

NaCo 21/22 Assignment Report, Group 13

Emanuele Greco, Jasper Kooij & Oda Aggerholm

Leiden Institute of Advanced Computer Science, The Netherlands

Abstract. This report contains the results and descriptions of the assignment for the course Natural Computing. The first part gives an introduction to the genetic algorithm (GA), including a literature review on different operators of the algorithm, and a description of a GA implemented in Python. Moreover, a recap and analysis of an application of the GA used to assign teachers to different classes at a university is included. The second part gives a brief introduction into cellular automata (CA) and describes an implementation of it in Python, using the implemented GA, to solve the inverting problem. It is investigated how the GA performs when changing internal settings such as the operators described. Lastly, a recap and analysis of an application of CA to model the development of epidemics is given.

1 Introduction

Genetic algorithms are evolutionary algorithms that reflect natural selection by the mantra *survival of the fittest*. This makes them useful for many search and optimization problems. The general GA makes use of three different evolutionary operators — namely recombination, selection and mutation. The general idea behind these operators and examples of different ways of implementing them are described in this report. Moreover, an actual implementation is made in Python along with a review of a real life implementation of the algorithm outside the field of computer science. For this part, an implementation of the algorithm to generate a course schedule and assigning teachers to the different courses at a department of a university is described and discussed.

The report also covers cellular automata. CAs take inspiration in the interaction between neighbors in nature such as e.g., neighboring cells. Based on a CA, the GA implemented in Python is applied to solve the inverting problem. The performance of the GA based on different operator combinations and objective functions is investigated using the IOHprofiler [1]. Lastly, a literature review of a real life implementation using CA to model the development of epidemics is given to further describe the application range of CA.

2 Genetic Algorithm

The genetic algorithm is an algorithm inspired by evolution. It takes an initial population and selects the fittest individuals to produce offsprings that are then added to the next generation [6]. It starts with an initial random population. A population can be seen as a group of chromosomes each containing a set of variables representing the alleles. In general, the algorithm uses three different evolutionary operators: namely *recombination*, *mutation* and *selection*. In the selection phase, the algorithm assigns probabilities to the individuals and selects them to create the next generation. After the individuals have been selected, they need to be paired to create the new generation. Much like in nature where a female and male individual are paired to generate a new

generation, two individuals are combined to generate two new individuals by the algorithm. This is what is called recombination or *crossover*.

After the creation of the new generation, the last evolutionary operator can be applied to the children - namely mutation. Usually the rate of the mutations is not high in genetic algorithms as this makes the process similar to a random search. A small level of randomness can however be used by using mutations and be beneficial due to the fact that it helps maintain diversity of the population [7]. So, mutations make small local changes in the population, meanwhile recombination combines information from good individuals which generates bigger changes. In the following subsections, different ways of implementing recombination, mutation and selection are described.

2.1 Recombination

Recombination, also called crossover, is a way to simulate the mating in nature between a female and male individual. Recombination is usually done on two selected individuals, 'parents', of the population, and is a way of combining the two individuals to produce a new generation with new individuals, 'children'.

The simplest recombination technique used in the GA is *single-point crossover*. In this technique, the paired 'parent' individuals are cut at a random crossover site. The parts after the crossover site are then exchanged to form the two new children [3]. Fig. 1 shows how single-point crossover is applied to two paired individuals to generate individuals for the new generation. Other versions of this recombination technique can be used. An example is two-point crossover where two random points are chosen and only the parts between these points are exchanged.

Another type of recombination is uniform *three-parent crossover* on a binary representation. The basic idea behind this technique is that instead of having two parents contribute to the child, you have three. The offspring is then created by a comparison of the bits of the three randomly selected parents. In each position, if the bit of parent 1 and parent 2 are the same, this is chosen as the value of the offspring. If they are not the same, the value of parent 3 is chosen [4]. Fig. 2 shows an example of this.

As seen from the example given in Fig. 1, the single-point crossover is quite simple. Therefore, it has a high capability of generating illegal offsprings. It suffers from positional bias, as it is more likely to keep together genes that are close to each other, and can never keep genes from opposite ends of the string together [2]. Therefore, it is not suitable for very complex applications where illegal offsprings are easily generated.

The three-parent crossover is a bit more complex than the single-point crossover. By using more than two parents, more information is gathered which can aid the process of evolution to gain better results for the next generation - for instance in the problem of generating time tables [4].

2.2 Mutation

Mutation is another evolutionary operator. It introduces an extra level of randomness to the algorithm and in this way help maintaining diversity of the population. Usually, the mutation rate in the GA is set low to avoid the GA to become similar to a random search [7].

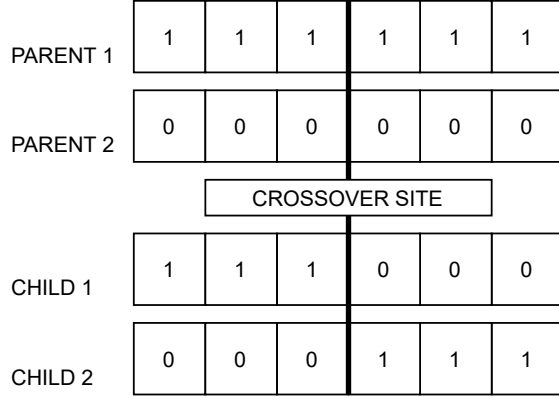


Fig. 1: Example of single-point crossover. Modified from [3].

