Diese Arbeit wurde vorgelegt am
Lehr- und Forschungsgebiet Theorie der hybriden Systeme

Optimierung von Windpark- und Heliostat-Layouts mit evolutionären Algorithmen

Evolutionary Optimization of Wind Farm and Heliostat Layouts

Masterarbeit
Informatik

Juni 2020

Vorgelegt von Presented by	Laurids Vollmann Düppelstrasse 16 52068 Aachen Matrikelnummer: 344500 laurids.vollmann@rwth-aachen.de
Erstprüfer First examiner	Prof. Dr. rer. nat. Erika Ábrahám Lehr- und Forschungsgebiet: Theorie der hybriden Systeme RWTH Aachen University
Zweitprüfer Second examiner	Prof. Dr. rer. nat. Thomas Noll Lehrstuhl für Software Modeling and Verification RWTH Aachen University
Externer Betreuer External supervisor	Dr. rer. nat. Pascal Richter Steinbuch Centre for Computing Karlsruhe Institute of Technology

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen meiner Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Dasselbe gilt sinngemäß für Tabellen und Abbildungen. Diese Arbeit hat in dieser oder einer ähnlichen Form noch nicht im Rahmen einer anderen Prüfung vorgelegen.

Aachen, im Juni 2020

Danksagung

Ich möchte an dieser Stelle meinem Betreuer Pascal Richter dafür danken, dass ich diese Arbeit im Rahmen eines interessanten praktischen Projektes durchführen konnte. Er stand mir jederzeit für Fragen zur Verfügung und hat es immer wieder geschafft mich zu motivieren.
Großer Dank gilt meinen Eltern die mich mein ganzes Studium lang unterstützt haben und damit diesen Abschluss erst ermöglicht haben.

Laurids Vollmann

Contents

1. Introduction

In the wake of energy revolution the importance of power plants relying on renewable energy sources cannot be underestimated. With every year passing more and more new renewable power plants are built. Renewable energy sources have convincing advantages in the long run, and will play a crucial role implementing the changes needed to fulfill the Paris Agreement [7]. Energy production accounts for the largest part of global emissions [3], therefore CO2 neutral energy production is a key factor for the reduction of greenhouse gas emissions. Renewable energy sources have a positive environmental record and only generate construction and maintenance costs for a continuous operation in contrast to traditional power plants that constantly require fuel for energy production. Moreover, wind and central receiver systems can be implemented in areas not fit for agriculture or human living space. Or to rephrase it, they can exploit low quality construction land. While hydropower is currently the predominant form of renewable energy production (Figure 1) it relies heavily on geological features that can be harvested (water basins with a more or less steady inlet). Solar and wind power on the other hand are much less dependant on environmental parameters. Both exhibit a much stronger relative growth. Central receiver systems can be constructed on any sufficiently flat area with a sufficient amount of sunshine hours per year. A dry flat piece of desert is a fitting building ground for a central receiver system [22]. Wind farms offer an additional benefit. They are not even required to be built on mainland, in fact offshore wind farms experience even better wind conditions than farms build onshore. The wind in the ocean is stronger, more steady and its direction has less fluctuation due to the flat surface. While facing higher construction costs, as building a wind turbine on the ocean ground is inherently more difficult and expensive than building on solid ground, offshore wind farms can draw from a nearly unlimited space of possible locations all of which do not affect any other constructions at all. While both solar and wind power plants can be build in almost any region choosing an appropriate site with steady strong wind throughout the year respectively enough sunshine hours is crucial for electricity production. As this thesis only deals with central receiver systems and offshore wind farms, the following examples will be limited to these types of power plants.

1.1. Problem Description

Electricity is a homogeneous good – meaning that it has no inherent quality that is distinguishable between different producers. Price of production therefore becomes the most significant factor when competing on the market. The fact that the electricity is being produced cleanly becomes irrelevant if the production costs become unprofitably high. In order to increase the attractiveness of such power plants beyond the environmental factor, the financial benefit the plant produces needs to be maximized. As a first step, models are used to calculate power plant efficiency by simulating for example the influence of wind turbines on each other. From this simulation the expected revenue of a planned power plant can be calculated. Knowing the expected profitability

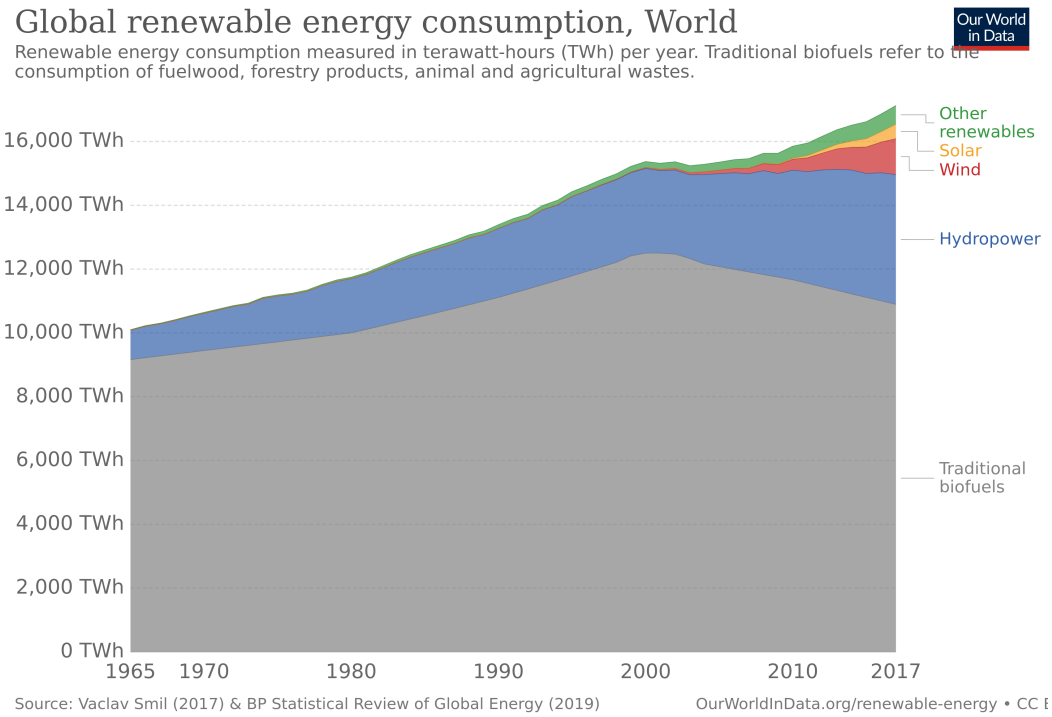


Figure 1: Global renewable energy consumption [10]. Solar- and windpower show a strong updward trend in the last 10 years.

Figure 2: Turbines in an offshore wind farm

of a given project beforehand with a certain confidence already increases its chances of being realized mainly because risk management becomes much simpler. Computing optimized layout parameters (e.g. placement of turbines or mirrors) in an automated fashion is the next logical step to take. For both the wind farm and the central receiver system the construction parameters mainly consist of a layout that dictates where wind turbines, respectively mirrors are built. Automating the generation of an optimized layout reduces the amount of human work needed in the planning of a power plant without sacrificing the consideration of environmental parameters of the location. This enables companies interested in the construction of power plants to reliably compare a larger number of different building sites. The certainty of the expected revenue as well as the certainty that the power plant itself is built in an optimal way greatly increases the attractiveness of such a project. In previous works, simulations that compute the energy production of a wind farm respectively central receiver system configuration have been developed and successfully evaluated against industry standards (Soltrace [9] for central receiver systems and Openwind [6] for wind farms). The simulations are now an integral part of an optimizer that (among other strategies) employs a *genetic algorithm* (also called *GA*) to obtain optimal layouts for either power plant. In the following, we will give a quick overview of the setup of both power plants, potential constraints of building sites and the parameters to be optimized.

1.1.1. Offshore Wind Farms

In addition to the wind turbines that are clearly required in a wind farm and apart from the cabling there is no other system needed for power production. As the turbines are enclosed systems on their own, the only other problem to be solved is an efficient

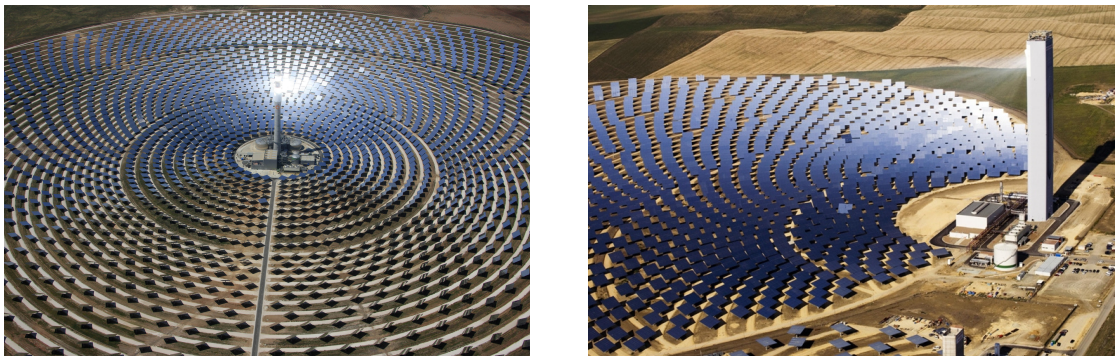
cabling of the turbines. This problem however, is not considered here.

The turbines are defined by height above sea, rotor diameter, rotor blade profile and ultimately their energy production characteristics. While these parameters are used for the computation of energy production in the simulation, they are fixed for the whole wind farm and not part of the optimization process.

The building site is defined by an area wherein turbines can be placed, and restricted areas within the building area where the construction of turbines is not possible (steep surface, shipwrecks, ...). Moreover, the topography of the ocean floor influences the construction costs. The result of the optimization process consists of a set of positions that describe the placement of each turbine. The amount of turbines to be placed is fixed and needs to be specified as a parameter.

The transportation of electric energy to the mainland with all its inherent challenges like the potential necessity of a transformer station is also not considered here. However, the cost of the internal cabling is considered in the simulation.

1.1.2. Central Receiver Systems



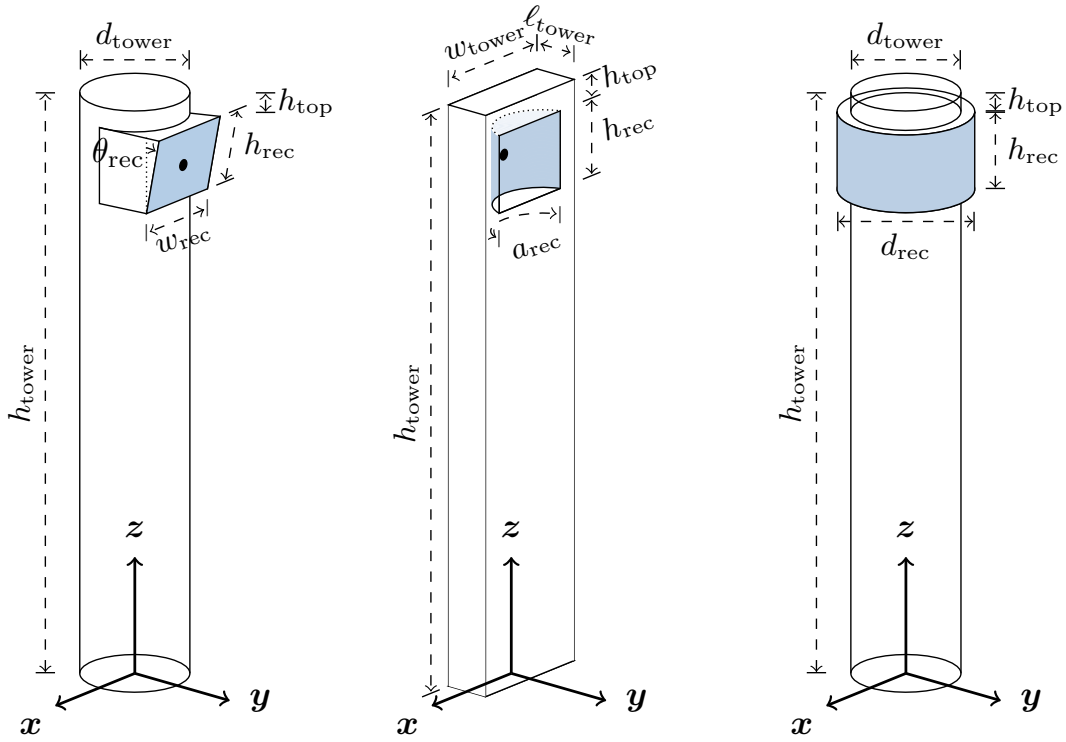
(a) Tower at the center of heliostat field

(b) Tower at the edge of heliostat field

Figure 3: Pictures of real-life central receiver systems

Similar to a wind farm a central receiver system aims to produce electrical energy (and not just thermal energy). Its setup is more complicated than a wind farm in the way that it comprises more systems working together. The largest amount of building space is occupied by the set of large mirrors that are meant to reflect the sunlight that arrives at their surfaces and concentrate it towards a small area where it is collected.

The reflectors are called *heliostats* and they reflect the sunlight towards the *thermal receiver* which is the second element in the energy production pipeline. The thermal receiver is usually mounted on a tower which can be situated inside the field or at its edge as seen in Figure 3. Depending on the tower's location different designs for the thermal receiver are necessary (see Figure 4). Its task is to absorb the radiant energy and convert it into heat. The heat is ultimately used to power a steam turbine that drives a generator.



(a) Flat tilted cavity receiver (b) Cylindric cavity receiver (c) Cylindric external receiver

Figure 4: Different receiver types on different tower types. The figure is derived from [47]

In order to convert the heat gained from the sun to electrical power, an intermediate step is required. The thermal receiver is not used to produce steam from water directly but rather heats a *heat transfer fluid*. Molten salt is often used as heat transfer fluid but other materials are possible. The usage of aheat transfer fluid as an intermediate step offers the advantage of better storage capabilities which directly result in a higher potential adaptability of the whole system. Adaptability in electricity production is an advantageous property as it allows for different production and consumption characteristics. Energy consumption is highly correlated with time of day or singular events (e.g. television coverage of sport events cause spikes) therefore adaptability to consumer needs is of particular interest.

After the transfer of heat into the heat transfer fluid and potential storage thereof, a heat exchanger is used to produce steam driving a conventional steam turbine that is connected to a generator ultimately producing electricity.

Regarding the optimization problem, again the parameters of the individual systems are not optimized as they are fixed. This includes size and properties of the heliostats as well as characteristics of the systems occurring later in the process. Optimized are only the positions of the heliostats.

Note that for the sake of simplicity we will refer to both wind turbines and heliostats as *instances* as their roles in the optimization process are equivalent.

1.2. Related Work

When we talk about the optimization of a power plant the variables we are optimizing is the set of positions where the *instances* are placed and the target value is the *Levelized Cost of Electricity* which is calculated by the model. The possible methods for turbine or heliostat placement optimization do not really vary as the abstract mathematical problem is the same. In both cases the simulation of a valid solution (a valid placement of heliostats/turbines) takes a comparably long time and the space of possible solutions is large.

1.2.1. State of the Art for Wind Farm Layout Optimization

The process that produces an optimized wind farm layout consists of two main components. The optimization algorithm that decides how the layout shall be altered to achieve a better fitness, and the simulation that evaluates a layout to compute energy production metrics. The model used by us is called the *PARK* or *Jensen* model [32]. Other models include the *viscosity model* [12] and the *deep-array wake model* [21] but according to Maghnie [36] the *PARK* model is the most popular model used in wind farm optimization [28] and provides a good trade-off between speed and precision [50].

Investigation of wind farm layout optimization dates back as far as the late eighties with new attention drawn to the area after the work of Mosetti et al. in 1994 [39, 28, 36].

The methods used for the optimization range from simple rules over parameterized mathematical patterns to evolutionary optimization with subsequent post-processing.

Patel presents a crude rule of thumb for turbine placement. The optimal placement is claimed to be achieved using offset rows with eight to twelve rotor diameters distance in windward direction respectively three to five rotor diameters distance in crosswind direction as shown in Figure 5 [44]. Applying this rule results in a grid layout of the turbines.

Wagner et al. make use of the same wind farm model as we do but relies solely on a local search algorithm to optimize the layout of wind farms. As an additional contribution they claim to have improved the runtime of the model significantly by only reevaluating one turbine in each optimization step. The local search algorithm in use was constructed with the problem in mind to allow for problem specific optimizations [54].

Evolutionary algorithms in general and specifically the subclass of genetic algorithms are popular in wind farm optimization [53]. Other evolutionary approaches include *biogeography based optimizations* [13, 14] "which are inspired by the migration of species over time [51]" [36]. All approaches that employ evolutionary optimization are sensitive to the quality of their initial solutions [36] which makes fast generation of non-optimal-fitness solutions relevant nevertheless.

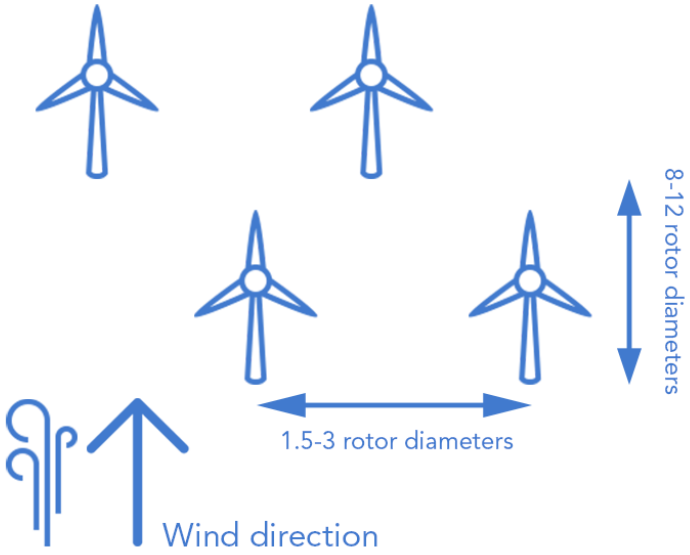


Figure 5: Rule of thumb for turbine placement.

Initial solutions are usually obtained by means of "grid patterns [39, 25, 56] or random positions [52]." [36]

A different approach is used by Fagerfjäll [20] and Ozturk and Norman [43] who make use of circle packing to produce a solution while still enforcing the minimum distance constraint.

According to Maghnie circle packing proves to be an effective way to generate wind farm layouts, however, there was no research conducted on the optimization of the circle packing mechanism [36]. Patterns in general are a fast way to generate reasonably well-performing layouts which can either be used directly or as an initial solution for further optimization steps. They also ensure the adherence to the minimal distance constraint between any two turbines.

Neubert et al. draw a comparison between symmetric layouts that are meant to please the beholder while still maintaining a high energy output, and stochastic layouts that do not fulfill symmetry conditions [41].

The use of genetic algorithms as means of turbine placement optimization has already proven successful for wind farms and has been implemented by various researchers including but not limited to [24] (González et al.) and [37] (Marmidis et al.).

Chen et al. goes one step further and considers turbines of different height in the GA optimization process.

A different evolutionary approach is implemented by Wan et al. [57]. They make use of an algorithm called *particle swarm optimization* (PSO) which is a different kind of evolutionary algorithm. Each particle in the swarm is one independent solution to the layout problem. The solutions may be obtained from patterns or be generated randomly. Additionally, a particle has a random velocity whose dimension scales with the number of turbines. Each iteration the fitness of each particle is recomputed and

the particles are moved according to past fitness values of this particular particle and the whole swarm. Particles will favor positions where they produced a good fitness in the past and the swarm will as a whole move to regions with better fitness. After several iterations the individual particles positions will converge. This approach seems to become increasingly popular in recent research [31, 58].

A well-known algorithm that is used for the post-processing step is *simulated annealing* which tries to replicate properties of hot material cooling down. Turbines are moved randomly, they are kept if the solution has improved, otherwise they are still kept with a certain probability that shrinks over time (cooling). This approach is used by Rivas et al. [48] and Bilbao and Alba [15].

1.2.2. State of the Art for Heliostat Field Layout Optimization

Again the problem is split into two sub-problems, simulation and optimization.

While the problems of turbine and heliostat placement are similar and should be able to be optimized using the same algorithms with minimal changes, the work on heliostat field layout optimization is much more scarce. Moreover, the research does not seem to date back as far as wind farm layout optimization.

Pitz-Paal et al. already use a genetic algorithm approach [46]. But they only use it to optimize a set of variables that among other things influence the heliostat pattern. Optimization of individual reflectors is not part of their optimizer. Apart from the genetic algorithm, they implemented also two other optimization algorithms one of which is a hill-climbing algorithm. The latter one was used to further explore the solutions produced by the GA as its parameter space was discretized.

According to Franke [22] all central receiver systems built to this day rely on patterns to specify the heliostat positions instead of the tweaking of individual heliostat positions. However, the patterns offer parameters that modify their shape. They include but are not limited to hexagonal, spiral, radial or alternating rows patterns (see Figure 6) [22]. A biomimetic pattern is introduced by Noone et al. [42]. Lutchman mentions an iterative approach which he calls the *field growth method* [35]. This method does not seem to have undergone thorough research so far. A hybrid approach that combines a GA and *particle swarm optimization* is presented by Li et al. [33].

1.2.3. Comparison to Our Method

In contrast to the established research which uses at most two different optimization techniques to optimize the layout, our approach of a Multi-Step Optimizer tries to salvage the benefits of multiple individual optimization mechanisms while minimizing the effect of their drawbacks.

1.3. Goals of This Thesis

Genetic algorithms offer a plethora of different parameters and extensions to influence their behavior [17]. Understanding how individual parameters influence each other

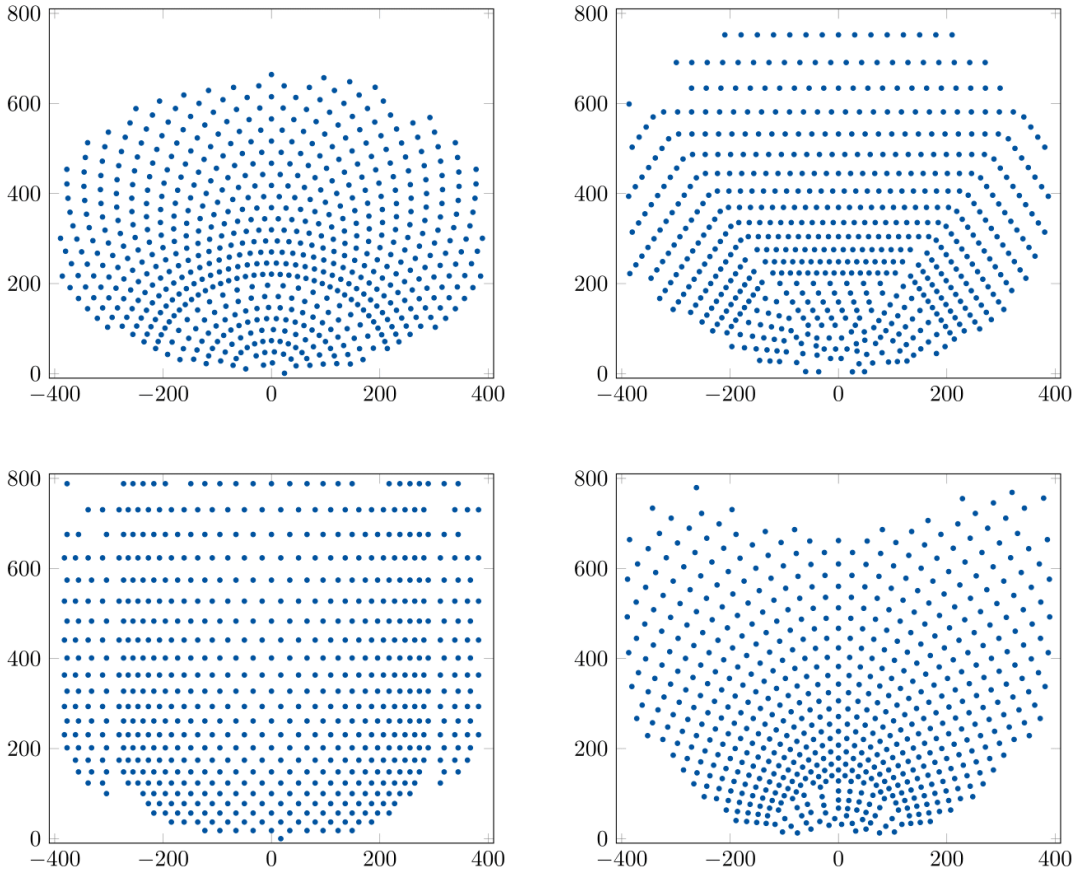


Figure 6: Different heliostat patterns. The tower is always located at $(0, 0)$. (all graphics from [22])

is still element of active research. There seems to be consensus that the choice of appropriate *mutation* and *crossover* routines is critical for the performance of the GA [30, 19]. Deb and Agrawal also mention that at the time of publication parametric studies that include pretty much random testing of different GA parameters are used to optimize a GA for the problem it is supposed to solve. The same approach is being used by us to enhance the performance of our GA. The performance of a GA strongly depends on the choice of hyper-parameters which we set out to determine. Additionally, we want to determine the place of the GA in the greater scheme of the Multi-Step Optimizer as we need to tweak its behavior keeping in mind that it is only one piece in an optimization chain.

