Evolutionary Algorithms: Power Distribution Network Reconfiguration (PDNR)

Gideon Hanse
s1630784
i.j.g.hanse@umail.leidenuniv.nl

Francesco Bovo
s2264951
f.bovo@umail.leidenuniv.nl

October 3, 2020

1 Introduction

Our task for this assignment was to implement the concept of the Genetic algorithm (GA) and the Monte Carlo algorithm (MC) in order to minimize the power loss in a power distribution network. In general, power is generated in power plants after which it is distributed to consumers through several substations and transmission lines. The way in which the power is distributed from the power plant to the consumers can be captured in a power distribution network. This network can be configured in many different ways by opening and closing switches where every configuration comes with it's own properties in terms of power loss, which is caused by resistance on the transmission lines. To minimize the power loss, it is necessary to find the optimal network configuration. Since there are a lot of possible configurations, it would take an incredibly long time to check the power loss for all of the possible configurations. Therefore, we implemented the GA and the MC algorithm in order to come up with a good network configuration in a short time with limited computational resources.
Both the GA and the MC algorithm are used to search for the best solution to problems with many possible solutions. The GA is based on the Darwinian principles of evolution in order to optimize solutions, whereas the MC algorithm uses the principle of randomized search based on best possible solutions. The algorithms are explained in more detail in the Implementation section.

2 Problem Description

The way in which power is distributed across the network can be changed by determining which switches are normally open and which are normally closed. The system we used for this assignment is called the 119 test system since there are 119 switches normally closed, as shown in Fig. 1. In the system, 15 switches have to be normally open, and it was our task to determine the best possible configuration of which switches had to be set to be normally open. A power distribution network typically consists of multiple loops, which are circuits

that return to the point where they originate. When re-configuring the network distribution there are two constraints to be hold [1]:

1. Cycle free: To avoid short circuits it is required to have at least one normally open switch per power loop.

2. No separated components: Every consumer should have power supply, therefore every loop should contain a maximum of one normally open switch.

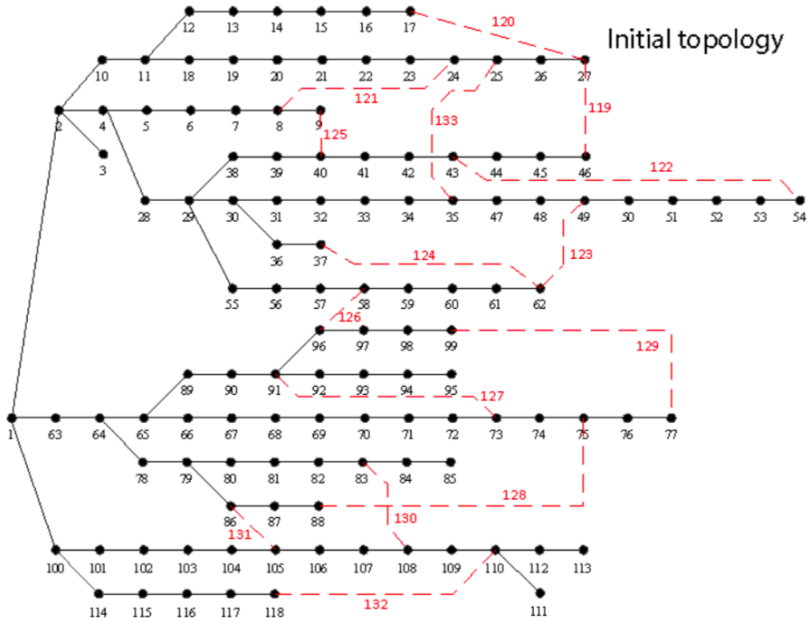

Figure 1: Initial topology of the 119 test system.

Following these constraints, it means that every loop should contain exactly one normally open switch. To check if these constraints are obeyed, we were provided with a valid_119 function which returns the value 1 for feasible configurations and 0 for configurations that are not feasible for the network. We were also provided with a function calculation_119 which returns the amount of power loss for a given configuration.
Both these functions had to be given a column integer vector of length 15 as input, which represents the locations of the switch to be normally open for all of the 15 loops.

