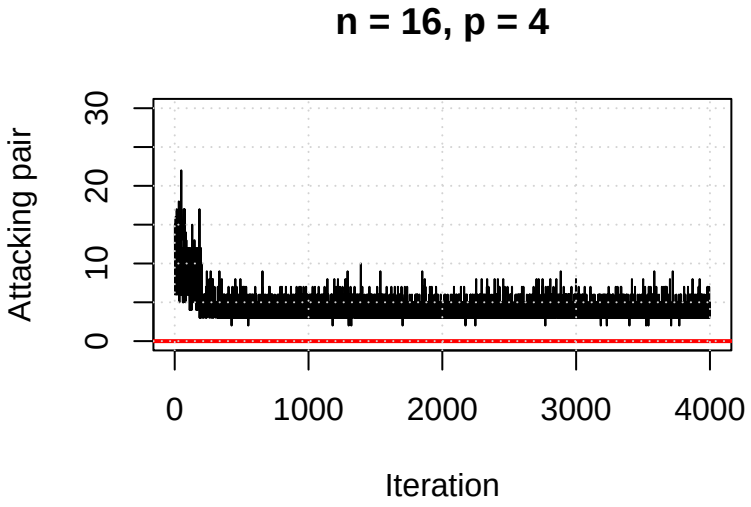
Encoding c crossover p value graphs:

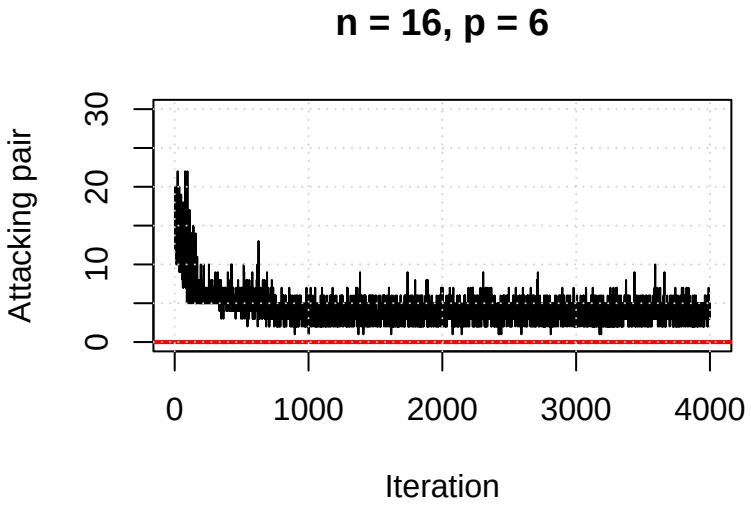


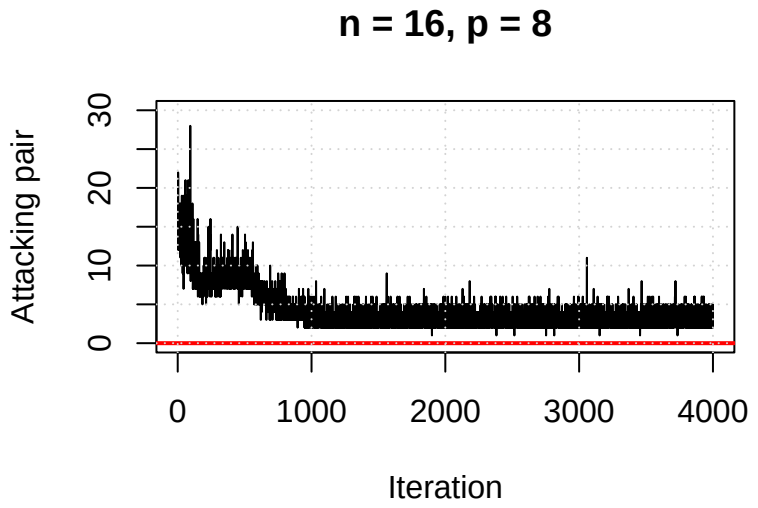





n = 8, p = 4
Attacking pair
0
5
10
20
0
1000
2000
3000
4000
Iteration


n = 16, p = 4
Attacking pair
0
10
20
30
0
1000
2000
3000
4000
Iteration


n = 16, p = 6
Attacking pair
0
10
20
30
0
1000
2000
3000
4000
Iteration



The genetic algorithm is implemented for three encodings. In each encoding, there is only one queen in each column. Population size is 20, maximum iteration is 4000 and best p value for crossover is selected by looking at the graphs above for each encoding. The code can be seen in Appendix. For each fitness function, a score is calculated between 0 (the worst) and 1 (the best). Fitness function indices and meanings are as follows:

- 1 - binary fitness score: check if there is a clash or not
- 2 - not attacked queen score: counts the number of not attacked queen
- 3 - attacking pair score: count the number of pairs that clash

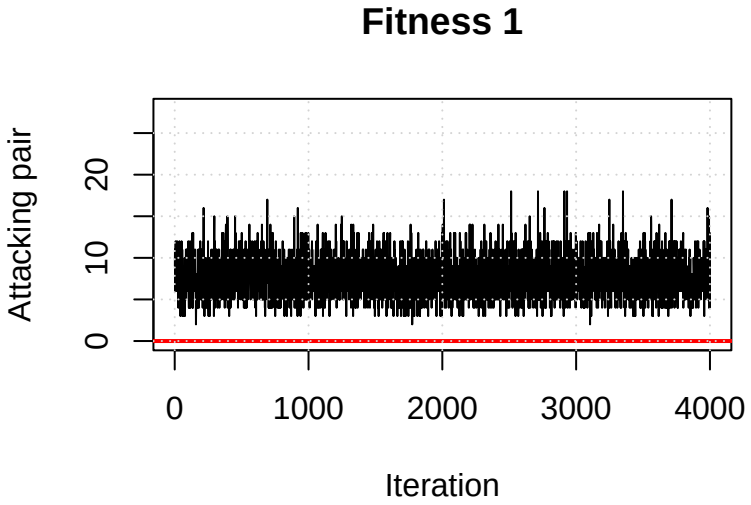
The fitness score value for each n, mutation probability and fitness function type is calculated. The results and some attacking pair queens plot for each encoding can be seen below. These graphs are used to compare different mutation probabilities and fitness function types.

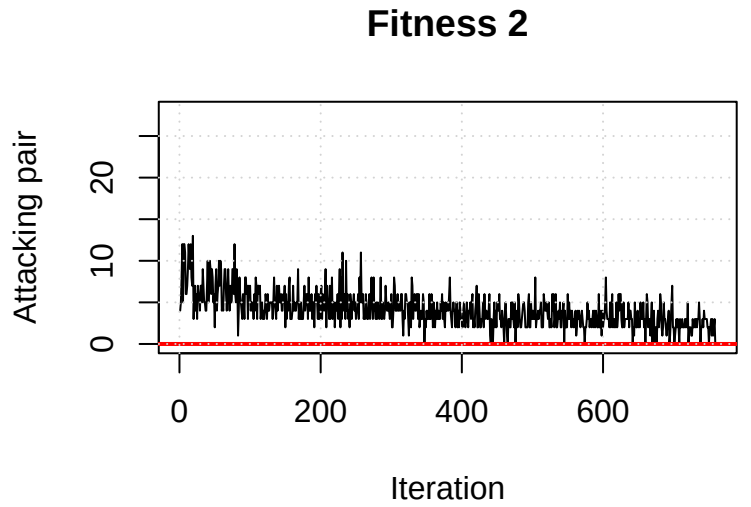
Encoding (a) Results

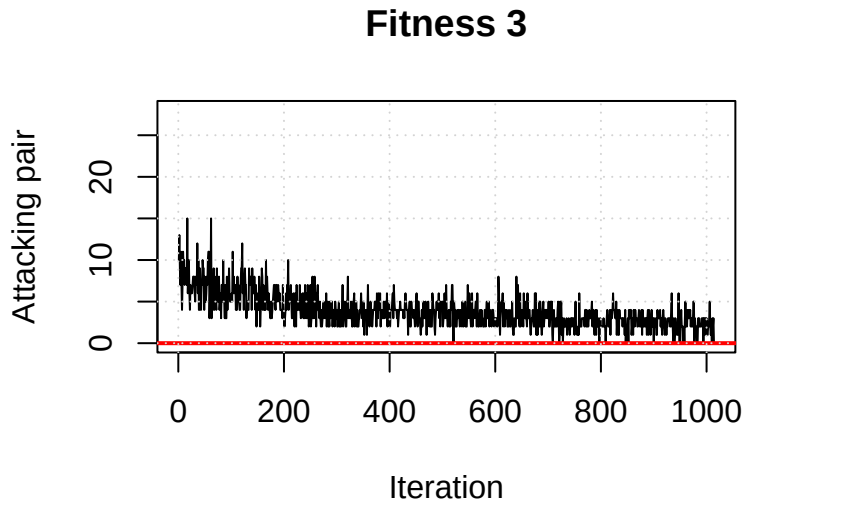
Encoding a stores the values as pairs inside a list. When there is 4 queens, the algorithm found a legal state and fitness score is 1 for each combination. When queen number is 8, fitness function 3 with mutation probability 0.1 and 0.9 give the score which is 0.9643. When queen number is 16, fitness function 3 with mutation probability 0.9 gives the highest score 0.1.

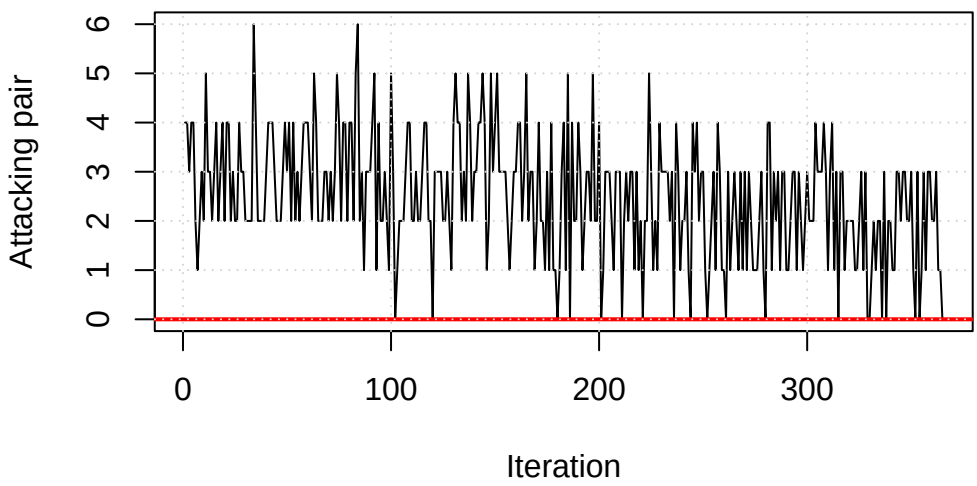
```
## n is 4
## Fitness function is 1 -- mutation probaility is 0.1 -- score is 1
## Fitness function is 2 -- mutation probaility is 0.1 -- score is 1
## Fitness function is 3 -- mutation probaility is 0.1 -- score is 1
## Fitness function is 1 -- mutation probaility is 0.5 -- score is 1
## Fitness function is 2 -- mutation probaility is 0.5 -- score is 1
## Fitness function is 3 -- mutation probaility is 0.5 -- score is 1
## Fitness function is 1 -- mutation probaility is 0.9 -- score is 1
## Fitness function is 2 -- mutation probaility is 0.9 -- score is 1
## Fitness function is 3 -- mutation probaility is 0.9 -- score is 1
## n is 8
## Fitness function is 1 -- mutation probaility is 0.1 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.1 -- score is 1
## Fitness function is 3 -- mutation probaility is 0.1 -- score is 0.9642857
## Fitness function is 1 -- mutation probaility is 0.5 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.5 -- score is 0.875
## Fitness function is 3 -- mutation probaility is 0.5 -- score is 1
```

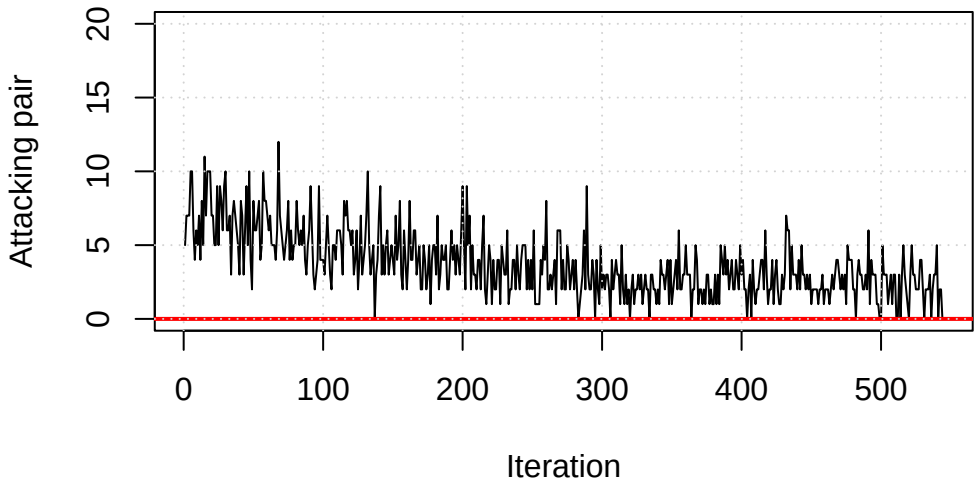
```
## Fitness function is 1 -- mutation probaility is 0.9 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.9 -- score is 0.85
## Fitness function is 3 -- mutation probaility is 0.9 -- score is 0.9642857
## n is 16
## Fitness function is 1 -- mutation probaility is 0.1 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.1 -- score is 0.8125
## Fitness function is 3 -- mutation probaility is 0.1 -- score is 0.9916667
## Fitness function is 1 -- mutation probaility is 0.5 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.5 -- score is 0.875
## Fitness function is 3 -- mutation probaility is 0.5 -- score is 0.9904167
## Fitness function is 1 -- mutation probaility is 0.9 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.9 -- score is 0.871875
## Fitness function is 3 -- mutation probaility is 0.9 -- score is 1

## Run time: 20.54009
```

