# Intelligent Systems Seminar Assignment 1

Jaka Milavec, Mateja Cerina

22.11.2021

## 1 Problem defnition

The goal of the game is to arrange the numbers and mathematical operators in such a way that the result of the equation comes as close to the target number as possible.

First we save the numbers and operators as a string vector. We set the target variable, and then calculate the length of the string vector and how many numbers and operators the equations we generate, are going to have.

For our program to work we must write the exp string vector in the right order. The numbers come first, and four operators follow. If we want to use less operators, we have to change the value of the `diffOperators` variable accordingly.

```
exp <- c("10", "25", "100", "5", "3", "+" ,"-", "/", "*")
target <- 2512

expSize <- length(exp)
diffOperators <- 4
numbNumbers <- expSize - diffOperators
numbOperators <- numbNumbers - 1
equationLength <- (numbNumbers * 2) - 1
```

## 2 Population

To generate a population we create a matrix that has `popSize` number of rows and the length of the equation as the number of columns. For each row in the matrix we then generate an individual.

To create valid equations, that follow the rules of our game, we select each index that represents a number once, and save it in the numbers vector. Then we generate the indexes for the operators. These can be repeated. Next we fill the vector that represents our equation and will be returned to the population function. So we can later evaluate the equation, we put the numbers in the odd positions and the operators in the even positions of the vector.

```
myInitPopulation <- function(object) {

  p <- matrix(nrow = object@popSize, ncol = object@upper)

  for (i in 1:nrow(p)) {

    t <- generateIndividual()

    p[i,] <- as.vector(t)
  }

  return(p)
}

generateIndividual <- function() {

  numbers <- c(sample(1:numbNumbers, numbNumbers))
  opIndex <- expSize - diffOperators + 1
  operators <- c(sample(opIndex:expSize, numbOperators, replace = T))

  t <- integer(equationLength)

  ix1 = 1
  ix2 = 1

  for (j in 1:equationLength) {

    if (j %% 2 != 0) {
      t[j] = numbers[ix1]
      ix1 = ix1 + 1
    }
    else {
      t[j] = operators[ix2]
      ix2 = ix2 + 1
    }

  }

  return(t)
}
```

## 3 Fitness function

The fitness function accepts one row of our generated population matrix. To get the actual equation, we first index our `exp` string vector by the `x` vector that gets passed into the function. Then we calculate the equation.

To check the fitness, we subtract the target number from the result, using the absolute value function in case our result is smaller than the target number.

Because we want to maximize the result of the fitness function, we divide 1 with the result of the subtraction. To avoid division by zero, if the result of our equation is the same as the target number, we add 1 in the denominator.

```
f <- function(x) {

  a <- exp[x]

  str = paste(a, collapse='')
  y <- eval(parse(text=str))

  fit <- 1 / (1 + abs(y - target))

  return(fit)
}
```

## 4 Mutation

For our mutation function we chose to implement three different versions. One that swaps operands, one that swaps operators and one that is based on the lecture slides. The latter one swaps a part of the equation with a randomly generated equation of smaller size.

### 4.1 Swap operands

This function creates a sequence of odd indexes. Using the sample function, we take two of the generated indexes and use them to index the mutant and the parent, and perform the swap.

```
myMutation_swapOperands <- function(object, parent) {

  mutate <- parent <- as.vector(object@population[parent,])
  n <- length(parent)

  odd <- seq(1,n, by=2)

  m <- sample(odd, size = 2)

  mutate[m[1]] <- parent[m[2]]
  mutate[m[2]] <- parent[m[1]]

  return(mutate)
}
```

### 4.2 Swap operators

This function works the same as the previous one, except we create a sequence of even indexes. This results in the swapping of numbers instead of operators.

```
myMutation_swapOperators <- function(object, parent) {

  mutate <- parent <- as.vector(object@population[parent,])
  n <- length(parent)

  even <- seq(2,n, by=2)

  m <- sample(even, size = 2)

  mutate[m[1]] <- parent[m[2]]
  mutate[m[2]] <- parent[m[1]]

  return(mutate)
}
```

### 4.3 Replace subtree

This function is based on the lecture slides. It selects a position for the replacement, and then calls a function to generate a subtree of appropriate size.