3 Implementation

Both of our algorithms have been implemented using Matlab version R2018b. The Genetic Algorithm was written on a Macbook Air, and the Monte Carlo Algorithm was implemented on a Macbook Pro machine. However, both algorithms have been also tested on Windows and Linux using Matlab.

3.1 Genetic Algorithm

The general outline of our Genetic Algorithm is as follows:

1. Initialize population
2. Check population for feasibility
3. Evaluate population
4. While function evaluations are smaller than 10000
    1. Select individuals from population
    2. Crossover selected individuals with probability
    3. Mutate crossed individuals with probability
    4. Population becomes offspring population
5. Output best offspring and it's power loss

We implemented this using the following functions:

3.1.1 Population initialization

To get started with the genetic algorithm, we initialized the population. We did this based on the initial network configuration as was given in the assignment, see Figure 1. We generated a population of size 100 by taking a random integer for each of the column vector indices, each bound to the maximum amount of switches per loop. For instance, the first power loop, which is represented by the first index in the column vector, we took a random integer between 1 and the length of this loop (Length 25 in this case). However, to limit the variability in the population and keep enough feasible configurations, not every integer in the column vector was changed. Instead, we used a mutation function that changed every integer from the initial configuration with a 0.20 chance.

3.1.2 Fitness calculation

We wrote a function to calculate the fitness of the whole population. Our function requires the population as input, provided as a matrix with 15 rows and N columns, where every column represents one individual. In order to calculate a fitness value for all of the individuals, we use the calculation_119 function to calculate the power loss, after which a function evaluation counter gets increased by one. Therefore, the function first checks the feasibility for all of the individuals. Infeasible individuals automatically receive a fitness of zero. For the feasible individuals, the fitness value is based on the power loss. Since it turned out to be hard to do mate selection based on a lowest score, we came up with the following function through trial and error to give higher fitness values to individuals with less power loss:

$$Fitness = \frac{1}{(powerloss * 10^4)^8 * 10^{-4}}$$

The function output is an array filled with the fitness values for all of the configurations in the population, with corresponding indices to the population matrix.

3.1.3 Mate selection

In order to select individuals we created a function which uses the population matrix and the fitness array as input. Since the proportional selection function seemed not to perform well, we decided to compute the tournament selection function. The key idea behind this method is that for a certain number of tournaments, in our case 100 as the size of the population, we compare the fitness value of two randomly selected individuals from the population and store the one with the higher fitness value. After all the tournaments, the selected individual will be the one with the highest fitness value. The output of this function is the selected individual in form of a 15 integer column vector.

3.1.4 Crossover

In order to keep the runtime as small as possible, we chose to do a simple single point crossover. Our crossover function has to be provided with a 15x2 matrix, containing two individuals represented as 15 integer column vectors. It also has to be given a crossover probability between zero and one. A balance between exploitation and exploration is found by implementing a probability of crossover. First, a random value between 0 and 1 is generated, which is compared to the given crossover probability. If the random value is larger than the crossover probability value, crossover is not applied. However, if the random value is smaller than the crossover probability, two crossover offspring are created with crossover point 7. This means that one of the offsprings has the first seven integers from mate one and the last eight integers from mate two. For the other offspring it's the other way around. The function outputs two offsprings in form of the input matrix. If crossover was not applied, it returns the input matrix.

3.1.5 Mutation

Our mutation function uses a single individual and the mutation probability as input. As explained in the section on population initialization, for every integer in the individual column vector, there is the possibility of being changed to a random integer between 1 and the length of the corresponding loop the column index represents. This is the case if a random generated value between zero and one is smaller than the mutation probability, which is generated separately for every integer in the 15 integer column vector. The function outputs the mutated individual.

3.1.6 Function evaluation counting

Since Matlab do not support function calls by reference, we had to create an additional function to be able to keep a global counter and increase it in the fitness calculation function. Instead of directly increasing the evaluation counter in the fitness calculation function, we call a function that increases the counter with one.