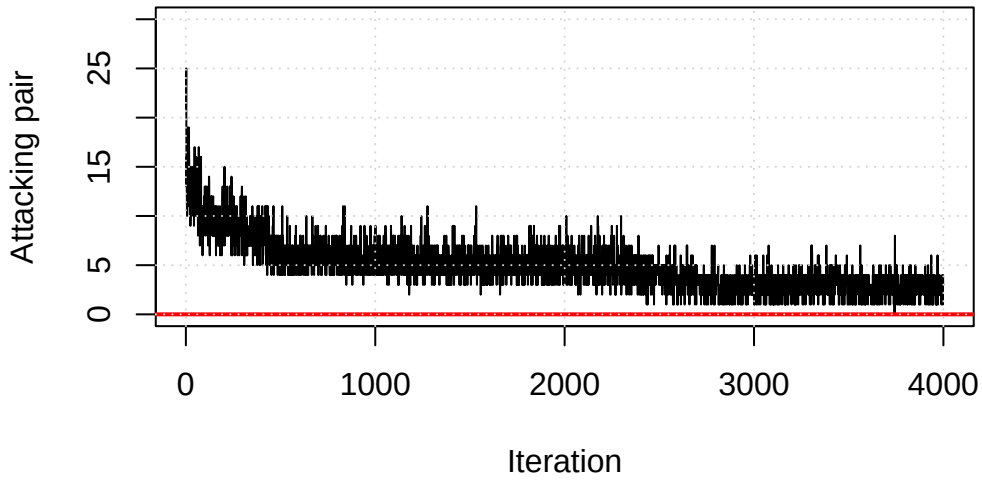





Fitness 3
Attacking pair
0
10
20
0
200
400
600
800
1000
Iteration


4 queens, mutation prob = 0.9, fitness type = 3
Attacking pair
0
1
2
3
4
5
6
0
100
200
300
Iteration


8 queens, mutation prob = 0.9, fitness type = 3
Attacking pair
0
5
10
15
20
0
100
200
300
400
500
Iteration




```
## Initial score: 0.4
```

```
## Legal configuration (n = 4):
```

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

```
## Initial score: 0.7410714
```

```
## Legal configuration (n = 8):
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    0    0    0    0    0    1    0    0
## [2,]    0    0    0    1    0    0    0    0
## [3,]    0    0    0    0    0    0    1    0
## [4,]    1    0    0    0    0    0    0    0
## [5,]    0    0    0    0    0    0    0    1
## [6,]    0    1    0    0    0    0    0    0
## [7,]    0    0    0    0    1    0    0    0
## [8,]    0    0    1    0    0    0    0    0
```

```
## Initial score: 0.855
```

```
## Legal configuration (n = 16):
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
##  [1,]    0    1    0    0    0    0    0    0    0     0     0     0     0
##  [2,]    0    0    0    0    0    1    0    0    0     0     0     0     0
##  [3,]    0    0    0    0    0    0    0    0    0     0     0     0     0
##  [4,]    0    0    0    0    0    0    0    0    1     0     0     0     0
##  [5,]    0    0    0    0    0    0    0    0    0     0     1     0     0
##  [6,]    0    0    0    0    0    0    0    0    0     0     0     0     0
##  [7,]    0    0    0    0    1    0    0    0    0     0     0     0     0
##  [8,]    0    0    1    0    0    0    0    0    0     0     0     0     0
##  [9,]    1    0    0    0    0    0    0    0    0     0     0     0     0
## [10,]    0    0    0    0    0    0    1    0    0     0     0     0     0
## [11,]    0    0    0    0    0    0    0    0    0     0     0     0     1
```

```
## [12,]    0    0    0    0    0    0    0    1    0     0     0     0     0
## [13,]    0    0    0    0    0    0    0    0    0     0     0     1     0
## [14,]    0    0    0    0    0    0    0    0    0     0     0     0     0
## [15,]    0    0    0    1    0    0    0    0    0     0     0     0     0
## [16,]    0    0    0    0    0    0    0    0    0     1     0     0     0
##       [,14] [,15] [,16]
##  [1,]     0     0     0
##  [2,]     0     0     0
##  [3,]     0     1     0
##  [4,]     0     0     0
##  [5,]     0     0     0
##  [6,]     0     0     1
##  [7,]     0     0     0
##  [8,]     0     0     0
##  [9,]     0     0     0
## [10,]     0     0     0
## [11,]     0     0     0
## [12,]     0     0     0
## [13,]     0     0     0
## [14,]     1     0     0
## [15,]     0     0     0
## [16,]     0     0     0
```

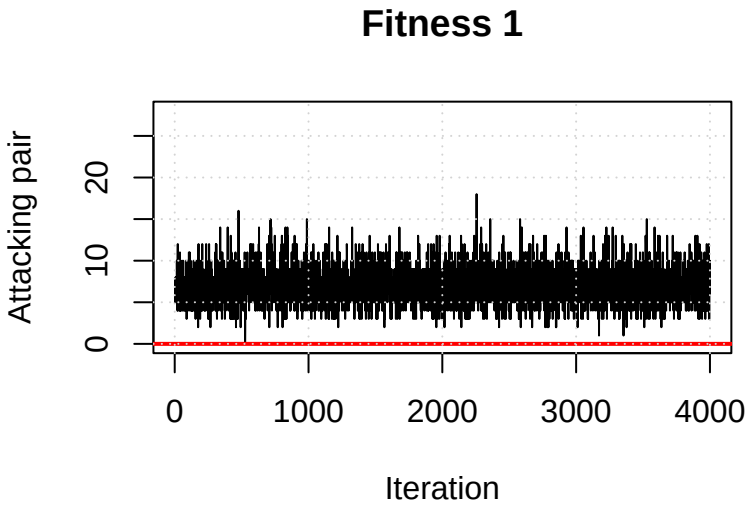
Encoding (b) Results

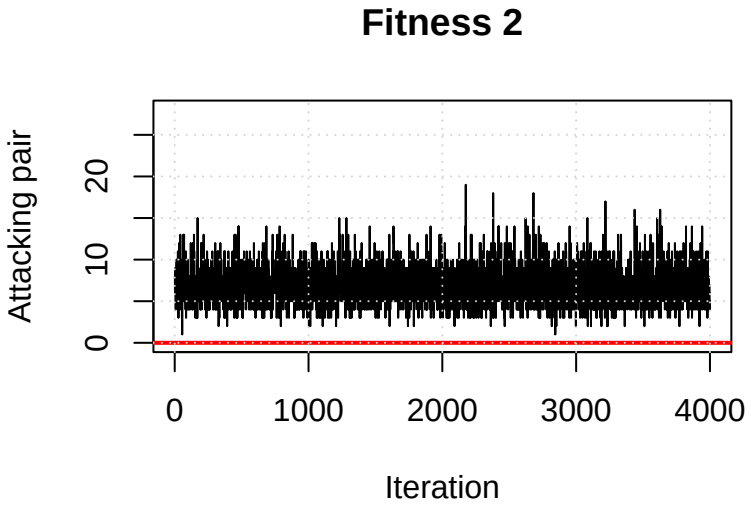
Encoding b is stores the values in a matrix in binary form, binary representation in each column represents the corresponding row. When there is 4 queens, the algorithm found a legal state and fitness score is 1 for each combination. When queen number is 8, fitness function 3 with mutation probability 0.9 give the highest score which is 0.9339. When queen number is 16, fitness function 3 with mutation probability 0.5 gives the highest score 0.9445.

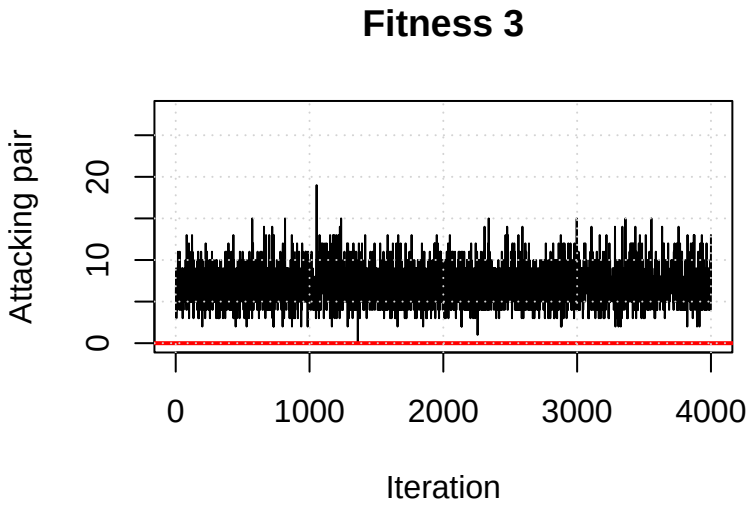
```
## n is 4
## Fitness function is 1 -- mutation probaility is 0.1 -- score is 1
## Fitness function is 2 -- mutation probaility is 0.1 -- score is 1
## Fitness function is 3 -- mutation probaility is 0.1 -- score is 1
## Fitness function is 1 -- mutation probaility is 0.5 -- score is 1
## Fitness function is 2 -- mutation probaility is 0.5 -- score is 1
## Fitness function is 3 -- mutation probaility is 0.5 -- score is 1
## Fitness function is 1 -- mutation probaility is 0.9 -- score is 1
## Fitness function is 2 -- mutation probaility is 0.9 -- score is 1
## Fitness function is 3 -- mutation probaility is 0.9 -- score is 1
## n is 8
## Fitness function is 1 -- mutation probaility is 0.1 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.1 -- score is 0.78125
## Fitness function is 3 -- mutation probaility is 0.1 -- score is 0.925
## Fitness function is 1 -- mutation probaility is 0.5 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.5 -- score is 0.75
## Fitness function is 3 -- mutation probaility is 0.5 -- score is 0.9178571
## Fitness function is 1 -- mutation probaility is 0.9 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.9 -- score is 0.75
## Fitness function is 3 -- mutation probaility is 0.9 -- score is 0.9303571
## n is 16
## Fitness function is 1 -- mutation probaility is 0.1 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.1 -- score is 0.678125
## Fitness function is 3 -- mutation probaility is 0.1 -- score is 0.9391667
```

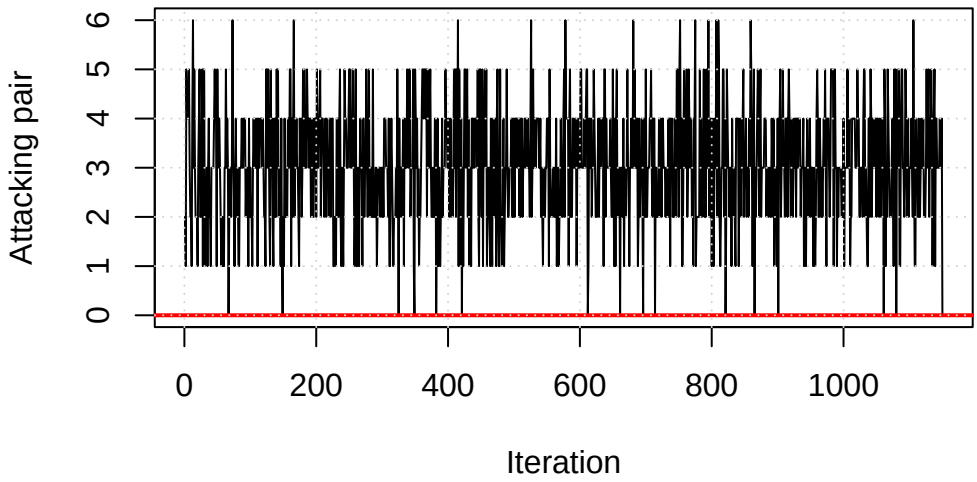
```
## Fitness function is 1 -- mutation probaility is 0.5 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.5 -- score is 0.696875
## Fitness function is 3 -- mutation probaility is 0.5 -- score is 0.9433333
## Fitness function is 1 -- mutation probaility is 0.9 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.9 -- score is 0.6875
## Fitness function is 3 -- mutation probaility is 0.9 -- score is 0.9429167
```