It is a simpler version of the one from the lectures, since it always chooses an odd index for replacement.

```
myMutation_replaceSubTree <- function(object, parent) {

  mutate <- parent <- as.vector(object@population[parent,])

  n <- length(parent)

  myIndex <- c(3, 5, 7)
  ix <- sample(1:3, 1, replace=F)
  randValue <- myIndex[ix]

  size <- randValue

  sIx <- seq(1,n - 3, by=2)
  startIx <- sample(sIx, size = 1)

  if (size + startIx > equationLength) {
    size = equationLength - startIx

    if (size %% 2 == 0) {
      size <- size - 1
    }
  }

  subTree <- generateSubTree(size)

  ixST = 1

  for (i in startIx:startIx + size) {
    mutate[i] <- subTree[ixST]
    ixST <- ixST + 1
  }

  return(mutate)
}
```

```
generateSubTree <- function(size) {

  oSize <- floor(size / 2)
  nSize <- oSize + 1
  opIndex <- expSize - diffOperators + 1

  numb <- sample(1:numbNumbers, nSize, replace=F)
  operator <- sample(opIndex:expSize, oSize, replace=T)

  equation <- integer(size)

  ixN = 1
  ixO = 1

  for (i in 1:size) {

    if (i %% 2 != 0) {

      equation[i] <- numb[ixN]
      ixN <- ixN + 1
    }
    else {

      equation[i] <- operator[ixO]
      ixO <- ixO + 1
    }

  }

  return(equation)
}
```

## 5 Crossover

Firstly we implemented the single point crossover. The function selects a crossover point and performs the operation based on it. One child gets the first part from the first parent and the second part from the second parent, the other child gets the first part from the second parent and the second part from the first parent. To ensure that each number is still only used once in each equation, the crossover is performed on operator indexes only.

```
sp_crossover <- function(object, parents) {

  fitness <- object@fitness[parents]
  parents <- object@population[parents,,drop = FALSE]
  n <- ncol(parents)
  children <- matrix(as.double(NA), nrow = 2, ncol = n)
  fitnessChildren <- rep(NA, 2)
  crossOverPoint <- sample(0:n, size = 1)

  if (crossOverPoint == 0) {

    children[1:2,] <- parents[2:1,]
    fitnessChildren[1:2] <- fitness[2:1]
  }
  else if (crossOverPoint == n) {

    children <- parents
    fitnessChildren <- fitness
  }
  else {

    overPoint <- 0

    for (i in 1:n) {

      if (i >= crossOverPoint) {

        overPoint <- 1
      }

      if (!(i %% 2) || !overPoint) {

        children[1,i]=parents[1,i]
        children[2,i]=parents[2,i]
      }
      else {

        children[1,i]=parents[2,i]
        children[2,i]=parents[1,i]
      }
    }
  }

  out <- list(children = children, fitness = fitnessChildren)
  return(out)
}
```

Our second crossover function creates children that are a mix of their parents. It gives one child the numbers of the first parent and the operators of the second, and the second child the numbers of the second parent and the operators of the first.

```
mix_crossover <- function(object, parents) {

  fitness <- object@fitness[parents]
  parents <- object@population[parents,,drop = FALSE]
  n <- ncol(parents)

  parent1 <- parents[1,]
  parent2 <- parents[2,]

  child1 <- integer(equationLength)
  child2 <- integer(equationLength)

  for (i in 1:equationLength) {

    if (i %% 2 != 0) {

      child1[i] <- parent1[i]
      child2[i] <- parent2[i]
    }
    else if (i %% 2 == 0) {

      child1[i] <- parent2[i]
      child2[i] <- parent1[i]
    }

  }

  children <- matrix(c(child1,child2), 2, byrow=TRUE)
  out <- list(children = children, fitness = rep(NA,2))

  return(out)
}
```

The third crossover is a version of the uniform crossover. The first child is a copy of the first parent and the second child is the copy of the second parent. Then we decide for each operator, based on a 50/50 chance, if we switch it with the operator of the other parent.

```
uniform_crossover <- function(object, parents) {

  fitness <- object@fitness[parents]
  parents <- object@population[parents,,drop = FALSE]
  n <- ncol(parents)

  parent1 <- parents[1,]
  parent2 <- parents[2,]

  child1 <- parent1
  child2 <- parent2

  for (i in 1:equationLength) {

    if (i %% 2 == 0) {

      flip1 <- sample(0:1, 1)
      flip2 <- sample(0:1, 1)

      if (flip1 == 1) {
        child1[i] <- parent2[i]
      }

      if (flip2 == 1) {
        child2[i] <- parent1[i]
      }
    }
  }

  children <- matrix(c(child1,child2), 2, byrow=TRUE)
  out <- list(children = children, fitness = rep(NA,2))

  return(out)
}
```