3.1.7 Main function: looping over generations

After initializing, checking and evaluating the population as described in section 3.1.1 and 3.1.2, our algorithm loops over generations as follows, as long as the function evaluation

counter is below 10000:
For every generation, the offspring population is initialized as an empty array. Mates are selected in pairs of two with the mate selection function, after which they run through the crossover function (with crossover probability 0.75) and mutation function (with mutation probability 0.1) respectively. The individuals that remain after crossover and mutation are added to the offspring population in pairs of two. This is repeated 50 times, since our population size is 100. Since this generated a new population, the current population gets replaced by the offspring population, and the fitness of the new population is calculated as described before. The individual with the lowest power loss so far is stored in the variable aopt and its power loss is stored in fopt, and the generation counter gets increased by one, after which the next generation is calculated. The algorithm outputs aopt as 15 integer column vector and fopt as single integer value.

3.2 Monte Carlo Algorithm

In this particular case, the Monte Carlo algorithm represents more a naive way to search for the optimal solution. The outline of this approach is more trivial than the Genetic Algorithm to understand as well as to implement. Generally, the core of this algorithm is to create, for a large number of times, an individual as random vector and take that individual that maximizes the fitness function.

For our problem, the general outline of our the Monte Carlo algorithm is as follows:

1. Load parameters
2. Initialize the vector and fitness values
3. For 10000 evaluations do
    1. Create an individual as random vector from lower bound to upper bound
    2. If valid, evaluate the fitness function
    3. Take the best fitness and the save the associated optimal vector
4. Output optimal vector found and it's power loss

As just described, the implementation of the Monte Carlo algorithm is quite straightforward, therefore the only parameters used are the lower and upper bounds to obtain the random individual and the size of the individual. These parameters have been used from "para.mat" file provided. Here, we also define the fitness function as described in section 3.1.2, however instead of giving the population as input parameter we give a random valid vector.
As for the Genetic Algorithm, the individual with the lowest power loss so far is stored in the variable aopt and its active power loss is stored in fopt. The algorithm outputs aopt as 15 integer column vector and fopt as single integer value.

4 Experiments

The experiments were performed using Matlab version R2018b on a Macbook Pro and they have been repeated on a Linux machine as well.
For both approaches we have run 20 experiments with 10000 function evaluations each. Table 1 shows the results in terms of active power loss and computation time both for the Monte Carlo and Genetic Algorithm. At first sight, we can clearly conclude that on average the Genetic Algorithm performs much better than Monte Carlo algorithm. This is not surprising since the latter one is entirely based on a random process. If we consider the Monte Carlo Search we can see that the average active power loss is quite high, meaning that even after 10000 iterations the algorithm is not able to find a better solution. Moreover, the standard deviation is also high, meaning that if we perform a large number of experiments is more likely to find values far from the mean. Basically, these results are telling us that the Monte Carlo algorithm doesn't perform well for our problem since it's too unpredictable and the it's not accurate. However, if we analyze the Genetic Algorithm results we can see that it performs quite well. The average active power loss is significantly lower than the Monte Carlo approach and the standard deviation is remarkably low. We noticed also that the minimum active power loss achieved by Genetic Algorithm, which corresponds to the best solution we obtained, was 869.7271 kW. This optimal result is associated to the following configuration of normally open switches:

$$Aopt = [21, 16, 1, 18, 18, 8, 10, 15, 10, 1, 2, 18, 12, 1, 19]$$

Figure 2 shows the performance of both algorithm on the 119 System. The X axis represents the number of function evaluations computed while the Y axis shows the fitness score in log scale. We can easily observe the difference in performance between the two approaches, as the GA converges to a good solution much faster than the MC algorithm. After about 1000 function evaluations it has found its best solution already. Also, the distance between the Monte Carlo Search and the Genetic Algorithm is quite large, meaning that the latter is more efficient for our problem and provides us with a better solution. As for the run times, the Genetic Algorithm takes much less time to complete than the Monte Carlo algorithm.

Table 1: Results after 10,000 function evaluations, averaged over 20 runs.