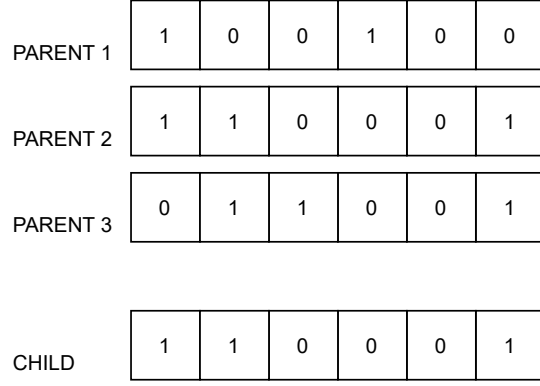


Fig. 2: Example of three-parent crossover. Modified from [4].

In the basic GA, the chromosomes are binary encoded. A simple type of mutation that can be used on this is the *bit flip* mutation where one or more random bits are selected and flipped. The bit flip mutation is carried out by selecting a probability, p_m , of inversion. p_m should be the same for every bit in the alleles. Again, to avoid too much randomness, p_m should be small - e.g., $p_m = 1/n$, where n is the number of bits in the allele [10].

Another type of mutation is the *swap mutation*. This mutation takes two randomly chosen alleles and swaps their positions. This mutation has a bigger applicability, as it can be applied in GAs that are not necessarily binary encoded.

2.3 Selection

The selection operator of the GA is heavily inspired from natural selection in nature. The probability of survival therefore depends on the fitness of the individuals. In the selection process, the GA assigns probabilities to the different individuals of the population and selects them for creating the next generation based on this [7].

One implementation of the selection operator is the *roulette selection*. As indicated by the name, this method is inspired by a roulette wheel. However, instead of each possibility having the same probability of being chosen, the probability for each individual to be chosen is proportional to its fitness. To be able to do so, the probability of selection of each individual has to be normalized, as the sum of the probabilities of a population of size n should be equal to 1. When doing so, the probability of selection of an individual, x_i , with fitness value $f(x_i)$ in an n -size population is given by [5]:

$$p_i = \frac{f(x_i)}{\sum_{j=1}^n f(x_j)} \quad (1)$$

This type of selection will have issues if you are working on populations with very big differences in fitness. If a chromosome has a very good fitness, it will be hard for any of the other chromosomes to get selected [8].

Another type of selection that takes this problem into account is *rank selection*. This method ranks the individuals based on their fitness and applies the probability based on the ranking instead, which gives each individual in the population a higher probability of being chosen. A downside to this is that it may lead to slower convergence, as the fittest individuals do not differ much from each other [8].

2.4 Representation

In our case, the genes are represented as binary strings, with a fixed size for every gene in the pool. This is a simple way of representing genes, and easy to work with. As stated earlier, binary representation is beneficial to the evolutionary operator mutation, because bit flips are an intuitive solution of mutating a single bit. Also, using binary values is more computationally efficient than using floating point values, for example.

Changing the representation of genes to something other than binary strings will bring the need to tweak the evolutionary operators, in order to account for the new representation. Computations like bit flips are not possible anymore, but using swap mutation as evolutionary operator is still possible. Also the fitness function has to be considered when changing to a non-binary representation. Will the bit values have a range? Can there be negative values? How will this affect the fitness of the gene?

Representations where each gene has a variable length are possible, but implementing crossover will be more complex. Other types of representations do exist, for example tree-like or graph-form representations, but these representations are used in a specific setting, for specific types of problems.

3 Implementation of GA

The implementation of the Genetic Algorithm was done in Python. It uses the IOHexperimenter [1] package to get a real value test problem on $\{0, 1\}^n$, and is encapsulated in a class called `GeneticAlgorithm`.

When initialized, it selects the operators to be used for the selection, recombination and mutation stages through internal variables. The design itself was built in a modular fashion from the start, to ensure the seamless substitution of a specific implementation of one operator for another. At this point, internal variables used by the operators are also initialized, using the values provided by the constants. The constants can be easily changed and are checked to be in between boundaries. Tab. 1 displays the parameters of the implementation including their default values and range of accepted values.

The class needs to be called with an instance of `ioh.problem.Integer` as the only parameter. The problem dimensionality n is then stored in a class variable and the initial population population array of M strings of length n is created. Every string is generated through the use of an uniformly distributed random function that returns a random bit.

Parameter	Default Value	Range
MAX_ITERATIONS	200	1-4000
INIT_POPULATION_NUMBER	200	1-4000
SELECTION_NUMBER	INIT_POPULATION_NUMBER/10	1-200
SELECTION_TOURNAMENT_SIZE	2	1- ∞
CROSSOVER_OFFSPRINGS	100	1-1000
MUTATION_NUMBER	CROSSOVER_OFFSPRING/2	1-200
MUTATION_PROBABILTY	$1/n$	1 - ∞
N_K_POINTS	1	1 - 10

Table 1: Parameters of the implementation.

After that the main loop begins. The MAX_ITERATIONS constant stores the number of times the loop should be iterated.