Our last crossover function is the two point crossover. As the name suggests, we select two points, where the crossover will occur. The first child gets the operators from the second parent on the indexes between the two crossover points, and the second one gets the operators from the first parent on the indexes between the two crossover points.

```
twoPoint_crossover <- function(object, parents) {

  fitness <- object@fitness[parents]
  parents <- object@population[parents,,drop = FALSE]
  n <- ncol(parents)

  children <- matrix(as.double(NA), nrow = 2, ncol = n)
  fitnessChildren <- rep(NA, 2)
  crossOverPoint <- sample(0:n, size = 2)

  cop1 = min(crossOverPoint)
  cop2 = max(crossOverPoint)

  for (i in 1:n) {

    if (!(i %% 2) && i >= cop1 && i <= cop2) {

      children[1,i]=parents[2,i]
      children[2,i]=parents[1,i]

    } else {

      children[1,i]=parents[1,i]
      children[2,i]=parents[2,i]
    }
  }

  out <- list(children = children, fitness = fitnessChildren)

  return(out)
}
```

## 6 Random search

For our random search function we need to generate individuals. We use the same code we used for generating individuals to fill our population.

```
generateIndividual <- function() {

  numbers <- c(sample(1:numbNumbers, numbNumbers))
  opIndex <- expSize - diffOperators + 1
  operators <- c(sample(opIndex:expSize, numbOperators, replace = T))

  t <- integer(equationLength)

  ix1 = 1
  ix2 = 1

  for (j in 1:equationLength) {

    if (j %% 2 != 0) {
      t[j] = numbers[ix1]
      ix1 = ix1 + 1
    }
    else {
      t[j] = operators[ix2]
      ix2 = ix2 + 1
    }

  }

  return(t)
}
```

The random search function is fairly simple. It keeps generating individuals and calculating the fitness function, until the result of the fitness function equals 1. When that occurs, we know we have found the solution to our problem and we can stop the time.

```
randomSearch <- function() {

  fit <- 0
  startTime_rs <- proc.time()

  while (fit != 1) {

    individual <- generateIndividual()
    fit <- f(individual)
  }

  time_rs <- proc.time() - startTime_rs
  print(time_rs)
}
```

## 7 Evaluation

This is the plot we get if we run the ga function with the two point crossover on the equation `exp <- c("10", "25", "100", "5", "3", "+" ,"-", "/", "*")` with the target number `target <- 2512`.