Algorithm	Average	Std Deviation	Minimum	Maximum	Avg. time
MC	1269.66	107.17	1094.98	1469.03	1203 sec.
GA	879.1	8.73	869.73	890.92	44.72 sec.

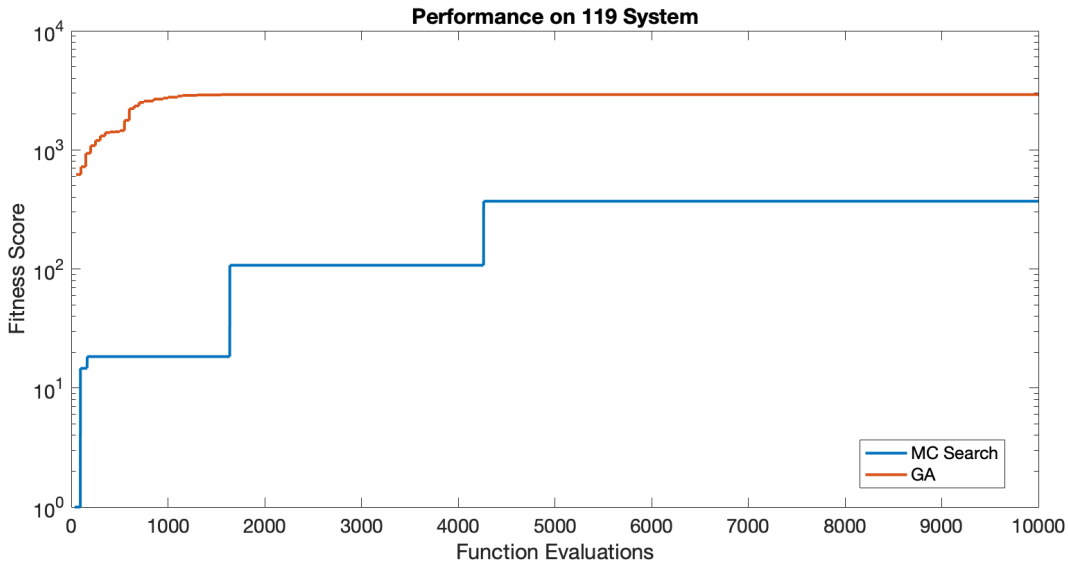

Figure 2: Performance of Genetic Algorithm and Monte Carlo Search on 119 System.

5 Conclusion and Discussion

To conclude, based on our results it is clear that the Genetic Algorithm that we implemented performs way better than the Monte Carlo algorithm. These findings are not surprising, since the Genetic Algorithm is more likely to only consider the good configurations, whereas the Monte Carlo algorithm is based entirely on randomly chosen configurations. The Genetic Algorithm does not only perform better based on the provided solutions, but also in terms of computational resources.

As the Genetic Algorithm provides really good solutions in very limited time, we can conclude that this Algorithm is very suitable to solve the problem of power distribution network reconfiguration. Although the Monte Carlo algorithm provides reasonably well solutions, its time complexity and final results are not very good, we would not recommend this algorithm in order to solve the current problem.

Although we have tried to optimize the performance of the proposed Genetic Algorithm, there are some aspects that could be considered in further research to make the algorithm perform even better. For instance, the way in which our proposed algorithm selects mates could be implemented in various ways. It could be that there are other ways that minimize the computation time, or improve results. Also, we chose to use the most simple form of crossover to minimize computation time. However, different forms of crossover, e.g. uniform crossover or k-point crossover, could be considered to optimize the algorithms results. As a last remark, one could also experiment with different population sizes to optimize the algorithms performance.

References

[1] Yang, K., Emmerich, M.T.M., Li, R., Wang, J., & Bäck, T. (2014). Power distribution network reconfiguration by evolutionary integer programming. In: Bartz-Beielstein T., Branke J., Filipi B., Smith J. (eds) Parallel Problem Solving from Nature PPSN XIII. PPSN 2014. Lecture Notes in Computer Science, vol 8672. Springer, Cham.