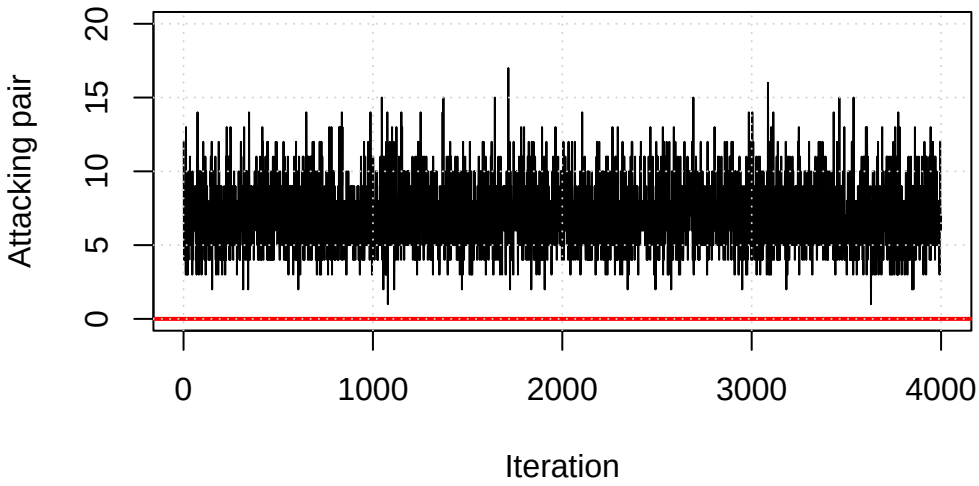
```
## Run time: 4.863443
```

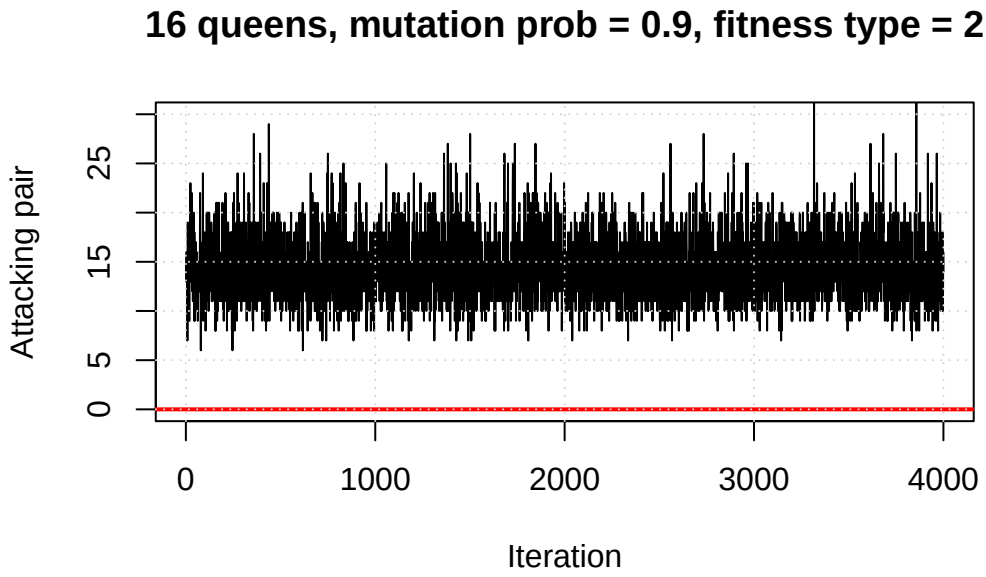





Fitness 3
Attacking pair
0
10
20
0
1000
2000
3000
4000
Iteration


4 queens, mutation prob = 0.9, fitness type = 2
Attacking pair
0
1
2
3
4
5
6
0
200
400
600
800
1000
Iteration


8 queens, mutation prob = 0.9, fitness type = 2
Attacking pair
0
5
10
15
20
0
1000
2000
3000
4000
Iteration




```
## Initial score: 0.6
```

```
## Legal configuration (n = 4):
```

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

```
## Initial score: 0.53125
```

```
## Legal configuration (n = 8):
```

```
## There's no legal configuration
```

```
## Initial score: 0.565625
```

```
## Legal configuration (n = 16):
```

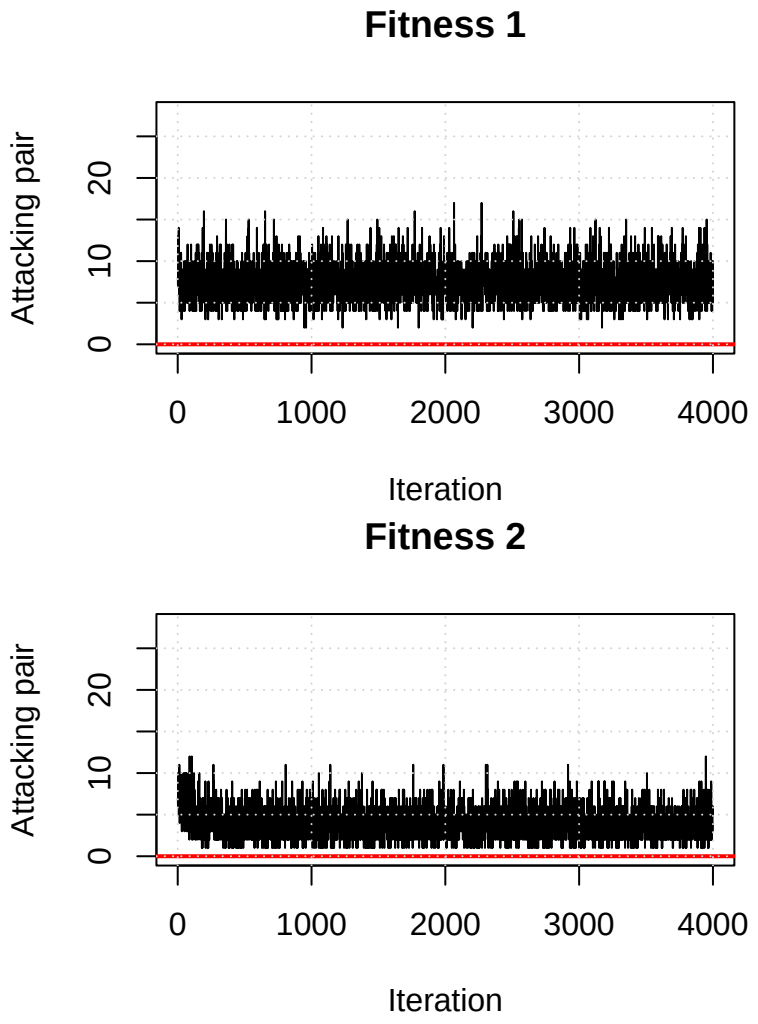
```
## There's no legal configuration
```

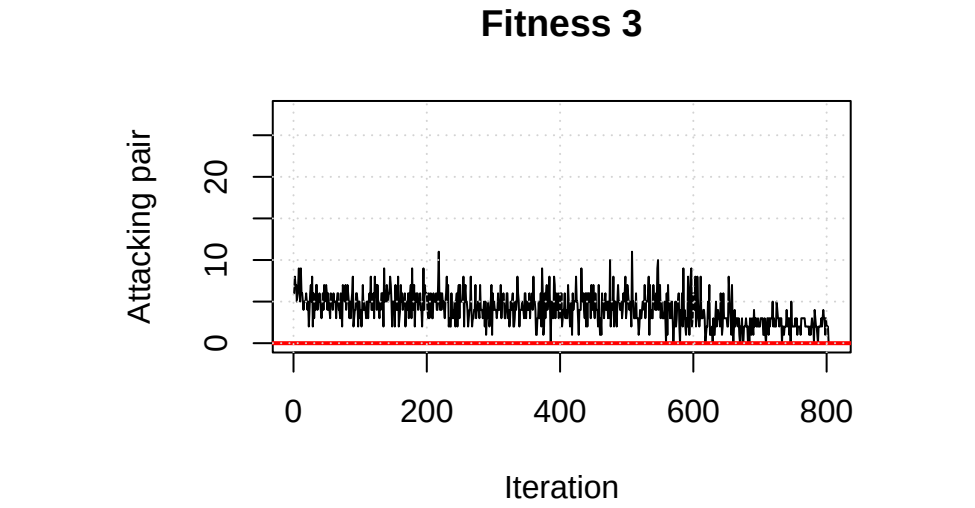
Encoding (c) Results

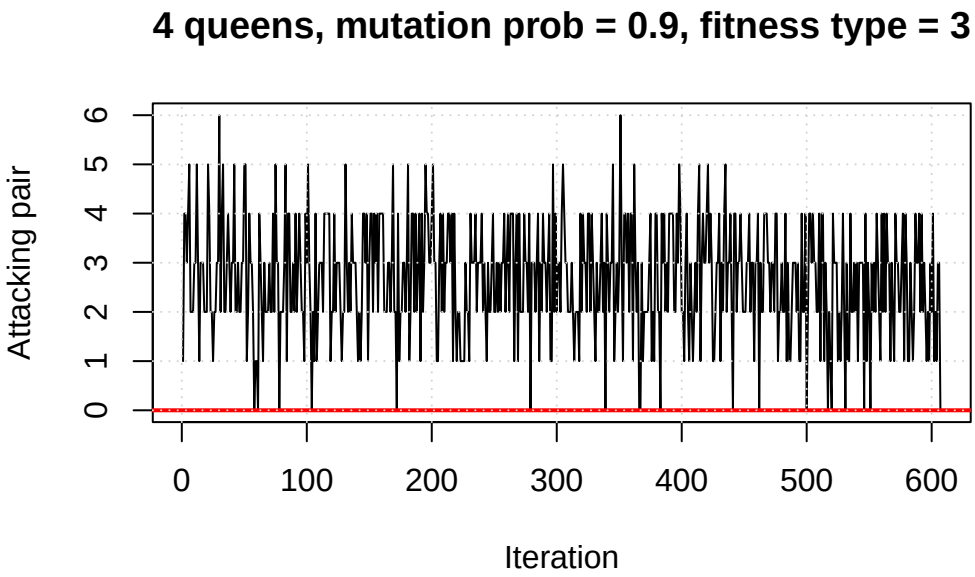
Encoding c stores the values as a 1xn matrix. The values in each row represent the row number. When there is 4 queens, the algorithm found fitness score is 1 for all combinations. When queen number is 8, fitness function 3 with 0.9 mutation probability gives score 0.9589. With 16 queens, fitness function 3 with mutation probability 0.9 gives score 0.9804.

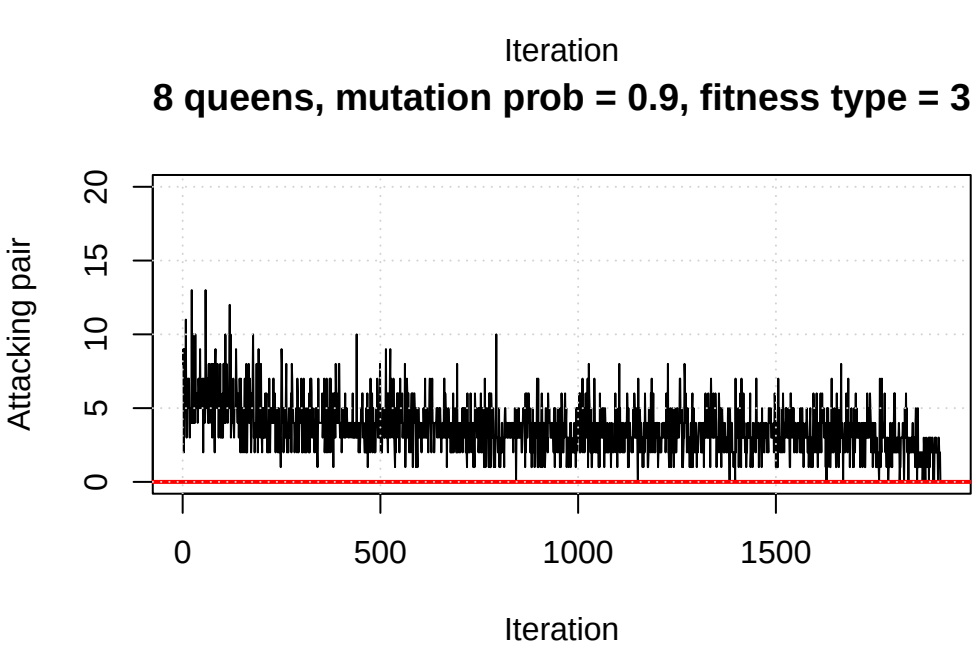
```
## n is 4
## Fitness function is 1 -- mutation probaility is 0.1 -- score is 1
## Fitness function is 2 -- mutation probaility is 0.1 -- score is 1
## Fitness function is 3 -- mutation probaility is 0.1 -- score is 1
## Fitness function is 1 -- mutation probaility is 0.5 -- score is 1
## Fitness function is 2 -- mutation probaility is 0.5 -- score is 1
## Fitness function is 3 -- mutation probaility is 0.5 -- score is 1
## Fitness function is 1 -- mutation probaility is 0.9 -- score is 1
## Fitness function is 2 -- mutation probaility is 0.9 -- score is 1
## Fitness function is 3 -- mutation probaility is 0.9 -- score is 1
## n is 8
```

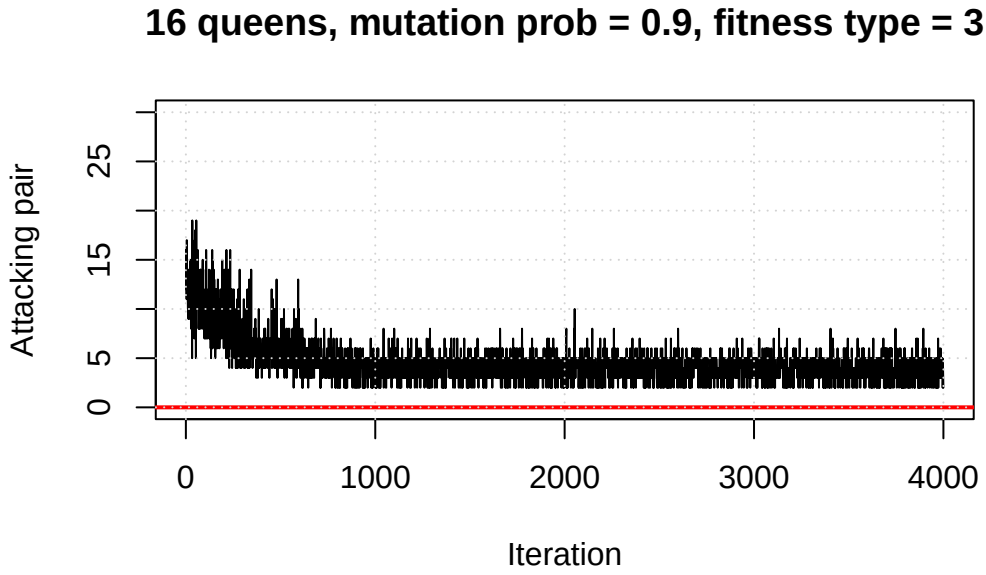
```
## Fitness function is 1 -- mutation probaility is 0.1 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.1 -- score is 0.875
## Fitness function is 3 -- mutation probaility is 0.1 -- score is 1
## Fitness function is 1 -- mutation probaility is 0.5 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.5 -- score is 1
## Fitness function is 3 -- mutation probaility is 0.5 -- score is 1
## Fitness function is 1 -- mutation probaility is 0.9 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.9 -- score is 0.8625
## Fitness function is 3 -- mutation probaility is 0.9 -- score is 0.9589286
## n is 16
## Fitness function is 1 -- mutation probaility is 0.1 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.1 -- score is 0.8125
## Fitness function is 3 -- mutation probaility is 0.1 -- score is 0.975
## Fitness function is 1 -- mutation probaility is 0.5 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.5 -- score is 0.9375
## Fitness function is 3 -- mutation probaility is 0.5 -- score is 0.975
## Fitness function is 1 -- mutation probaility is 0.9 -- score is 0
## Fitness function is 2 -- mutation probaility is 0.9 -- score is 0.93125
## Fitness function is 3 -- mutation probaility is 0.9 -- score is 0.9804167
## Run time: 41.97503
```