The first operator implemented is selection. The main one is a simple implementation of the rank selection operator called rank_selection. It is a function that computes the objective function value $f(\mathbf{x}_i)$ for each \mathbf{x}_i in both population and offspring_population, where offspring_population is the array populated by the crossover operator. After that the function simply select the individuals with the largest objective function value, the number of the individuals to choose is stored in the SELECTION_NUMBER. There is also an implementation of the tournament selection operator, where it samples q individuals from the union of initial_population and offspring_population and then chooses the individual with the largest objective function value from those q . The q is setted by SELECTION_TOURNAMENT_SIZE and the whole process of sampling and choosing is done for SELECTION_NUMBER times. Lastly, there is an implementation of the roulette wheel selection operator. It builds a fitness population based on the fitness of the individuals. The fitness determines the probability of the individual. Then it 'spins the wheel' and the winner is found.

The next operators are the recombination operators. We have two recombination operators: a k-point crossover operator and the three-parents crossover. The kPointsCrossover_recombination computes $n.k_points$ unique points which are smaller than the dimensionality, then perform the crossover for CROSSOVER_OFFSPRINGS times on two random parents. These parents are chosen uniformly from the population array. The resulting offspring is stored in the offspring_population array. The threeParentsCrossover_recombination selects three parents from which one offspring is generated. The function then checks every allele of the first two parents: If they have they same value, it will copy that allele value into its the new one, otherwis the third parent's allele value is used.

After that, mutation is performed through the use of one of either bitFlip_mutation or swap_mutation function. This functions select a number of uniformly random individuals from the offspring_population array equal to the MUTATION_NUMBER constant and apply a mutation. With MUTATION_PROBABILTY probability, the bitFlip_mutation works by taking each allele value incremented by one and then taking the modulo of the arity (the number of values each allele can have). The textttswap_mutation takes two random alleles and swap their position

Although minor implementation issues could be fixed, the implementation overall works sufficiently good.

4 Application of GA in Practice

The paper *An application of genetic algorithm methods for teacher assignment problems* [11] describes an application of the genetic algorithm to generate class schedules and assign teachers to courses at a university department. The GA was chosen for the problem to deal with the issue of multiple constraints. For the problem, the application was divided into two sub-issues, namely teacher assignment and course schedule. For the solution in the paper, it was assumed that all resources needed for a course were in immediate and sufficient supply. The teacher assignments and course scheduling were based on the following factors:

- *Teacher assignment*: The courses offered, the teacher’s professional knowledge, the minimum teaching hours required for a teacher by the university, and the limit of overtime hours.
- *Scheduling*: Avoiding conflicts with teacher schedules, classrooms and ensuring continuity of each course.

To ensure the quality of the solution, three hard and two non-hard constraints were set. The hard constraints were reaching minimum teaching requirement for each teacher, having overtime within reasonable range and the teacher’s academic qualification and preferences. The non-hard constraints were that overtime should not make a difference in scheduling and an instructor should not be appointed to too many different classes. Since the values of the parameters’ coding space of handling problems can satisfy all constraints of the problem, the problem of the paper can be considered a *non-constrained* problem.

Variable	Description
$C = C_1, C_2, \dots, C_n$	The set of courses offered.
$t = t_1, t_2, \dots, t_p$	The teachers in a department.
$W_{t_i} = C_a, C_b, \dots, C_x$	The courses teacher t_i can teach (all teachers cannot teach all courses), where the order is based on the willingness of the teacher.
Φ_{C_i}	The set of teachers eligible to teach course C_i .

Table 2: The sets used in the application of the GA.

Tab. 2 displays the different sets that were defined in order to solve the problem. For the implementation of the algorithm, the problem was represented by a onedimensional matrix as displayed in Fig. 3. Here, the order represents different courses and the elements in the matrix represent appointed teachers for each course. Cf. Φ , given in Tab. 2, each course cannot be randomly appointed to a teacher as not all teachers can teach all courses. In the population each chromosome as displayed in Fig. 3 therefore represents a different course schedule.

4.1 Genetic Operators

For the implementation of the algorithm, the genetic operators *selection*, *mutation*, and *crossover* were used.

The mutational operator was implemented in three steps: 1) randomly selecting a mutational chromosome within the population, S_X , 2) randomly selecting a course, C_y , in S_X , and 3) randomly selecting a teacher in Φ_{C_y} and exchange it into matrix C_y .

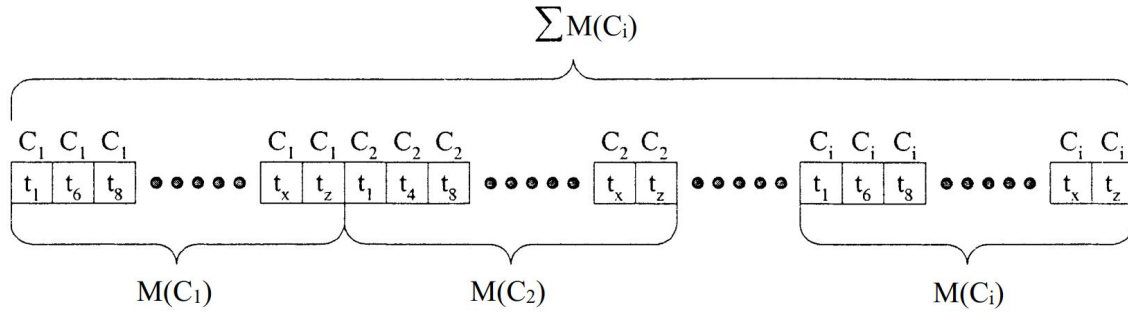


Fig. 3: Genetic representation of the teacher assignment problem. Adapted from [11].

The crossover operator was implemented in three steps that were repeated until all chromosomes were finished in the crossover process: 1) randomly selecting paired chromosomes S_A and S_B , 2) randomly choosing crossover point in S_A and S_B , 3) exchanging the parameter's combination based on the crossover point.