Unfortunately, the findings of previous GA parameter configuration research are not applicable as our gene encoding and crossover operation are non-standard and do not adhere to common best-practices (which is unfortunately unavoidable, more on this later in Section 3.5.4), making existing recommendations meaningless. Furthermore, these parameters usually diverge between different families of problems.

Additionally, we implement changes in the codebase to add the modifications to the

GA that we want to examine, and also perform general refactoring and cleanup work.

1.4. Outline

Section 2 will cover the underlying models used to compute the performance of a given offshore windpark or central receiver system. Then in Section 3 we present the structure of the genetic optimization, what its individual steps do and how it can be configured. Section 4 explains the principle of a Multi-Step Optimizer and what the purpose of the GA in the greater picture is. Section 5 gives an overview of work on the codebase that was carried out over the course of this thesis. In Section 6 we present the test cases that are used to evaluate our optimization process. In Section 7 we present the main part of this work and show which experiments were conducted in order to find optimal GA parameters. In the end, Section 8 summarizes our results and Section 9 gives an outlook on possible future work that is beyond the scope of this thesis.

2. Models

This section will give an overview of the underlying simulation models for offshore wind farms and central receiver systems. The tools used to simulate and optimize central receiver systems and offshore wind farms are called *SunFlower* and *WindFlower*.

Both models take the characteristics of the construction site, the weather and the individual components as input. Additionally, they expect a set of (x, y) positions that represents the positions of the instances to be simulated. The output of the simulation is a set of multiple economically relevant values including *Annual Energy Production* as well as internal metrics about the goodness of individual positions. More on this later in Section 2.1.4.

2.1. Offshore Wind Farm Model

This section summarizes the model description given by Netz [40]. The model that computes the power output of a given wind farm consists of four different sub-models namely wind, wake, power generation and cost model. The models form a logical chain and need to be evaluated in order. Each model processes the output of the previous model. Additionally, problem parameters are defined. They define the area in which the wind farm is to be constructed as well as restricted areas where for an arbitrary reason no turbines are allowed.

2.1.1. Wind Model

The wind model is based on real measurements made by the Fino3 measurement station in the North Sea that collects real-life information about wind speed and wind direction at different heights[2]. The raw wind data consists of two list of (timestamp, min-**value**, max-**value**, deviation) tuples where **value** is wind speed in the one and wind direction in the other list. This representation is impractical and takes way too much space for a constant reevaluation of the subsequent models. The data is therefore converted in a format that allows a fast lookup of specific wind speed and wind direction combinations.

The data passed to the next model consists of wind direction bins that span over a range of directions. Each bin is associated with the probability that wind is coming from that direction at any point in time. Additionally, each bin is associated with an approximation of the wind speed in that direction. Weibull curves are used to describe the wind speed and -direction (see Equation (1)). This is a standard approach that is widely used when a mathematical approximation of wind speeds is needed [26, 49]. Thus, the model consists of two parts, the probability of the direction and the two Weibull parameters that describe the curve in each bucket. The number of individual bins is one of the essential if not the most important parameter to influence the accuracy-speed trade-off of the model.

$$h_{\lambda,\kappa}(x) = \frac{\kappa}{\lambda}\left(\frac{x}{\lambda}\right)^{\kappa-1} e^{-(x/\lambda)^{\kappa}} \tag{1}$$

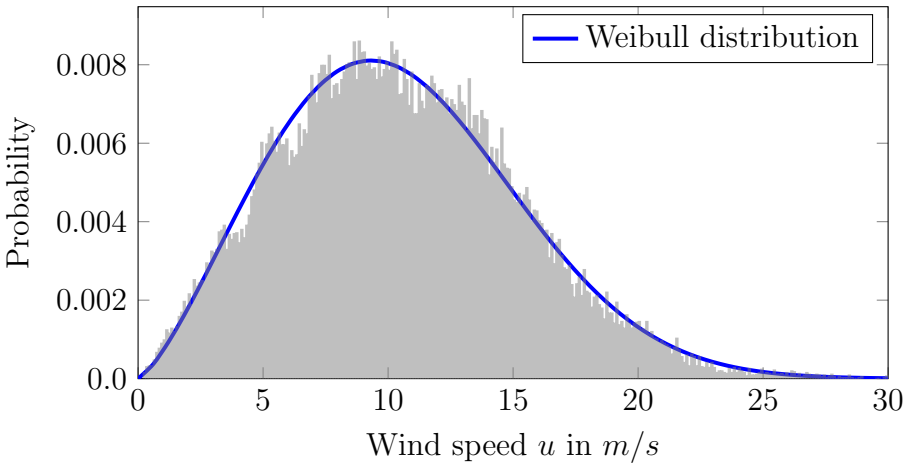


Figure 7: Weibull approximation of measurements.

2.1.2. Wake Model

Figure 8: Rare weather condition visualizing turbulences of the turbines.

When a turbine converts the kinetic energy of the wind into electric power, naturally the energy of the wind decreases. The turbine affects a cone of air behind it, its *wake* as seen in Figure 8. In the wake the wind speed is lower than in front of the turbine. For a valid simulation of a wind farm the mutual influence of wind turbines on each other needs to be considered when calculating the total power produced. We use the farm model developed by Jensen [32]. Computing the influence of turbines on each other is by far the costliest of the sub-models. To reduce complexity it makes several

simplifications. Firstly, it only computes an approximation of the windflow in the wake. Instead, of a flow field that would be able to depict turbulences at the edges of the wake, only a wind speed that solely depends on the horizontal distance from the rotor and the wind speed in front of the turbines is computed. Moreover, a constant wind speed (over the span of the rotor) is assumed. Considering that the rotor ranges from 20m to 120m above sea level and the wind data used for simulation was captured at a height of 100m this is a reasonable simplification. The model also requires a ground roughness to compute the amount of friction between air and ocean. This is set to a fixed value that proved to be a sufficient approximation [32]. Aforementioned abstractions require the wind turbines to be placed at a distance of at least three rotor diameters which in our case is 300m. For smaller spaces the model does not produce a valid output. One computation of the model only produces a result for one specific wind direction, so multiple runs are required to give estimations for all wind directions. Tweaking the number of different wind directions simulated is the essential parameter to compute a result of lower accuracy more quickly.

The loss of energy arising from the time windows when the wind is changing direction but the turbines are not yet adjusted is considered by assuming a fixed loss percentage.

2.1.3. Power Generation Model

In the power generation model the attenuated wind speed calculated by the wake model is mapped to the amount of energy produced by each turbine. This is the simplest sub-model as the correlation can easily be represented by a function $f(speed) = power$ in W. This function is unique to the type of wind turbine. An example can be found in Figure 9.

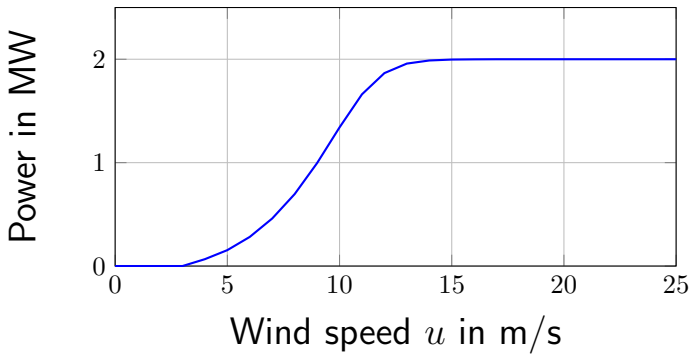


Figure 9: Example of a power curve mapping wind speed to electrical output.

2.1.4. Cost Model

Purpose of the *cost model* is to give a realistic estimation of value and revenue of the project. It provides different target functions that can be used to evaluate the quality of a given solution. Basis for all of them is the *Gross Annual Energy Production* (AEP)

which is computed with the output of previous models. First this value is adjusted with a loss percentage that compensates for maintenance, technical problems and inaccuracies of the models. The adjusted value is called *Net Annual Energy Production*. This is already one of the target functions. The *Levelized Cost of Electricity* ($LCOE$) measures the price per kWh over the lifetime of the farm. *Net Present Value* (NPV), *Internal Rate of Return* (IRR) and *Payback Period* (PP) are common characteristics used in financial mathematics to describe a project. Refer to [22] for the exact formulas.

For the optimization usually the AEP is used for fitness calculation. The AEP is the foundation upon which the other values are computed. It is also important to note that we are able to extract the contribution of each individual wind turbine to the overall AEP, this will be relevant later for the optimization.

2.2. Central Receiver System Model

The sub-models have a strict logical order and pass their output to the next model. Problem parameters consist of the region in which the power plant is planned and again restricted areas where no structures can be constructed. Moreover, information about the elevation of the ground can be provided.

2.2.1. Optical Model

The first sub-model contains information about the relevant environmental parameters at the given location. As the power plant draws its energy solely from the sun we need to describe its path and irradiation (luminosity) over the day. The position of the sun is given by two angles from which we derive azimuth - the position w.r.t. the geographic North and altitude - the position w.r.t. the horizon.

2.2.2. Ray Tracing Model

The task of the ray tracing model is to compute the amount of radiation that reaches the thermal receiver on top of the tower. Multiple receiver types exist, three of them are illustrated in Figure 4. It needs to consider the position of the heliostats installed on the ground at different positions of the sun during the day. At each position rays are traced from the sun onto the heliostat field and if their path is not obstructed by anything, ultimately onto the thermal receiver. Our model implements different ray tracers but here we only use a *Monte Carlo Ray Tracer*, meaning that the direction of rays simulated is not given by a pattern or formula but rather randomly generated.

One heliostat usually consists not of one but many small mirrors, each with a different orientation. The possible shapes of the smaller mirrors called facets are either triangular or rectangular. The simulated rays from the sun are reflected by the small mirror segments towards the receiver. The loss introduced by the partial reflection of the light is modeled using the *reflectivity* percentage which is usually around 95%.

Cosine effect A major source introducing loss of efficiency is the cosine effect. It describes the decrease of reflective area that occurs when the heliostats are not oriented perpendicular to the direction of the incoming light. Due to the movement of the sun and the fact that they need to reflect light towards the receiver the mirrors cannot be installed in a fixed position but need to rotate to minimize the loss of reflective area at different sun position.

Shading and blocking There are different obstacles that can hinder the propagation of light towards the tower, all of which have to be considered. This includes the shadow cast by the tower on the heliostat field. Moreover, heliostats can influence each other in two different ways. They may either *block* a ray on the way from their intended heliostat towards the receiver. A ray is *shaded* if it hits a heliostat different from the one it was intended for, the worst case scenario would comprise two heliostats placed in close proximity on one line with the dominant sun direction. The heliostat in the front would shade the other one almost completely. The last alternative is a ray that hits its intended heliostat and is not blocked by any other heliostat but simply does not hit the receiver.

Practical obstacles The ray tracing model also considers some more practical problems that influence the amount of energy that reaches the receiver. Firstly, over time dust begins to cover the surface of the reflectors reducing their reflective capabilities. For larger heliostat fields it is also necessary to consider atmospheric attenuation by particles in the atmosphere. Imperfections in the reflective surface of the heliostats also introduce an error when light is not reflected in the expected direction. This can also happen when the rotatable heliostats end up not pointing in the ideal direction due to deviations in the control of the rotation.

2.2.3. Thermal Model

After the ray tracing model produced a flux map for the area of the thermal receiver in the next step we compute how much of the incoming radiation energy is available for energy production and how much falls victim to various losses. Simplified the thermal receiver is made from a set of parallel tubes through which the heat transfer fluid flows in alternating directions. The first source of loss is the fact that the receiver is not able to absorb 100% of the incoming radiation and will reflect a certain percentage. Moreover, after absorbing the energy from the light rays it is directly re-emitted into the environment via infrared radiation. Energy is also lost through convection; the tubes conducting the heat transfer fluid are warmer than the ambient temperature and will therefore heat up the air around the tubes. The warmer air around the tube then dissipates into the still ambient temperature air further away from the tube removing energy from the heat transfer fluid in that process.

Storage As previously mentioned, one advantage of the two-fluid design is the ability to efficiently store the thermal energy using insulated tanks filled with hot heat transfer

fluid We assume the storage to be large enough and disregard losses originating from heat dissipation from the tanks.

2.2.4. Electrical model

The last part to be modeled is the conversion from heat transfer fluid into electricity using a steam turbine and an generator. It consists of a simple lookup table which is partly visualized in Figure 10. The data is bilinearly interpolated between data points.

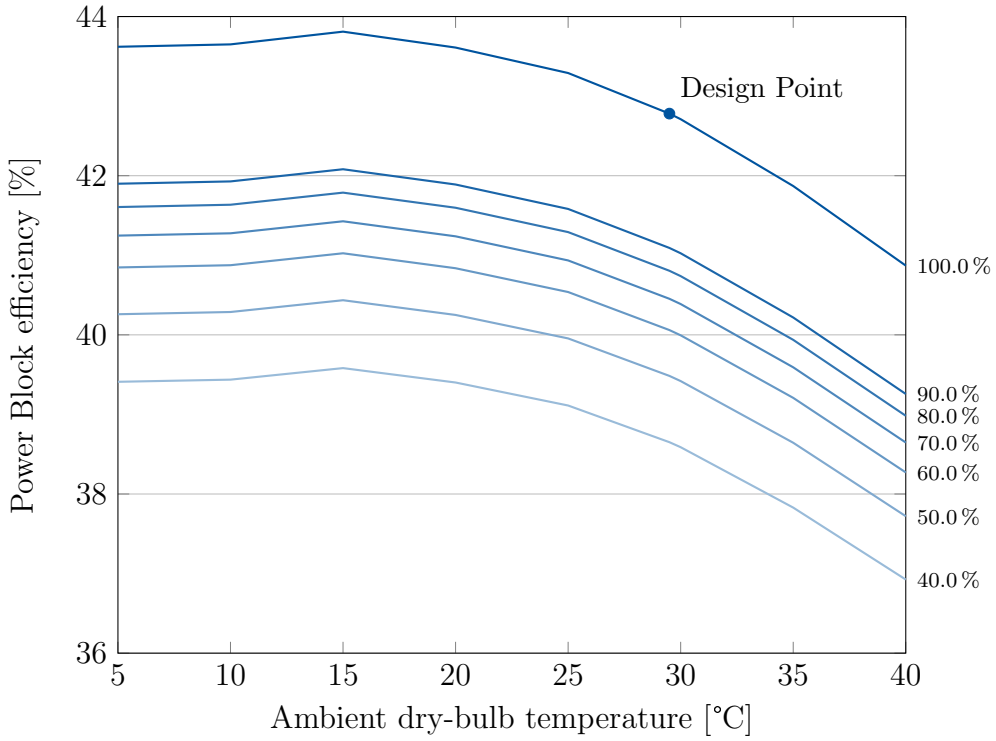


Figure 10: Characteristic Diagram of a 100 MW_{th} power conversion unit. The lines represent the temperature-dependent efficiencies for different loads. (Graphic from [22])

2.2.5. Cost Model

The cost model is analog to the offshore simulation again yielding *AEP*, *LCOE*, *NPV*, *PP*, and *IRR* as output. Like in the wind model, we can obtain the contribution of individual heliostats to the overall fitness.

3. Genetic Algorithm Overview

In this section, we are going to present the general mechanics of genetic algorithms and the adaptions necessary for our application. The initial work towards genetic algorithms was performed in 1960 by Holland [29].

3.1. Evolution in Nature

In a simplified way, in nature every member of a species is defined by its set of *genes*. They decide which physical and character traits it exhibits. The multitude of genes found in one individual is called his *chromosome*. The multitude of all occurring genes in the species' population is called the species' *gene pool*. The species lives in a certain environment whose challenges it has to face. There will always be chromosomes that are more suitable for some environment than others. Possibly because one individual succeeded in developing a completely new physical trait or because a mutation resulted in a better social strategy — while others did not.

"Succeeding" in nature always means better survivability and therefore more chances to produce offspring. This we call *fitness*. The fitness in the optimization problem for wind farms and central receiver systems corresponds to the annual energy production of the power plant or the resulting economical profitability of the power plant. To maintain and improve the fitness over time and adapt to environmental changes nature uses the principle *survival of the fittest*. Usually the individuals that are better adapted to the environment live longer and thereby have a greater chance to propagate and pass their genes to the next generation. This causes a gradual improvement of the whole species (gene pool of the species).

However, passing unaltered genes to the next generation is ineffective. This is why almost all non-primitive organisms can only produce *offspring* using two individuals of their species. When two individuals *breed* and produce offspring their chromosomes combine in some way to form new genes that contain (hopefully beneficial) features from both of their parents. The survivability of the combination of parental genes will, with some variety, be similar to the ones of their parents. Some offspring will perform better than both of its parents, some will perform worse. Again, better performing offspring will spread their genes with a higher probability. Thus, *recombining* features of the parents can result in a better performing individual.

In addition to the recombination by breeding sometimes *mutations* are introduced. Mutations are random changes to some genes in the chromosome. They have a small random impact on the fitness of the individual to help avoid low diversity in the gene pool. Every once in a while a mutation randomly improves the fitness of the individual and by survival of the fittest these mutations will propagate through the population of the species.

One of the main problems that hinders successful evolution is a limited variability in the gene pool. A small gene pool impairs the species ability to develop new beneficial genes as mutations only happen slowly over time and nature has no other way of randomly introducing genes so adaption becomes harder or impossible (the gene pool converges to all individuals being equal), compare Figure 13 for a mathematical representation of that problem.

Genetic algorithms Genetic algorithms try to mimic the evolution process found in nature hence the more general name *evolutionary algorithm*.

They are used to optimize problems where the solution space is too big for exhaustive search to be viable. The problem consists of parameters that constrain the set of solutions and determine the fitness of individual solutions. However, we need to be able to evaluate the *fitness* of a given solution in a short amount of time. The fitness function takes a valid solution as input and computes the quality of that solution w.r.t. the given problem. In our case the fitness function corresponds to the simulation of the power plant with given environmental parameters. The higher the fitness of the solution the better. Different fitness functions can produce very diverse solutions on the same problem. As solutions correspond to individuals in nature, their fitness is the survivability of the individual. The environment is the combination of problem parameters and the fitness function that evaluates solutions. Each iteration of recombining the existing population into new individuals is called a *generation*.

3.2. Similarity Between Wind Farm and Central Receiver System

We do use the same algorithms for both the wind farm and the centralreceiversystem. Employing the exact same algorithm for two seemingly different problems may sound strange. However, from a mathematical point of view the problems are very similar, Figure 12 illustrates this. In both cases the solutions consist only of a set of unordered positions. These are the only mutable variables in the optimization process, no other variables are changed. The restrictions imposed on the positions are similar, too. Both turbines and heliostats may not be placed in direct vicinity of each other. Moreover, maximizing pairwise distance between instances has a beneficial effect in both applications. Turbines and heliostats placed close to one another will steal resources from each other.

From the optimizers point of view both simulations can be treated as a blackbox that is fed a set of positions and spits out a score for this particular set. Nonetheless, the similarities do not guarantee an analogous performance of the GA. After all the two simulations employ completely different methods to compute the fitness. Another substantial difference is the dimension of a solution. A wind farm usually consist of about 100 turbines and even the largest at the time of writing do not exceed 200 turbines. Centralreceiversystems on the other hand can reach more than 100 thousand individual heliostats.

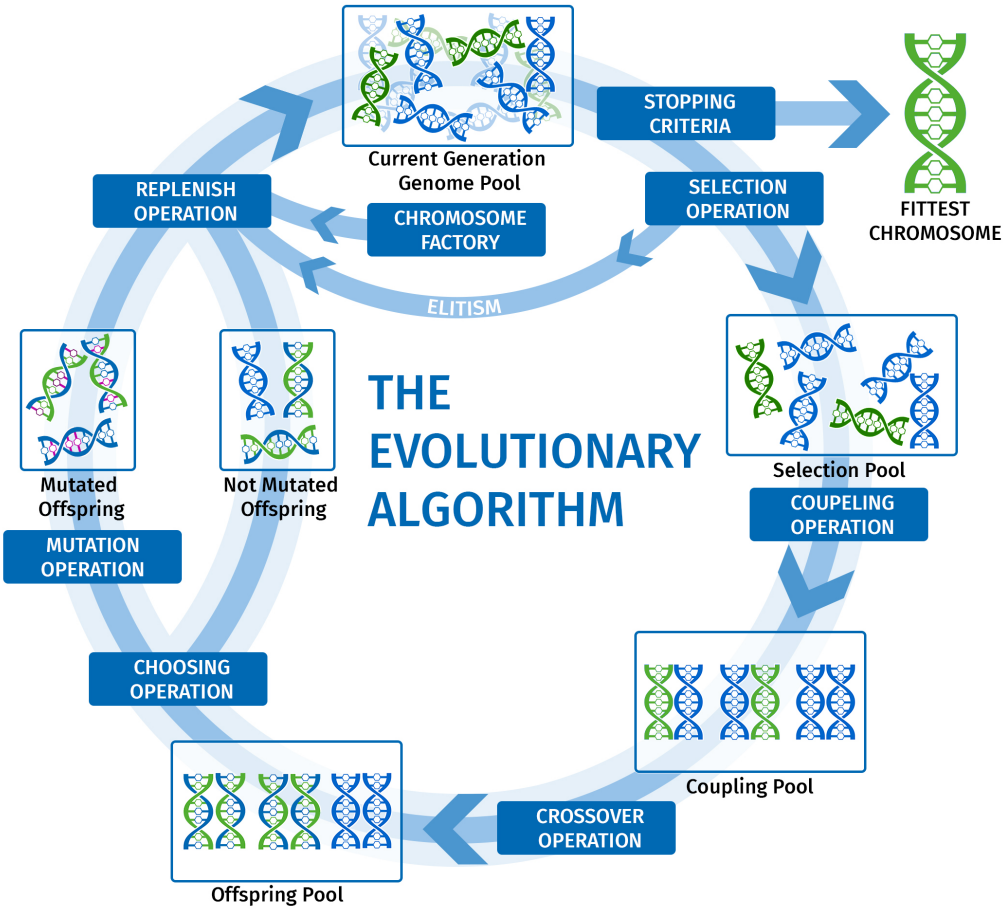


Figure 11: Simplified iteration of an evolutionary algorithm. The algorithm starts at the top with the current generation. Following a test for the *Stopping Criteria*, each chromosome is evaluated and ranked. The best chromosomes are copied to the next generation unchanged (*Elitism*). Based on the ranking chromosomes are selected (*Selection Operation*). Only the selected chromosomes are paired to couples (*Coupling Operation*) and recombined to form new Chromosomes (*Crossover Operation*). A part of the offspring is chosen (*Choosing Operation*) to get mutated (*Mutation Operation*). All offsprings are then added to the new generation. Some randomly generated ones (*Chromosome Factory*) are injected into the new generation as well. (Image and caption from [40])

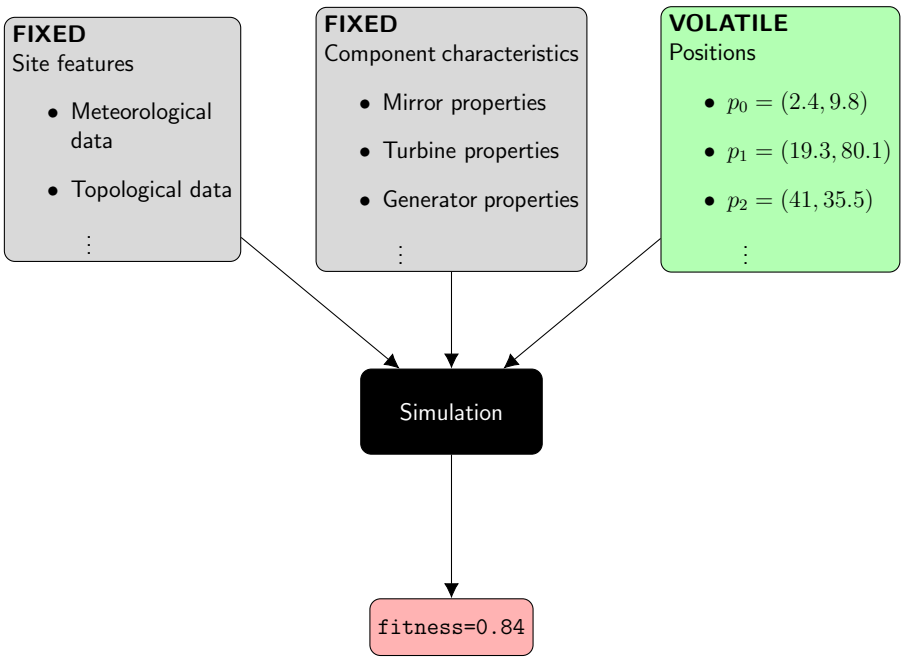


Figure 12: Illustration of a simulation of a wind farm / central receiver system as a black box

3.3. Fitness Functions

The fitness function needs to represent the quality of a solution by assigning a value from 0 (worst) to 1 (best). In this section, we present the fitness functions used for wind farms and central receiver systems and argue why it makes sense to use them.