Fitness 3
Attacking pair
0
10
20
0
200
400
600
800
Iteration


4 queens, mutation prob = 0.9, fitness type = 3
Attacking pair
0
1
2
3
4
5
6
0
100
200
300
400
500
600
Iteration


8 queens, mutation prob = 0.9, fitness type = 3
Attacking pair
0
5
10
15
20
0
500
1000
1500
Iteration




```
## Initial score: 0.6916667
```

```
## Legal configuration (n = 4):
```

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

```
## Initial score: 0.8035714
```

```
## Legal configuration (n = 8):
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    0    0    0    0    0    1    0    0
## [2,]    0    0    1    0    0    0    0    0
## [3,]    0    0    0    0    1    0    0    0
## [4,]    0    0    0    0    0    0    0    1
## [5,]    1    0    0    0    0    0    0    0
## [6,]    0    0    0    1    0    0    0    0
## [7,]    0    1    0    0    0    0    0    0
## [8,]    0    0    0    0    0    0    1    0
```

```
## Initial score: 0.92375
```

```
## Legal configuration (n = 16):
```

```
## There's no legal configuration
```

Conclusion

- Only encoding a found a legal configuration for 16 queen problem
- Encoding a found a legal configuration for all n and encoding c found a legal configuration for n=4 and n = 8 but encoding b could only found for n = 4.
- Fitness function 1 (binary) is the worst. If there's any clashes, it directly assigns zero to score and since the algorithm limits the solutions explored by the algorithm, it cannot make a improvement in the population. It is the most aggressive fitness function to find a legal configuration.

- Even though fitness function 3 (attacking pair) gives higher score, fitness function 2 (not attacked queen) is more aggressive at finding legal configuration.
- Mutation probability 0.9 is the best among other values. High mutation probability explores the solution space more.
- As n increases, initial score is increasing but it is getting harder to find a legal configuration.
- Initial score values in encoding b is the lowest. When n = 16, initial score of encoding c is higher than encoding a but encoding c couldn't find a legal configuration.
- Encoding b has the lowest run time and encoding c has the highest run time.
- From the values I get, encoding a and mutation probability 0.9 worked best.

Question 2: EM algorithm

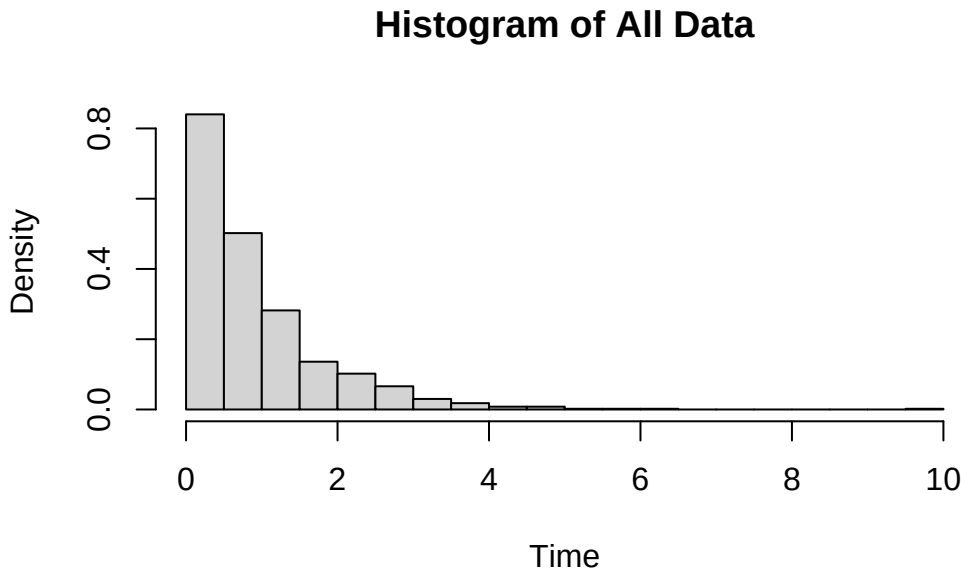
The data file censoredproc.csv contains the time after which a certain product fails. Some of these measurements are left-censored (cens=2)—i.e., we did not observe the time of failure, only that the product had already failed when checked upon. Status cens=1 means that the exact time of failure was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

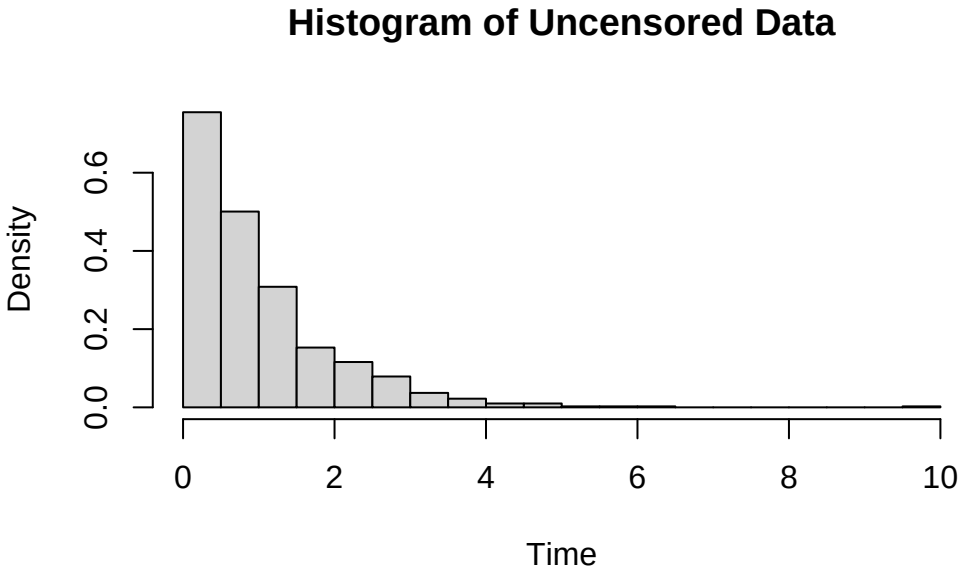
cens = 1 : exact time of failure observed (observed-uncensored)
cens = 2 : the product had already failed when checked upon (censored)

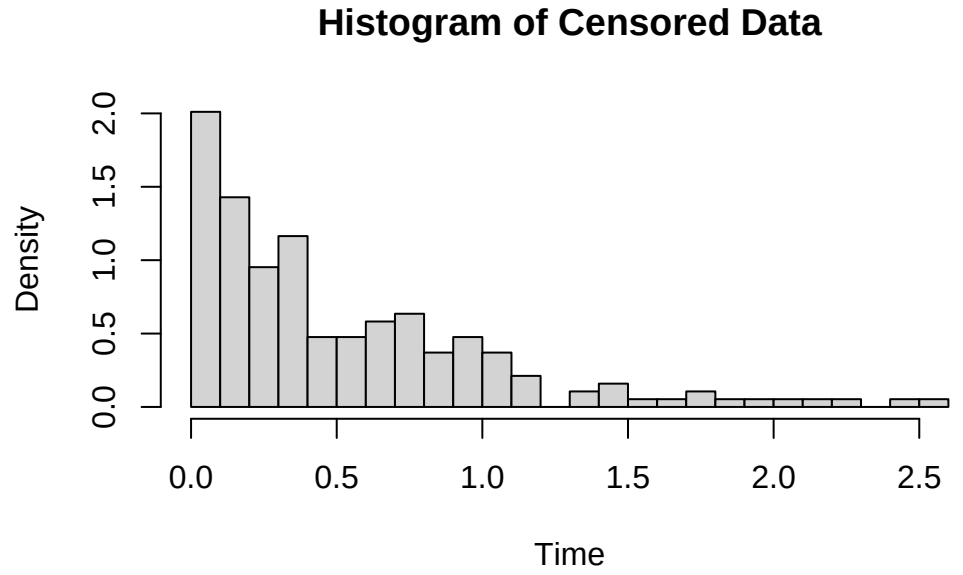
Part 1)

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

Answer: The histogram of all data and uncensored data can be seen below. Both of the histograms remind exponential distribution.







Part 2)

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

Answer: The density function of exponential and truncated exponential function (Truncated distribution 2023) can be seen below, x is the failure time and t is the censor time.

$$f(x|\lambda) = \lambda e^{-\lambda x}, \text{ where } x \geq 0, \lambda > 0$$

$$f(x|\lambda, t) = \frac{\lambda e^{-\lambda x}}{1 - e^{-\lambda t}}, \text{ where } 0 \leq x \leq t, \lambda > 0$$

Uncensored data values come from the exponential λ distribution, while censored data from a truncated exponential distribution. Likelihood function can be seen below, U represents the set of uncensored data and C represents the censored data.

$$\prod_{i \in U} \lambda e^{-\lambda x_i^{(1)}} \cdot \prod_{j \in C} \frac{\lambda e^{-\lambda x_j^{(2)}}}{1 - e^{-\lambda t_j}}$$

In the censored data we know the t value (censor time) but we don't know the x (exact failure time). Let's derive using the conditional expectation of x.

$$E[X|X < t] = \int_0^t x \cdot \frac{\lambda e^{-\lambda x}}{1 - e^{-\lambda t}} dx$$

$$E[X|X < t] = \frac{\lambda \int_0^t x e^{-\lambda x} dx}{1 - e^{-\lambda t}}$$

$$\lambda \int_0^t x e^{-\lambda x} dx = \lambda \Big[(-\frac{1}{\lambda} x - \frac{1}{\lambda^2}) e^{-\lambda x} \Big]_0^t = \lambda \Big[(-\frac{1}{\lambda} t - \frac{1}{\lambda^2}) e^{-\lambda t} + \frac{1}{\lambda^2} \Big]$$

$$E[X|X < t] = \frac{\lambda \Big[(-\frac{1}{\lambda} t - \frac{1}{\lambda^2}) e^{-\lambda t} + \frac{1}{\lambda^2} \Big]}{1 - e^{-\lambda t}}$$

We will find the x values for each to using the equation above. Now let's take the logarithm of the likelihood function. The loglikelihood function is as follows, since we know $x^{(1)}$, $x^{(2)}$ and t values, we can calculate the loglikelihood

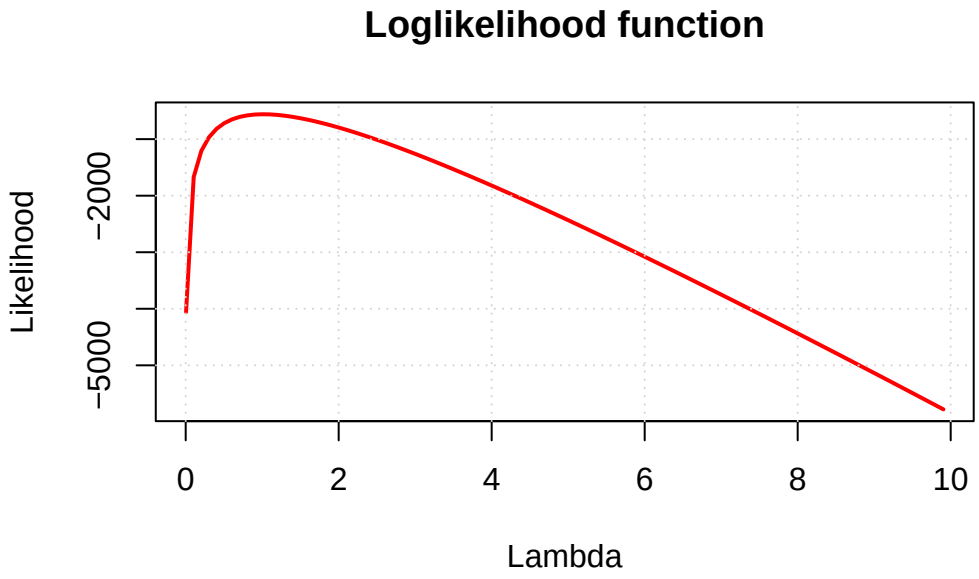
$$\ln(\prod_{i \in U} \lambda e^{-\lambda x_i^{(1)}}) \cdot \ln(\prod_{j \in C} \frac{\lambda e^{-\lambda x_j^{(2)}}}{1 - e^{-\lambda t_j}})$$

Part 3)

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

Answer: The loglikelihood function is as follows. There's an optimal lambda value that maximizes the loglikelihood between 0 and 2.

```
# Loglikelihood function
llik_fnc <- function(lambda, cens1_time, cens2_time){
  t <- cens2_time
  numerator <- lambda * ((((-t/lambda)-(1/lambda^2))*exp(-lambda*t)) + (1/lambda^2))
  denominator <- 1-exp(-lambda*t)
  censored_x <- numerator/denominator # find the expected failure time from censored data

  a <- lambda*exp(-lambda*censored_x)
  b <- 1-exp(-lambda*t)
  llik_censored <- log(a/b)
  llik_uncensored <- log(lambda*exp(-lambda*cens1_time))
  loglikelihood <- sum(llik_uncensored) + sum(llik_censored)
  return(loglikelihood)
}
```



Part 4)

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

Answer: Since I used optim function, the iteration count is NA in the "Brent" method.