The genetic operators were applied to the chromosomes according to each chromosome's fitness function. The fitness function was based on how well the above mentioned constraints were satisfied - and a penalty applied accordingly.

4.2 Results

For the problem defined in the paper, the ideal case, where the teacher willingness for all courses is maximal, was assigned a satisfaction of 1800. This level of satisfaction is obviously not possible to achieve. After 400 generations of evolution, a satisfaction of 1465 was achieved, leading to a satisfaction fraction of $1465/1800 = 0.813$, which is considered high. Running the algorithm to achieve this result took 24 s, which is quite a difference from the five days it usually takes to do the job manually. Moreover, the algorithm is flexible as parameters can easily be changed e.g., if a teacher would like to change time for a class.

Based on the more or less simple nature of the problem, and the fact that it was possible to model it as non-constrained, the GA seems like a good choice for the given problem. As scheduling is a quite crucial task at university departments, the advancement in the time used to solve the problem and the simplicity of generating new solutions is quite remarkable.

However, it should be considered in the future to incorporate all necessities in the solution. Depending on the type of department the problem is solved for, the supplies are an important factor to consider. Taking for instance a department of Chemistry, it may be quite important to incorporate the availability of laboratories and equipment. This would improve the setup a lot and make the solution generated by the algorithm easier to apply in reality. Also, it may be worth to consider using e.g. three-parent crossover in the recombination phase which was seen to decrease the number of generations to get a valid solution than the two-parent for increasing size of the population in [4].

5 Cellular Automata

Cellular automata takes inspiration from the interaction between neighbors in nature. This can be things such as neighboring cells in biological organisms as well as growth of crystals. In computer science, the basics of cellular automata is done on a number of cells positioned in an n -dimensional grid. The cells have a finite number of states that are updated every discrete time step based on itself, its neighboring cells and a transition rule [9].

For our project, we recreated cellular automata (CA) in order to solve the inverting problem. The cellular automata class can be used to simulate individuals inside populations for the genetic algorithm. Eventually, an individual will contain the solution to the inverting problem, described in the next section.

Individuals will be a cellular automation consisting of a one-dimensional array of cells, each containing a value. Values can be either binary (0,1) or ternary (0,1,2). This results in a bit string which can be easily manipulated.

Cellular automata change state in time steps. Each time step, transition rules are applied to every cell in the CA. This transition rule set is bound to the CA and cannot change during execution.

The transition rules determine the next value of a cell, depending on the neighbourhood of that cell. The neighbourhood is defined as the cell itself and the direct neighbours of the cell. For the cells at the beginning or end of the array, the neighbor outside of the array is set to zero. In a CA with binary values, there are $2^3 = 8$ possible states a neighbourhood can be in. A transition rule defines a next value for every neighbourhood. This results in a total of $2^{2^3} = 256$ possible transition rules for a binary value set.

When three values are allowed in the cells of a CA, the number of possible transition rules expands exponentially, namely $3^{3^3} = 7.625\,597\,5 \times 10^{12}$.

With all this implemented, we can simulate cellular automata with a specific transition rule for some predefined amount of timesteps. This is used to solve the inverting problem.

6 Solving the Inverting Problem

The inverting problem we have to solve in the setting of cellular automata requires us to use a genetic algorithm. The premise of the inverting problem is to find an initial state, which will result in a given state in a given amount of timesteps. So, the required state, the transition rule and the amount of time steps are predetermined. We need to figure out the initial state.

The obvious naïve way to do this is by trying every possible starting state and then simulating the CA to check if the required state is reached. This very quickly becomes too computationally expensive and it is not a good way to solve the inverting problem, especially for larger CAs and more timesteps.

The way we try to solve it is by using a genetic algorithm. The population consists of CAs at initial state CA_i . Every generation the population gets evaluated, mutated and selected. To evaluate individuals, an objective function is needed to rank individuals based on the difference between the found result and the given result. If there is no difference, the objective function returns a

value of 1 and the solution is found. Otherwise, a value between 0 and 1 is returned. Based on this value, population gets selected.

There are two different selection operators implemented, each with its own way of selecting population for the next generation. We can choose which operator to use to compare them in effectiveness. The two selection operators are:

1. Tournament selection
2. Rank selection

A part of the population is also mutated every generation. Two mutation operators are implemented, each with its own properties. The two mutation operators are:

1. Bitflip mutation
2. Swap mutation

The last two operators are the recombination operators. These operators make sure the new generation consists of the right amount of individuals. The two recombination operators are:

1. k-points crossover
2. Three-parent crossover

All these possible operator combinations need to be tested to figure out the best combination for our environment. That's why we need to run $2 \times 2 \times 2 = 8$ tests for every inverting problem we have been given.

The input for the program consists of a number of different problems. These have the format shown in Tab. 3:

dimensionality k	rule number	timesteps T	CA at timestep T
2	30	1	[0,1,1,1,1,0,0,1,0,1,0,0,0,1,1,0]

Table 3

6.1 Performance of the GA

The IOAnalyzer [1] was used to test the performance of the GA. To do so, each of the two objective functions were tested for the experiments given in the assignment. For the experiments, we chose the two combinations: 1) k-point crossover, bitflip mutation, rank selection, and objective function r-contiguous, 2) k-point crossover, bitflip mutation, rank selection, and Hamming objective function. This is because these algorithms seemed to give the best results in terms of solving the inverting problem. Our configuration used 10 parents, 200 offspring and 100 mutation on the offspring population.

