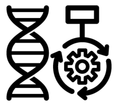
Intelligent Systems

Seminar Assignment 1 - Genetic Algorithms

November 18, 2021

by Marcel Martinšek and Žiga Medved



Genetic Algorithms

Overview

Genetic Algorithms are programs which are based on the the ideas from the Darwinian evolution. They are basically an iterative search and optimization technique, that is capable of solving complex problems.

Problem definition

The goal of this assignment was to solve a simple mathematical game with the help of the genetic algorithms. In this mathematical game, you are given a set of numbers, a set of mathematical operators and a target number. Your goal is then to arrange the numbers and mathematical operators in such manner that the result of the equation is as close or even identical as the target number.

An example:

- You are assigned the numbers 6, 20, 10, 5 and operators +,-,/,*
- And the target number 135

A valid solution for the target would be 6*20+10+5.

We had a few rules which made the problem a bit easier, the rules were:

- Each number can only be used once
- Each operator can be used multiple times
- Your solutions should not include brackets

Tasks

1. Population

For the first task we had to generate a starting population. The problem here was the formatting of our solutions that would be compatible with the *R GA library*.

We chose to work with permutations. Because the mentioned library doesn't know how to work with permutations of strings, we represented them as indices of a string vector.

The code below shows the mentioned approach.

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

Our population initialization function works like this:

- The input of our function is a vector of length 5+4 where the first 5 indexes represent the given numbers and the other 4 indexes represent the operators
- First we create a population matrix where each row will contain an individual
- We then randomly pick a number between 1 and 5, that tells us how many operands we will use and therefore the length of the individual
- With the picked number we can then choose how many operands and operators we need to sample
- And lastly we insert the sampled operands and operators into our population matrix

The code below implements the process explained above.

```
myInitPopulation <- function(object) {
    maxLen <- object.popSize
    pop <- matrix(0, maxLen, 10)

    for (i in 1:maxLen) {
        len <- sample(1:5, 1)

        ioperands <- sample(1:5, len, replace = FALSE)

        ioperators <- sample(6:9, len - 1, replace = TRUE)

        p <- insert(ioperands, 2:length(ioperands), ioperators)

        pop[i, seq(1, length(p))] <- p
    }
    pop
}
```

2. Fitness function

Every Genetic Algorithm needs a tool with which it can decide if a produced individual is either fit or not.

So our objective was to create a function which will measure the fitness of an individual.

The fitness function works like this:

- The input to the function is an expression vector
- First extract the data in the expression vector
- Evaluate the data with the function eval
- Last and the most important part is the actual measurement. We choose to subtract the individual's value from the target value
- The result of this function will be the difference between the two
- A nice feature here is to add weights to make near-miss results yield strongly better values than completely wrong ones

Corresponding code below.

```
myFitness <- function(expression) {
    expr_string <- paste(data[expression], collapse = "")
    expr_value <- eval(parse(text = expr_string))
    penalty <- abs(target - expr_value)
    score <- -penalty
    score
}
```

3. Crossover and mutation functions

Since Genetic Algorithms always select the most optimal individuals it can happen that they will be represented multiple times in the upcoming generations. If we have a smaller population, then this could cause a problem because we couldn't find the optimal global solution.

To avoid this problem we use crossover operations as reproduction techniques, where the offspring is created by exchanging genes from the parents.

Another powerful operator is mutation. The purpose of mutation is to keep diversity within the population and prevent premature convergence.

In this task we had to come up with our own crossover and mutation functions that will generate valid equations.

1. Our first **mutation** function is based on swapping two elements of the same type (operand/operator) it works like this:
   - The input to the function are two parameters:
     - object, which is a GA object
     - parent, which is a number that tells which object(individual) was selected for the mutation
   - Then it randomly decides whether to swap two operators or two operands
   - Actually swapping the elements and returning the mutated individuals

The snippet below shows the implementation.

```
myMutation <- function(object, parent) {
    mutate <- parent <- as.vector(object@population[parent, ])

    if (sample(0:1, 1)) {
        m <- sample(seq(2, exprMaxLen, by = 2), size = 2, replace = FALSE)
    }else {
        m <- sample(seq(1, exprMaxLen, by = 2), size = 2, replace = FALSE)
    }

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

, ],"mutate:",mutate))
    return(mutate)
}
```

2. Our first **crossover** is based on swapping random snippets from both of the parents and the resulting children will just be a combination of the parents. The function handles four cases which will be described below.