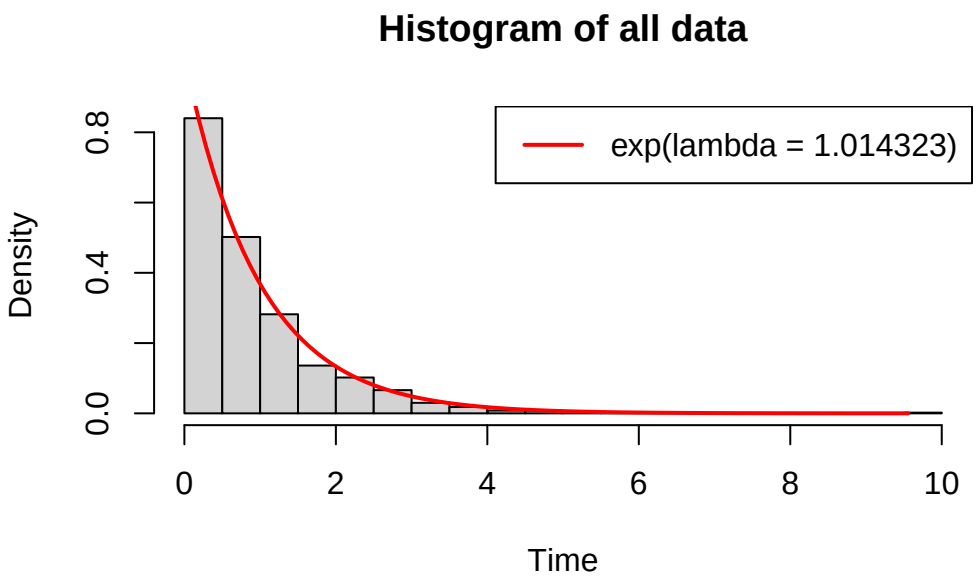
```
EM_algorithm <- function(lambda, cens1_time, cens2_time){
  negative_llik_fnc <- function(lambda, cens1_time, cens2_time){
    return(-llik_fnc(lambda, cens1_time, cens2_time))
  }

  result <- optim(par = lambda,
                  fn = negative_llik_fnc,
                  cens1_time = cens1_time, cens2_time = cens2_time,
                  method = "Brent",
                  lower = 0.001, upper = 10)
  optimal_lambda <- result$par
  return(optimal_lambda)
}

optimal_lambda <- EM_algorithm(100, cens1$time, cens2$time)
cat("Optimal lambda:", optimal_lambda, "\n")
```

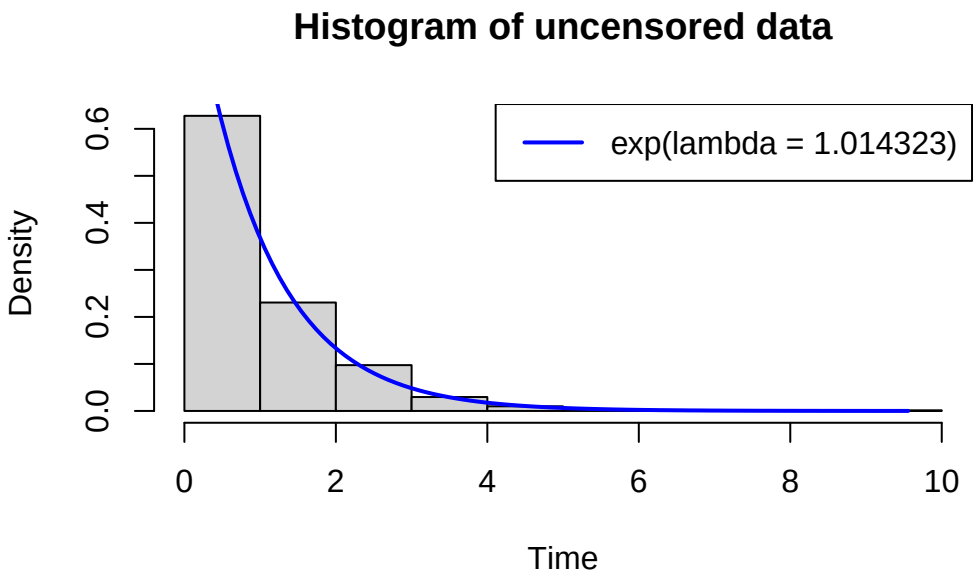
```
## Optimal lambda: 1.014323
```

Part 5)

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

Answer: The histograms can be seen below





Part 6)

Question: Study how good your EM algorithm is compared to usual maximum likelihood estimation with data reduced to only the uncensored observations. To this end we will use a parametric bootstrap. Repeat 1000 (reduce if computational time is too long—but carefully report the running times) times the following procedure:
(a) Simulate the same number of data points as in the original data, from the exponential $\hat{\lambda}$ distribution.
(b) Randomly select the same number of points as in the original data for censoring. For each observation for censoring—sample a new time from 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.

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.

Answer: The usual MLE estimator is as follows:

$$\hat{\lambda} = \frac{n}{\sum x_i}$$

```r
lambda_hat <- nrow(cens1) / sum(cens1$time)
cat("MLE on uncensored data:", lambda_hat, "\n")
```

```
## MLE on uncensored data: 1.004758
```

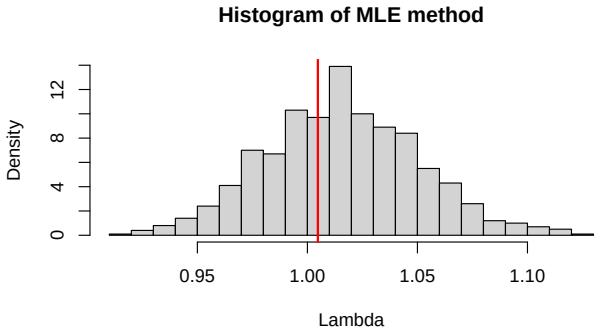
We will generate exponential $\hat{\lambda}$ distribution. All the values are failure times, to apply EM algorithm we need to find the censor time on censored data. We used to formula below:

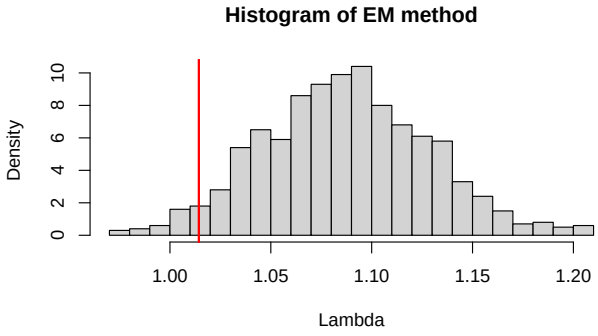
$$E[t|t > x] = \frac{\int_x^\infty t \cdot \lambda e^{-\lambda t} dt}{1 - e^{-\lambda x}}$$

$$\lambda \int_x^\infty t e^{-\lambda t} dt = \lambda \Big[(-\frac{1}{\lambda} t - \frac{1}{\lambda^2}) e^{-\lambda t} \Big]_x^\infty = -\lambda \Big[(-\frac{1}{\lambda} x - \frac{1}{\lambda^2}) e^{-\lambda x} \Big]$$

$$E[t|t > X] = \frac{-\lambda \Big[(-\frac{1}{\lambda} x - \frac{1}{\lambda^2}) e^{-\lambda x} \Big]}{e^{-\lambda x}}$$

```r
set.seed(12345)
B <- 1000
MLE_vector <- numeric(length = B)
EM_vector <- numeric(length = B)

for (b in 1:B){
  # Simulate data from exponential distribution
  generated_data <- rexp(nrow(df2), rate = optimal_lambda)
  uncensored_data <- sample(generated_data, size = nrow(cens1))
  censored_data_init <- generated_data[!generated_data %in% uncensored_data]

  # Find the censor time
  x <- censored_data_init # failure time
  above <- -optimal_lambda*(((-x/optimal_lambda) - (1/optimal_lambda^2)) * exp(-x*optimal_lambda))
  below <- exp(-x*optimal_lambda)
  censored_data <- above/below # censor time
  MLE_vector[b] <- length(uncensored_data) / sum(uncensored_data)
  EM_vector[b] <- EM_algorithm(100, uncensored_data, censored_data)
}
```






```
## Bias of MLE: 0.0108616
```

```
## Bias of EM: 0.07305739
```

```
## Variance of MLE: 0.001217778
```

```
## Variance of EM: 0.001693446
```

The red lines show the λ values found from the original data. By looking at the histograms and values above, it can be observed that EM method is more biased. Since we find censor time using expectation, it might cause this bias. Moreover variance of EM algorithm is slightly higher.

References

Truncated distribution. 2023. "Truncated Distribution — Wikipedia, the Free Encyclopedia." https://en.wikipedia.org/wiki/Truncated_distribution.