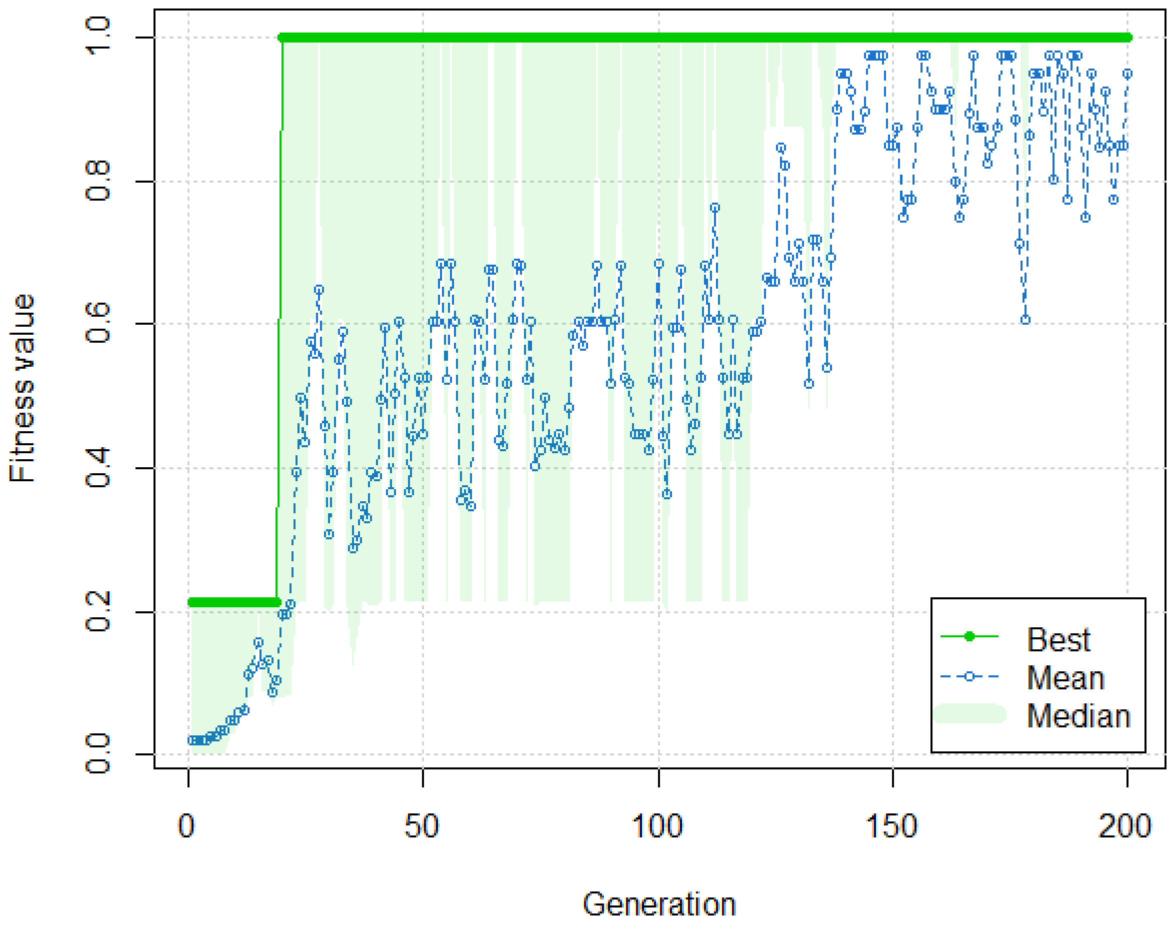


Figure 1: Two point crossover

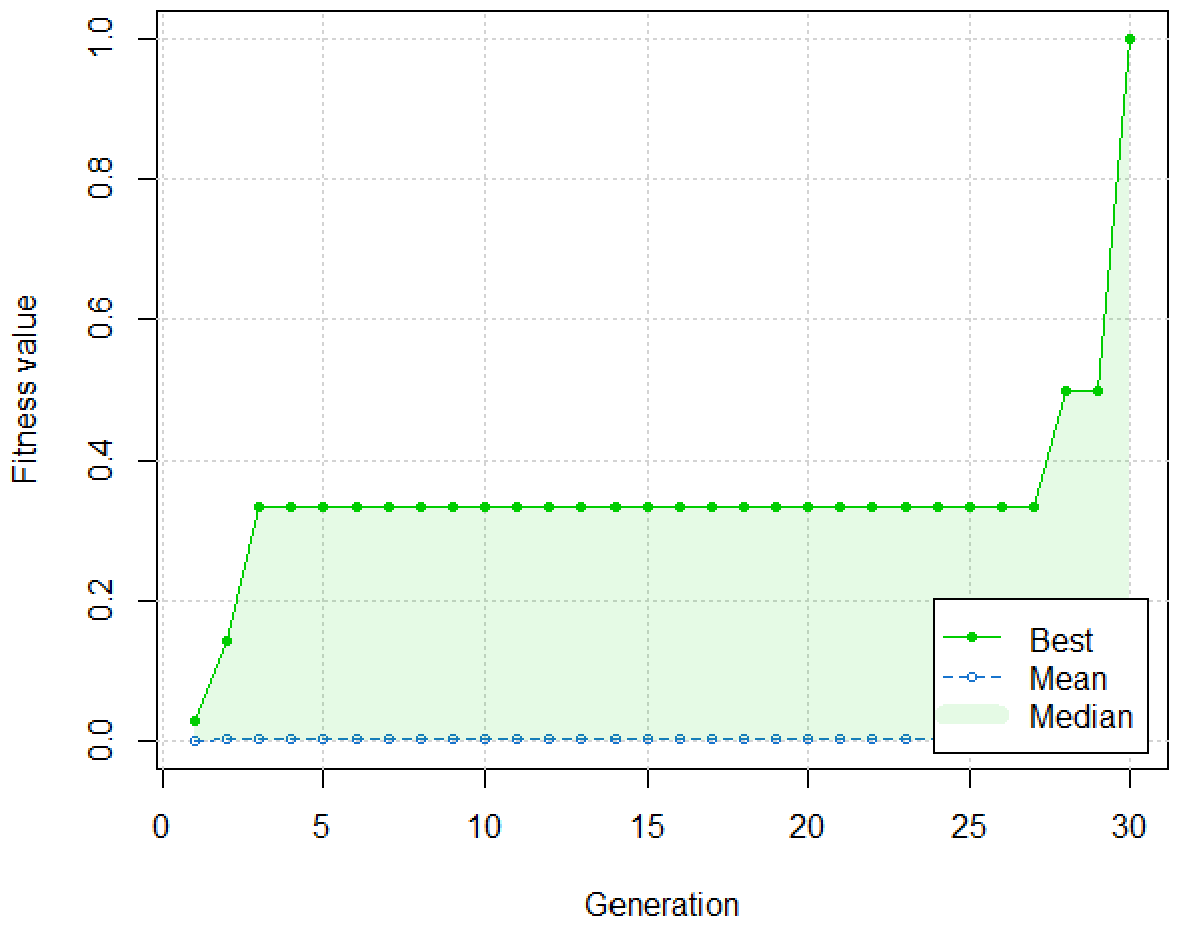
This is how we run the ga() function.

```
startTime_ga <- proc.time()

GA <- ga(
      type = "permutation",
      fitness = f,
      lower = 1,
      upper = numbNumbers * 2 - 1,
      popSize = 10,
      maxiter = 200,
      population = myInitPopulation,
      mutation = myMutation_swapOperands,
      #mutation = myMutation_swapOperators,
      #mutation = myMutation_replaceSubTree,
      pmutation = 0.05,
      pcrossover = 0.8,
      #crossover = sp_crossover,
      #crossover = mix_crossover,
      crossover = twoPoint_crossover,
      #crossover = uniform_crossover,
      elitism = 3,
      keepBest = TRUE)

time_ga <- proc.time() - startTime_ga
print(time_ga)

plot(GA)

exp[GA@bestSol[[length(GA@summary[,1])]][1,]]
```