We had a lot of issues plotting the different tests in the same plot in the IOAnalyzer (see Appendix B for further explanation). Therefore we have only added the fixed-target and fixed-budget plots of one successful test case of the two algorithms (two out of the 20 experiments).

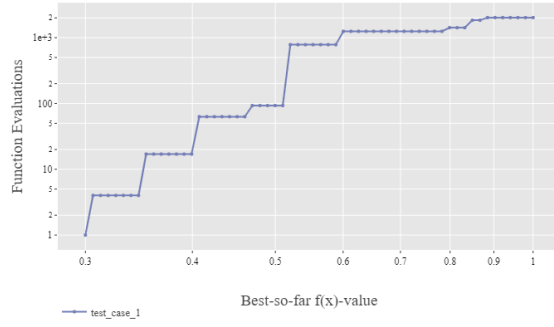


Fig. 4: Fixed-Target result with r-contiguous objective function.

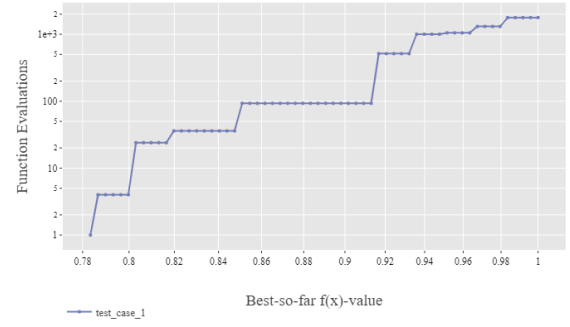


Fig. 5: Fixed-Target result with Hamming objective function.

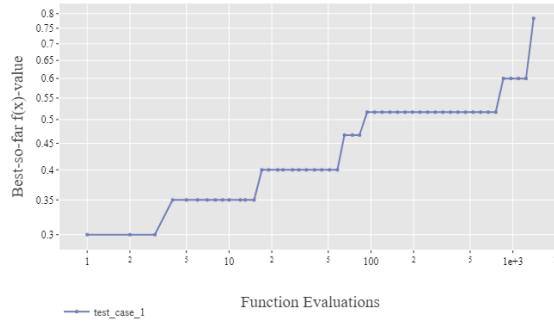


Fig. 6: Fixed-Budget result with r-contiguous objective function.

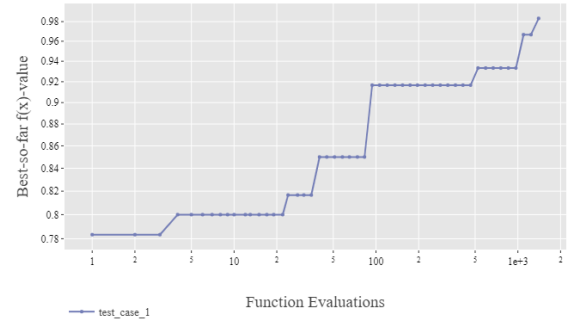


Fig. 7: Fixed-Budget result with Hamming objective function.

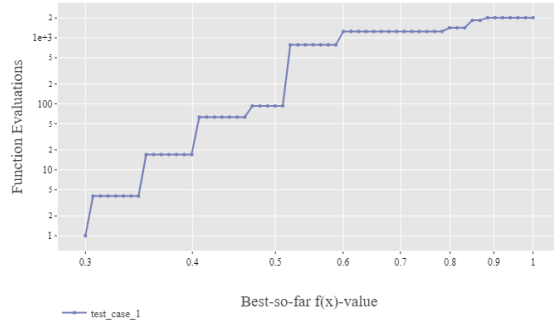


Fig. 8: Fixed-Target result with r-contiguous objective function and random search method.

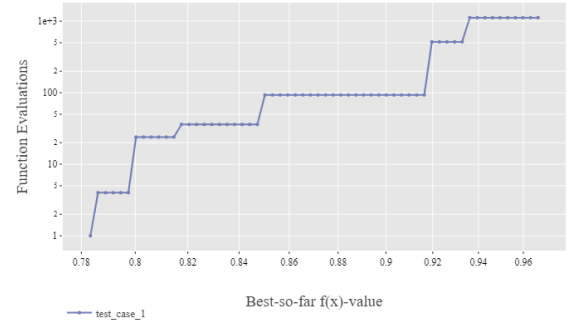


Fig. 9: Fixed-Target result with Hamming objective function and random search method.

However, in Appendix A we have added the test runs for all test problems given for the assignment for the algorithms mentioned above. Here it is seen that the problems that don't generate a solution do not seem to have a progress in the graphs either. It is moreover seen that solutions are only

generated for the problems with few iterations - meaning that these are the only problems our algorithm is able to solve. The two test cases with 99 iterations were removed as the algorithm got 'stuck' (it took more than two hours to finish the run) when these were added. Once removed, running the 8 remaining test cases on each algorithm took around 6-7 minutes except for when using tournament selection where it took around 40 minutes.

From Figs. 4 and 5 it is seen that more function evaluations are used to find the target value for the algorithm using the r-contiguous objective function compared to the Hamming. From a fixed-target perspective it thus seems that the Hamming objective function is best.