Appendix

```
knitr::opts_chunk$set(echo = TRUE)
# --------------------------- QUESTION 1 ---------------------------------------
# encoding a ---------------------------------------------------------------------
# store as pairs in a list
set.seed(12345)
encoding_a <- function(n){
  index_list <- list()
  r <- sample(1:n, n, replace = TRUE)
  c <- c(1:n)
  for (i in 1:n){
    vect <- c(r[i],c[i])
    index_list <- append(index_list, list(vect))
  }
  return(index_list)
}

# layout of the chessboard
generate_board_a <- function(encoded_list_a){
  n <- length(encoded_list_a)
  first_elements <- lapply(encoded_list_a, function(x) x[1])
  second_elements <- lapply(encoded_list_a, function(x) x[2])
  matrix_a <- matrix(0, nrow = n, ncol = n)
  for (i in 1:n){
    matrix_a[first_elements[[i]], second_elements[[i]]] <- 1
  }
  return(matrix_a)
}
set.seed(12345)
crossover_a <- function(parent1, parent2, p){
  n <- length(parent1)
  child1 <- c(parent1[1:p], parent2[(p+1):n])
  child2 <- c(parent2[1:p], parent1[(p+1):n])
  if (runif(1) <= 0.5){
    return(child1)
  } else{
    return(child2)
  }
}
set.seed(12345)
mutate_a <- function(encoding){
  n <- length(encoding)
  select_idx <- sample(1:n, size = 1)
  available_positions <- setdiff(1:n, encoding[[select_idx]][1])
  encoding[[select_idx]][1] <- sample(available_positions, size = 1)
  return(encoding)
}
# Fitness functions
diagonal_check <- function(queen1, queen2) {
  abs(queen1[1] - queen2[1]) == abs(queen1[2] - queen2[2])
}
```

```r
row_check <- function(queen1, queen2) {
  queen1[1] == queen2[1]
}
# --------------fitness function 1 ----------------------------------------------
binary_fitness_a <- function(encoding){
  n <- length(encoding)
  score <- 1
  for (i in 1:(n-1)){
    for (j in (i+1):n){
      if (diagonal_check(encoding[[i]], encoding[[j]])){
        score <- 0
        break
      }
      if(row_check(encoding[[i]], encoding[[j]])){
        score <- 0
        break
      }
    }
  }
  return(list(score = score))
}
# --------------fitness function 2 ----------------------------------------------
attacked_queen_a <- function(encoding){
  attacked <- 0
  n <- length(encoding)
  attacking_pair <- 0
  for (i in 1:(n-1)){
    for (j in (i+1):n){
      if ( diagonal_check(encoding[[i]], encoding[[j]]) ||
           row_check(encoding[[i]], encoding[[j]]) ){
        attacked <- attacked + 1
        break
      }
    }
  }
  score <- 1-(attacked/n)
  return(list(attacked_queen = attacked, score = score))
}
# --------------fitness function 3 ----------------------------------------------
attacking_pair_fitness_a <- function(encoding){
  n <- length(encoding)
  attacking_pair <- 0
  for (i in 1:(n-1)){
    for (j in (i+1):n){
      if (diagonal_check(encoding[[i]], encoding[[j]])){
        attacking_pair <- attacking_pair + 1
      }
      if(row_check(encoding[[i]], encoding[[j]])){
        attacking_pair <- attacking_pair + 1
      }
    }
  }
  score <- 1-(attacking_pair / ((n*(n-1))/2))
```

```
  result_list <- list(attacking_pair = attacking_pair, score = score)
  return(result_list)
}
set.seed(12345)
genetic_algorithm_a <- function(fitness_assignment, n, mut_prob, p,
                                population_size = 20, max_iter = 4000){
  if (fitness_assignment == 1){
    population <- list()
    fitness_score_vector <- numeric()
    for (i in 1:population_size){
      encoding <- encoding_a(n)
      population <- append(population, list(encoding))
      fitness_score_vector[i] <- binary_fitness_a(encoding)$score
    }
    init_score <- sum(fitness_score_vector)/length(population)

    attacking_pair_vector <- numeric()
    for (iter in 1:max_iter){
      parents <- sample(1:population_size, size = 2, replace = FALSE)
      p1 <- population[[parents[1]]]
      p2 <- population[[parents[2]]]
      child <- crossover_a(p1, p2, p = p)
      victim_idx <- order(fitness_score_vector)[1]
      if (runif(1) <= mut_prob){
        child <- mutate_a(child)
      }
      population[[victim_idx]] <- child
      fitness_score_vector[victim_idx] <- binary_fitness_a(child)$score
      attacking_pair_vector[iter] <- attacking_pair_fitness_a(child)$attacking_pair
      if (sum(fitness_score_vector) == length(population)){
        break
      }
    }
    new_score <- sum(fitness_score_vector)/length(population)
    result <- list(init_score = init_score,
                   new_score = new_score,
                   fitness_score_vector = fitness_score_vector,
                   attacking_pair_vector = attacking_pair_vector,
                   new_population = population)
    return(result)
  }
  # ------------------------------------------------------------------------
  if (fitness_assignment == 2){
    population <- list()
    fitness_score_vector <- numeric()
    for (i in 1:population_size){
      encoding <- encoding_a(n)
      population <- append(population, list(encoding))
      fitness_score_vector[i] <- attacked_queen_a(encoding)$score
    }
    init_score <- sum(fitness_score_vector)/length(population)

    attacking_pair_vector <- numeric()
```

```
  for (iter in 1:max_iter){
    parents <- sample(1:population_size, size = 2, replace = FALSE)
    p1 <- population[[parents[1]]]
    p2 <- population[[parents[2]]]
    child <- crossover_a(p1, p2, p = p)
    victim_idx <- order(fitness_score_vector)[1]
    if (runif(1) <= mut_prob){
      child <- mutate_a(child)
    }
    population[[victim_idx]] <- child
    fitness_score_vector[victim_idx] <- attacked_queen_a(child)$score
    attacking_pair_vector[iter] <- attacking_pair_fitness_a(child)$attacking_pair
    if (sum(fitness_score_vector) == length(population)){
      break
    }
  }
  new_score <- sum(fitness_score_vector)/length(population)
  result <- list(init_score = init_score,
                 new_score = new_score,
                 fitness_score_vector = fitness_score_vector,
                 attacking_pair_vector = attacking_pair_vector,
                 new_population = population)
  return(result)
}
# ------------------------------------------------------------------------
if (fitness_assignment == 3){
  population <- list()
  fitness_score_vector <- numeric()
  for (i in 1:population_size){
    encoding <- encoding_a(n)
    population <- append(population, list(encoding))
    fitness_score_vector[i] <- attacking_pair_fitness_a(encoding)$score
  }
  init_score <- sum(fitness_score_vector)/length(population)

  attacking_pair_vector <- numeric()
  for (iter in 1:max_iter){
    parents <- sample(1:population_size, size = 2, replace = FALSE)
    p1 <- population[[parents[1]]]
    p2 <- population[[parents[2]]]
    child <- crossover_a(p1, p2, p = p)
    victim_idx <- order(fitness_score_vector)[1]
    if (runif(1) <= mut_prob){
      child <- mutate_a(child)
    }
    population[[victim_idx]] <- child
    fitness_score_vector[victim_idx] <- attacking_pair_fitness_a(child)$score
    attacking_pair_vector[iter] <- attacking_pair_fitness_a(child)$attacking_pair
    if (sum(fitness_score_vector) == length(population)){
      break
    }
  }
  new_score <- sum(fitness_score_vector)/length(population)
```

```
    result <- list(init_score = init_score,
                   new_score = new_score,
                   fitness_score_vector = fitness_score_vector,
                   attacking_pair_vector = attacking_pair_vector,
                   new_population = population)
    return(result)
  }
}
# encoding b ------------------------------------------------------------------
# binary representation
number2binary <- function(number, noBits) {
  binary_vector <- rev(as.numeric(intToBits(number)))
  binary_vector <- binary_vector[-(1:(length(binary_vector) - noBits))]
  return(binary_vector)
}

binary2number <- function(binary_vector) {
  binary_string <- paste(binary_vector, collapse = "")
  decimal_number <- as.integer(strtoi(binary_string, base = 2))
  return(decimal_number)
}

set.seed(12345)
encoding_b <- function(n){
  individual <- matrix(0, nrow = ceiling(log2(n)), ncol = n)
  queen_pos <- sample(1:n, replace = FALSE)
  for (col in 1:n){
    binary_pos <- number2binary(queen_pos[col], log2(n))
    individual[,col] <- binary_pos
  }
  return(individual)
}

# layout of the chessboard
generate_board_b <- function(encoded_matrix_b){
  n <- ncol(encoded_matrix_b)
  matrix_b <- matrix(0, nrow = n, ncol = n)
  for (i in 1:n){
    ifelse(binary2number(encoded_matrix_b[,i]) == 0,
         row_idx <- n,
         row_idx <- binary2number(encoded_matrix_b[,i]))
    matrix_b[row_idx, i] <- 1
  }
  return(matrix_b)
}
set.seed(12345)
crossover_b <- function(parent1, parent2, p){
  n <- ncol(parent1)
  parent1 <- encoding_b(n)
  parent2 <- encoding_b(n)

  child1 <- cbind(parent1[, 1:p], parent2[, (p+1):n])
  child2 <- cbind(parent2[, 1:p], parent1[, (p+1):n])
```

```
  if (runif(1) <= 0.5){
    return(child1)
  } else{
    return(child2)
  }
}
set.seed(12345)
mutate_b <- function(encoding){
  n <- ncol(encoding)
  select_col <- sample(1:n, size = 1)
  available_row_pos <- setdiff(1:n, binary2number(encoding[,select_col]))
  new_row_pos <- sample(available_row_pos, size = 1)
  encoding[, select_col] <- number2binary(new_row_pos, log2(n))
  return(encoding)
}
diagonal_check <- function(queen1, queen2) {
  abs(queen1[1] - queen2[1]) == abs(queen1[2] - queen2[2])
}

row_check <- function(queen1, queen2) {
  queen1[1] == queen2[1]
}
# --------------fitness function 1 ----------------------------------------------
binary_fitness_b <- function(encoding){
  n <- ncol(encoding)
  score <- 1
  for (i in 1:(n-1)){
    for (j in (i+1):n){
      if (diagonal_check(c(binary2number(encoding[,i]), i),
                         c(binary2number(encoding[,j]), j))){
        score <- 0
        break
      }
      if(row_check(c(binary2number(encoding[,i]), i),
                   c(binary2number(encoding[,j]), j))){
        score <- 0
        break
      }
    }
  }
  return(list(score = score))
}
# --------------fitness function 2 ----------------------------------------------
attacked_queen_b <- function(encoding){
  attacked <- 0
  n <- ncol(encoding)
  attacking_pair <- 0
  for (i in 1:(n-1)){
    for (j in (i+1):n){
      if ( diagonal_check(c(binary2number(encoding[,i]), i),
                          c(binary2number(encoding[,j]), j)) || row_check(c(binary2number(encoding[,i])
        attacked <- attacked + 1
        break
```

```
      }
    }
  }
  score <- 1-(attacked/n)
  return(list(attacked_queen = attacked, score = score))
}
# --------------fitness function 3 ----------------------------------------------
attacking_pair_fitness_b <- function(encoding){
  n <- ncol(encoding)
  attacking_pair <- 0
  for (i in 1:(n-1)){
    for (j in (i+1):n){
      if (diagonal_check(c(binary2number(encoding[,i]), i),
                         c(binary2number(encoding[,j]), j))){
        attacking_pair <- attacking_pair + 1
      }
      if(row_check(c(binary2number(encoding[,i]), i),
                   c(binary2number(encoding[,j]), j))){
        attacking_pair <- attacking_pair + 1
      }
    }
  }
  score <- 1-(attacking_pair / ((n*(n-1))/2))
  result_list <- list(attacking_pair = attacking_pair, score = score)
  return(result_list)
}
set.seed(12345)
genetic_algorithm_b <- function(fitness_assignment, n, mut_prob, p,
                                population_size = 20, max_iter = 4000){
  if (fitness_assignment == 1){
    population <- list()
    fitness_score_vector <- numeric()
    for (i in 1:population_size){
      encoding <- encoding_b(n)
      population <- append(population, list(encoding))
      fitness_score_vector[i] <- binary_fitness_b(encoding)$score #score calc
    }
    init_score <- sum(fitness_score_vector)/length(population)

    attacking_pair_vector <- numeric()
    for (iter in 1:max_iter){
      parents <- sample(1:population_size, size = 2, replace = FALSE)
      p1 <- population[[parents[1]]]
      p2 <- population[[parents[2]]]
      child <- crossover_b(p1, p2, p = p)
      victim_idx <- order(fitness_score_vector)[1]
      if (runif(1) <= mut_prob){
        child <- mutate_b(child)
      }
      population[[victim_idx]] <- child
      fitness_score_vector[victim_idx] <- binary_fitness_b(child)$score # score calc
      attacking_pair_vector[iter] <- attacking_pair_fitness_b(child)$attacking_pair
      if (sum(fitness_score_vector) == length(population)){
```

```
      break
    }
  }
  new_score <- sum(fitness_score_vector)/length(population)
  result <- list(init_score = init_score,
                 new_score = new_score,
                 fitness_score_vector = fitness_score_vector,
                 attacking_pair_vector = attacking_pair_vector,
                 new_population = population)
  return(result)
}
# ------------------------------------------------------------------------------
if (fitness_assignment == 2){
  population <- list()
  fitness_score_vector <- numeric()
  for (i in 1:population_size){
    encoding <- encoding_b(n)
    population <- append(population, list(encoding))
    fitness_score_vector[i] <- attacked_queen_b(encoding)$score #score calc
  }
  init_score <- sum(fitness_score_vector)/length(population)

  attacking_pair_vector <- numeric()
  for (iter in 1:max_iter){
    parents <- sample(1:population_size, size = 2, replace = FALSE)
    p1 <- population[[parents[1]]]
    p2 <- population[[parents[2]]]
    child <- crossover_b(p1, p2, p = p)
    victim_idx <- order(fitness_score_vector)[1]
    if (runif(1) <= mut_prob){
      child <- mutate_b(child)
    }
    population[[victim_idx]] <- child
    fitness_score_vector[victim_idx] <- attacked_queen_b(child)$score # score calc
    attacking_pair_vector[iter] <- attacking_pair_fitness_b(child)$attacking_pair
    if (sum(fitness_score_vector) == length(population)){
      break
    }
  }
  new_score <- sum(fitness_score_vector)/length(population)
  result <- list(init_score = init_score,
                 new_score = new_score,
                 fitness_score_vector = fitness_score_vector,
                 attacking_pair_vector = attacking_pair_vector,
                 new_population = population)
  return(result)
}
# ------------------------------------------------------------------------------
if (fitness_assignment == 3){
  population <- list()
  fitness_score_vector <- numeric()
  for (i in 1:population_size){
    encoding <- encoding_b(n)
```

```
      population <- append(population, list(encoding))
      fitness_score_vector[i] <- attacking_pair_fitness_b(encoding)$score #score calc
    }
    init_score <- sum(fitness_score_vector)/length(population)

    attacking_pair_vector <- numeric()
    for (iter in 1:max_iter){
      parents <- sample(1:population_size, size = 2, replace = FALSE)
      p1 <- population[[parents[1]]]
      p2 <- population[[parents[2]]]
      child <- crossover_b(p1, p2, p = p)
      victim_idx <- order(fitness_score_vector)[1]
      if (runif(1) <= mut_prob){
        child <- mutate_b(child)
      }
      population[[victim_idx]] <- child
      fitness_score_vector[victim_idx] <- attacking_pair_fitness_b(child)$score # score calc
      attacking_pair_vector[iter] <- attacking_pair_fitness_b(child)$attacking_pair