Wind farm When we compute the simulation for a wind farm WindFlower gives us the option to choose whether we want to use a wake model or not. For the computation of the actual energy output we need to perform the wake computation. To compute the fitness of a solution however, we compute an additional simulation without wake model. This gives us the theoretical perfect score of a solution because we completely disregard detrimental interactions between the turbines. Dividing the actual result with wake model by the theoretical optimal one gives us the fitness value. It has a guaranteed upper bound of 1 because no solution can exceed the energy output of the simulation that does not take the wake effect into account. Fitness computed this way also trivially has a lower bound of 0 as we cannot produce a negative amount of energy. To obtain the fitness of a singular turbine we just divide its actual energy production by its theoretical maximum.

Central receiver system The ray tracer shoots a certain amount of rays towards each heliostat. The rays may either hit the heliostat and be reflected towards the receiver which corresponds to a contribution of energy to the total output or either be shaded (the ray hits another heliostat first), be blocked (the ray hits the back of

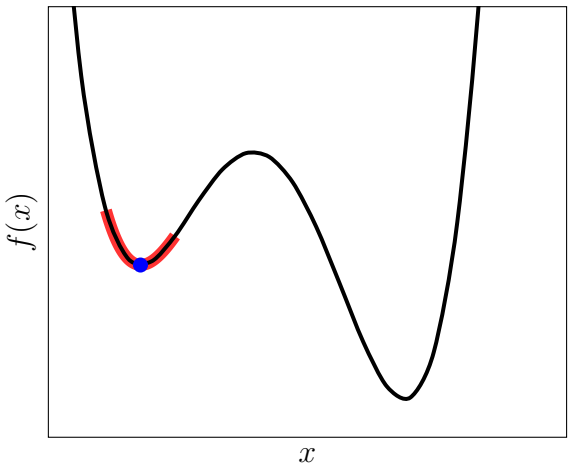


Figure 13: Local optimum problem: The global optimum is not reachable from the local optimum (blue) via small changes in x (reachable area in red).

another heliostat on the way back), hit the back of the receiver tower on its way to the heliostat field or plainly miss the receiver after being successfully reflected back by the heliostat. All the latter ones do not correspond to a contribution of energy. We now divide the number of rays that hit the receiver by the number of rays shot in total. Similar to the wind farm we can compute this metric for the whole heliostat field or individual heliostats providing us with the fitness metric for the GA.

3.4. Encoding Positions

One goal of Netz' analysis [40] was to evaluate performance and usability of discrete vs. continuous position encoding for the instances (in his case only wind turbines). In both cases the chromosomes are a set of positions without any ordering meaning that a chromosome is equivalent to a randomly shuffled version of itself. As a result he ruled out the continuous encoding as unsuitable. In comparison with the discrete positions it exhibits slower convergence rates and worse final fitness. In the continuous mapping an instance can potentially be placed anywhere in the allowed building area. This greatly increases the solution space slowing down the GA. In contrast, the discrete mappings divide the area in distinct cells with a configurable precision (number of cells). Instances can then only be placed at the center of a cell. Thus, reducing the amount of possible solutions.

3.5. Phases in GA

In the following we describe the general process of a genetic algorithm and how it is influenced by the parameters given in Table 1.

The problem of low gene pool diversity Similar to the process in nature, a gene pool with low diversity is something to avoid at all cost. If the algorithm finds a local

Domain	Notation	JSON parameter	Description
$\mathbb{N}^+$	N_{pop}	population_size	Number of chromosomes in the population
$\mathbb{N}^+$	$N_{\text{P\_elite}}$	parent_elitism	Amount of elitism in selection pool
$\mathbb{B}$	$B_{\text{prefer\_better}}$	prefer_better_parents	Choose parents for breeding depending on their performance
$\mathbb{N}^+$	N_{elite}	elitism	Number of elite chromosomes, that are passed through to the next generation
$[0, 1]$	P_{rand}	random_fraction	Percentage of random chromosomes
$[0, 1]$	μ	mutation_strength	Mutation strength (of individual mutations)
$\mathbb{N}^+$	$n_{\mu_{\min}}$	pos_to_mutate_min	Minimum number of genes to mutate
$\mathbb{N}^+$	$n_{\mu_{\max}}$	pos_to_mutate_max	Maximum number of genes to mutate
$[0, 1]$	p_c	mutation_rate	Mutation rate (probability to get chosen for mutation)
$\mathbb{N}^+$	ρ	discrete_map_resolution	Results in ρ^2 distinct positions
$\mathbb{N}^+$	$N_{\text{max\_it}}$	stopping_criteria_max_it	Maximum amount of iterations in the GA

Table 1: Tweakable parameters of the GA.

optimum of the fitness function in the beginning and proceeds to select only chromosomes at that local optimum it will not be able to "break free" from that optimum later on and never considers the global optimum of the fitness function as it cannot be reached using only mutations of local optimum-solutions (see Figure 13). This is why the genetic algorithm contains mechanics that (re)introduce random solutions that are not bred from the previous generation.

3.5.1. Seeding the Population

In order to combine chromosomes to produce better performing offspring we first need to seed the GA with a set of valid solutions to the problem. While they do not have to be optimized in any way, providing initial solutions that already have a high fitness greatly reduces the total runtime of the optimization process. We will quickly lay out how to compute good initial solutions later on (multi-step optimizer).

3.5.2. Selecting Chromosomes for Breeding

In each generation only a subset of the last generation is selected for breeding. If $B_{\text{prefer\_better}}$ is $true$, a parent's probability to become selected for breeding is not uniformly distributed but depends on its fitness. Given fitness f_i for chromosome with index i its selection probability is given by

$$p_i = \frac{f_i}{\sum_{j=1}^{N_{\text{pop}}} f_j} \tag{2}$$

If $B_{\text{prefer\_better}}$ is $false$ all chromosomes have the same selection probability.

The composition of the selection pool can be further influenced with the parameter $N_{\text{P\_elite}}$. It ensures that the top performing chromosomes are used for breeding at least once, however, they are not necessarily bred with each other. For the first $N_{\text{P\_elite}}$ breeding operations the algorithm selects one of the top chromosomes as one parent. The other parent is selected as usual according to $B_{\text{prefer\_better}}$.

Note that the specific amount of offspring created is not given directly via parameters but derived using the following formula:

$$N_{\text{offspring}} = N_{\text{pop}} \cdot (1 - P_{\text{rand}}) - N_{\text{elite}}$$

3.5.3. Coupling

After chromosomes are chosen for the breeding operations in the selection stage, pairs of chromosomes are formed to generate one new offspring. A chromosome can be a part of multiple breeding operations. The coupling implemented by Netz is a simple random coupling meaning the pair of the two parent chromosomes is not chosen to perform well under a certain metric but is chosen at random. Note that using a non-zero value for $N_{\text{P\_elite}}$ does not contradict this as the breeding partner is still chosen randomly.

3.5.4. Crossover

The crossover process describes how an offspring is constructed from his parents. All methods discussed assume that exactly two parents are used in the breeding process. However, producing new chromosomes from more than two parents is theoretically possible. We will quickly describe common crossover operators (also see Figure 14) and why their application is neither advisable nor usable with chromosomes encoding a set of positions.

Traditional crossover methods *One-Point Crossover* randomly selects an offset within the length of the chromosome. Both parent chromosomes are cut in two halves at this

position. Now, two different children are possible, one will contain the genes up to the offset from parent A and genes from parent B starting from the offset, the other one vice versa.
The extension of one-point crossover is **n-point crossover** which is based on exactly the same principle but relies on more than one crossover point. Again two different children are conceivable.
Finally, in the most extreme case each gene is randomly assigned from one of the parents. We call this **uniform crossover**.

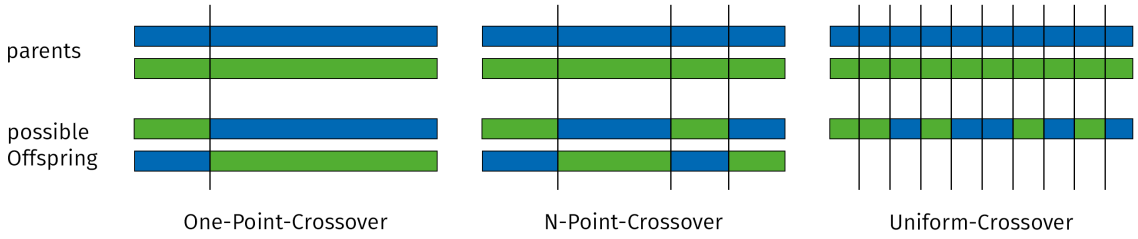
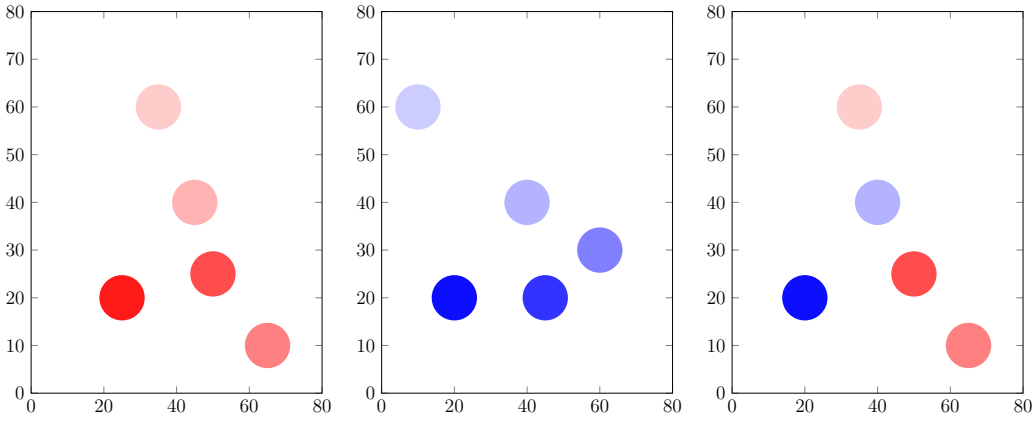


Figure 14: Illustration of classical crossover operations. Graphic from Richter [47].

Apart from the fact that these crossover operations disregard constraints like keeping a minimum distance between instances that we impose on solutions to be valid, they make assumptions about the nature of the chromosome. They assume that the position of a gene conveys meaning about what property that specific gene is encoding. They expect a gene that represents a specific feature to always be at the same position. However, in our encoding the set of genes represents just the set of turbine/heliostat positions. Randomly reordering our chromosome would not have an impact on the semantics of the chromosome. Therefore, using one of the three aforementioned crossover operations is not possible. Hence, we employ a new approach.

Position-specific crossover methods In the following we will present the crossover methods developed by Richter [47] and quickly discuss their respective effectiveness and efficiency.

Zero-Step Crossover In the first step, the genes (instance positions) are sorted into one list according to their fitness in their respective parent. Then they are removed from the list best to worst heliostat and inserted into the offspring. At each insertion we check if the offspring still fulfills the minimum distance constraints. If there is a conflict, the heliostat is skipped. We stop after successfully inserting N_{pop} positions from the sorted list. If the list is depleted and the offspring still has less than N_{pop} we insert valid random positions until we reach N_{pop}.



(a) Parent A (b) Parent B (c) Offspring

Figure 15: Illustration of a *Zero-step crossover*, higher saturation indicates higher individual fitness. Graphic adapted from Richter [47]

One-Step crossover This crossover method again makes use of individual instance performance within its original configuration. As in the *Zero-Step Crossover* the instances of both parent chromosomes are first merged into a combined layout according to their performance in their original layout. Instances that would introduce a conflict in the layout (turbines or heliostats violating the minimum distance condition) are dropped and no longer considered. The difference to *Zero-Step Crossover* is that we do not stop adding positions into the layout after reaching the threshold of N_{pop} (the case with less than N_{pop} instances is handled in the same way though). Then, the configuration is re-evaluated to obtain information about the instances performance in the new layout. The worst-performing instances are dropped until the desired amount is reached. This additional re-simulation of the layout tries to address the weakness that a good performance of a position in a parent layout does not necessarily correlate with a good performance in the offspring layout.

Multi-Step Crossover One problem in the *One-step crossover* still persists: As not all positions of the parent layout will be present in the offspring layout, the contribution of an individual position to the overall layout-fitness may still diverge between parent and offspring. Therefore, Richter defines another operator. In *Multi-step crossover* each position is simulated on its own first. The resulting fitness is then again used to sort all parent positions into a list. The best individual position is added to the offspring layout. In every iteration each non-conflicting position is simulated as if it was added to the current offspring layout. The one yielding the best result is actually added to the offspring.

Comparison of Crossover Operations The crossover operations yield the intuitive ordering of fitness and speed. *Zero-step crossover* is the fastest operation followed by

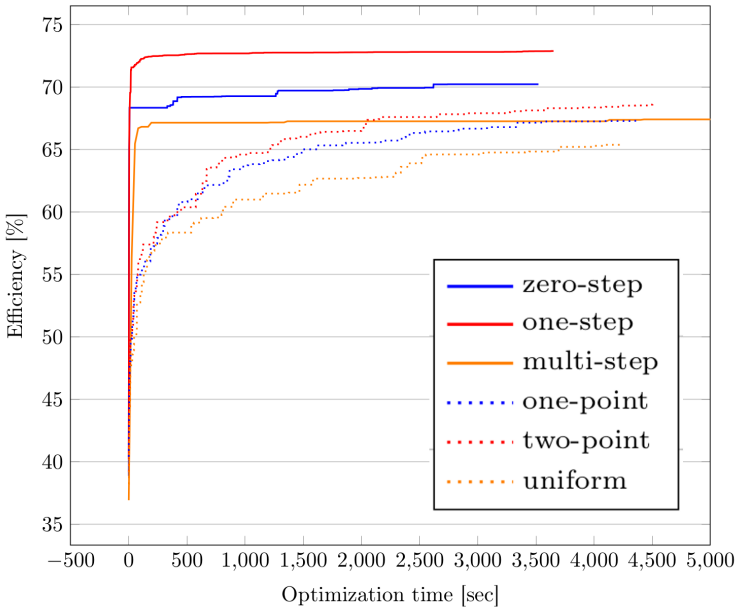


Figure 16: Visualization of the crossover methods plotted against the overall optimization time. Image from Richter [47]

One-step crossover and *Multi-step crossover*. Conversely, the quality of the generated offspring is highest using *Multi-step crossover*. In practice, it is not worth to expend the additional computational cost to achieve the highest offspring fitness with *Multi-step crossover*. *One-step crossover* has the best fitness/performance trade-off as seen in Figure 16.

3.5.5. Choosing and Mutation

The choosing operation chooses chromosomes for mutation. Our GA makes use of the simplest mechanism to pick individuals that will be mutated. The parameter p_c controls the probability which determines whether a chromosome is mutated or not. Each individual has the same chance to be picked and on average $p_c \times N_{\text{pop}}$ will be mutated. As the choosing and mutation operation are highly interdependent and due to the simplicity of the selection scheme we will call p_c a mutation variable.

Because the recombination of existing (and possibly randomly generated) chromosomes into offspring is not enough to uphold the variability in the gene pool, chromosomes are *mutated* before they are copied to the next generation. The mutation operation serves two purposes. First, it contributes to a higher diversity in the gene pool as it makes random changes to the genes of some chromosomes. Second it improves the GA's search behavior, by making the GA reach more positions in the search space during its runtime.

To influence the mutation multiple parameters are at our disposal. With $n_{\mu_{\min}}$ and $n_{\mu_{\max}}$ the range of mutated instances is defined. A number r is chosen uniformly from

the range then r turbines or heliostats are moved according to the operation described hereafter.

Mutating an Instance An instance or rather its position *pos* on the layout is mutated by randomly generating a random position pos_{rand} and moving *pos* towards pos_{rand}. Here, the third parameter influencing the mutation operation comes into play, μ. It defines how much *pos* is moved towards the random position with $\mu = 0$ corresponding to no movement at all and $\mu = 1$ correspond to replacing *pos* with pos_{rand}.

Setting either of the three parameters to 0 (respectively $[0, 0]$ for the interval) completely disables mutation.

3.5.6. Replacement and Replenishment

In each generation the gene pool is renewed with offspring produced by breeding. We call this *replacement*. To maintain a higher variability in the gene pool, randomly generated chromosomes are introduced as well. This is called *replenishment*.

Elitism is a common modification that is usually implemented. Elitism ensures that the best N_{elite} performing individuals of the current generation are copied to the next generation unconditionally. If the amount of elitism N_{elite} is at least 1 it ensures that the best solution of each generation is at least as good as the best solution of the previous one. Elitism is not limited to the best solution, copying multiple chromosomes into the next generation is also an option.

In our GA all individuals from the previous generation except for the elites are replaced. Moreover, we produce exactly as many offspring as needed to form a new generation. There exist approaches where the next generation is formed from offspring, random chromosomes, elites *and* other members of the previous generation. The reason behind this is not to add an offspring to the new generation if it is worse than each member of the previous generation. In order to reduce the amount of tweakable parameters we choose not to adopt this concept.

4. Multi-step Optimization

The motivation to use a Multi-Step Optimizer originates from the fact that we want to avoid the shortcomings of each individual step and speed up the whole process. Multi-step optimization is a general term for an optimizer that relies on different chained processes in order to improve some characteristics of the optimization, in our case the speed and quality of the final result.

To speed up the whole process we first compute initial solutions using pattern generators that are handed over to the GA as initial solutions. The GA runs until it is no longer able to improve the quality of the solutions over a given number of generations. This is one of the shortcomings of a GA, after its population converges to a solution, further improvements are slow and only caused by the algorithms inherent randomness. To avoid this flaw we stop the GA at this point and hand over its best solution to the local search (for an explanation of its mechanism, see Section 4.1) which is better suited for micro-optimization. With each step in the Multi-Step Optimizer the search space – and hence the computation time – increases while the amount of potential improvement decreases.

Computing initial solutions Initial solutions are the starting point the rest of the optimization chain relies upon. They need to provide an adequate level of solution quality. What ”adequate” means depends on the underlying problem. In some cases randomly generated solutions may suffice as input for the next step.

The next step in our case is the GA. As previously mentioned, providing the GA with initial solutions that are better than arbitrary valid solutions can greatly reduce the runtime of the algorithm because the GA does not need to achieve the fitness level of the first step in the optimization chain on its own. It is not necessary to rely on (slow) genetic optimization to discover the effectiveness of a primitive heuristic. A primitive heuristic may be to structure heliostats or turbines in a mathematical pattern. For example a grid pattern for wind turbines. The fitness of the structured pattern will generally outperform random solutions and accelerate the convergence toward a high-fitness solution as there are more high-fitness solutions available to be bred. Nevertheless, seeding the GA with a high percentage of similar heuristics will have a detrimental effect on the optimization as it lowers the variability of the gene pool.

We make use of three different methods to generate initial solutions, namely *patterns*, *close packing* (WindFlower only) and *randomized* solutions.

The close packing algorithm – which is only available for turbine layouts – takes the available construction area as input and in the first step tries to fit as many instances as possible in it while already taking into consideration pairwise influence between instances (see Figure 17 for a depiction of the efficiency of two neighboring turbines). Because turbines have a minimal distance between neighboring instances, the algorithm boils down to *sphere-packing* in 2D. In the next step the densely packed layout is simulated, the instances are ordered by fitness and the lowest performing positions

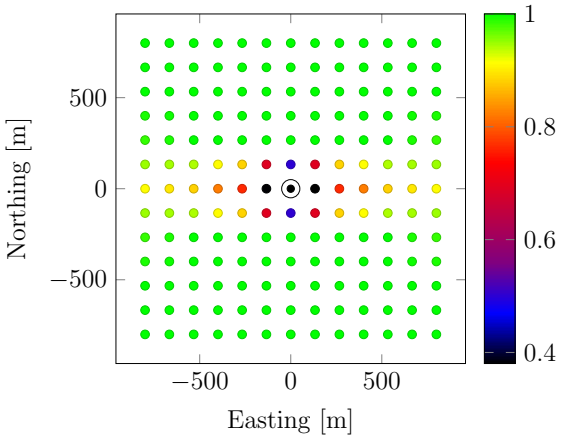


Figure 17: Efficiency matrix used for close packing of wind turbines. Wind direction is left to right. Graphic from Maghnie [36].

are dropped from the layout. This achieves a good performance using only one simulation step. A more detailed description of that method is given by Maghnie [36].

The other non-trivial way to generate solutions are parameterized patterns. The patterns could be generated with random parameters or parameters known to perform well, but our approach goes one step further. The parameters of each pattern are optimized using the *Globalized Bounded Nelder-Mead method* to explore the parameter space [27]. This approach requires multiple simulation steps per pattern but yields better results and does not require any initialization.

4.1. Local Search

The local search aims to fine-tune layouts that already exhibit a high fitness by moving individual instances. The switch from GA to local search should happen as soon as the fitness curve of the GA reaches a plateau that exhibits only minor improvements over extended periods of time.
We exploit the circumstance that only a fraction of the layout has to be re-simulated in the model if only one instance changed position since the last simulation run.
In the case of a wind farm this is due to the limited range over which turbines can physically influence each other. The effect of the wake decays to an irrelevant amount if two turbines are more than r meters apart.
Similarly, a heliostat cannot influence another heliostat over unbounded distances. The two ways a heliostat can impact the performance of a second instance are *shading* and *blocking*. *Shading* means that the first heliostat intercepts rays on the way from the sun to the second heliostat. *Blocking* means that the first heliostat intercepts rays on the way from the second heliostat to the receiver.

Both decrease the performance of the second heliostat. It is important to note that

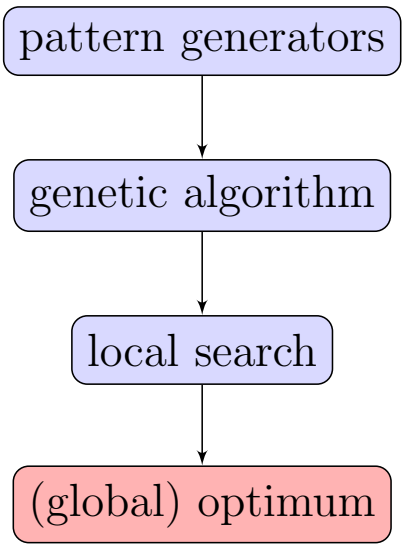


Figure 18: Optimization methods in the multi-step optimizer.

this influence is in practice only possible over a limited distance. In theory the length of a shadow cast by an object on the surface of the earth can be arbitrarily long as the sun approaches the horizon. We do, however, not care about such edge cases, as the energy density on the heliostats quickly falls of as the sun reaches the horizon. Therefore, we can set a distance after which we do not consider the effects of *shading* and *blocking* anymore (at least in the optimization routine of the local search, the end result still takes this into consideration).
The local search uses the locality of the impact to speed up the evaluation. It iterates over all instances in the layout. Each one is moved into multiple directions. When the instance is moved from position p to p', two sets that represent the immediate neighborhood of p and p' are computed. S corresponds to the instances in the affected radius around p, S' analog to the ones around p'. After the instance is moved to p' only $S \cup S'$ has to be re-evaluated in the model. If the fitness is higher than before the relocation of the instance it is kept, otherwise a new target position for the same instance is tested or the algorithm continues with the next instance.
The radius is chosen using empirical data rather than a theoretic foundation. Multiple values for r have been tested and been compared to the original implementation which re-simulated the whole layout in each step.

5. Extension of the Codebase

In addition to the investigation of GA parameters, general work on the codebase was necessary to enable the usage of new parameters and tweaks to the algorithm while at the same time improving and refactoring the codebase of both Wind- and SunFlower.

5.1. Improving Consistency

One main goal of the refactoring was to make the whole project more consistent. This includes but is not limited to command-line parameters, configuration access, error handling and file access.