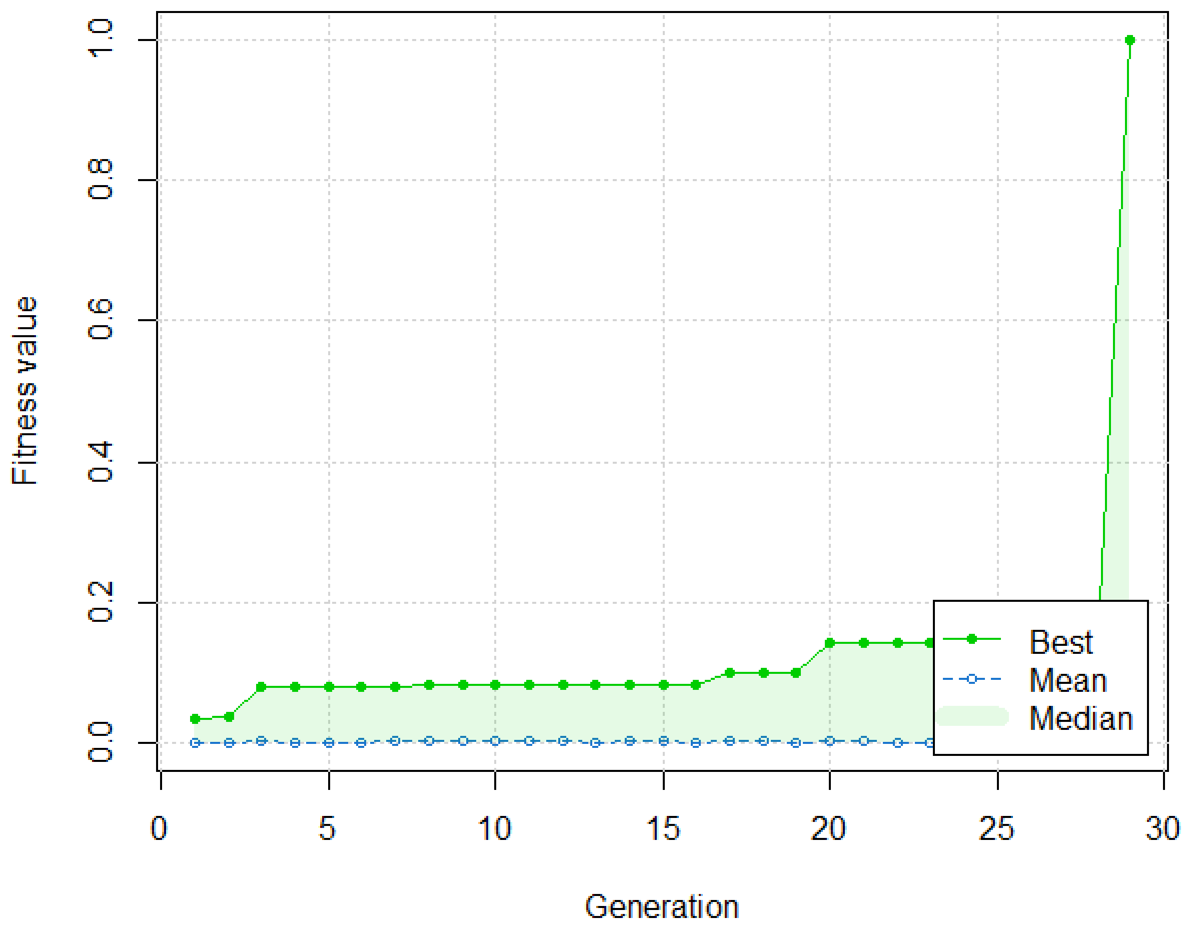
To compare the run times of the genetic algorithm and the random search we run the following code.

```
research_f <- function(){
  time_brute <- 0
  time_ga <- 0

  for(it in 1:10){


    time_brute <- time_brute + randomSearch()


    startTime_ga <- proc.time()

    GA <- ga(
      type = "permutation",
      fitness = f,
      lower = 1,
      upper = numbNumbers * 2 - 1,
      popSize = 1000,
      maxiter = 400,
      population = myInitPopulation,
      #mutation = myMutation_swapOperators,
      mutation = myMutation_swapOperands,
      #mutation = myMutation_replaceSubTree,
      pmutation = 0.1,
      pcrossover = 0.8,
      #crossover = mix_crossover,
      #crossover = twoPoint_crossover,
      crossover = sp_crossover,
      elitism = TRUE,
      maxFitness=1,
      keepBest = TRUE)

    time_ga <- time_ga + (proc.time() - startTime_ga)
  }

  plot(GA)
  print(time_brute[3]/10)
  print(time_ga[3]/10)
}
```