Figs. 6 and 7 show the fixed-budget results for the two objective functions. Here it is seen that the runtime is given as 'Function Evaluations', which means that the graphs pretty much are similar to the fixed-target ones, just with the axes flipped. However, it is worth noting that the graph in Fig. 7 is steeper than the one in Fig. 6. So, depending on the limits of your budget (e.g., if it is up to 100 function evaluations), one may argue that the algorithm with the r-contiguous objective function is better.

Comparing with the random search method algorithms in Figs. 8 and 9, it is seen that the implemented algorithms actually don't seem to work very well compared to a random search. For the r-contiguous objective function, no big differences are seen, meanwhile for the Hamming objective function it is seen that the random search method actually seems to perform a little bit better by being a bit steeper towards a target value of 1. This could indicate that the implemented algorithms may not be very good - at least for problems with many iterations. However, it is hard to make that conclusion in total as the plots displayed here are only based on very few runs on a single test case. We realise that it would have been much nicer to display the runs for all test cases in the plots for the different operator combinations in the style of the plots in Appendix A, as it would then be easy to see which operator combination works best on which problem. But as described in Appendix B we did not manage to do this.

The reason that some of the runs show no progress may be because they get stuck in a local maxima. This may be due to the large number of iterations causing small changes to have a greater impact. The issue may be resolved, or improved, by changing the selection strategy so that it includes more 'bad' candidates or increasing the number of mutations. Both of these solutions will lead to a higher diversity, which can help prevent the algorithm from getting stuck. Another reason could be the fact that the the objective function used were not well suited for the problem. Given the chaotic behaviour of the CAs, simple methods like the one used may not be able to express a significant similarity value.

7 Application of CA in Practice

The paper *Modelling epidemics using cellular automata* [12] describes an application of cellular automata for describing and modelling the development of epidemics. The use of CA was chosen because it is able to overcome some of the issues of traditional mathematical models within the field of mathematical epidemiology that often neglect the local characteristics of the spreading process and do not include variable susceptibility of individuals. Using CA, it is possible to simulate the individual contact process, the effects of individual behavior, the spatial aspects of the epidemic spreading, and the effects of mixing patterns of the individuals. The aim of the paper is to serve a basis for development of other algorithms to simulate real epidemics based on

real data (relevant these days...). The population was divided into three categories: 1) susceptible: individuals who can get the disease, 2) infected: individuals infected with the disease able to pass it on to others, and 3) recovered: individuals who are immune to the disease because they have recovered from it.

The main features of the epidemic and the environment considered in the paper are:

- Epidemic is not lethal, and no birth, immigration or emigration is considered - the total amount of population (and thereby the population of each cell) is constant.
- The population distribution is inhomogeneous: the total population living in each cell is different. The total population of cell (i,j) is N_{ij} .
- The way of infection is the contact between an infected and a healthy individual.
- Healthy individuals who have gotten the disease and recovered from it is immune, so they are not susceptible to the disease again.
- People can move from one cell to another if there is some type of transport - they can go outside and come back each time step.
- When an infected individual arrives in a cell, the number of healthy individuals contacted by him is the same independently of the total amount of people in the cell.

7.1 Description of the CA

For the simulations of the paper, a two-dimensional array of 50×50 cells was used.

The state of each cell is given by the portion of each class (susceptible, infected or recovered) in the cell in each time step. This was visualized using a grey level code where white represents state 0 where no individuals are infected, and black state 1 where all individuals are infected. Several individuals were considered in each cell instead of one in each cell as usually done in many CAs. This is because the proposed model is trying to simulate epidemic spreading in large regions. For the model, different cells can have different densities and different 'across cell' traversal or mobility properties.

The state of each cell was based on the transition rules of the CA, described by three different calculations:

1. Calculating the number of infected individuals in the cell at a particular time step given by the portion of infected individuals which have not been recovered, and by the number of susceptible individuals from the last time step who have been infected by the infected individuals of the cell. Moreover, a factor calculating the number of individuals who have been infected by visitors is added depending on set parameters describing the possibilities of travelling between the cells (movement factor).
2. Calculating the number of recovered individuals as sum of recovered from last time step plus the infected in last time step that are now recovered.
3. Calculating the portion of susceptible individuals of the cell at the time step as the portion of susceptible from last time step that have not been infected.

The neighborhoods used for simulations were Von Neumann and Moore neighborhoods.

7.2 Results

The model was tested on three cases. In the first, the population of each cell was the same and the connections between all cells constant. Here, circular successive epidemic fronts with the starting point of the epidemic as center were seen as expected.

In a second case, the population of each cell was not constant. Instead, most of the population was concentrated in the eastern cells. Here it was seen that the epidemic moved faster towards the west (the lower populated cells). This makes sense mathematically based on the definition of the CA where each infected individual in a cell infects the same amount of susceptible individuals no matter the size of the population in the cell. However, if you want to simulate real life, this result seems kind of 'upside-down' as you would expect the epidemic to spread faster in highly populated areas instead of low populated.

In the third case, there were no constant connections between the cells. Here, it was seen that the epidemic spread faster towards the areas with more connections between the cells, which makes sense. In this step, vaccination was also taken into account by adding a parameter standing for the portion of susceptible who have been vaccinated since last time step.

It makes sense why they chose to use a CA in the paper. It has a lot of important factors incorporated such as different population sizes in areas, means of transportation between areas, vaccination and of course the susceptibility of the individuals. It is a quite neat way to model something quite complex using relatively simple structure and parameters.