Previously SunFlower and WindFlower used a custom written config parser to load information from the `.json` files that contain the parameters to configure optimizer and simulation. This approach comes with several advantages and disadvantages. Parsing in C++ makes it easy to do advanced post-processing on the configuration and is straightforward for small projects which is probably why it was chosen in the beginning. The config files were parsed into specialized `struct`'s that held information for e.g. sub-models of the simulation which were passed around using constructors and setter methods. The main problem in the config parser was that it was hard to debug why a specific value in a config `struct` did not have the value given in a config file or why it was set at all. This originated from the fact that the parser first loaded one large `.json` file that contained all possible config values. Missing bits in the actual config files were filled using this information. Moreover, the parsing methods allowed for default values to be passed, which were often but not always hardcoded in the parsing logic. Lastly the names of the variables in the config `struct`'s did not always match the JSON key or sometimes matched a different JSON key than they were representing.

To address these problems, the syntax definition for the config files was completely outsourced using *JSON Schema* [11]. The schemata define how the `.json` files are supposed to look like and are even able to do type-checking and encode dependencies, basic if-else structures, value ranges and enums for string values. This accounts for roughly 90% of the C++ parser code. Moreover, the *JSON Schema* allows for documentation of the semantics of the JSON keys.

The parser code is replaced by checking all config files with the appropriate `.schema.json` file using the library `json-schema-validator` [45] in combination with a JSON library by Lohmann [34]. This allows for config files to be added to the project with almost zero C++ code overhead and provides useful errors if the config file is malformed.
The validated JSON variable is stored in a global singleton that can be accessed in the following way:

```
#include "GlobalDataSimulation.h"

void initData() {
  GlobalDataSimulation::init(
    "path/to/config/folder",
    "path/to/schemata/folder"
  );
}

double sumBoundaryLongitude() {
  const json& site_settings =
    GlobalDataSimulation::getInstance().site_settings;
  double sum = 0;
  for (const json& coord: site_settings["boundary"]) {
    sum += coordinate_pair["longitude"].get<double>();
  }
  return sum;
}
```

Listing 1: `initData()` needs to be called once, then everybody can read the config by including the header and accessing the global singleton.

There is no need to pass around config parameters as they can be easily accessed from anywhere in the code. The overhead arising from needing to access values via the JSON key is negligible in all but the most time-critical code sections where we fall back to reading the values into actual variables.
Of course this approach also comes with some disadvantages. The code example Listing 1 also illustrates that the access to individual config values is very convoluted. Also, while *JSON Schema* allows for elaborate syntax checks, it is not able to do any post-processing on the config data.

In some cases this is still necessary and the code merely moved to a different place. The migration to the new config parser and config access allowed us to get rid of more than 1500 LOC across both projects.

Another goal of the refactoring efforts was to fail fast on errors. In the past missing or misformed parameters or semantic contradictions were ignored during execution (and sometimes silently dropped) or corrected during runtime by inserting default values for conflicting variables. This type of behavior is inconsistent at best and leads to results that do not correspond to the given configuration at worst. Instead, errors should be detected as early as possible and cause the optimization to terminate. This relieves the user from the responsibility to read the logfiles after the optimization process to make sure that no inconsistencies occurred during the process which may or may not have been logged.

We also strived to update and replace libraries (e.g. drop *JsonCpp* [4] for *JSON* by Lohmann [34]) or drop them if they were no longer needed. A significant downside of *JsonCpp* was the fact that the library does not fail if one tries to retrieve a non-existing key of a JSON object but just returns the default value of that variable type (e.g. 0 for `int`). This is not desirable as typos in the JSON key name go mostly unnoticed and only become apparent if the end result of a simulation is off the expected value by a considerable margin. The new library by Lohmann throws an exception if we try to retrieve a non-existing key or try to extract a different data type than the one that is present, **failing fast** in both cases which is the desired behavior. Similar behavior not necessarily related to used libraries could be found at multiple locations throughout the codebase. Values outside of a certain interval of legitimate values were just clamped to the interval unnoticed or replaced by a default value. Again we want to adopt the principle of failing fast and alerting the user or developer about the issue instead of silently continuing operation.

Moreover, we tried to make better use of existing library dependencies. An example would be the transition to `Boost.Program_options` from the widely used `Boost` library [16] for consistent and concise generation of command line options replacing manual parsers that were duplicated in every executable.

During the work on the codebase it became evident that the separation of both Sun- and WindFlower in a *Simulation* and *Optimization* repository was not beneficial to the project and just existed due to the history of the projects. The *Optimization* and *Simulation* repositories of each project were subsequently merged removing the need for some glue code between them and as a side effect reducing the complexity of the build system.

Another task tackled was the standardization of the output files produced by different tools in one repository but also between Sun- and WindFlower. To this point, many of the binaries built write non-standard `.csv` files to hardcoded paths the content of which in most cases is not needed anymore. The goal was to provide one comprehensive `.json` file with detailed information about the economic values and optimization process and one concise `.csv` file containing only the most relevant information (including the positions of the best optimization result).

5.2. Choosing a new GA Library

During the thesis the need for a new library handling the evolutionary algorithm arose. The previously used library caused problems during compilation on various machines. Moreover, its implementation forced a considerable amount of boilerplate-code upon the project and was hard to maintain. Therefore, the search for an alternative began. The library had to fulfill several knock out criteria. Every call to the evaluation of the simulation was required to be parallelizable as this is by far the most time consuming part of the optimization. The amount of time spent in the logic of the GA is negligible. The other exclusion criterion refers to the encoding of chromosomes. We need to be

able to specify a custom genome encoding that is not limited to bitvectors (which many libraries are).

Having fulfilled these requirements the main selection criterion is ease of use and the extensibility / maintainability of the library with trivial compilation procedures being a plus. The libraries evaluated were the existing *GeneiAL* [23], *GAlib* [55], *Open BEAGLE* [5], *openGA* [38] and *Evolving Objects* (EO) [1].
`openGA` – the contestant that was adopted in the end – especially stood out due to its small codebase and its implementation in a single file. Also its simple structure facilitates even fundamental changes to its algorithm. Beneficial is also the fact that it is implemented using a recent C++ standard (C++17).

	GAlib	BEAGLE	openGA	GeneiAL	EO
custom chromosome	✓	✗	✓	✓	✓
parallelization	✓	✓	✓	✓	✓
multi-objective	✗	✓	✓	✓	✓
boilerplate-code	●●●	●●○	●●●	●○○	●○○

Table 2: Comparison of C++ GA libraries. The estimation of boilerplate complexity was derived from studying the examples of the respective libraries.

6. Test Cases

Test cases should differ in amount of instances (turbines / heliostats), weather conditions and layout. However, they should not contain excessive amounts of instances (especially for Sunflower).
In order to ensure the transferability of our results we choose test cases that cover a large spectrum of the existing and potentially coming real-world implementations of either power plant. Environmental and building-site-dependent parameters include but are not limited to: variation of weather conditions (number of sun-hours, standard deviation of wind directions, ...), existence and size of forbidden areas (non-suitable for construction) and overall size and shape of the construction site. Naturally, testing a large variety of these parameters is infeasible but we aim to select test cases that exhibit a variety of characteristics for the optimization results to generalize.

6.1. PS10 Central Receiver System

PS10 is a solar power plant situated in southern Spain near Seville. It consists of 624 heliostats and is the smaller of the two central receiver system test cases. The plant was constructed from 2004 until early 2007 underlining the high-risk aspect of these projects which makes optimization and predictability crucial. The plant produces around 23 GW h per year.

6.2. Gemasolar Central Receiver System

The Gemasolar power plant is also located in the province of Seville in Spain. However, it consists of 2650 heliostats and produces 80 GW h per year. This test case was chosen to analyze the differences of two power plants that differ only in the amount of installed heliostats while the environment boundary conditions are mostly the same.

6.3. Sandbank Wind Farm

Sandbank is an offshore wind farm operated by *Vattenfall* off the coast of Germany in the North Sea. It consists of 72 wind turbines spread over an area of 60 km^2 [8]. It is the smallest test case in our selection and the only one containing restricted areas.

7. Parameter Study

Goal of this thesis This thesis aims to be an evaluation, extension and concretization of Netz' results [40]. He already measured the influence of different GA parameters on the convergence speed and quality of the obtained results. We now want to give a more specific description of the variables' influence and the interplay of different parameters. Ideally, we are able to identify the relevant parameters early on and can focus on an in depth examination of a few important ones instead of analyzing many low-impact options of the algorithm. Moreover, we try to discover new means to positively influence the behavior of the GA that either reduce computation time or improve the quality of the result (fitness of the model).

The result of the thesis consists of several new pieces of information. The most basic one is a description of the relevant parameters for the GA and how each one influences the behavior of the algorithm. The key result consists of a suggestion how to configure the genetic algorithm for maximal efficiency. It is obtained by means of a multitude of experiments that run the optimization process with different configurations to analyze the interplay of the GA parameters. One experiment consists of four parts. The configuration parameters to be tested. They are depicted in one or several tables that translate into the set of GA configurations that are tested. Second we lay out what results we expect from these runs and why. Finally, the results of the tests and their analysis are given. The results are in the form of a plot usually plotting the fitness of the best performing chromosome against the total number of simulated layouts.

Determinism vs. randomness Most GA parameters have the inherent property that they influence the amount of determinism in the algorithm. High elitism values are associated with determinism because they cause a large quantity of the current population to be also present in the next generation. Contrary, a high probability to mutate a chromosome causes the algorithm to introduce much more randomness in its procedure.
Finding good configuration parameters for the GA can be viewed as finding a good balance between those two antipodal components. Choosing the amount of determinism too high can result in an insufficient exploration of the search space while a high amount of randomness cripples the evolutionary properties of the GA.

Optimal behavior of the GA We want to optimize the parameters of the GA. When optimizing something we need to know what we are aiming for; in our case how the convergence behavior of the GA should look like.
In this thesis we refrain from the usage of both the pattern generators (to seed the GA with sufficiently good solutions) and the local optimization (that would normally be applied after the GA for a more efficient local search) as they achieve different levels of fitness in SunFlower and WindFlower and thereby make the results (of the complete optimization chain) less comparable across the two problems.

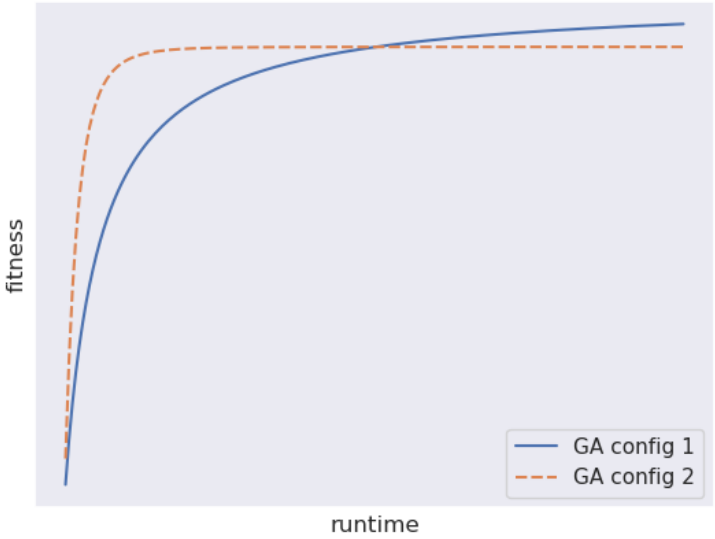


Figure 19: Possible convergence behaviors.

When specifying what we want to optimize for, it is important to keep in mind that the GA has its place in the multi-step optimization and is usually not operated on its own. This influences how the *optimal convergence behavior* looks like. We do explicitly not aim for GA parameters that achieve the highest fitness but take a long time to do so. The last bit of additional fitness of the layout can be achieved much more efficiently by the local search algorithm that takes as input the best solution of the GA. This is why a convergence behavior that converges to a "good" fitness in a short amount of time (GA configuration 2) is preferred over a configuration that takes a long time to achieve a "very good" fitness that offers only a small benefit over the "good" result (GA configuration 1, qualitative examples for this are illustrated in Figure 19). The local search is algorithmically much more fitted to perform the final micro-optimizations and the combination of the "good" GA followed by the local optimization will achieve the same fitness as the "very good" GA usually faster than the "very good" GA on its own.

Comparing Different Configurations When facing the task of comparing results obtained with sometimes vastly different GA configurations or configurations run on different test cases, the problem of comparing the results in a meaningful way presents itself. It is fairly easy to compare two results qualitatively as it is evident if a certain configuration performs good or bad on two different test cases, but we also want allow for a quantitative comparison of two different configurations executed on the same test case.
The x-axis should ideally depict a measure that scales linearly with the runtime of the

algorithm while still being comparable on different hardware setups. This is not easy because hardware can differ in multiple essential parameters influencing the runtime of the GA including but not limited to: CPU power (MIPS), CPU instruction set, CPU cache speed and size, memory bandwidth and size, etc. Another problem introduced by the test setup is that multiple tests are run in parallel but without any certainty of how many are operating simultaneously at any point in time eliminating the runtime in seconds as valid measure. A more robust way to compare to configurations is to count how many calls to the simulation are necessary to achieve the same fitness (one call being equivalent to the computation of the energy output for one specific layout). This has not only the advantage that it compensates for the impact of different population sizes but also includes evaluation calls that occur in all stages of the GA. Our crossover operation for example makes use of evaluation of a layout to produce the offspring. Furthermore, comparing two different performance plots becomes more intuitive. The comparison of fitness values after a specific amount of evaluations is meaningful even if the GA configurations differ.
The underlying assumption legitimating this is that the amount of time spent in the logic of the genetic algorithm is vanishingly small in comparison to the time spent in the simulation model. Counting the number of function calls enables a sufficient level of precision and comparability over different GA configurations and different hardware with varying load as it is completely independent from the execution speed.
This method still has some flaws though. Due to the inherent randomness of the algorithm we cannot guarantee that each evaluation call has to deal with the same amount of instances placed in the layout; again the notable offender is the crossover method. Also, other time-consuming operations take place during optimization that are not covered by counting the evaluation calls, most importantly the generation of random layouts or the addition of random positions to existing layouts. This makes it especially hard to compare two configurations that use vastly different values of P_{rand} using this method.
By implementing counting the evaluation calls we also address a flaw in the analysis of Netz where the fitness was plotted against the number of the generations in the GA. This however is not viable when comparing different runs executed with different configurations.

Usage of multi-step optimization in the experiments Despite disposing of a functional multi-step optimizer we abstain from using it in our experiments. The first step in the multi step optimization is the generation of patterns that are used to seed the GA. The achieved fitness using patterns however varies greatly in the wind farm and central receiver system optimization. To preserve a higher level of comparability we do not rely on patterns as initial solutions to the GA and instead only use random solutions as initialization.

parameter	value(s)
$N_{\texttt{pop}}$	100
$N_{\texttt{P\_elite}}$	0
$B_{\texttt{prefer\_better}}$	true
$N_{\texttt{elite}}$	1
$P_{\texttt{rand}}$	0.1
μ	0.2
$n_{\mu_{min}}$	3
$n_{\mu_{max}}$	9
p_c	0.1
$N_{\texttt{max\_it}}$	1000

Table 3: Reference GA parameters

7.1. Explanation of Visualizations

For the visualization of the optimization process we include the relevant optimizer settings in the graphs to allow for easier assignment of the figures to the corresponding experiment. As already mentioned we run each optimization setting multiple times, five times in our case. All runs are plotted in the same figure, resulting in what looks like thick lines in the graph which really are different runs that yield very similar results.
It is worth noting that a high absolute fitness value does not mean the solution is actually good, it may just be a property that emerges from the problem parameters. Large numbers of instances seem to be penalized by our fitness calculation across both wind farms and central receiver systems. We can however use the first generation (which only consists of random solutions) as a baseline for our evaluation and examine the relative improvement of the fitness value as well as the characteristics of the line for our analysis.
It is important to note that the graphs are cut off as soon as the performance of the best solution stalled in order to only show the interesting part of the performance curve. As we always plot five runs this means the fitness curve seems to thin out towards the end due to some of them being cut off sooner.

The parameters roughly rely on the numbers used by Netz. Another parameter left unchanged in the experiments is the resolution of the discretization ρ which Netz has shown to have no impact as long as it is large enough. Undersized values lead to a lower final fitness which is explained by the coarse discretization that prohibits micro-optimization.
After inspecting the first results (Appendix A.2.1), we noticed that there was never an improvement in fitness after a longer stagnation. Hence, we decided to implement an additional stopping condition to limit the amount of generations without improvement and therefore runtime of the optimization. In addition to the upper bound on the number of generations given by $N_{\text{max\_it}}$ we employ an early stopping mechanism relying on

exponential moving average. In practice, it has proven to be a much stronger limiting factor than plainly limiting the amount of iterations.

7.2. Experiments

Note: If any parameter is not specified in any of the following tables it is set to the reference value (Table 3).
Not all experiments were performed on all test cases. The test cases for central receiver systems in general and Gemasolar due to its large number of heliostats in particular took a very long time to compute. The optimization process for a single configuration often exceeded one month of CPU time on our machine.

7.2.1. Trade-off Between Population Size and Iterations

Netz claims that a higher population results in a faster convergence. This makes sense if we do not compensate for the increased number of evaluations that result from the larger population. We investigate this phenomenon by increasing the population size while reducing the maximum amount of iterations. The parameters N_{max_it} and N_{pop}will be configured mirror-inverted in the range `[10,100,1000,10000]` (Table 4). The results will then be plotted against the number of evaluations instead of the generation of the evolutionary algorithm. A potential pitfall in this experiment is that we do not know how to sensibly adjust the rest of the parameters that also influence the algorithm. In the experiment we will scale N_{elite} linearly with the population size and keep the rest of the parameters the same.

Expected Results We expect to find a sweet spot somewhere in the middle because both large population, few iterations and small population, many iterations have weaknesses.
A small population cannot hold enough diversity to avoid fast convergence to a local optimum and few iterations prohibit the beneficial effects of repeated recombination of valid solutions. Therefore, we expect either $N_{pop} = 100$ or $N_{pop} = 1000$ to perform best.

Results The graphs of the results can be found in Appendix A.2.1. Note that these were not performed with the actual amount of heliostats in PS10 to reduce the runtime; this also why we cannot compare them to the results of the pattern optimizer in Appendix A.1.
The results for the PS10 test case suggest that our assumption that the ideal balance between population size and iterations lies somewhere between 100 individuals at 1000 generations and 1000 individuals and 100 generations is correct.
On the other hand, the results of the Sandbank test case do not support these claims. For Sandbank the largest population at very few iterations performed the best.

parameter	value(s)	parameter	value(s)
$n_{\mu_{\text{min}}}$	3	$n_{\mu_{\text{min}}}$	3
$n_{\mu_{\text{max}}}$	15	$n_{\mu_{\text{max}}}$	15
μ	0.75	μ	0.75
$B_{\text{prefer\_better}}$	False	$B_{\text{prefer\_better}}$	False
N_{pop}	10	N_{pop}	1000
N_{elite}	1	N_{elite}	100
P_{rand}	0.1	P_{rand}	0.1
N_{max_it}	10000	N_{max_it}	100
$n_{\mu_{\text{min}}}$	3	$n_{\mu_{\text{min}}}$	3
$n_{\mu_{\text{max}}}$	15	$n_{\mu_{\text{max}}}$	15
μ	0.75	μ	0.75
$B_{\text{prefer\_better}}$	False	$B_{\text{prefer\_better}}$	False
N_{pop}	100	N_{pop}	10000
N_{elite}	10	N_{elite}	1000
P_{rand}	0.1	P_{rand}	0.1
N_{max_it}	1000	N_{max_it}	10

Table 4: Parameters used to test the trade-off between population size and maximum iterations

7.2.2. Favoring Better Individuals for Crossover

The parameter $B_{\text{prefer\_better}}$ controls whether individuals with a higher fitness have a higher chance to be selected for crossover. Intuitively, this is meant to avoid breeding individuals that have a low fitness value while not omitting them completely. However, it is possible that enabling this parameter leads to a faster convergence to an unwanted local optimum. Moreover, the breeding low with high fitness or low with low fitness may still result in a well-performing individual.

Expected Results We expect that $B_{\text{prefer\_better}}$ enabled leads to faster convergence. We expect the influence of the parameter to be small to moderate as it does not cause extensive changes to the algorithm.

parameter	value(s)
$B_{\text{prefer\_better}}$	true
$B_{\text{prefer\_better}}$	false

Table 5: Parameters to test the usefulness of $B_{\text{prefer\_better}}$

Results The graphs of the results can be found in Appendix A.2.3. Our assumption that giving well-performing individuals a higher selection chance will increase the convergence rate is not supported by the results of both PS10 and Sandbank. The differences in the curve are marginal. Nonetheless, enabling the parameter does yield a higher final fitness in both results.

7.2.3. Ensuring Breeding of Best Individuals

This test is similar to the previous in the way that it favors high-performing individuals. Only this time we do not adjust their chance to be bred but ensure that the top $N_{\text{P\_elite}}$ individuals are used for at least one crossover operation. To evaluate the effect of this parameter we set it to multiple values as shown in Table 6.

parameter	value(s)
$N_{\texttt{P\_elite}}$	0
$N_{\texttt{P\_elite}}$	1
$N_{\texttt{P\_elite}}$	5
$N_{\texttt{P\_elite}}$	10

Table 6: Evaluating whether enforcing that the top x individuals take part in breeding operations is beneficial.

Expected Results As in the previous test we expect this setting to increase the convergence rate. However, we assume this parameter to have a more serious impact on the algorithm. The convergence may be affected negatively way, causing the optimization to stop early on a local maximum.

Results The graphs of the results can be found in Appendix A.2.4. Neither Gemasolar nor PS10 nor Sandbank show significant differences for different values of the parameter.

7.2.4. Required Amount of Elitism

Elitism is an apparent beneficial modification to the genetic algorithm. But how much elitism is required exactly with regard to the population size? We are going to test multiple amounts of elitism ranging from no elitism at all to an unreasonably high value. Note that the configurations contain values that result in no elitism to support our claim that an arbitrary amount of elitism is absolutely required for the algorithm to work. The concrete values are given in Table 7.

Expected Results We expect to see a significant difference between no elitism at all and any amount elitism. We believe that even for a large population a rather small percentage of elitism is sufficient. We expect the differences between any of the configurations where elitism is active to be marginal.

parameter	value(s)
$N_{\texttt{elite}}$	0
$N_{\texttt{elite}}$	5
$N_{\texttt{elite}}$	10

Table 7: Different amount of elite chromosomes

Results The graphs in Appendix A.2.8 do not exhibit any meaningful differences which is particularly surprising. Completely disabling elitism should yield considerably worse results.

7.2.5. High Amounts of Randomness

In these experiments we want to vastly increase the amount of randomness in the optimization process (see Table 8). In order to achieve this, only a small amount of elites and few individuals from the offspring pool are copied into the next generation. The vast majority is going to consist of random individuals. What we hope to achieve by this is that the few well-performing chromosomes (elites and offspring from the previous generation) are going to be bred mainly with random chromosomes. As the selection pool is larger than $N_{\text{elite}} + N_{offspring}$ and thanks to the random coupling operation, couples of one well-performing and one random individual will be the main source of offspring.

Expected Results We hope that the combination of established solutions and completely random solutions is more fruitful than breeding mostly well-performing chromosomes with each other as they may have already converged to be very similar. A drawback of this strategy is the vast amount of random solutions that is inserted each generation. They will induce many costly evaluations using the simulation. We expect this to perform worse than the reference configuration.

parameter	value(s)
$P_{\texttt{rand}}$	0.6

parameter	value(s)
$P_{\texttt{rand}}$	0.8

parameter	value(s)
$P_{\texttt{rand}}$	0.9

Table 8: Parameters that cause high amount of randomness in the optimizer with some variations.

Results The Sandbank graphs in Appendix A.2.5 support our suspicion that large percentages of random chromosomes do not improve the convergence behavior. For Gemasolar the impact of the parameter is negative as well but less pronounced than the wind farm result.

7.2.6. Randomization of Amount of Mutated Instances

When a chromosome is chosen for mutation a random number r is drawn uniformly from the interval $[n_{\mu_{\min}}, n_{\mu_{\max}}]$. Then r instances are mutated in this chromosome. In this experiment we want to investigate if the randomization has a substantial impact on the convergence behavior of the GA. To test it we run the GA with 4 different intervals $[n_{\mu_{\min}}, n_{\mu_{\max}}]$. All share the same expected value and will — in the long run — mutate roughly the same amount of instances.

Expected Results We do not expect a significant influence on the behavior of the GA from this modification.

parameter	value(s)
$n_{\mu_{\min}}$	6
$n_{\mu_{\max}}$	6
$n_{\mu_{\min}}$	5
$n_{\mu_{\max}}$	7

parameter	value(s)
$n_{\mu_{\min}}$	2
$n_{\mu_{\max}}$	10
$n_{\mu_{\min}}$	0
$n_{\mu_{\max}}$	12

Table 9: Parameters that go continually from an interval containing one value to the largest interval possible

Results The graphs of the results can be found in Appendix A.2.6. The simulations do not show any differences neither in final fitness nor in the fitness curve. The choice of $n_{\mu_{\min}}$ and $n_{\mu_{\max}}$ seems to be irrelevant as long as $\frac{n_{\mu_{\min}}+n_{\mu_{\max}}}{2}$ stays the same. I suggest dropping this parameter entirely to reduce the dimension of the configuration space by one.