      if (sum(fitness_score_vector) == length(population)){
        break
      }
    }
    new_score <- sum(fitness_score_vector)/length(population)
    result <- list(init_score = init_score,
                   new_score = new_score,
                   fitness_score_vector = fitness_score_vector,
                   attacking_pair_vector = attacking_pair_vector,
                   new_population = population)
    return(result)
  }
}
# encoding c ------------------------------------------------------------------
set.seed(12345)
encoding_c <- function(n){
  individual <- matrix(0, nrow = 1, ncol = n)
  individual[1,] <- sample(1:n, replace = FALSE)
  return(individual)
}

generate_board_c <- function(encoded_matrix_c){
  n <- ncol(encoded_matrix_c)
  matrix_c <- matrix(0, nrow = n, ncol = n)
  for (i in 1:n){
    row_idx <- encoded_matrix_c[,i]
    matrix_c[row_idx ,i] <- 1
  }
  return(matrix_c)
}
set.seed(12345)
crossover_c <- function(parent1, parent2, p){
  n <- ncol(parent1)
  child1 <- matrix(0, nrow = 1, ncol = n)
  child2 <- matrix(0, nrow = 1, ncol = n)
```

```
  child1[1, ] <- c(parent1[, 1:p], parent2[, (p+1):n])
  child2[1, ] <- c(parent2[, 1:p], parent1[, (p+1):n])

  if (runif(1) <= 0.5){
    return(child1)
  } else{
    return(child2)
  }
}
mutate_c <- function(encoding){
  n <- ncol(encoding)
  select_col <- sample(1:n, size = 1)
  available_position <- setdiff(1:n, encoding[1,select_col])
  encoding[1, select_col] <- sample(available_position, size = 1)
  return(encoding)
}
diagonal_check <- function(queen1, queen2) {
  abs(queen1[1] - queen2[1]) == abs(queen1[2] - queen2[2])
}

row_check <- function(queen1, queen2) {
  queen1[1] == queen2[1]
}
# --------------fitness function 1 ---------------------------------------------
binary_fitness_c <- function(encoding){
  n <- ncol(encoding)
  score <- 1
  for (i in 1:(n-1)){
    for (j in (i+1):n){
      if (diagonal_check(c(encoding[,i], i), c(encoding[,j], j))){
        score <- 0
        break
      }
      if(row_check(c(encoding[,i], i),c(encoding[,j], j))){
        score <- 0
        break
      }
    }
  }
  return(list(score = score))
}
# --------------fitness function 2 ---------------------------------------------
attacked_queen_c <- function(encoding){
  attacked <- 0
  n <- ncol(encoding)
  attacking_pair <- 0
  for (i in 1:(n-1)){
    for (j in (i+1):n){
      if ( diagonal_check(c(encoding[,i], i), c(encoding[,j], j))
           ||
          row_check(c(encoding[,i], i),c(encoding[,j], j)) ){
        attacked <- attacked + 1
        break
```

```
      }
    }
  }
  score <- 1-(attacked/n)
  return(list(attacked_queen = attacked, score = score))
}
# --------------fitness function 3 ----------------------------------------------
attacking_pair_fitness_c <- function(encoding){
  n <- ncol(encoding)
  attacking_pair <- 0
  for (i in 1:(n-1)){
    for (j in (i+1):n){
      if (diagonal_check(c(encoding[,i], i), c(encoding[,j], j))){
        attacking_pair <- attacking_pair + 1
      }
      if(row_check(c(encoding[,i], i),c(encoding[,j], j))){
        attacking_pair <- attacking_pair + 1
      }
    }
  }
  score <- 1-(attacking_pair / ((n*(n-1))/2))
  result_list <- list(attacking_pair = attacking_pair, score = score)
  return(result_list)
}
set.seed(12345)
genetic_algorithm_c <- function(fitness_assignment, n, mut_prob, p,
                                population_size = 20, max_iter = 4000){
  if (fitness_assignment == 1){
    population <- list()
    fitness_score_vector <- numeric()
    for (i in 1:population_size){
      encoding <- encoding_c(n)
      population <- append(population, list(encoding))
      fitness_score_vector[i] <- binary_fitness_c(encoding)$score #score calc
    }
    init_score <- sum(fitness_score_vector)/length(population)

    attacking_pair_vector <- numeric()
    for (iter in 1:max_iter){
      parents <- sample(1:population_size, size = 2, replace = FALSE)
      p1 <- population[[parents[1]]]
      p2 <- population[[parents[2]]]
      child <- crossover_c(p1, p2, p)
      victim_idx <- order(fitness_score_vector)[1]
      if (runif(1) <= mut_prob){
        child <- mutate_c(child)
      }
      population[[victim_idx]] <- child
      fitness_score_vector[victim_idx] <- binary_fitness_c(child)$score # score calc
      attacking_pair_vector[iter] <- attacking_pair_fitness_c(child)$attacking_pair
      if (sum(fitness_score_vector) == length(population)){
        break
      }
```

```
  }
  new_score <- sum(fitness_score_vector)/length(population)
  result <- list(init_score = init_score,
                 new_score = new_score,
                 fitness_score_vector = fitness_score_vector,
                 attacking_pair_vector = attacking_pair_vector,
                 new_population = population)
  return(result)
}
# ------------------------------------------------------------------------
if (fitness_assignment == 2){
  population <- list()
  fitness_score_vector <- numeric()
  for (i in 1:population_size){
    encoding <- encoding_c(n)
    population <- append(population, list(encoding))
    fitness_score_vector[i] <- attacked_queen_c(encoding)$score #score calc
  }
  init_score <- sum(fitness_score_vector)/length(population)

  attacking_pair_vector <- numeric()
  for (iter in 1:max_iter){
    parents <- sample(1:population_size, size = 2, replace = FALSE)
    p1 <- population[[parents[1]]]
    p2 <- population[[parents[2]]]
    child <- crossover_c(p1, p2, p)
    victim_idx <- order(fitness_score_vector)[1]
    if (runif(1) <= mut_prob){
      child <- mutate_c(child)
    }
    population[[victim_idx]] <- child
    fitness_score_vector[victim_idx] <- attacked_queen_c(child)$score # score calc
    attacking_pair_vector[iter] <- attacking_pair_fitness_c(child)$attacking_pair
    if (sum(fitness_score_vector) == length(population)){
      break
    }
  }
  new_score <- sum(fitness_score_vector)/length(population)
  result <- list(init_score = init_score,
                 new_score = new_score,
                 fitness_score_vector = fitness_score_vector,
                 attacking_pair_vector = attacking_pair_vector,
                 new_population = population)
  return(result)
}
# ------------------------------------------------------------------------
if (fitness_assignment == 3){
  population <- list()
  fitness_score_vector <- numeric()
  for (i in 1:population_size){
    encoding <- encoding_c(n)
    population <- append(population, list(encoding))
    fitness_score_vector[i] <- attacking_pair_fitness_c(encoding)$score #score calc
```

```
    }
    init_score <- mean(fitness_score_vector)

    attacking_pair_vector <- numeric()
    for (iter in 1:max_iter){
      parents <- sample(1:population_size, size = 2, replace = FALSE)
      p1 <- population[[parents[1]]]
      p2 <- population[[parents[2]]]
      child <- crossover_c(p1, p2, p)
      victim_idx <- order(fitness_score_vector)[1]
      if (runif(1) <= mut_prob){
        child <- mutate_c(child)
      }
      population[[victim_idx]] <- child
      fitness_score_vector[victim_idx] <- attacking_pair_fitness_c(child)$score # score calc
      attacking_pair_vector[iter] <- attacking_pair_fitness_c(child)$attacking_pair
      if (sum(fitness_score_vector) == length(population)){
        break
      }
    }
    new_score <- mean(fitness_score_vector)
    result <- list(init_score = init_score,
                   new_score = new_score,
                   fitness_score_vector = fitness_score_vector,
                   attacking_pair_vector = attacking_pair_vector,
                   new_population = population)
    return(result)
  }
}
set.seed(12345)
par(mfrow = c(1, 2))
p1 <- genetic_algorithm_a(fitness_assignment = 2, n = 4, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 1)
p2 <- genetic_algorithm_a(fitness_assignment = 2, n = 4, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 2)

plot(p1$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n= 4, p = 1", ylim = c(0,6))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(p2$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 4, p = 2", ylim = c(0,6))
abline(h = 0, col = "red", lwd = 2)
grid()
set.seed(12345)
p2 <- genetic_algorithm_a(fitness_assignment = 2, n = 8, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 2)
p3 <- genetic_algorithm_a(fitness_assignment = 2, n = 8, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 3)
p4 <- genetic_algorithm_a(fitness_assignment = 2, n = 8, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 4)

plot(p2$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
```

```
     main = "n = 8, p = 2", ylim = c(0,20))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(p3$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 8, p = 3", ylim = c(0,20))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(p4$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 8, p = 4", ylim = c(0,20))
abline(h = 0, col = "red", lwd = 2)
grid()
set.seed(12345)
p4 <- genetic_algorithm_a(fitness_assignment = 2, n = 16, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 4)
p6 <- genetic_algorithm_a(fitness_assignment = 2, n = 16, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 6)
p8 <- genetic_algorithm_a(fitness_assignment = 2, n = 16, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 8)

plot(p4$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 16, p = 4", ylim = c(0,30))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(p6$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 16, p = 6", ylim = c(0,30))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(p8$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 16, p = 8", ylim = c(0,30))
abline(h = 0, col = "red", lwd = 2)
grid()
set.seed(12345)
par(mfrow = c(1, 2))
p1 <- genetic_algorithm_b(fitness_assignment = 2, n = 4, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 1)
p2 <- genetic_algorithm_b(fitness_assignment = 2, n = 4, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 2)

plot(p1$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n= 4, p = 1", ylim = c(0,6))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(p2$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 4, p = 2", ylim = c(0,6))
abline(h = 0, col = "red", lwd = 2)
grid()
set.seed(12345)
p2 <- genetic_algorithm_b(fitness_assignment = 2, n = 8, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 2)
p3 <- genetic_algorithm_b(fitness_assignment = 2, n = 8, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 3)
p4 <- genetic_algorithm_b(fitness_assignment = 2, n = 8, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 4)
```

```
plot(p2$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 8, p = 2", ylim = c(0,20))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(p3$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 8, p = 3", ylim = c(0,20))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(p4$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 8, p = 4", ylim = c(0,20))
abline(h = 0, col = "red", lwd = 2)
grid()
set.seed(12345)
p4 <- genetic_algorithm_b(fitness_assignment = 2, n = 16, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 4)
p6 <- genetic_algorithm_b(fitness_assignment = 2, n = 16, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 6)
p8 <- genetic_algorithm_b(fitness_assignment = 2, n = 16, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 8)

plot(p4$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 16, p = 4", ylim = c(0,30))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(p6$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 16, p = 6", ylim = c(0,30))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(p8$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 16, p = 8", ylim = c(0,30))
abline(h = 0, col = "red", lwd = 2)
grid()
set.seed(12345)
par(mfrow = c(1, 2))
p1 <- genetic_algorithm_c(fitness_assignment = 2, n = 4, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 1)
p2 <- genetic_algorithm_c(fitness_assignment = 2, n = 4, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 2)

plot(p1$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n= 4, p = 1", ylim = c(0,6))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(p2$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 4, p = 2", ylim = c(0,6))
abline(h = 0, col = "red", lwd = 2)
grid()
set.seed(12345)
p2 <- genetic_algorithm_c(fitness_assignment = 2, n = 8, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 2)
p3 <- genetic_algorithm_c(fitness_assignment = 2, n = 8, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 3)
p4 <- genetic_algorithm_c(fitness_assignment = 2, n = 8, mut_prob = 0.5,
```

```
                          population_size = 20, max_iter = 4000, p = 4)

plot(p2$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 8, p = 2", ylim = c(0,20))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(p3$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 8, p = 3", ylim = c(0,20))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(p4$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 8, p = 4", ylim = c(0,20))
abline(h = 0, col = "red", lwd = 2)
grid()
set.seed(12345)
p4 <- genetic_algorithm_c(fitness_assignment = 2, n = 16, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 4)
p6 <- genetic_algorithm_c(fitness_assignment = 2, n = 16, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 6)
p8 <- genetic_algorithm_c(fitness_assignment = 2, n = 16, mut_prob = 0.5,
                          population_size = 20, max_iter = 4000, p = 8)

plot(p4$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 16, p = 4", ylim = c(0,30))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(p6$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 16, p = 6", ylim = c(0,30))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(p8$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "n = 16, p = 8", ylim = c(0,30))
abline(h = 0, col = "red", lwd = 2)
grid()
genetic_algorithm <- function(n, mut_prob, encoding_type, fitness_type){
  if (encoding_type == "a"){
    if (fitness_type == 1){
      if (n == 4){
        return(genetic_algorithm_a(fitness_assignment = 1, n, mut_prob, p = 2))
      }
      if (n == 8){
        return(genetic_algorithm_a(fitness_assignment = 1, n, mut_prob, p = 4))
      }
      if (n == 16){
        return(genetic_algorithm_a(fitness_assignment = 1, n, mut_prob, p = 4))
      }
    }
    if (fitness_type == 2){
      if (n == 4){
        return(genetic_algorithm_a(fitness_assignment = 2, n, mut_prob, p = 2))
      }
      if (n == 8){
        return(genetic_algorithm_a(fitness_assignment = 2, n, mut_prob, p = 4))
```

```
    }
    if (n == 16){
      return(genetic_algorithm_a(fitness_assignment = 2, n, mut_prob, p = 4))
    }
  }
  if (fitness_type == 3){
    if (n == 4){
      return(genetic_algorithm_a(fitness_assignment = 3, n, mut_prob, p = 2))
    }
    if (n == 8){
      return(genetic_algorithm_a(fitness_assignment = 3, n, mut_prob, p = 4))
    }
    if (n == 16){
      return(genetic_algorithm_a(fitness_assignment = 3, n, mut_prob, p = 4))
    }
  }
}
if (encoding_type == "b"){
  if (fitness_type == 1){
    if (n == 4){
      return(genetic_algorithm_b(fitness_assignment = 1, n, mut_prob, p = 1))