A nice improvement to the setup, however, could have been to change the factor in which how many people are infected based on how closely populated the areas are. In their model, it is the same area for each cell but with different population sizes. It would be nice to be able to also model how closely people live together so that not as many in the low populated areas would get infected as the one e.g. in the cities.

8 Conclusion

This report has given an insight into the broad applicability of evolutionary algorithms. With regards to the design and application of genetic algorithms both through a literature review and our own implementations. It is seen that different combinations and variants of the three main operators of the GA, recombination, selection and mutation, give the algorithm a broad applicability - for instance applying it to a CA as we have done. Moreover, it is seen that you can take inspiration from nature not only for computation but also for modelling different scenarios - as for example the development of epidemics using CA. Combining these two things, GA and CA, show powerful as well as it can be used to solve e.g., the inverting problem as described in this report. However, as mentioned in Sec. 6.1, our algorithm implementation seemed to work best for solving problems with fewer iterations. When more iterations were added, the performance was comparable to the one of a random search, which may be because the problem is highly complex for such large problems.

A Test Runs

Figs. 10, 10, 12 and 13 shows the fixed-target plots for the different test problems given in the assignment. The algorithms used are with operator combination bitflip mutation, k-point crossover and rank selection as well as a random search. The two test cases with 99 iterations have been removed, the ones displayed are in the order given in the input.csv.

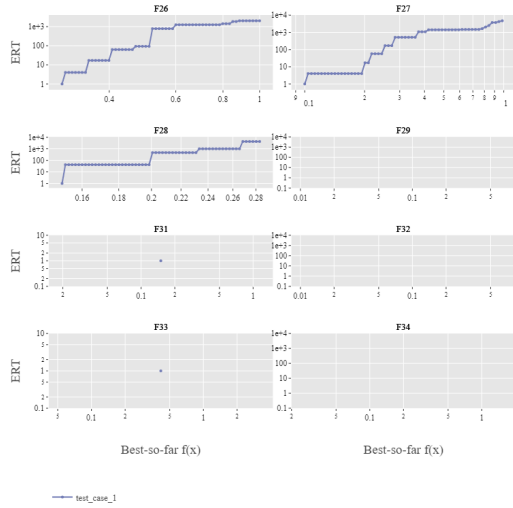


Fig. 10: Fixed-Target result with r-contiguous objective function.

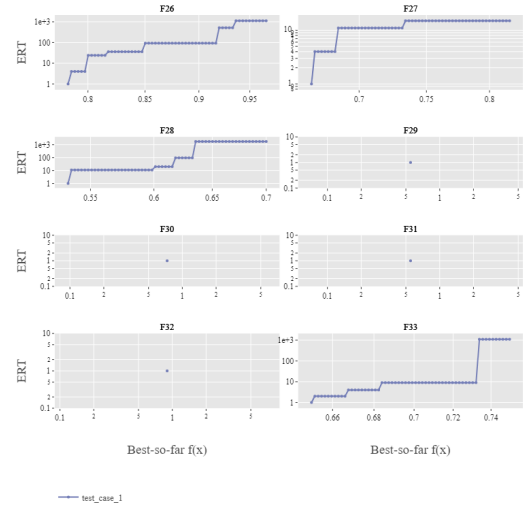


Fig. 11: Fixed-Target result with Hamming objective function.

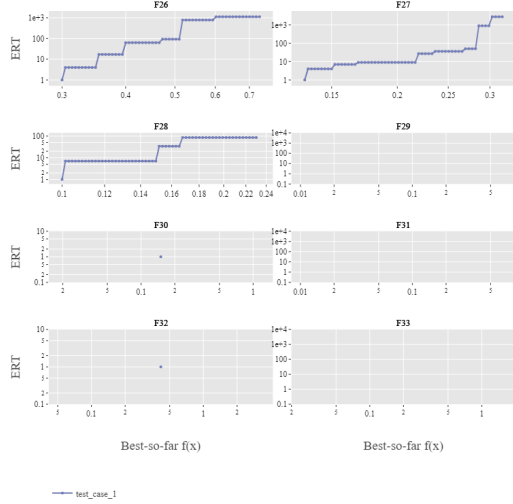


Fig. 12: Fixed-Target result with random search and r-contiguous objective function.

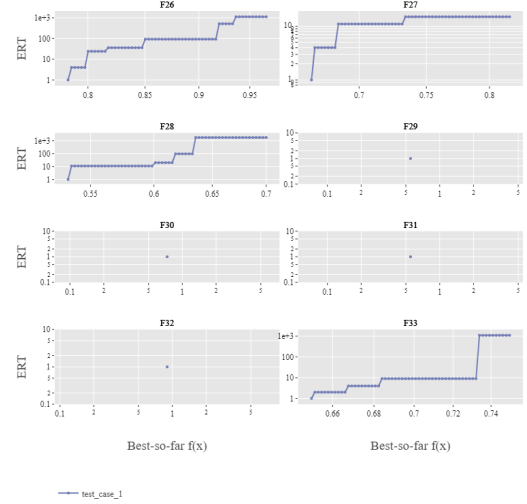


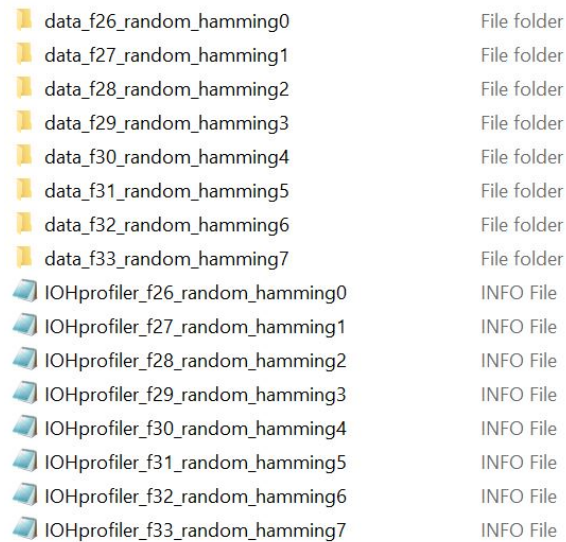
Fig. 13: Fixed-Target result with random search and Hamming objective function.