7.2.7. Interplay of Mutation Variables

The mutation of random chromosomes is an essential mechanism for better search space exploration and to avoid early convergence towards a local optimum. Tweaking these parameters is crucial for the performance of the GA. The three parameters responsible for controlling mutation are expected to strongly influence each other. Therefore, we need to test a lot of configurations to get an understanding about their pairwise relationships. Table 10 contains 27 different configurations (three options for each parameter).

Expected Results We expect the combination where two or more of the parameters are set to the largest possible value to perform badly because we chose to set the highest of the three options to a fairly extreme value. In configurations where only one value is set to its "high" option the extremes could balance each other out in an interesting way.

parameter	value(s)
$n_{\mu_{\texttt{min}}}$	3
$n_{\mu_{\texttt{max}}}$	3
μ	[0.01, 0.05, 0.4]
p_c	[0.01, 0.1, 0.4]
$n_{\mu_{\texttt{min}}}$	10
$n_{\mu_{\texttt{max}}}$	10
μ	[0.01, 0.05, 0.4]
p_c	[0.01, 0.1, 0.4]
$n_{\mu_{\texttt{min}}}$	20
$n_{\mu_{\texttt{max}}}$	20
μ	[0.01, 0.05, 0.4]
p_c	[0.01, 0.1, 0.4]

Table 10: Various combinations for the three parameters relevant for mutation (each parameter represented as a list is tested with any other combination of parameters yielding 27 configurations)

.

Results Again, the results do not exhibit significant differences.

8. Discussion and Conclusion

The results of the optimization of almost all test cases are inconclusive as most of them yielded the same fitness up to a very small error which we attribute to random chance despite the fact that each configuration was run five times. We are only able to conclude that one parameter is superfluous (we do not need to provide an interval for the number of mutated instances, on value is enough) and that two of our configurations are detrimental to performance due to them being negative outliers. One is the configuration that does not favor better performing individuals (see Appendix A.2.3) the other one being the configuration with excessive amounts of randomness (see Appendix A.2.5). In both cases the negative impact on the GA is not surprising.

Summarizing, we can say that despite sometimes vastly different configurations neither wind farm nor central receiver system test cases exhibited large enough differences that would allow us to trace differences in fitness or the fitness curve back to individual parameters. Only intuitively "bad" configurations like injecting excessive amounts of random solutions into the system produced substantial differences. As these configurations were expected to perform badly and were only chosen to verify our expectations we were not able to generate new knowledge from the results.

Nevertheless, most of our results for PS10 – the best one having a fitness of 0.491 (Figure 49) – were able to outperform the pattern optimizer (Table 12) with a fitness of 0.458 by a significant margin. This result could not be replicated for the Gemasolar test case. This proves that the usage of a GA can be effective. It is however, not efficient; the amount of time necessary to reach this level of fitness is tremendous, even if we consider that the early stopping mechanism could (and should) be configured more aggressive. It would still be a runtime of hours for the pattern optimizer vs. days for the GA.

Seeing the GA outperform the pattern optimizer might suggest to further pursue the usage of the GA as part of our optimizer. On the other hand it only did so for the smaller test case indicating that its performance will deteriorate further when facing larger test cases (which is backed by the problems presented in the next section). Moreover, there are multiple inherent problems in the mechanism of the GA which cannot be fixed with further tweaks. While we did not test it, we expect using a Multi-Step Optimizer only consisting of a pattern optimizer and the local search to always outperform an optimization chain that also includes the GA.

Our final conclusion is that the utilization of the GA should not be further investigated.

9. Future Work

In this section we will present some ideas that came up during the work on this thesis but were out of scope or simply impossible to test as the simulations take such a long time to compute. Additionally, we present an alternative to the GA and discuss its strong and weak points as well as inherent problems of the genetic algorithm.

9.1. Further Experiments

This section contains experiments we thought of but which due to time constraints did not make the cut. Note that we do not expect any of these to remedy the structural problems of the GA.

9.1.1. Mutating using Local Search

We also considered a mutation mechanism that makes use of a portion of the local search algorithm. While the normal mutation moves an instance over a distance dictated by the value of μ, the local search mutation moves the instance within a small radius of its original position. Furthermore it makes sure that the fitness is not negatively affected by the mutation and also tries to move in multiple directions. If it is not able to improve the fitness of the layout using this approach, the instance is not moved at all. Using local mutations could help tackle the problem of frequent complete resimulations.

9.1.2. Even higher selection chance for good individuals

The experiments show that it is beneficial to favor well-performing individuals for breeding. Currently, an individual with high fitness has a linearly higher chance of selection. It might be interesting to further amplify the effect beyond a linear relationship.

9.1.3. Constant selection chance, enforce breeding best

Somehow contrary to the previous suggestion, giving each individual the same selection chance while still enforcing the best individuals to take part in breeding could yield the benefits of breeding the best individuals while also maintaining more sources of randomness.

9.2. Problems using a GA

In this section we will present some shortcomings of the GA and reasons why this optimization tool might not be well-suited to our problem.

Non-local changes between optimization iterations The main problem that emerges from the usage of a GA for position optimization is that there are no small-scale changes between iterations. While the GA has not converged, the crossover operation will produce a layout that is different from all the layouts that have been evaluated so far. It will differ from both of its parents globally not locally. This means we cannot harness the evaluation results of either of its parents. So necessarily we need a full resimulation of the new layout. This problem becomes increasingly grave with growing instance-count. In contrast, the runtime of the local search algorithm does not increase with layouts containing more instances - its runtime and potential improvement stay constant. This and the fact that the algorithm runtime is much more predictable makes it more suitable for larger problem instances as short iterations with small fitness increases are favorable over long iterations with larger fitness jumps (assuming that one full resimulation withing the GA will also lead to linearly larger improvements in fitness).

Sensitivity to parameters Genetic algorithms in general and our implementation in particular provide a wide range of configuration parameters to influence their behavior.

Not only are the configuration options plentiful, GAs in general (Note that our GA does not have this property, it is mostly indifferent to parameter changes) are also very sensitive to changes in some of these options. We say some because not all parameters influence the behavior of the GA in a meaningful way. In this work we wanted to find out which parameters are really relevant and worth tweaking and which can just be left at a standard value all the time. However, the parameters that are relevant usually have a very high impact on the GAs behavior and need to be tweaked well for it to run at its full potential. Depending on the problem instance the theoretical optimal parameters may or may not change. So we need to make a compromise to obtain a set of options that perform reasonably well in most circumstances or deduce a formula to adjust parameters with regard to the problem instance.
Both of these are viable options, however, using an algorithm that is less sensitive to configuration or does not need to be configured at all is preferable.

Sensitivity to initial solutions Genetic algorithms need a starting point from which they start improving the solution. Due to the inherent convergence of the GA

the characteristics and the quality of the initial solutions have a strong influence on future solutions generated by the GA. A potential risk is that a GA seeded with an optimized mathematical pattern will never generate solutions that diverge from that pattern, despite random solutions being injected into the population every generation, simply because it is unlikely that an optimized pattern is improved by random changes or by breeding it with a random solution.

Non-standard genome structure The genome structure we use is a set of positions (layout) where traditionally the genome encodings used in genetic algorithms are vectors of bits or numbers. Albeit, there is no meaningful way to encode a layout in an

ordered vector since the order in which the positions are sorted has no influence on the semantic of the vector – the layout stays the same. This disallows us to use tried and tested crossover operations from literature and may even hamper the whole mechanism of the GA.

No utilization of domain knowledge and problem parameters Our implementation does not try to make use of any existing domain knowledge from either wind farms or central receiver systems nor do we consider additional information we have about the problem instance. For wind farms, a useful piece of information could be the top three dominant wind directions, for central receiver systems the position of the tower and the distance of the site to the equator. None of this information is used in the optimization. On the one hand this simplifies the integration of the algorithm because it only needs to operate on layouts and not care about the problem parameters, on the other hand we discard valuable information that could be used to develop more sophisticated optimization methods.

9.3. Additional Tweaks

Over the course of this thesis we had more ideas on how to improve the optimizer or related parts of the code through additional parameters or new concepts some of which we will present here. Note that they are not limited to usage with the GA but can be beneficial to any optimization algorithm.

9.3.1. Sector-wise Optimization

To combat the problem of total resimulation between optimization iterations, one might divide the construction area into multiple (rectangular) sectors each of which is optimized on its own in the first phase. After each sector has been optimized (treating all the other sectors as empty), in the second phase, the solutions are merged and discontinuities at the borders are resolved.
While it solves the problem of requiring full resimulations (now only one sector needs to be evaluated) it creates a lot of new complications. For example, it may not be beneficial to have the same amount of instances in each sector, and optimizing all sectors individually without accounting for that makes the first phase complexity pointless and just shifts the problem to the second phase. Therefore, the algorithm has to account for the fact that instances might not be evenly distributed across the construction area. One could solve this by randomly moving an instance to a (neighboring) sector at any given iteration. Over time this will result in an appropriate distribution of instances. Unfortunately, this operation (random exchange of instances) defeats the independence of each sector in the first optimization step. Another approach could be to optimize each sector with more instances than needed and to just drop some off at the end – this however will likely result in non-optimal layouts.

9.3.2. Dynamic Accuracy of Simulations

Improving the runtime of the GA is crucial, therefore dynamically adjusting the accuracy at which the simulations are performed is a good way to increase speed, especially in early generations. Notice that the accuracy of the simulation only needs to be high enough to distinguish good from bad solutions. That means that especially in early generations we can get away with low accuracy simulations, as the population has not started to converge and the difference in fitness between individuals is still high. As the algorithm progresses and differences in fitness decrease, the accuracy needs to be ramped up to be able to pick the better of two solutions.

WindFlower For WindFlower the accuracy could be tweaked by changing the amount of simulated wind directions or time slots. Both of these parameters are discrete and therefore do not allow for a smooth adjustment of the accuracy, moreover, they require an interpolation of the wind data upon change which is costly on its own and increases the complexity of the algorithm. However, the runtime of WindFlower was not a large problem in practice making this tweak superfluous. This may change when we start optimizing wind farms that have an amount of turbines that approach the number of heliostats in a large central receiver system.

SunFlower Adjusting the accuracy in SunFlower is much easier, can be done by changing only one parameter and needs little to no changes to the codebase. The parameter in question is the number of sunrays that the ray tracer shoots at the heliostats to calculate the percentage of mirror surface that ends up reflecting rays to the receiver. Although also being a discrete parameter, its value is usually high enough (a few thousand) to allow for fine-grained control. Similar to the wind farm we could also adjust the number of simulated day and time slots. The negative implications of changing these parameters are similar to the ones mentioned for WindFlower (algorithm complexity, recomputation of data) and make them an inferior choice compared to the number of rays.
This adjustment could prove essential for SunFlower as it linearly impacts the runtime of the simulation (halving the number of rays will roughly halve the runtime of one simulation).

9.3.3. Lazy Generation of Random Solutions

Most of the time spent during optimization is not in the logic of the optimization algorithms but in the simulation of solutions. In order to reduce the amount of simulations that need to be run in the first place we can exploit some properties of random solutions. First, their fitness is always roughly the same; the fitness of any random solution generated over the course of one optimization process will bunch around one μ with a comparably small σ. That means after generating a random solution we already

know its performance to a high extent without ever simulating it. Second, their fitness will always be worse than the best individual in our population (except for the very first generations). Third, a given random solution does not have a high priority of being chosen for breeding. This is partly because favoring well-performing individuals for breeding has turned out to be beneficial and partly because even with a constant selection probability for all individuals a single given individual has a relatively high chance of not being chosen.
Combining this information we can infer that random solutions all evaluate to the same bad fitness and (therefore) a significant percentage of them are not considered for breeding. This means not only do we not need to evaluate a random solution until it is chosen for breeding, we can also completely skip generating one and just add a placeholder to the population at zero cost without negative side-effects. In the next section we will illustrate why skipping not only evaluation but also generation can be a valuable performance gain.

9.3.4. Smart Generation of Random Solutions

Random solutions play a a significant role in a GA and even with lazy generation applied a large amount of them needs to be generated throughout the optimization process. We noticed that in SunFlower a surprising amount of time is spent with their generation. We assume this to be the cause due to the way we generate random solutions and an inherent property of heliostat fields. Whenever we need a random solution of a certain size n, we iteratively add random positions to a layout while assuring that each new position does not conflict with the existing layout (overlapping instances or minimum distance between distance not adhered to). This process incurs $\mathcal{O}(n^2)$ distance checks between instances. This is true for both Sun- and WindFlower and quadratic growth itself is not the problem.
This way of generating random solutions becomes a problem when we need to randomly generate a layout that is densely packed into our building site. As the number of instances we want to place in the layout n approaches the maximum amount of instances that can theoretically be placed using random placement $n_{\max}$ the process slows down *significantly* (Note that $n_{\max}$ is not a precise number due to randomness and also not close to the amount of instances that can be placed using classical sphere packing in 2D).
The reason this process is slow for n approaching $n_{\max}$ is quite intuitive. Suppose we want to generate a layout for $n = 3000$ (reasonable amount of heliostats for central receiver systems) with $n_{\max} = 3500$. As we iteratively add instances to the layout the probability of a random position hitting a free spot in our layout becomes smaller and smaller. Still, each new random position needs to be checked against the existing layout resulting in a runtime exceeding $\mathcal{O}(n^2)$ by a lot.

Possible solution WindFlower already uses circle packing as a quick way to generate good initial solutions. We can overhaul our random generation process by computing

a circle packing of instances in the construction area and using it as a base to generate random solutions. This only needs to be perfomed once. Generating a random solution from the circle packing would be performed by first randomly removing instances from the layout until we hit n and in a second step moving each instance for a random distance in a random direction. The second step will still produce collisions between overlapping instances, but they are much easier to fix as we can reduce the distance moved or change the direction in which we move the instance. This approach also mitigates the endless search for a valid layout because we can always move conflicting positions back to the original positions taken from the circle packing. In the worst case we can just skip the second step completely, generating a valid layout at zero cost that still exhibits randomness to a certain extent.

9.3.5. Remove the GA?

As outlined in the previous paragraphs it does not seem possible to keep the GA applicable for large problem instances. At this point we need to ask ourselves whether we want to keep the GA at all when it is so hard to adapt it into an algorithm that scales better with large problem instances. In the next section we will present one possible alternative and discuss its strength as well as similarities to the GA.

9.4. PSO as Alternative to the GA

As already mentioned before, *Particle Swarm Optimization* is an established means of optimization for wind farm layouts. It shares the disadvantage of the GA that all the changes to layouts are non-local. An advantage of PSO is the dimension of the configuration parameter space which is much smaller than the one of the GA meaning that tweaking the algorithm for different problem classes might be easier. However, this might be a naive assessment of the situation as a lot of the configuration parameters we have to deal with emerge from our custom and problem specific modification to the GA algorithm. Assuming we tweak PSO to a similar extent this perceived advantage will quickly vanish as our custom modifications quickly inflate the amount of tweakable parameters.

References

[1] Evolving objects. URL http://eodev.sourceforge.net.

[2] Fino 3. https://www.fino3.de/en/.

[3] International emissions. URL http://www.c2es.org/content/international-emissions.

[4] Jsoncpp. URL https://github.com/open-source-parsers/jsoncpp.

[5] Open beagle, . URL https://github.com/chgagne/beagle.

[6] Openwind homepage. https://aws-dewi.ul.com/software/openwind, .

[7] Paris agreement. URL https://treaties.un.org/pages/ViewDetails.aspx?src=TREATY&mtdsg_no=XXVII-7-d&chapter=27&clang=_en.

[8] Sandbank. https://powerplants.vattenfall.com/sandbank.

[9] Soltrace homepage. https://www.nrel.gov/csp/soltrace.html.

[10] Global renewable energy consumption. https://ourworldindata.org/renewable-energy, 2016.

[11] Json schema specification, 2019. URL https://json-schema.org/specification.html.

[12] John F Ainslie. Calculating the flowfield in the wake of wind turbines. *Journal of Wind Engineering and Industrial Aerodynamics*, 27(1-3):213–224, 1988.

[13] Jagdish Chand Bansal and Pushpa Farswan. Wind farm layout using biogeography based optimization. *Renewable Energy*, 107:386–402, 2017.

[14] Jagdish Chand Bansal, Pushpa Farswan, and Atulya K Nagar. Design of wind farm layout with non-uniform turbines using fitness difference based bbo. *Engineering Applications of Artificial Intelligence*, 71:45–59, 2018.

[15] Martin Bilbao and Enrique Alba. Simulated annealing for optimization of wind farm annual profit. In *2009 2nd International Symposium on Logistics and Industrial Informatics*, pages 1–5. IEEE, 2009.

[16] Boost. Boost C++ Libraries. http://www.boost.org/, 2020.

[17] Jason Brownlee. Clever algorithms. *Nature-Inspired Programming Recipes*, 436, 2011.

[18] Ying Chen, Hua Li, Kai Jin, and Qing Song. Wind farm layout optimization using genetic algorithm with different hub height wind turbines. *Energy Conversion and Management*, 70:56–65, 2013.

[19] Kalyanmoy Deb and Samir Agrawal. Understanding interactions among genetic algorithm parameters. In *FOGA*, pages 265–286, 1998.

[20] Patrik Fagerfjäll. Optimizing wind farm layout: more bang for the buck using mixed integer linear programming. *Chalmers University of Technology and Gothenburg University*, page 111, 2010.

[21] Sten Tronæs Frandsen. *Turbulence and turbulence-generated structural loading in wind turbine clusters*. 2007.

[22] Linus Franke. Modeling and optimization of large scale solar tower power plants, 2018.

[23] Geneial. Genetic algorithm library geneial. URL: `http://www.geneial.org`.

[24] Javier Serrano González, Angel G Gonzalez Rodriguez, José Castro Mora, Jesús Riquelme Santos, and Manuel Burgos Payan. Optimization of wind farm turbines layout using an evolutive algorithm. *Renewable energy*, 35(8):1671–1681, 2010.

[25] SA Grady, MY Hussaini, and Makola M Abdullah. Placement of wind turbines using genetic algorithms. *Renewable energy*, 30(2):259–270, 2005.

[26] Joseph P Hennessey Jr. Some aspects of wind power statistics. *Journal of applied meteorology*, 16(2):119–128, 1977.

[27] Wanja Hentze. Solar tower power plant optimization using the globalized bounded nelder-mead method, 2019.

[28] José Herbert-Acero, Oliver Probst, Pierre-Elouan Réthoré, Gunner Larsen, and Krystel Castillo-Villar. A review of methodological approaches for the design and optimization of wind farms. *Energies*, 7(11):6930–7016, 2014.

[29] John H Holland. Iterative circuit computers. In *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*, pages 259–265, 1960.

[30] Tzung-Pei Hong, Hong-Shung Wang, Wen-Yang Lin, and Wen-Yuan Lee. Evolution of appropriate crossover and mutation operators in a genetic process. *Applied Intelligence*, 16(1):7–17, 2002.

[31] Peng Hou, Weihao Hu, Mohsen Soltani, and Zhe Chen. Optimized placement of wind turbines in large-scale offshore wind farm using particle swarm optimization algorithm. *IEEE Transactions on Sustainable Energy*, 6(4):1272–1282, 2015.

[32] Niels Otto Jensen. *A note on wind generator interaction*. Number 2411 in Risø-M. 1983.

[33] Chao Li, Rongrong Zhai, Hongtao Liu, Yongping Yang, and Hao Wu. Optimization of a heliostat field layout using hybrid pso-ga algorithm. *Applied Thermal Engineering*, 128:33–41, 2018.

[34] Niels Lohmann. json. URL `https://github.com/nlohmann/json`.

[35] Shanley Lawrence Lutchman. *Heliostat field layout optimization for a central receiver*. PhD thesis, Stellenbosch: Stellenbosch University, 2014.

[36] Marwa Maghnie. Simulation and layout optimization of offshore wind farms, 2019.

[37] Grigorios Marmidis, Stavros Lazarou, and Eleftheria Pyrgioti. Optimal placement of wind turbines in a wind park using monte carlo simulation. *Renewable energy*, 33(7):1455–1460, 2008.

[38] Arash Mohammadi, Houshyar Asadi, Shady Mohamed, Kyle Nelson, and Saeid Nahavandi. Openga, a c++ genetic algorithm library. In *Systems, Man, and Cybernetics (SMC), 2017 IEEE International Conference on*, pages 2051–2056. IEEE, 2017.

[39] GPCDB Mosetti, Carlo Poloni, and B Diviacco. Optimization of wind turbine positioning in large windfarms by means of a genetic algorithm. *Journal of Wind Engineering and Industrial Aerodynamics*, 51(1):105–116, 1994.

[40] Lukas Netz. Analysis on discretization of gene parameters in evolutionary algorithms, 2017.

[41] A Neubert, A Shah, and W Schlez. Maximum yield from symmetrical wind farm layouts. In *Proceedings of DEWEK*, pages 17–18, 2010.

[42] Corey J Noone, Manuel Torrilhon, and Alexander Mitsos. Heliostat field optimization: A new computationally efficient model and biomimetic layout. *Solar Energy*, 86(2):792–803, 2012.

[43] U Aytun Ozturk and Bryan A Norman. Heuristic methods for wind energy conversion system positioning. *Electric Power Systems Research*, 70(3):179–185, 2004.

[44] Mukund R Patel. Design, analysis, and operation, wind and solar power system, 2006.

[45] pboettch. json-schema-validator. URL `https://github.com/pboettch/json-schema-validator`.

[46] Robert Pitz-Paal, Nicolas Bayer Botero, and Aldo Steinfeld. Heliostat field layout optimization for high-temperature solar thermochemical processing. *Solar energy*, 85(2):334–343, 2011.

[47] Pascal Richter. Simulation and optimization of solar thermal power plants, 2017.

[48] Rajai Aghabi Rivas, Jens Clausen, Kurt S Hansen, and Leo E Jensen. Solving the turbine positioning problem for large offshore wind farms by simulated annealing. *Wind Engineering*, 33(3):287–297, 2009.

[49] A. Sarkar, Gaurav Gugliani, and Sneh Deep. Weibull model for wind speed data analysis of different locations in india. *KSCE Journal of Civil Engineering*, 03 2017. doi: 10.1007/s12205-017-0538-5.

[50] Rabia Shakoor, Mohammad Yusri Hassan, Abdur Raheem, and Yuan-Kang Wu. Wake effect modeling: A review of wind farm layout optimization using jensen's model. *Renewable and Sustainable Energy Reviews*, 58:1048–1059, 2016.

[51] Dan Simon. Biogeography-based optimization. *IEEE transactions on evolutionary computation*, 12(6):702–713, 2008.

[52] C Szafron. Offshore windfarm layout optimization. In *2010 9th international conference on environment and electrical engineering*, pages 542–545. IEEE, 2010.

[53] Angelo Tesauro, Pierre-Elouan Réthoré, and Gunner Chr Larsen. State of the art of wind farm optimization. In *EWEA 2012-European Wind Energy Conference & Exhibition*. European Wind Energy Association (EWEA), 2012.

[54] Markus Wagner, Jareth Day, and Frank Neumann. A fast and effective local search algorithm for optimizing the placement of wind turbines. *Renewable Energy*, 51: 64–70, 2013.

[55] Matthew Wall. Galib. URL `http://lancet.mit.edu/ga/`.

[56] Chunqiu Wan, Jun Wang, Geng Yang, Xiaolan Li, and Xing Zhang. Optimal micro-siting of wind turbines by genetic algorithms based on improved wind and turbine models. In *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, pages 5092–5096. IEEE, 2009.

[57] Chunqiu Wan, Jun Wang, Geng Yang, and Xing Zhang. Optimal micro-siting of wind farms by particle swarm optimization. In *International Conference in Swarm Intelligence*, pages 198–205. Springer, 2010.

[58] JunHua Zhao, Fushuan Wen, Zhao Yang Dong, Yusheng Xue, and Kit Po Wong. Optimal dispatch of electric vehicles and wind power using enhanced particle swarm optimization. *IEEE Transactions on industrial informatics*, 8(4):889–899, 2012.

A. Appendix

A.1. Results of SunFlower Pattern Optimization

Tables 11 and 12 contain the fitness of different patterns that were optimized using the Globalized Bounded Nelder-Mead method.