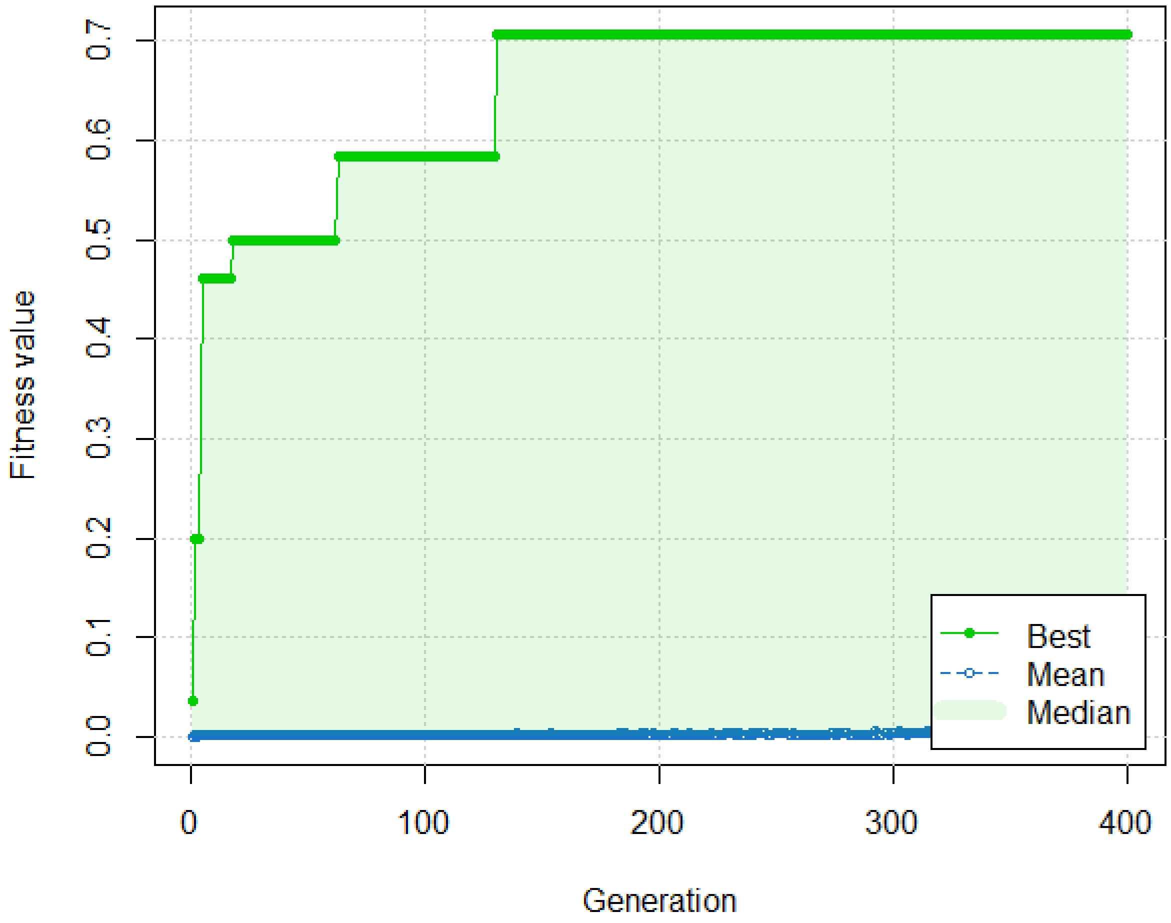
When used on the equation `exp <- c("10", "70", "7", "2", "15", "3", "12","+" ,"-", "/", "*")` with the target number `target <- 1444`, we get `time_ga = 6.285` seconds and `time_brute = 7.364` seconds.