B Analysis Issues

This appendix explains the issues we had with the IOHanalyzer. We would have liked to be able to upload data from test runs of all the possible algorithms (the different possible operator and object function combinations). This way we could have made a plot to compare the performance of each of the algorithms of the different test cases. However, we did not manage to do so. The following describes the method we used to generate and analyze the data.

The logger was created using `ioh.logger.Analyzer(store_positions=true)` and attached to every problem as can be seen in `implementation.py` lines 267 and 285. For each operator and objective function combination, the algorithm was run and the logger thus activated. This way, an `IOHprofiler_fx_y.info` with corresponding folder containing a `.dat` file was created for each of the input states given in `ca_input.csv`. The folder structure is seen in Fig. 14. The 'functions' are the different test functions/problems (given for the assignment). It thus has the same format as described in the *Data Format* section of the IOHprofiler [1]. However, when uploading the data to the IOHanalyzer, we were not able to get multiple plots in the same graph, but only the different results plotted in different graphs using the *Multiple Functions* tab.

We also tried to do the data generation differently so that instead of containing the 10 different tests on one algorithm, we ran one of the tests on all the different algorithm implementations. This did however not change anything when it came to generating the plots in the IOHanalyzer.



data_f26_random_hamming0	File folder
data_f27_random_hamming1	File folder
data_f28_random_hamming2	File folder
data_f29_random_hamming3	File folder
data_f30_random_hamming4	File folder
data_f31_random_hamming5	File folder
data_f32_random_hamming6	File folder
data_f33_random_hamming7	File folder
IOHprofiler_f26_random_hamming0	INFO File
IOHprofiler_f27_random_hamming1	INFO File
IOHprofiler_f28_random_hamming2	INFO File
IOHprofiler_f29_random_hamming3	INFO File
IOHprofiler_f30_random_hamming4	INFO File
IOHprofiler_f31_random_hamming5	INFO File
IOHprofiler_f32_random_hamming6	INFO File
IOHprofiler_f33_random_hamming7	INFO File

Fig. 14: Folder structure of generated data for the IOHanalyzer.

What we would have liked to do was to plot the results for each operator and objective function combinations in the same graph for each of the 10 input values. This way we could have easily compared the performance and analyzed which combination works best on the different problems.

We realize that we may just have misunderstood how the IOHanalyzer works - which is a shame now that we actually had generated the data. We're still not quite sure what went wrong - we were able to generate all the plots we wanted on the test data already existing for the IOHanalyzer, the only issue was when we uploaded our own data.

References

1. Doerr, C., Wang, H., Ye, F., van Rijn, S., Bäck, T.: Iohprofiler: A benchmarking and profiling tool for iterative optimization heuristics. CoRR **abs/1810.05281** (2018), <http://arxiv.org/abs/1810.05281>
2. Eiben, A.E., Smith, J.E.: Introduction to evolutionary computing: Genetic algorithms. University Lecture
3. Hasançebi, O., Erbatır, F.: Evaluation of crossover techniques in genetic algorithm based optimum structural design. Computers and Structures **78**(1), 435–448 (2000). [https://doi.org/10.1016/S0045-7949\(00\)00089-4](https://doi.org/10.1016/S0045-7949(00)00089-4)
4. Herath, A.K., Wilkins, D.E.: Timetabling with three-parent genetic algorithm: a preliminary study. Proceedings of the 5th International Conference of Control, Dynamic Systems, and Robotics (CDSR'18) pp. 127–1–127–9 (2018)
5. de Luca, G.: Roulette selection in genetic algorithms (2020), <https://www.baeldung.com/cs/genetic-algorithms-roulette-selection>
6. Mallawaarachichi, V.: Introduction to genetic algorithm - including example code (2017), <https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3>
7. Mirjalili, S.: Genetic Algorithm, pp. 43–55. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-319-93025-1_4
8. Obitco, M.: Selection - introduction to genetic algorithms (1998), <https://www.obitko.com/tutorials/genetic-algorithms/selection.php>
9. Prof Thomas Bäck, Dr Anna Kononova, D.V.: Cellular automata. University Lecture (2021)
10. Prof Thomas Bäck, Dr Anna Kononova, D.V.: Evolutionary algorithms: Introduction. University Lecture (2021)
11. Wang, Y.Z.: An application of genetic algorithm methods for teacher assignment problems. Expert Systems With Applications **22**(4), 295–302 (2002). [https://doi.org/10.1016/S0957-4174\(02\)00017-9](https://doi.org/10.1016/S0957-4174(02)00017-9)
12. White, S.H., del Rey, A.M., Sánchez, G.R.: Modeling epidemics using cellular automata. Applied Mathematics and Computation **186**(1), 193–202 (2007). <https://doi.org/10.1016/j.amc.2006.06.126>