Pattern	fitness
biomimetic	0.0697
contracted honeycombs	DNF
hexagonal	0.0679
regular cornfield	0.1110
staggered cornfield	0.1121

Table 11: Results of the pattern optimization for the Gemasolar test case

Pattern	fitness
biomimetic	0.4313
contracted honeycombs	0.4580
hexagonal	0.2215
regular cornfield	0.3843
staggered cornfield	0.4427

Table 12: Results of the pattern optimization for the PS10 test case

A.2. Plots of Optimization Performance

A.2.1. Population Size vs. Iterations

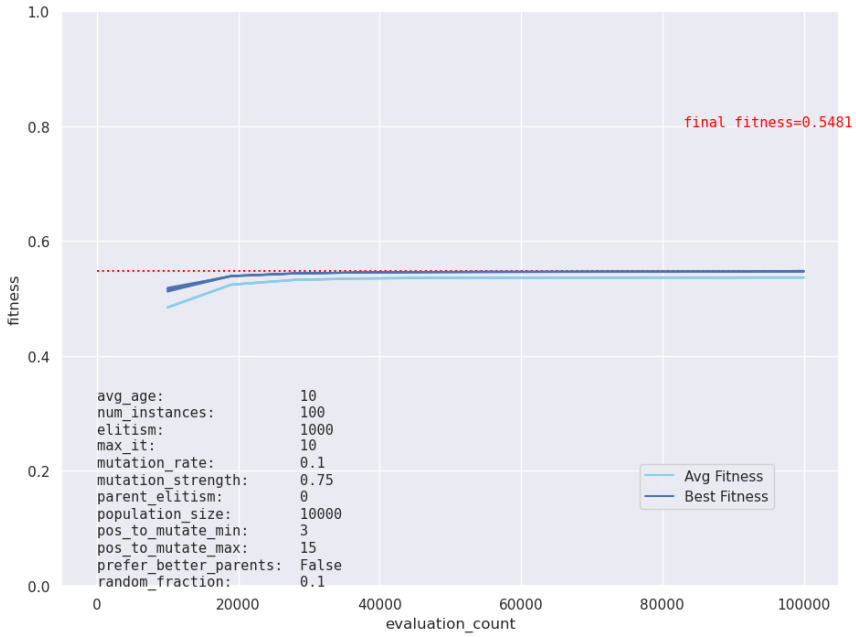


Figure 20: PS10: Very high population size, very few iterations

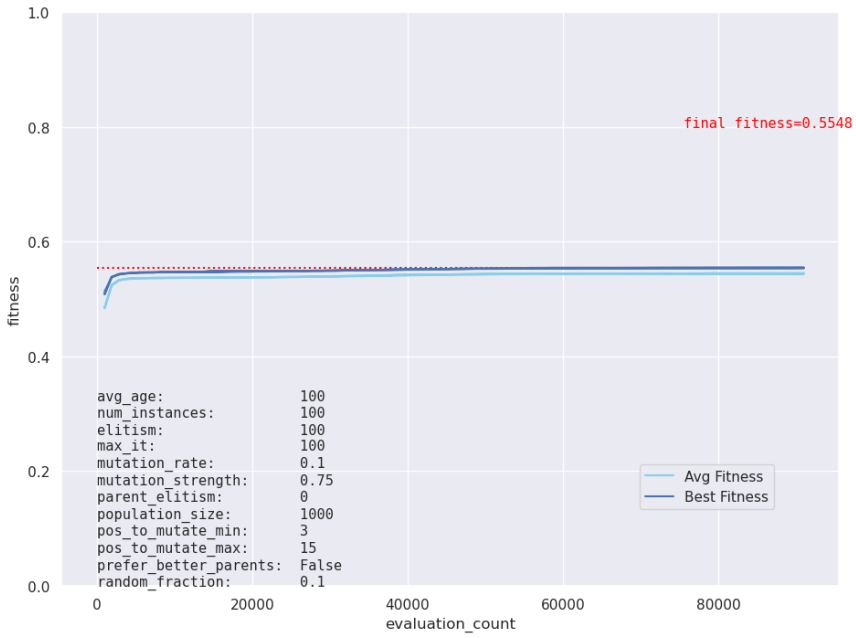


Figure 21: PS10: High population size, few iterations

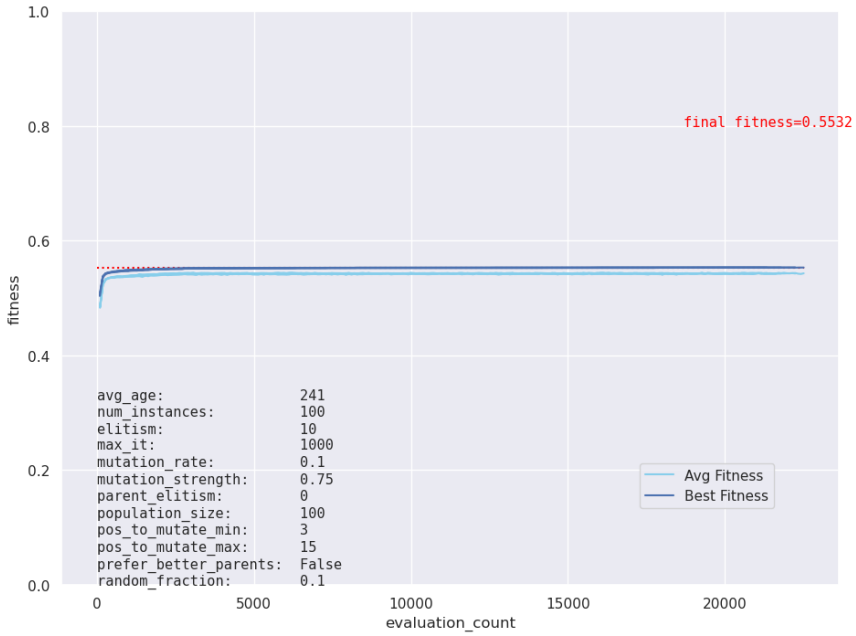


Figure 22: PS10: Reference population size, reference iterations

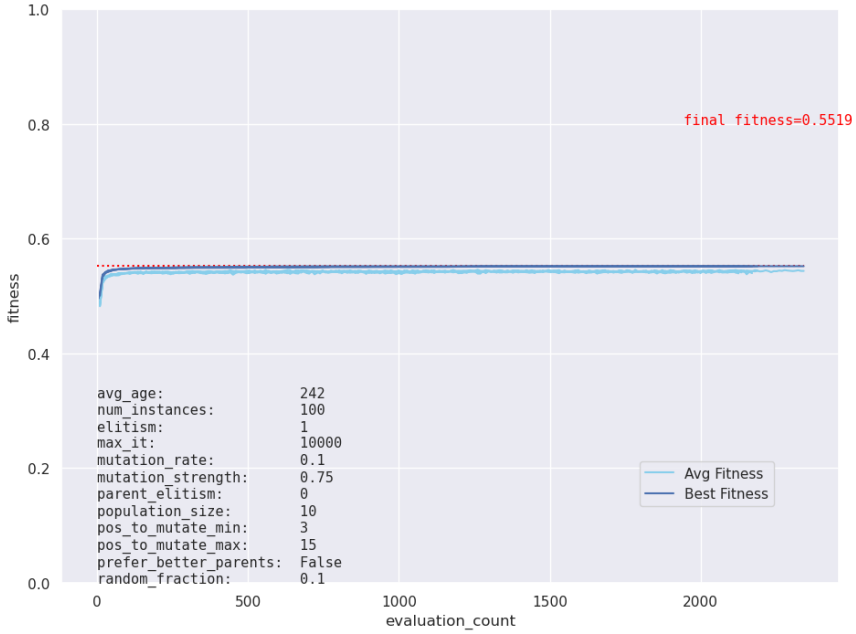


Figure 23: PS10: Small population size, many iterations

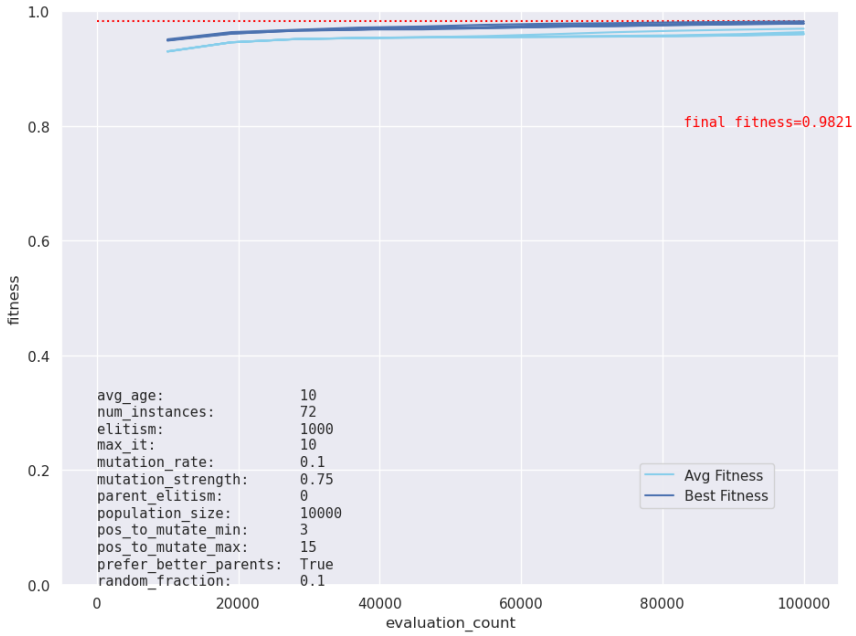


Figure 24: Sandbank: Very high population size, very few iterations

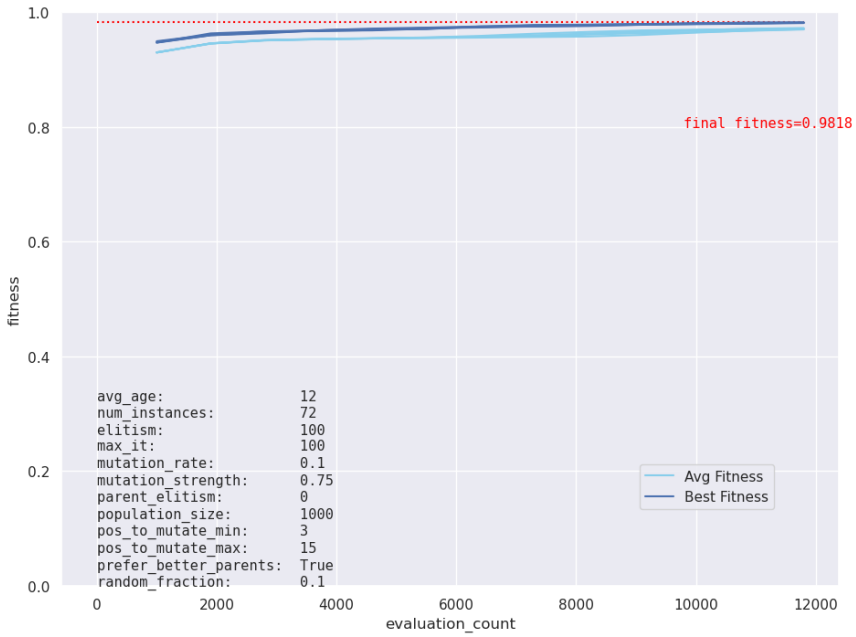


Figure 25: Sandbank: High population size, few iterations

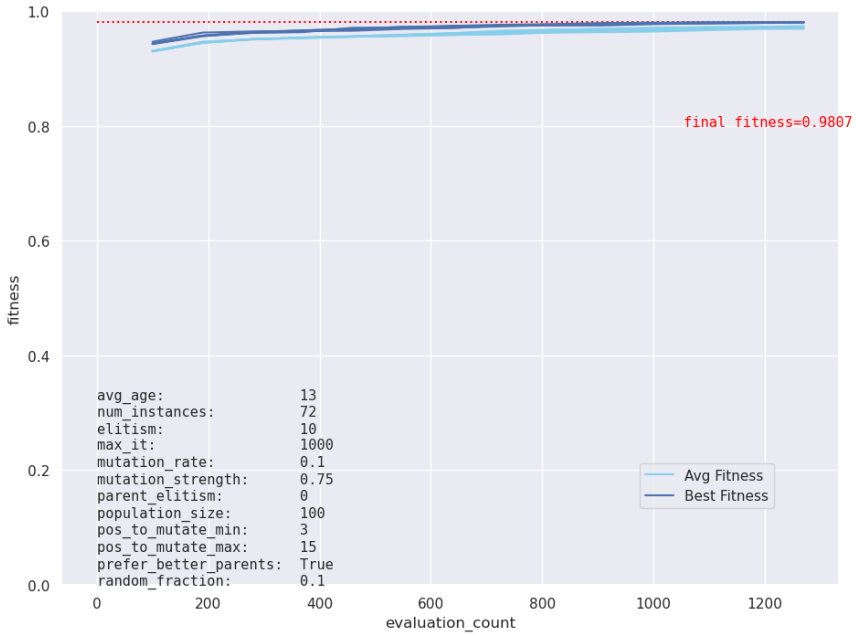


Figure 26: Sandbank: Reference population size, reference iterations

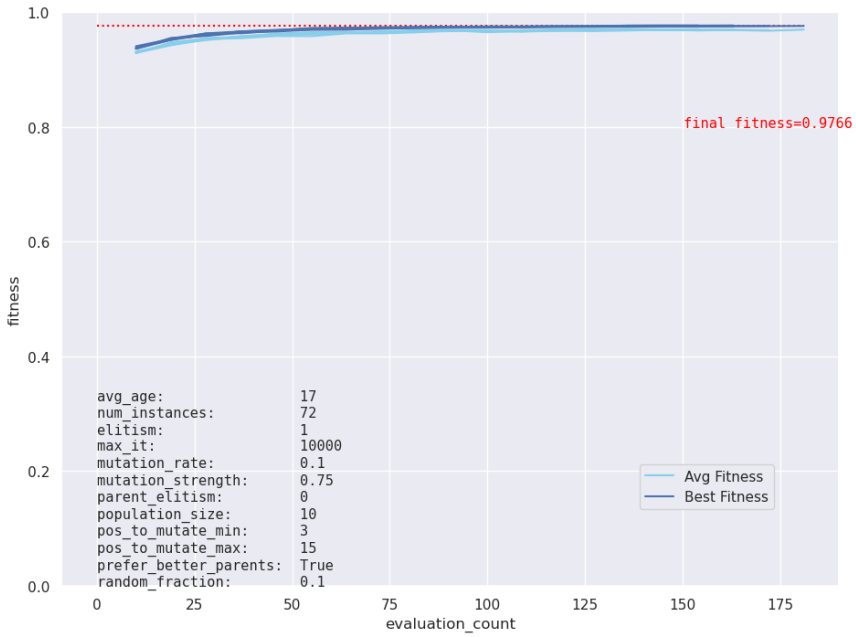


Figure 27: Sandbank: Small population size, many iterations

A.2.2. Reference configuration

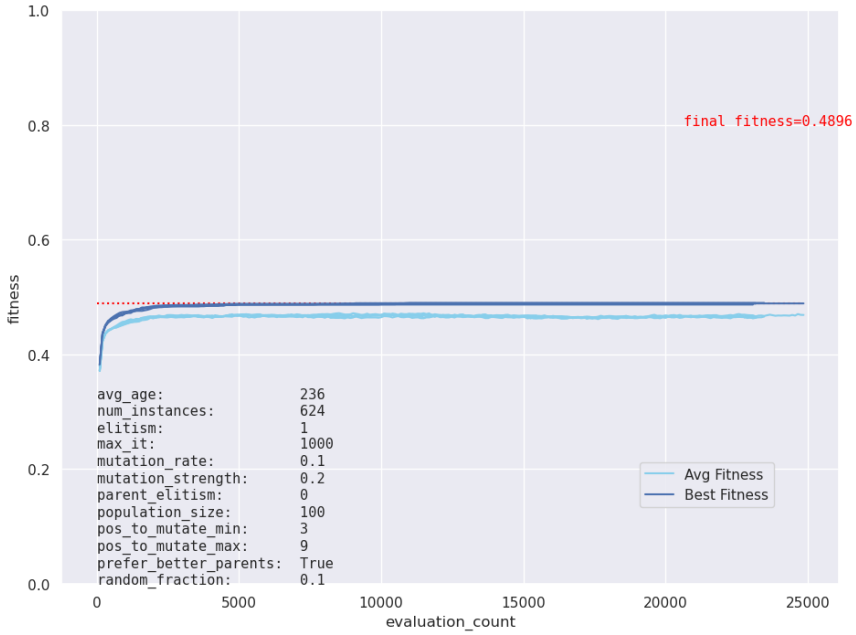


Figure 28: PS10: Reference configuration

A.2.3. Breeding Probability depending on Fitness

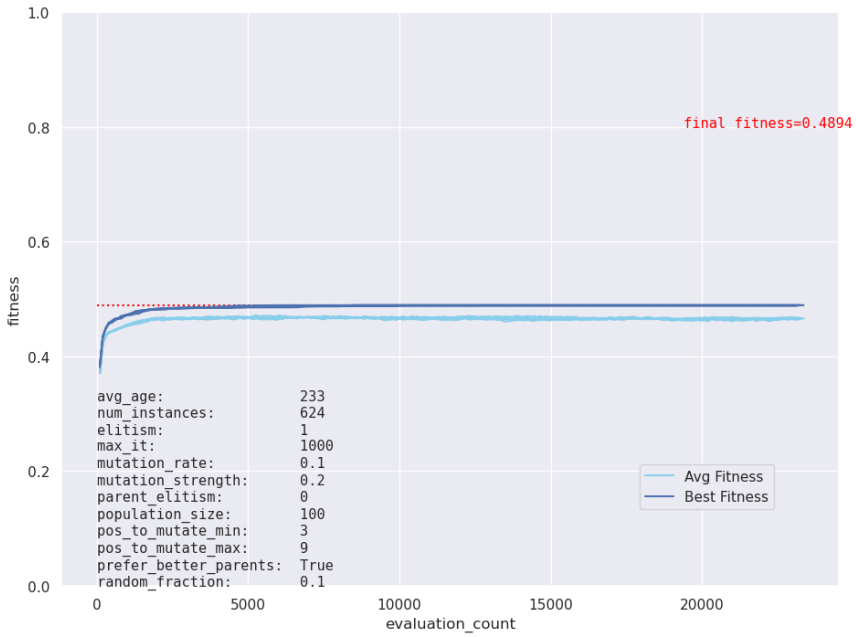


Figure 29: PS10: Prefer breeding genes with higher fitness

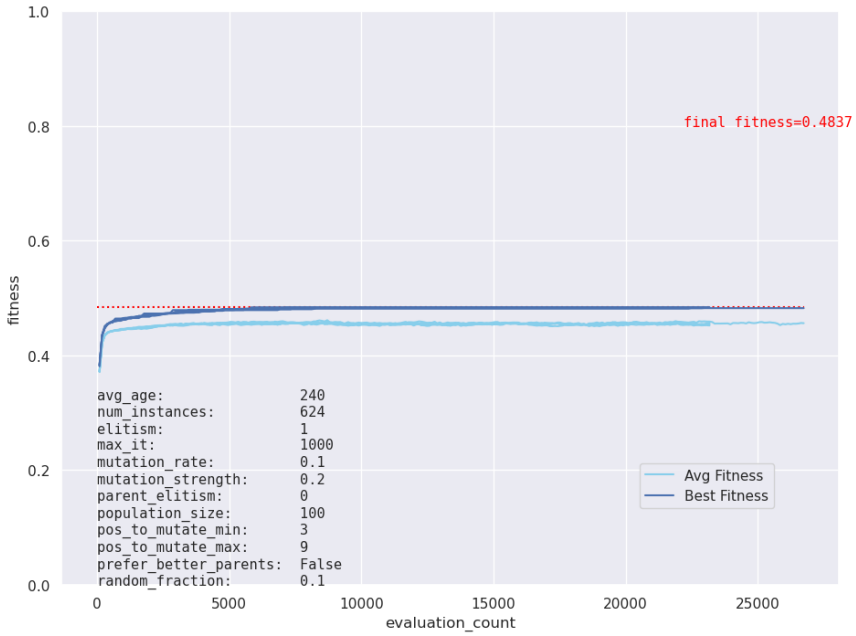


Figure 30: PS10: All genes have equal breeding probability