    }
    if (n == 8){
      return(genetic_algorithm_b(fitness_assignment = 1, n, mut_prob, p = 4))
    }
    if (n == 16){
      return(genetic_algorithm_b(fitness_assignment = 1, n, mut_prob, p = 8))
    }
  }
  if (fitness_type == 2){
    if (n == 4){
      return(genetic_algorithm_b(fitness_assignment = 2, n, mut_prob, p = 1))
    }
    if (n == 8){
      return(genetic_algorithm_b(fitness_assignment = 2, n, mut_prob, p = 4))
    }
    if (n == 16){
      return(genetic_algorithm_b(fitness_assignment = 2, n, mut_prob, p = 8))
    }
  }
  if (fitness_type == 3){
    if (n == 4){
      return(genetic_algorithm_b(fitness_assignment = 3, n, mut_prob, p = 1))
    }
    if (n == 8){
      return(genetic_algorithm_b(fitness_assignment = 3, n, mut_prob, p = 4))
    }
    if (n == 16){
      return(genetic_algorithm_b(fitness_assignment = 3, n, mut_prob, p = 8))
    }
  }
}
if (encoding_type == "c"){
```

```
    if (fitness_type == 1){
      if (n == 4){
        return(genetic_algorithm_c(fitness_assignment = 1, n, mut_prob, p = 2))
      }
      if (n == 8){
        return(genetic_algorithm_c(fitness_assignment = 1, n, mut_prob, p = 4))
      }
      if (n == 16){
        return(genetic_algorithm_c(fitness_assignment = 1, n, mut_prob, p = 6))
      }
    }
    if (fitness_type == 2){
      if (n == 4){
        return(genetic_algorithm_c(fitness_assignment = 2, n, mut_prob, p = 2))
      }
      if (n == 8){
        return(genetic_algorithm_c(fitness_assignment = 2, n, mut_prob, p = 4))
      }
      if (n == 16){
        return(genetic_algorithm_c(fitness_assignment = 2, n, mut_prob, p = 6))
      }
    }
    if (fitness_type == 3){
      if (n == 4){
        return(genetic_algorithm_c(fitness_assignment = 3, n, mut_prob, p = 2))
      }
      if (n == 8){
        return(genetic_algorithm_c(fitness_assignment = 3, n, mut_prob, p = 4))
      }
      if (n == 16){
        return(genetic_algorithm_c(fitness_assignment = 3, n, mut_prob, p = 6))
      }
    }
  }
}
set.seed(123)
start_time <- Sys.time()
n_vector <- c(4,8,16)
mut_prob_vector <- c(0.1, 0.5, 0.9)
fitness_assignment_vector <- c(1,2,3)
for(n in n_vector){
  cat("n is", n, "\n")
  for (mut_prob in mut_prob_vector){
    for(fitness_assignment in fitness_assignment_vector){
      result <- genetic_algorithm(n, mut_prob, encoding_type = "a", fitness_assignment)
      cat("Fitness function is", fitness_assignment, "-- mutation probaility is",
          mut_prob, "-- score is", result$new_score, "\n")
    }
  }
}
end_time <- Sys.time()
elapsed_time <- end_time - start_time
cat("Run time:", elapsed_time, "\n")
```

```r
set.seed(12345)

fitness1 <- genetic_algorithm(n = 8, mut_prob = 0.9, encoding_type = "a", fitness_type = 1)
fitness2 <- genetic_algorithm(n = 8, mut_prob = 0.9, encoding_type = "a", fitness_type = 2)
fitness3 <- genetic_algorithm(n = 8, mut_prob = 0.9, encoding_type = "a", fitness_type = 3)

plot(fitness1$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "Fitness 1", ylim = c(0,28))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(fitness2$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "Fitness 2", ylim = c(0,28))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(fitness3$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "Fitness 3", ylim = c(0,28))
abline(h = 0, col = "red", lwd = 2)
grid()
set.seed(1234)
y1 <- genetic_algorithm(n = 4, mut_prob = 0.9, encoding_type = "a", fitness_type = 3)
y2 <- genetic_algorithm(n = 8, mut_prob = 0.9, encoding_type = "a", fitness_type = 3)
y3 <- genetic_algorithm(n = 16, mut_prob = 0.9, encoding_type = "a", fitness_type = 3)

plot(y1$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "4 queens, mutation prob = 0.9, fitness type = 3", ylim = c(0,6))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(y2$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "8 queens, mutation prob = 0.9, fitness type = 3", ylim = c(0,20))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(y3$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "16 queens, mutation prob = 0.9, fitness type = 3", ylim = c(0,30))
abline(h = 0, col = "red", lwd = 2)
grid()
cat("Initial score:", y1$init_score, "\n")
cat("Legal configuration (n = 4):\n")
if (any(y1$fitness_score_vector == 1)){
  best_indices <- which(y1$fitness_score_vector == 1)
  generate_board_a(y1$new_population[[best_indices[1]]])
} else{
  cat("There's no legal configuration\n")
}

cat("Initial score:", y2$init_score, "\n")
cat("Legal configuration (n = 8):\n")
if (any(y2$fitness_score_vector == 1)){
  best_indices <- which(y2$fitness_score_vector == 1)
  generate_board_a(y2$new_population[[best_indices[1]]])
} else{
  cat("There's no legal configuration\n")
}
```

```r
cat("Initial score:", y3$init_score, "\n")
cat("Legal configuration (n = 16):\n")
if (any(y3$fitness_score_vector == 1)){
  best_indices <- which(y3$fitness_score_vector == 1)
  generate_board_a(y3$new_population[[best_indices[1]]])
} else{
  cat("There's no legal configuration\n")
}
set.seed(123)
start_time <- Sys.time()
n_vector <- c(4,8,16)
mut_prob_vector <- c(0.1, 0.5, 0.9)
fitness_assignment_vector <- c(1,2,3)
for(n in n_vector){
  cat("n is", n, "\n")
  for (mut_prob in mut_prob_vector){
    for(fitness_assignment in fitness_assignment_vector){
      result <- genetic_algorithm(n, mut_prob, encoding_type = "b", fitness_assignment)
      cat("Fitness function is", fitness_assignment, "-- mutation probaility is",
          mut_prob, "-- score is", result$new_score, "\n")
    }
  }
}
end_time <- Sys.time()
elapsed_time <- end_time - start_time
cat("Run time:", elapsed_time, "\n")
set.seed(12345)
fitness1 <- genetic_algorithm(n = 8, mut_prob = 0.5, encoding_type = "b", fitness_type = 1)
fitness2 <- genetic_algorithm(n = 8, mut_prob = 0.5, encoding_type = "b", fitness_type = 2)
fitness3 <- genetic_algorithm(n = 8, mut_prob = 0.5, encoding_type = "b", fitness_type = 3)

plot(fitness1$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "Fitness 1", ylim = c(0,28))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(fitness2$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "Fitness 2", ylim = c(0,28))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(fitness3$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "Fitness 3", ylim = c(0,28))
abline(h = 0, col = "red", lwd = 2)
grid()
set.seed(123)
y1 <- genetic_algorithm(n = 4, mut_prob = 0.9, encoding_type = "b", fitness_type = 2)
y2 <- genetic_algorithm(n = 8, mut_prob = 0.9, encoding_type = "b", fitness_type = 2)
y3 <- genetic_algorithm(n = 16, mut_prob = 0.9, encoding_type = "b", fitness_type = 2)

plot(y1$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "4 queens, mutation prob = 0.9, fitness type = 2", ylim = c(0,6))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(y2$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
```

```
     main = "8 queens, mutation prob = 0.9, fitness type = 2", ylim = c(0,20))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(y3$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "16 queens, mutation prob = 0.9, fitness type = 2", ylim = c(0,30))
abline(h = 0, col = "red", lwd = 2)
grid()
cat("Initial score:", y1$init_score, "\n")
cat("Legal configuration (n = 4):\n")
if (any(y1$fitness_score_vector == 1)){
  best_indices <- which(y1$fitness_score_vector == 1)
  generate_board_b(y1$new_population[[best_indices[1]]])
} else{
  cat("There's no legal configuration\n")
}

cat("Initial score:", y2$init_score, "\n")
cat("Legal configuration (n = 8):\n")
if (any(y2$fitness_score_vector == 1)){
  best_indices <- which(y2$fitness_score_vector == 1)
  generate_board_b(y2$new_population[[best_indices[1]]])
} else{
  cat("There's no legal configuration\n")
}

cat("Initial score:", y3$init_score, "\n")
cat("Legal configuration (n = 16):\n")
if (any(y3$fitness_score_vector == 1)){
  best_indices <- which(y3$fitness_score_vector == 1)
  generate_board_b(y3$new_population[[best_indices[1]]])
} else{
  cat("There's no legal configuration\n")
}
set.seed(123)
start_time <- Sys.time()
n_vector <- c(4,8,16)
mut_prob_vector <- c(0.1, 0.5, 0.9)
fitness_assignment_vector <- c(1,2,3)
for(n in n_vector){
  cat("n is", n, "\n")
  for (mut_prob in mut_prob_vector){
    for(fitness_assignment in fitness_assignment_vector){
      result <- genetic_algorithm(n, mut_prob, encoding_type = "c", fitness_assignment)
      cat("Fitness function is", fitness_assignment, "-- mutation probaility is",
          mut_prob, "-- score is", result$new_score, "\n")
    }
  }
}
end_time <- Sys.time()
elapsed_time <- end_time - start_time
cat("Run time:", elapsed_time, "\n")
set.seed(12345)
fitness1 <- genetic_algorithm(n = 8, mut_prob = 0.9, encoding_type = "c", fitness_type = 1)
```

```r
fitness2 <- genetic_algorithm(n = 8, mut_prob = 0.9, encoding_type = "c", fitness_type = 2)
fitness3 <- genetic_algorithm(n = 8, mut_prob = 0.9, encoding_type = "c", fitness_type = 3)

plot(fitness1$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "Fitness 1", ylim = c(0,28))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(fitness2$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "Fitness 2", ylim = c(0,28))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(fitness3$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "Fitness 3", ylim = c(0,28))
abline(h = 0, col = "red", lwd = 2)
grid()
set.seed(123)
y1 <- genetic_algorithm(n = 4, mut_prob = 0.9, encoding_type = "c", fitness_type = 3)
y2 <- genetic_algorithm(n = 8, mut_prob = 0.9, encoding_type = "c", fitness_type = 3)
y3 <- genetic_algorithm(n = 16, mut_prob = 0.9, encoding_type = "c", fitness_type = 3)

plot(y1$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "4 queens, mutation prob = 0.9, fitness type = 3", ylim = c(0,6))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(y2$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "8 queens, mutation prob = 0.9, fitness type = 3", ylim = c(0,20))
abline(h = 0, col = "red", lwd = 2)
grid()
plot(y3$attacking_pair_vector, type = "l", ylab="Attacking pair", xlab = "Iteration",
     main = "16 queens, mutation prob = 0.9, fitness type = 3", ylim = c(0,30))
abline(h = 0, col = "red", lwd = 2)
grid()
cat("Initial score:", y1$init_score, "\n")
cat("Legal configuration (n = 4):\n")
if (any(y1$fitness_score_vector == 1)){
  best_indices <- which(y1$fitness_score_vector == 1)
  generate_board_c(y1$new_population[[best_indices[1]]])
} else{
  cat("There's no legal configuration\n")
}

cat("Initial score:", y2$init_score, "\n")
cat("Legal configuration (n = 8):\n")
if (any(y2$fitness_score_vector == 1)){
  best_indices <- which(y2$fitness_score_vector == 1)
  generate_board_c(y2$new_population[[best_indices[1]]])
} else{
  cat("There's no legal configuration\n")
}

cat("Initial score:", y3$init_score, "\n")
cat("Legal configuration (n = 16):\n")
if (any(y3$fitness_score_vector == 1)){
```

```r
  best_indices <- which(y3$fitness_score_vector == 1)
  generate_board_c(y3$new_population[[best_indices[1]]])
} else{
  cat("There's no legal configuration\n")
}
# -------------------------- QUESTION 2 --------------------------------------
# Data manipulation
df2_init <- read.csv("censoredproc.csv")
split_data <- strsplit(as.character(df2_init$time.cens), ";")
split_matrix <- do.call(rbind, split_data)
df2 <- as.data.frame(split_matrix)
colnames(df2) <- c("time", "cens")

df2$time <- as.numeric(df2$time)
df2$cens <- as.factor(df2$cens)

cens1 <- subset(df2, cens == 1) # uncensored -  observed
cens2 <- subset(df2, cens == 2) # censored
hist(df2$time, main = "Histogram of All Data", freq = FALSE, breaks = 20, xlab = "Time")
hist(cens1$time, main = "Histogram of Uncensored Data", freq = FALSE, breaks = 20, xlab = "Time")
hist(cens2$time, main = "Histogram of Censored Data", freq = FALSE, breaks = 20, xlab = "Time")
# Loglikelihood function
llik_fnc <- function(lambda, cens1_time, cens2_time){
  t <- cens2_time
  numerator <- lambda * ((((-t/lambda)-(1/lambda^2))*exp(-lambda*t)) + (1/lambda^2))
  denominator <- 1-exp(-lambda*t)
  censored_x <- numerator/denominator # find the expected failure time from censored data

  a <- lambda*exp(-lambda*censored_x)
  b <- 1-exp(-lambda*t)
  llik_censored <- log(a/b)
  llik_uncensored <- log(lambda*exp(-lambda*cens1_time))
  loglikelihood <- sum(llik_uncensored) + sum(llik_censored)
  return(loglikelihood)
}
lambda_values <- seq(0.005, 10, by = 0.1)
lamda_values_y <- sapply(lambda_values, function(lambda) llik_fnc(lambda, cens1$time, cens2$time))
plot(lambda_values, lamda_values_y, type = "l", col = "red", lwd = 2, main = "Loglikelihood function", y
grid()
EM_algorithm <- function(lambda, cens1_time, cens2_time){
  negative_llik_fnc <- function(lambda, cens1_time, cens2_time){
    return(-llik_fnc(lambda, cens1_time, cens2_time))
  }

  result <- optim(par = lambda,
                  fn = negative_llik_fnc,
                  cens1_time = cens1_time, cens2_time = cens2_time,
                  method = "Brent",
                  lower = 0.001, upper = 10)
  optimal_lambda <- result$par
  return(optimal_lambda)
}
```

```r
optimal_lambda <- EM_algorithm(100, cens1$time, cens2$time)
cat("Optimal lambda:", optimal_lambda, "\n")
hist(df2$time, main = "Histogram of all data", freq = FALSE, breaks = 20, xlab = "Time")
x <- seq(0, max(df2$time), length.out = 100)
y <- dexp(x, rate = optimal_lambda)
lines(x, y, col = "red", lwd = 2)
legend("topright", legend = "exp(lambda = 1.014323)", col = "red", lwd = 2)
hist(cens1$time, breaks = 10, main = "Histogram of uncensored data", freq = FALSE, xlab = "Time")
x <- seq(0, max(cens1$time), length.out = 100)
y <- dexp(x, rate = optimal_lambda)
lines(x, y, col = "blue", lwd = 2)
legend("topright", legend = "exp(lambda = 1.014323)", col = "blue", lwd = 2)
lambda_hat <- nrow(cens1) / sum(cens1$time)
cat("MLE on uncensored data:", lambda_hat, "\n")
set.seed(12345)
B <- 1000
MLE_vector <- numeric(length = B)
EM_vector <- numeric(length = B)

for (b in 1:B){
  # Simulate data from exponential distribution
  generated_data <- rexp(nrow(df2), rate = optimal_lambda)
  uncensored_data <- sample(generated_data, size = nrow(cens1))
  censored_data_init <- generated_data[!generated_data %in% uncensored_data]

  # Find the censor time
  x <- censored_data_init # failure time
  above <- -optimal_lambda*(((-x/optimal_lambda) - (1/optimal_lambda^2)) * exp(-x*optimal_lambda))
  below <- exp(-x*optimal_lambda)
  censored_data <- above/below # censor time
  MLE_vector[b] <- length(uncensored_data) / sum(uncensored_data)
  EM_vector[b] <- EM_algorithm(100, uncensored_data, censored_data)
}
par(mfrow = c(1, 2))
hist(MLE_vector, main = "Histogram of MLE method", freq = FALSE, xlab = "Lambda", breaks = 20)
abline(v = 1.004758 , col = "red", lwd = 2) #mle lambda from the original dataset

hist(EM_vector, main = "Histogram of EM method", freq = FALSE, xlab = "Lambda", breaks = 20)
abline(v = 1.014323, col = "red", lwd = 2) #optimal lambda from the original dataset
mle_bias <- mean(MLE_vector) - lambda_hat
em_bias <- mean(EM_vector) - optimal_lambda

cat("Bias of MLE:", mle_bias, "\n")
cat("Bias of EM:", em_bias, "\n")

cat("Variance of MLE:", var(MLE_vector), "\n")
cat("Variance of EM:", var(EM_vector), "\n")
```