We then ran the same program and equation, but with the two point crossover function instead. The elapsed time for the ga function was `time_ga = 4.072` seconds, and for the random search `time_brute = 4.666` seconds.



With the uniform crossover function the resulting times were `time_ga = 7.322` seconds and `time_brute = 9.389` seconds.



When using the mix crossover function the resulting times were `time_ga = 13.861` seconds and `time_brute = 3.292` seconds.

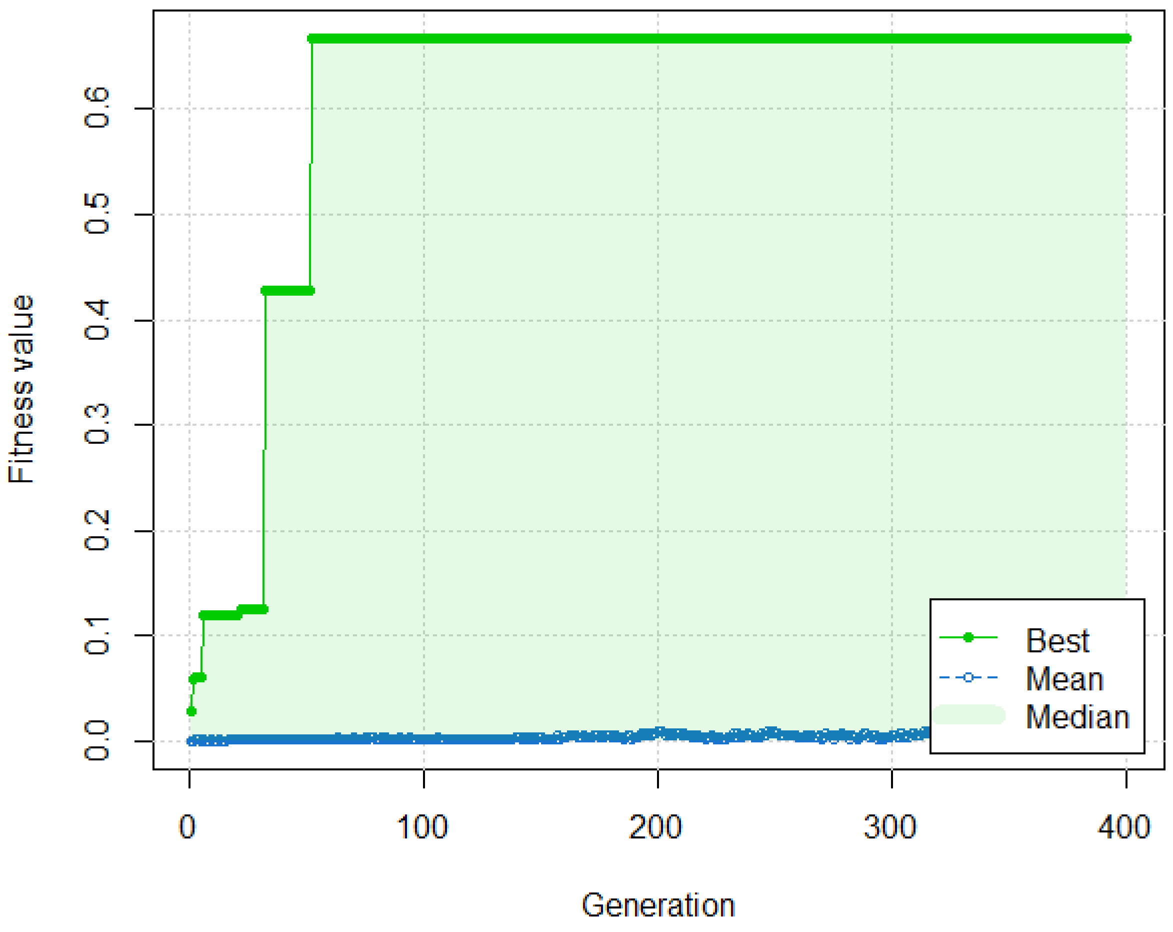
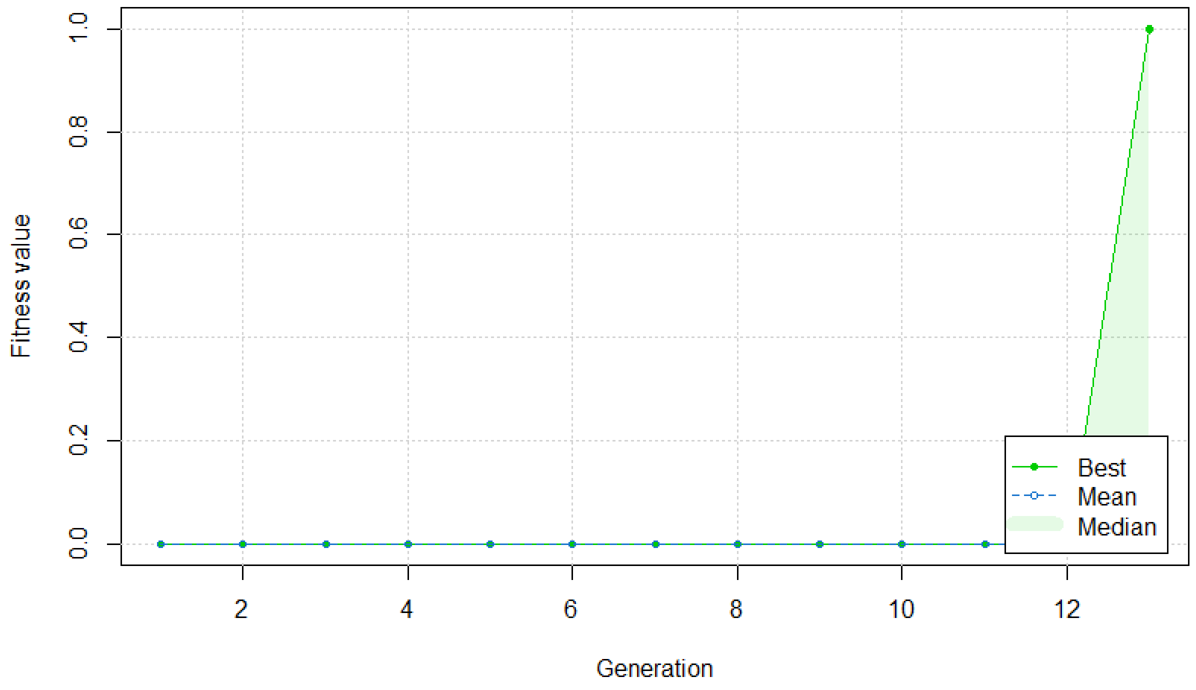


Figure 2: Mix crossover vs random search

From the elapsed times and plots, we can see that the mix crossover function performed worse than the random search function and that it did not find the solution to the equation. The best performing crossover functions were in our case the single point crossover and the two point crossover.

To further test the two point crossover function, we ran this equation `exp <- c(c(1:10),"+" ,"-", "/", "*")` and the target number `target <- 3628800`.

The elapsed time for our genetic algorithm function was `time_ga = 4.36`. The elapsed time for the random search function was `time_brute = 9.782`, so more than double of the genetic algorithm funcion.



From this we could conclude that if we kept increasing the difficulty, the random search algorithm would run forever, counting to infinity.