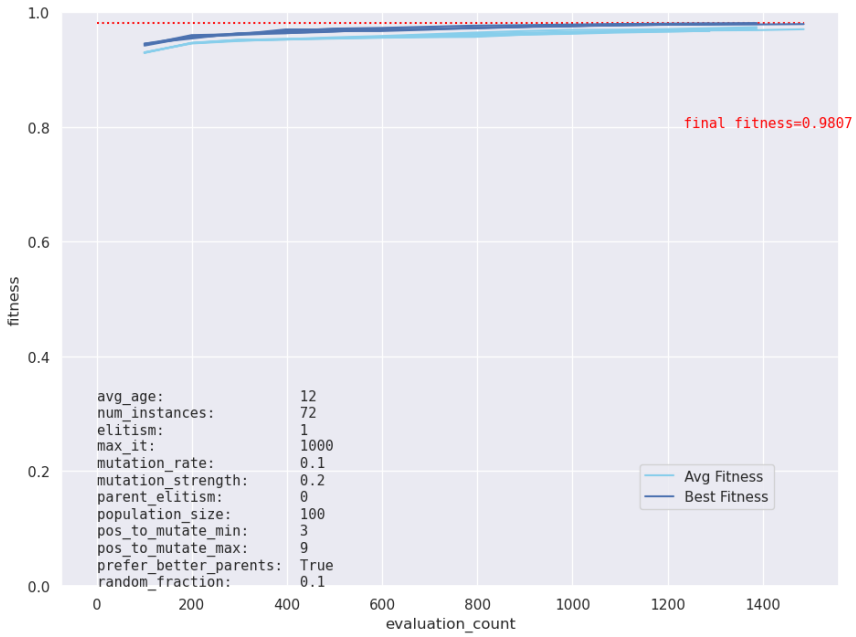


Figure 31: Sandbank: Prefer breeding genes with higher fitness

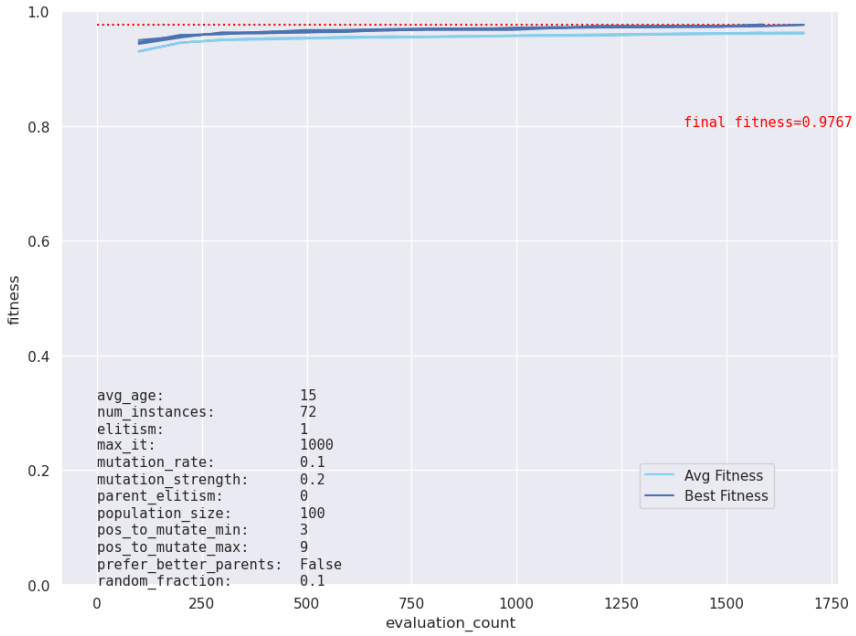


Figure 32: Sandbank: All genes have equal breeding probability

A.2.4. Enforce Breeding the Best Chromosomes

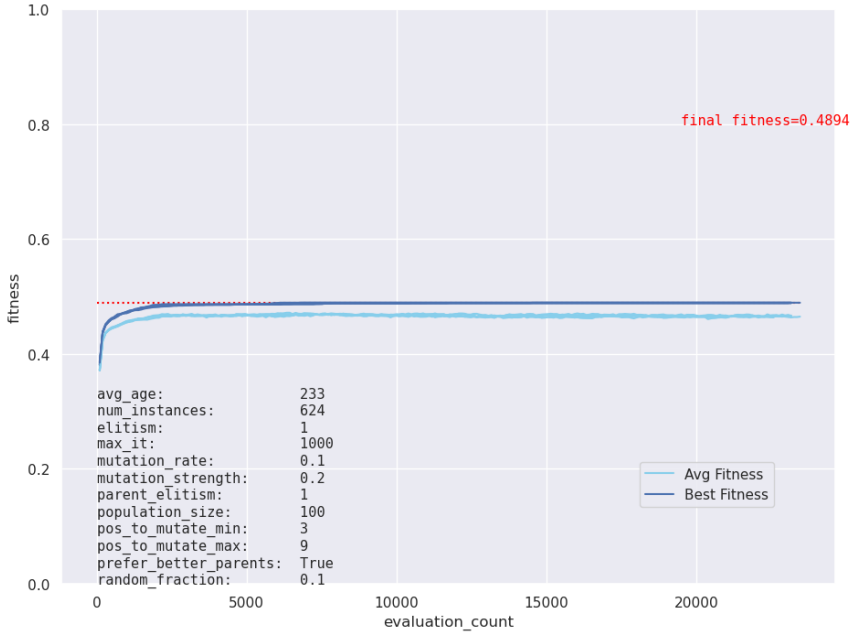


Figure 33: PS10: Always breed the best chromosome

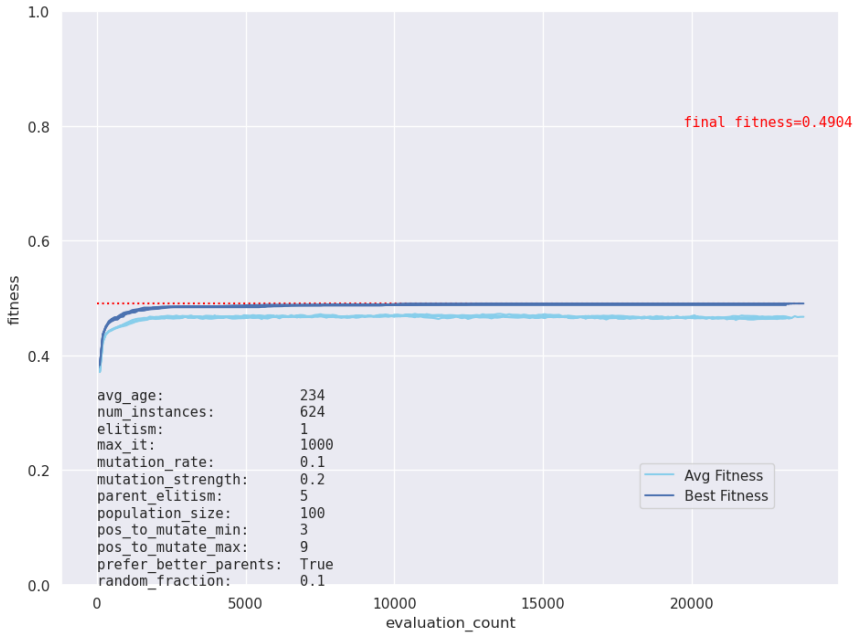


Figure 34: PS10: Always breed the best five chromosomes

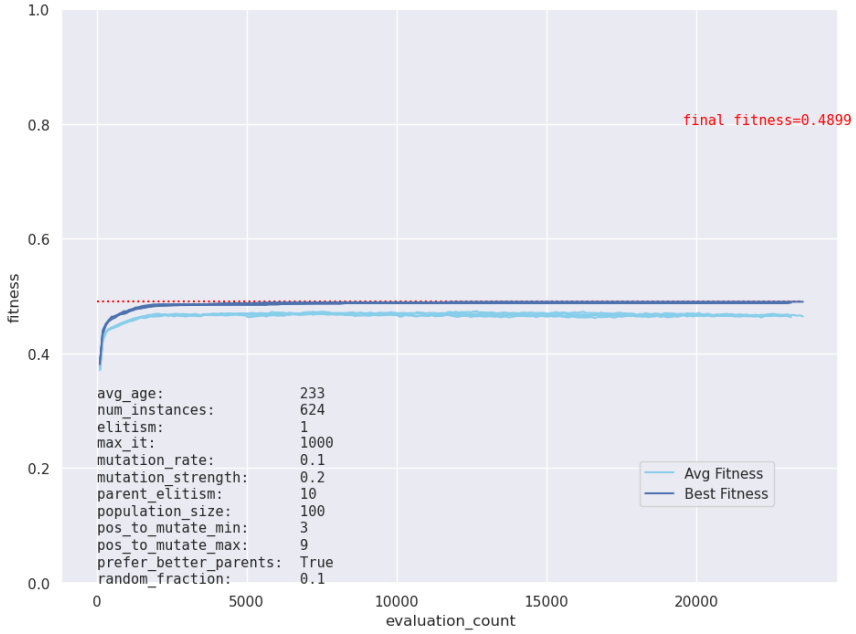


Figure 35: PS10: Always breed the best ten chromosomes

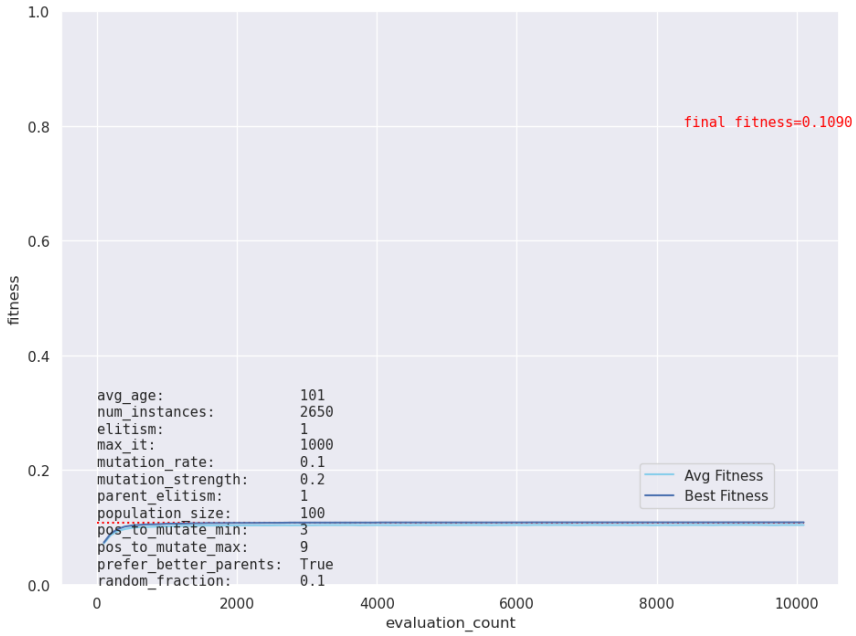


Figure 36: Gemasolar: Always breed the best chromosome

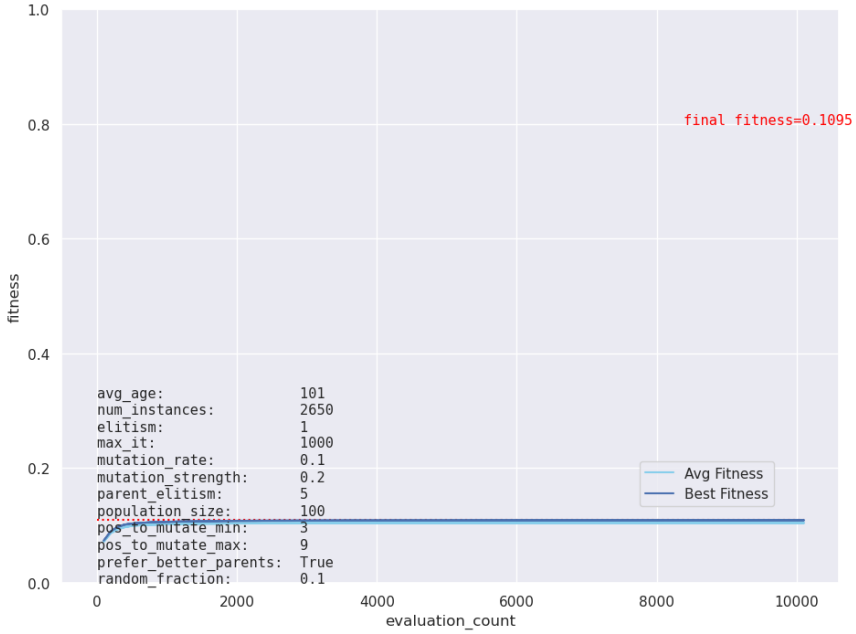


Figure 37: Gemasolar: Always breed the best five chromosomes

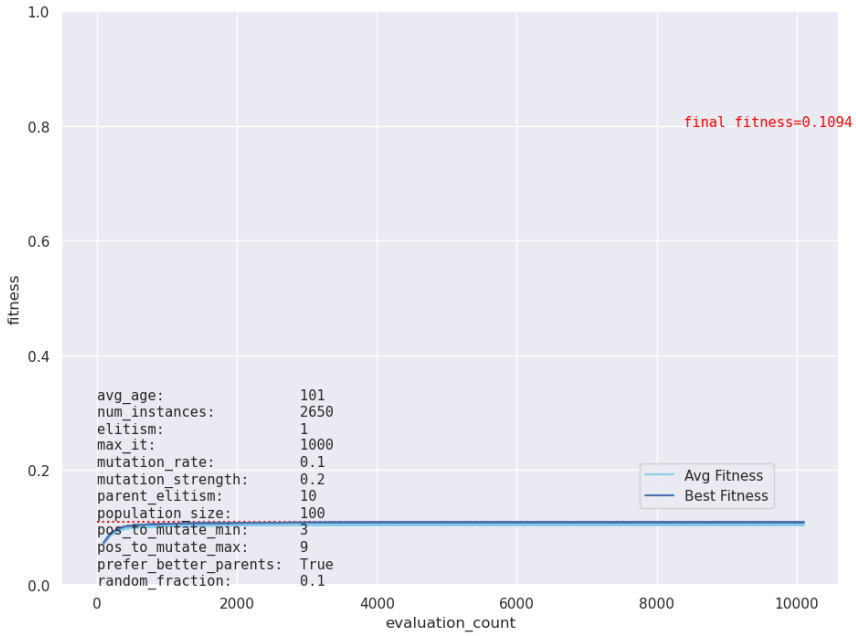


Figure 38: Gemasolar: Always breed the best ten chromosomes

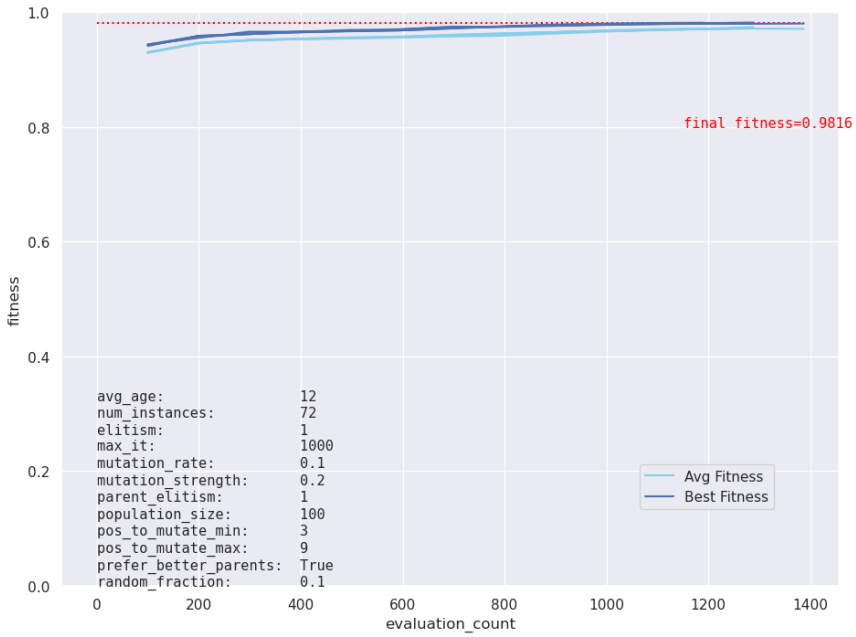


Figure 39: Sandbank: Always breed the best chromosomes

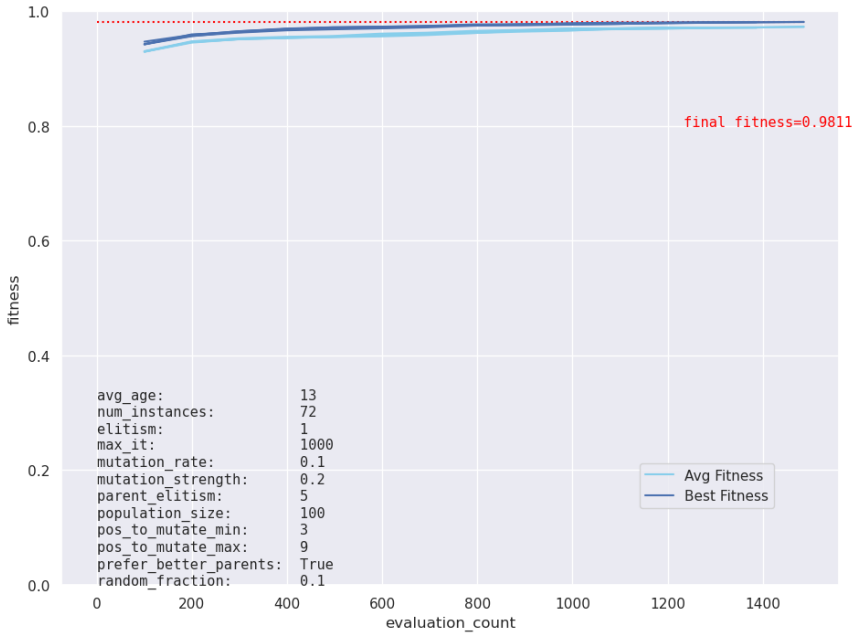


Figure 40: Sandbank: Always breed the best five chromosomes

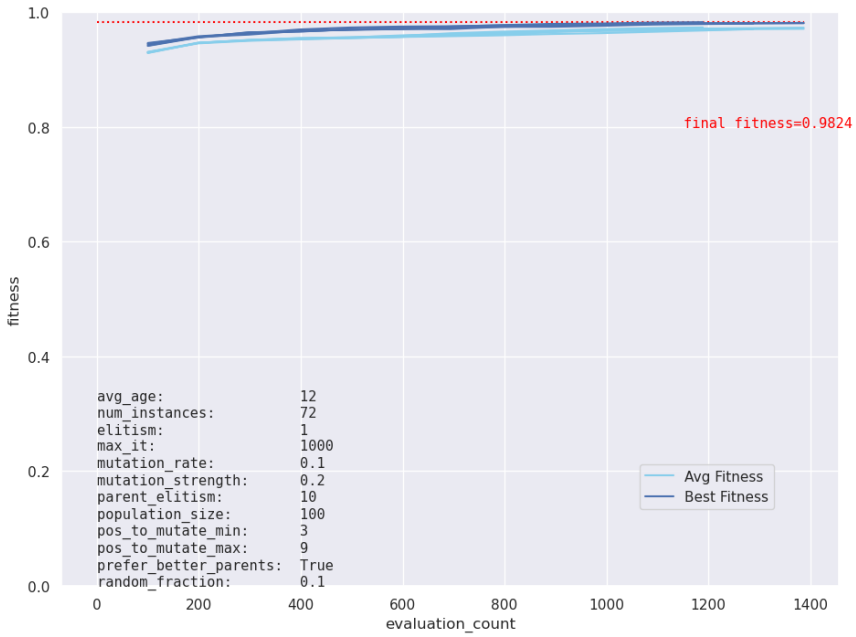


Figure 41: Sandbank: Always breed the best ten chromosomes

A.2.5. High Percentage of Random Solutions

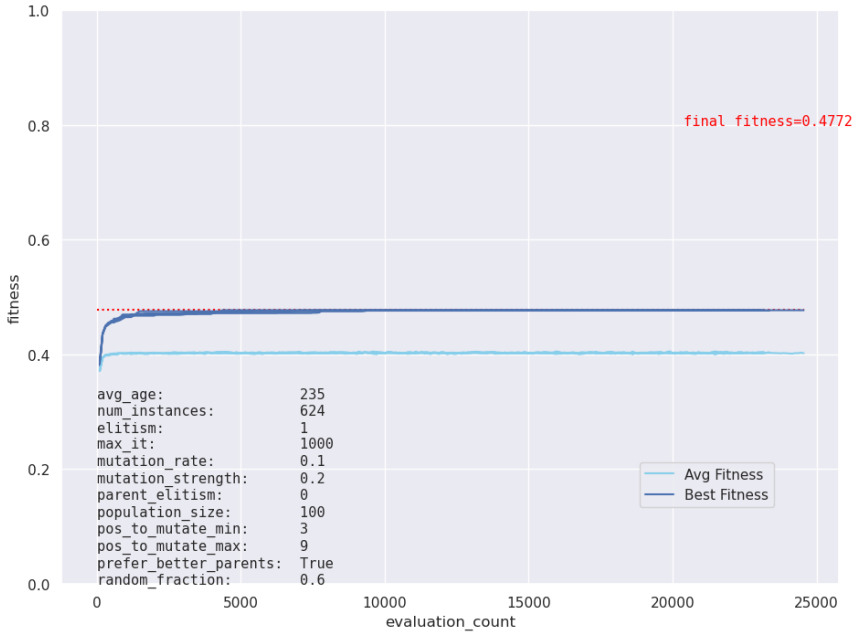


Figure 42: PS10: Inject high amounts of random chromosomes, 60%

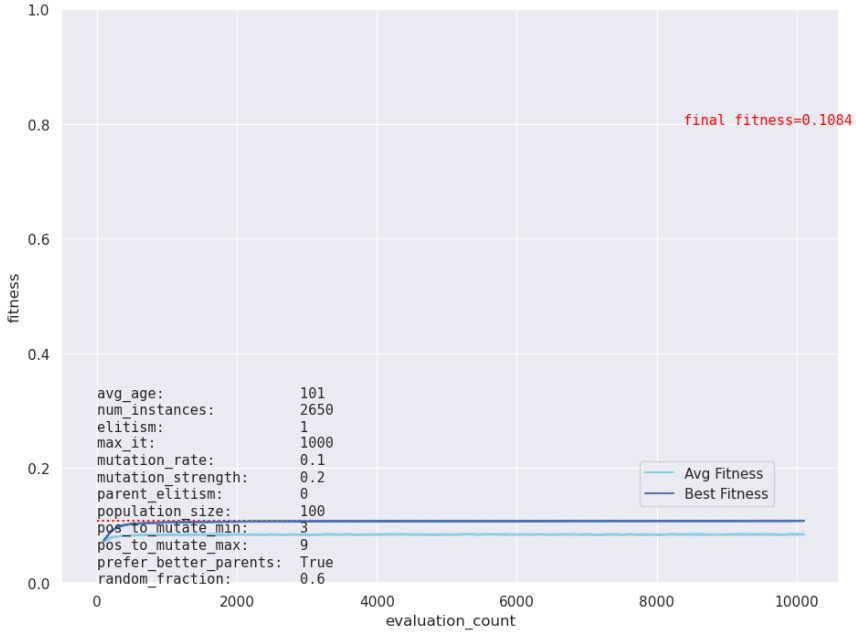


Figure 43: Gemasolar: Inject high amounts of random chromosomes, 60%

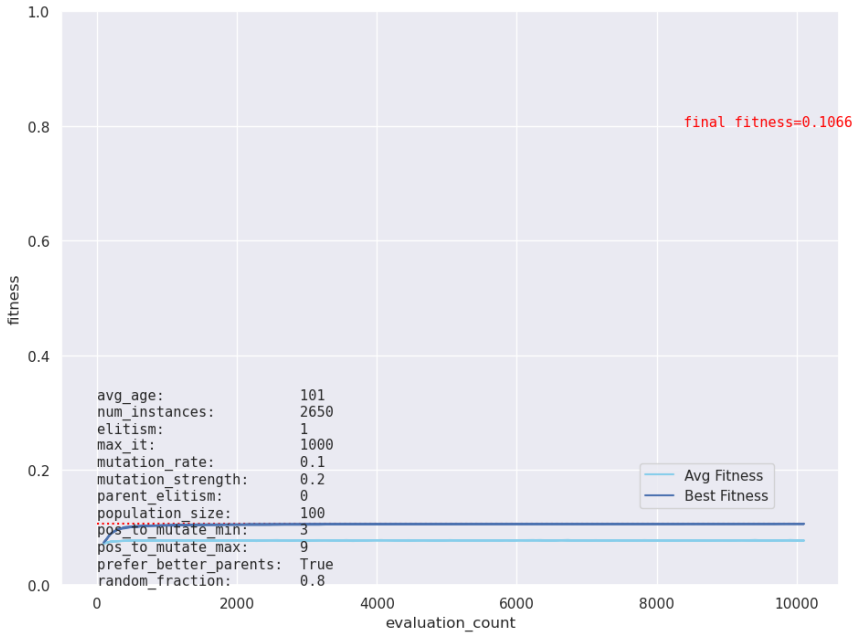


Figure 44: Gemasolar: Inject high amounts of random chromosomes, 80%

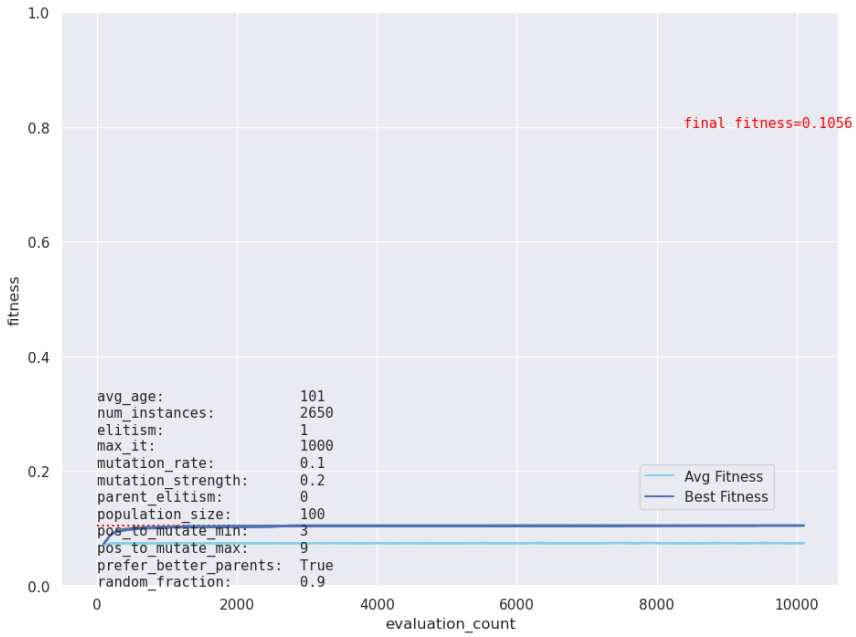


Figure 45: Gemasolar: Inject high amounts of random chromosomes, 90%