Our crossover looks like this:

- The input consists of two objects:
  - an object, which is of class GA
  - parents
- We sample a random index between the values one and the maximum length of the parent `sample(seq(1, parentLen), 1)`. This index tells us where our first snippet starts
- We apply the same approach to get the length of the first snippet. By sampling a value from one to the maximum length of the parent minus the snippet index from the step above `sample(seq(1, parentLen - snip_index[1] + 1), 1)`.
- We check what our index type is to keep the equation valid:
  - if it's odd -> we have a number, so we have to place the snippet on an odd index in the child
  - if it's even -> we have an operator, so we have to place the snippet on an even index in the child
- The next step is to find a random index in the other expression, that leaves at least snippet length space. The random index depends on the previous step. If the index type is odd we need to find an odd one and vice versa.
- The final part is the actual crossing:
  - we take the snippet from one parent and assign the values to the child on the indexes we determined in the previous steps
  - we now iterate over the child and assign elements from the other parent. The tricky part here is to find the elements which are not present in the snippet. Since the operators are allowed to repeat we just copy them, however the operands we must insert only those which appear in the parent and in the snippet
  - the whole process is identical for the second child
- The function returns a list of children and an undetermined fitness

The actual implementation:

```
myCrossover <- function(object, parents) {
    expr <- object@population[parents, ]
    childs <- matrix(0, 2, exprMaxLen)

    snip_index <- c(0, 0)

    snip_index[1] <- sample(seq(1, exprMaxLen), 1)
    snip_length <- sample(seq(1, exprMaxLen - snip_index[1] + 1), 1)

    snip_index_type <- (snip_index[1] + 1) %% 2 + 1 #1-odd or 2-even

    # set index in second expression that leaves at least snip1length space
    snip_index[2] <- snip_index_type
    if (snip_index_type > (exprMaxLen - snip_length + 1)) {
        snip_index[2] <- sample(seq(snip_index_type, exprMaxLen - snip_length + 1, by=2), 1) # nolint
    }

    # CROSS CHILDREN
    for (i in seq(1, length(parents))) {
        other_parent <- i %% 2 + 1
        snippet <- expr[other_parent, snip_index[other_parent]:(snip_index[other_parent] + snip_length - 1)] # no
lint

        childs[i, snip_index[i]:(snip_index[i] + snip_length - 1)] <- snippet

        insert_at <- which(childs[i, ] == 0)

        take_index <- 1
        take <- expr[i, seq(1, exprMaxLen, by = 2)]

        for (j in insert_at) {
            if (((j %% 2) == 0)) {
                childs[i, j] <- expr[i, j]
            }else {
                while (TRUE) {
                    if (!(take[take_index] %in% snippet)) {
                        childs[i, j] <- take[take_index]
                        take_index <- take_index + 1
                        break
                    }
                    take_index <- take_index + 1
                    if (take_index > length(take)) {
                        print("Error")
                    }
                }
            }
        }
        tryCatch(
            expr = {
                myFitness(childs[i, ])
            },
            error = function(e) {
                print("An error occured, please fix it")
            }
        )
    }
    return(list(children = childs, fitness = rep(NA, 2)))
}
```

3. Our second **mutation** function is a bit simpler, it changes n operators in the given expression into other operators.

The function works like this:

- The inputs are:
  - an object, which is of class GA
  - parent
- First we select a parent from the population
- Then we sample n operator indexes which will tell us where to change the operators
- Lastly we perform the swap

Snippet below shows the implementation:

```
myMutation2 <- function(object, parent) {

    mutate <- parent <- as.vector(object@population[parent, ])
    swapnumber <- 3

    m <- sample(seq(2, exprMaxLen, by = 2), size = swapnumber, replace = FALSE)
    operators
    for (i in seq(1, swapnumber)) {
        mutate[m[i]] <- sample(setdiff(operator_indices, parent[m[i]]), 1)
    }
    return(mutate)
}
```

4. Our last function and the second **crossover** is based on swapping operators from the first and the second parent.

The second crossover:

- The inputs are:
  - an object, which is of class GA
  - parents, which is a vector of two integers
- First we extract the parents from the population
- Now we just have to change the element at every even index in both children with the corresponding elements in a parent

Code snippet:

```
myCrossover2 <- function(object, parents) {
    expr <- object@population[parents, ]
    childs <- expr
    childs[1, seq(2, exprMaxLen, by = 2)] <- expr[2, seq(2, exprMaxLen, by = 2)]
    childs[2, seq(2, exprMaxLen, by = 2)] <- expr[1, seq(2, exprMaxLen, by = 2)]

    return(list(children = childs, fitness = rep(NA, 2)))
}
```

4. Evaluation

A Genetic Algorithm is basically a random search method that uses some kind of history, and because of that it can pick the most optimal solution from generation to generation. But is this really better than a completely random algorithm which starts with a blank sheet every time? In this section we compared our Genetic Algorithm with a completely random algorithm to test this assumption.