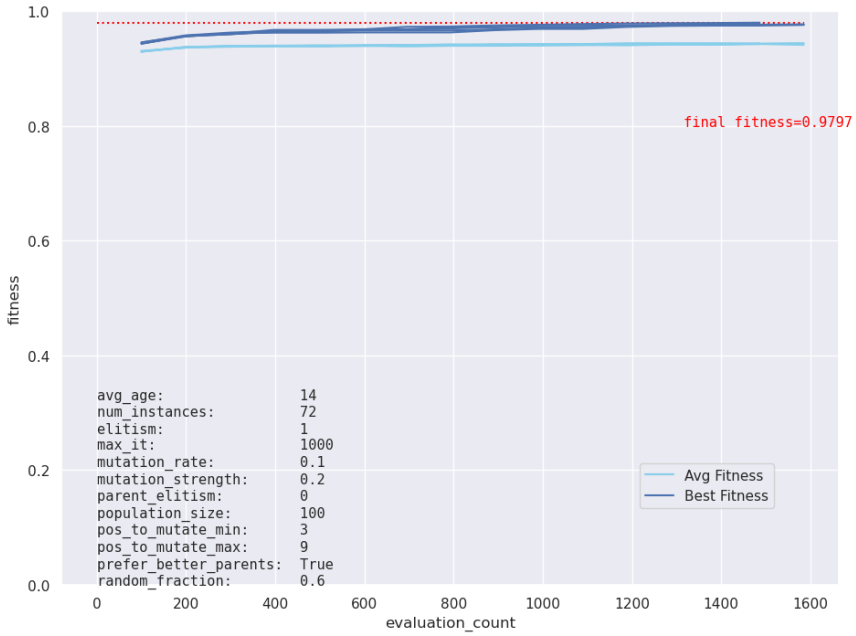


Figure 46: Sandbank: Inject high amounts of random chromosomes, 60%

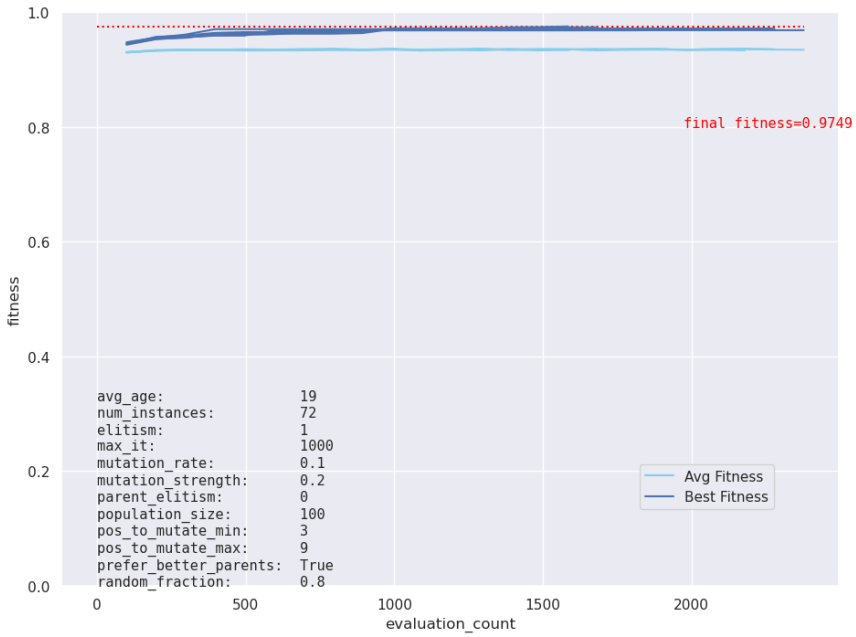


Figure 47: Sandbank: Inject high amounts of random chromosomes, 80%

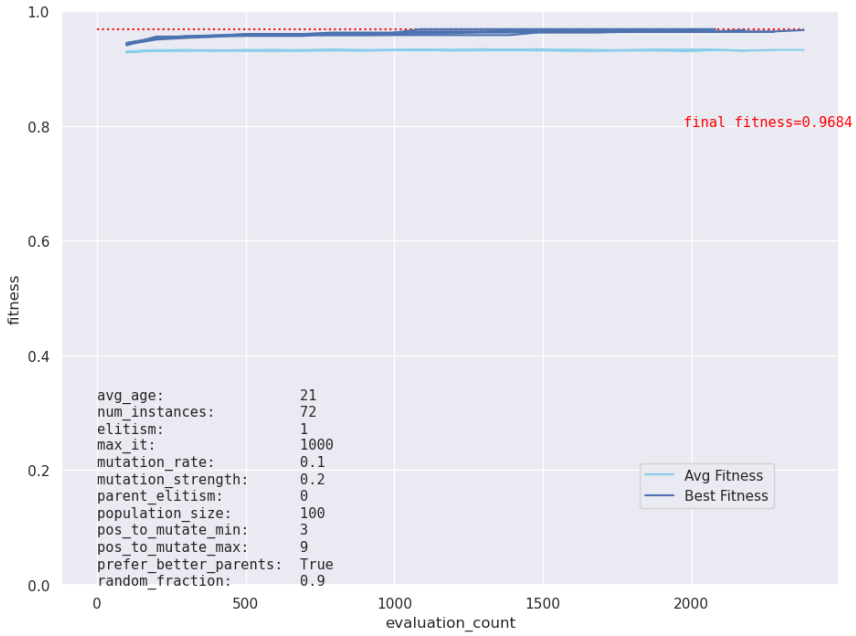


Figure 48: Sandbank: Inject high amounts of random chromosomes, 90%

A.2.6. Randomizing the Number of Mutated Instances

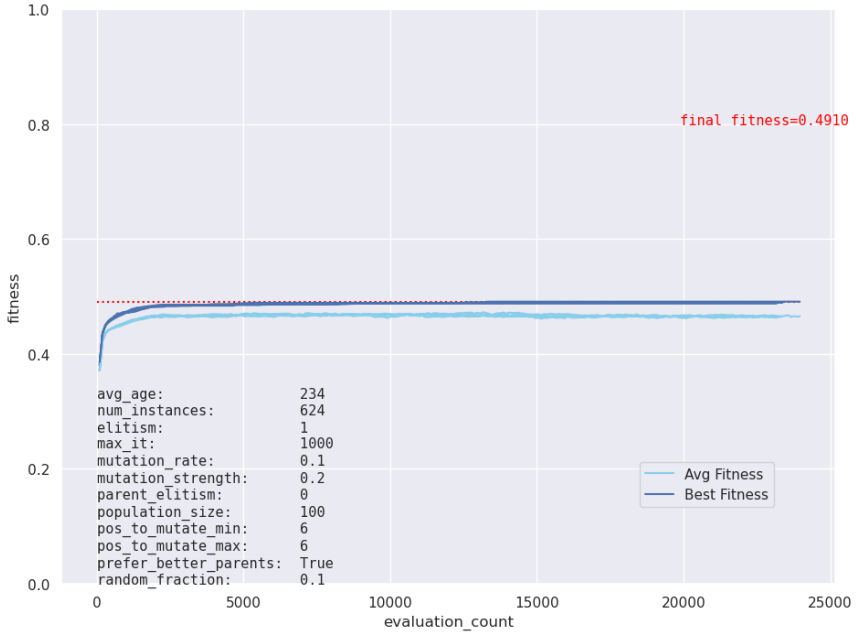


Figure 49: PS10: Always mutate the same amount of positions

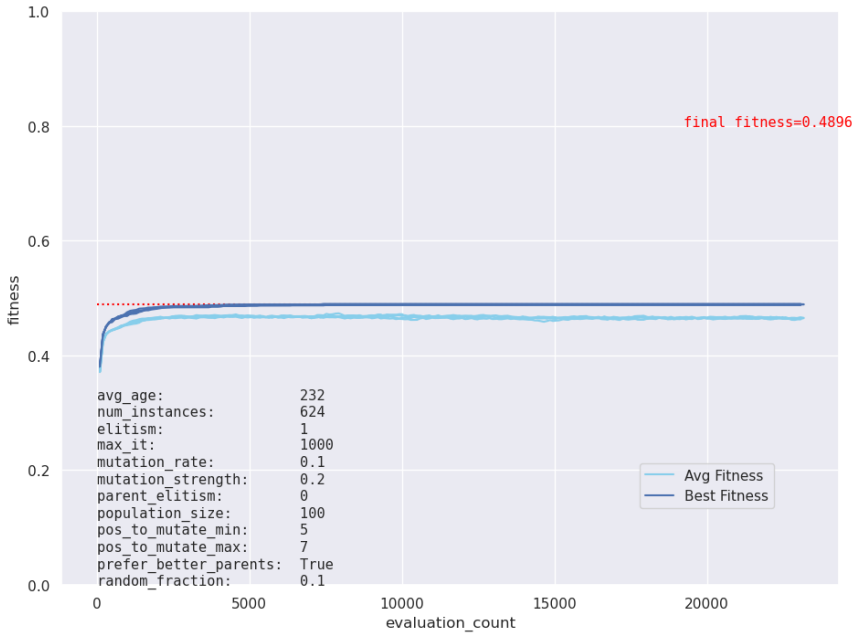


Figure 50: PS10: Light spread in number of mutated positions

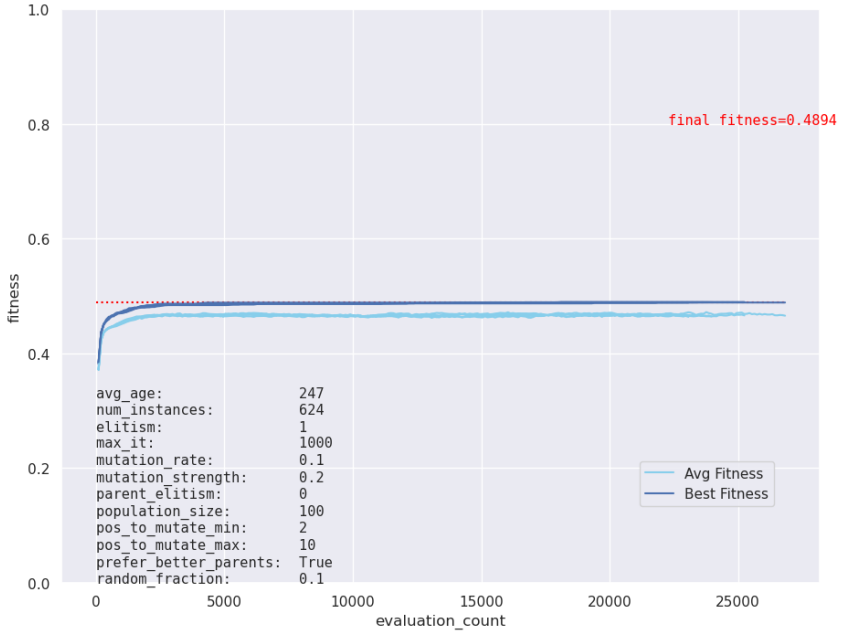


Figure 51: PS10: Medium spread in number of mutated positions

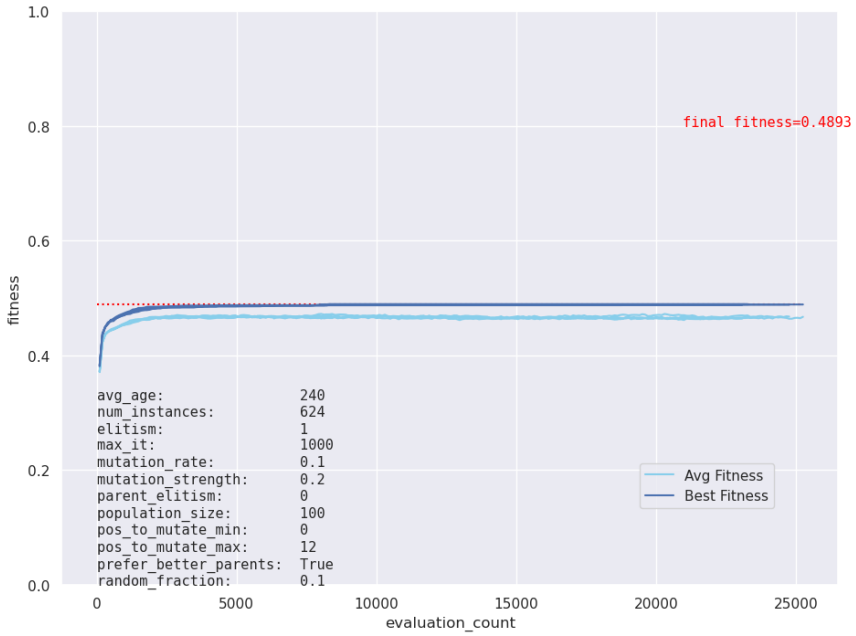


Figure 52: PS10: High spread in number of mutated positions

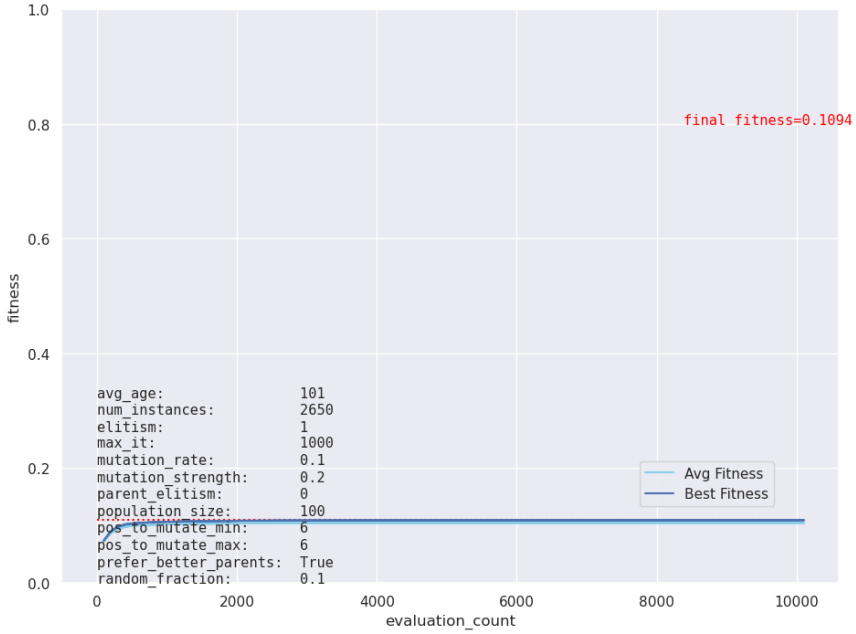


Figure 53: Gemasolar: Always mutate the same amount of positions

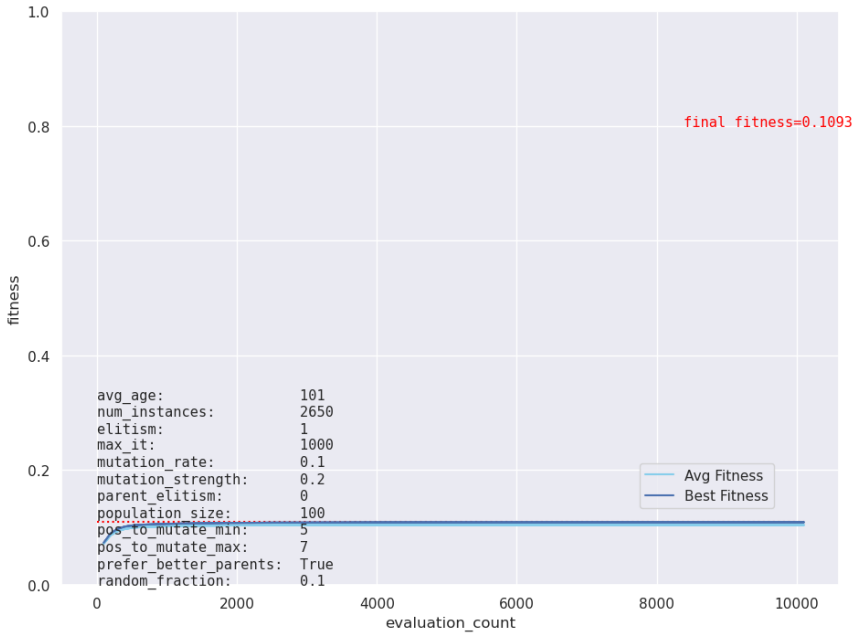


Figure 54: Gemasolar: Light spread in number of mutated positions

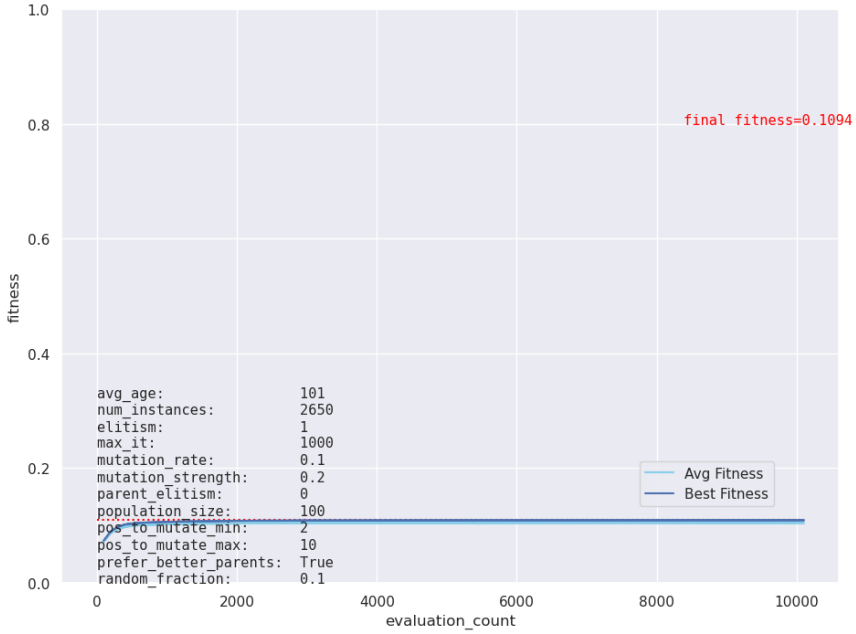


Figure 55: Gemasolar: Medium spread in number of mutated positions

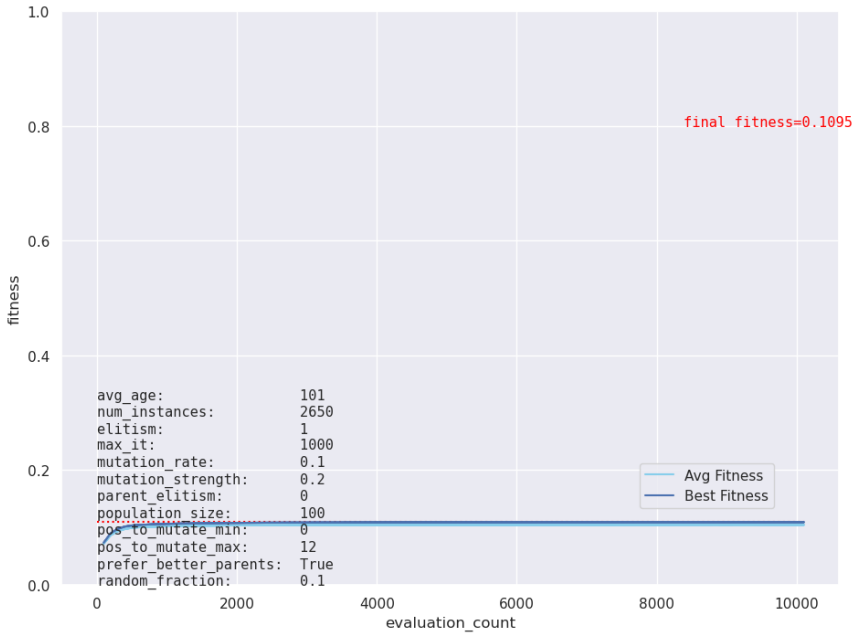


Figure 56: Gemasolar: High spread in number of mutated positions

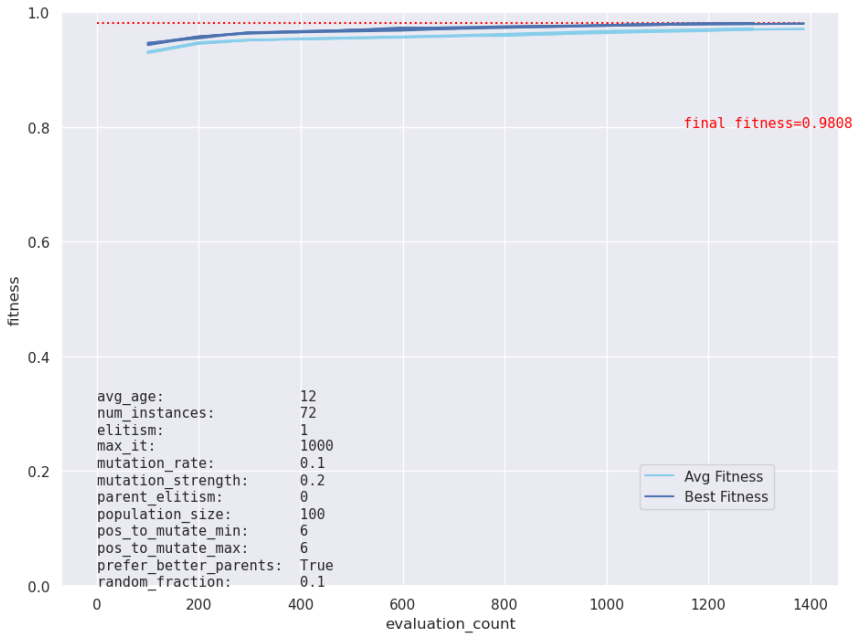


Figure 57: Sandbank: Always mutate the same amount of positions

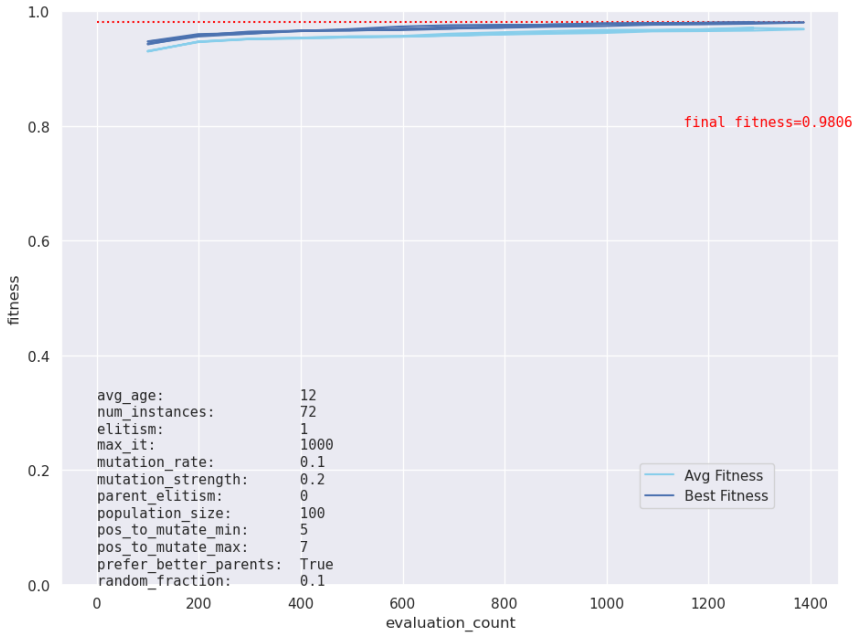


Figure 58: Sandbank: Light spread in number of mutated positions

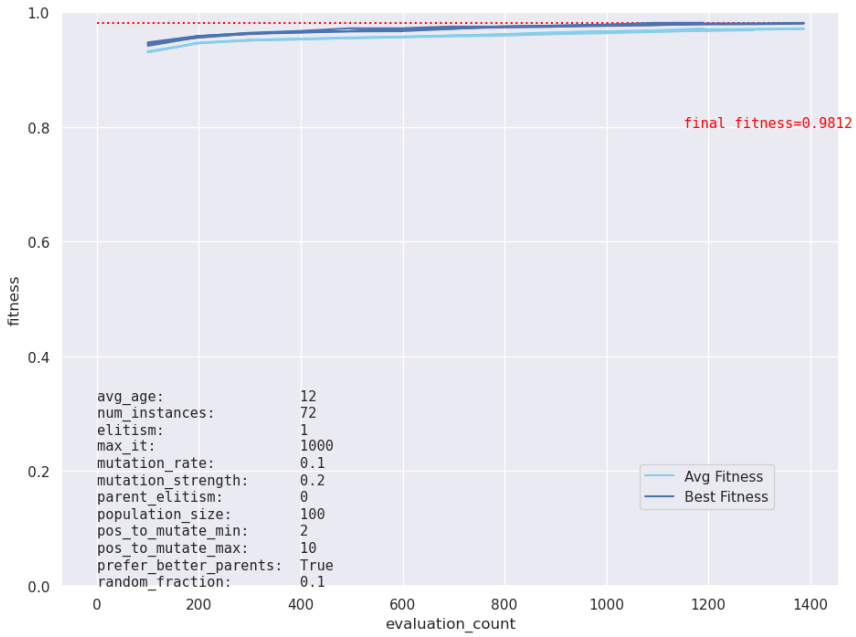


Figure 59: Sandbank: Medium spread in number of mutated positions

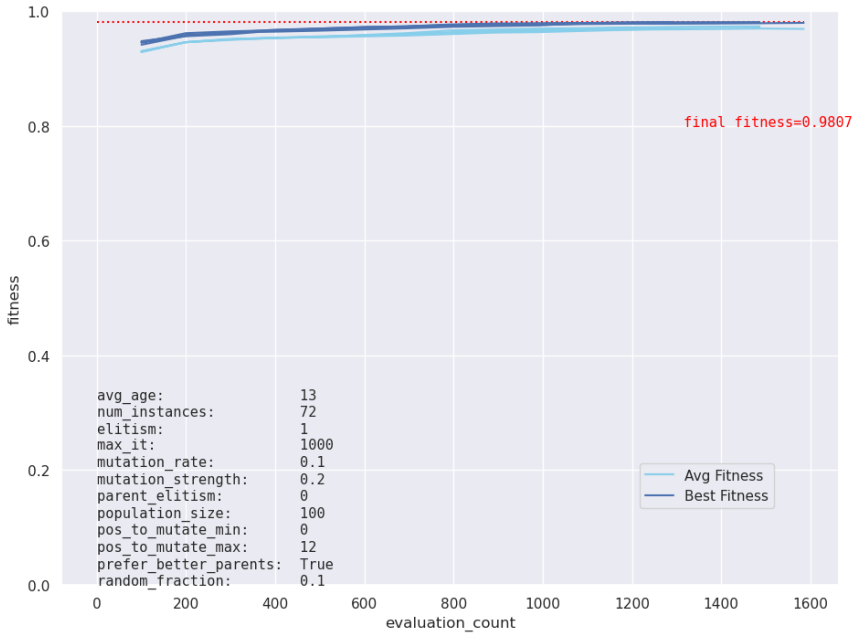


Figure 60: Sandbank: High spread in number of mutated positions

A.2.7. Mutation Variation

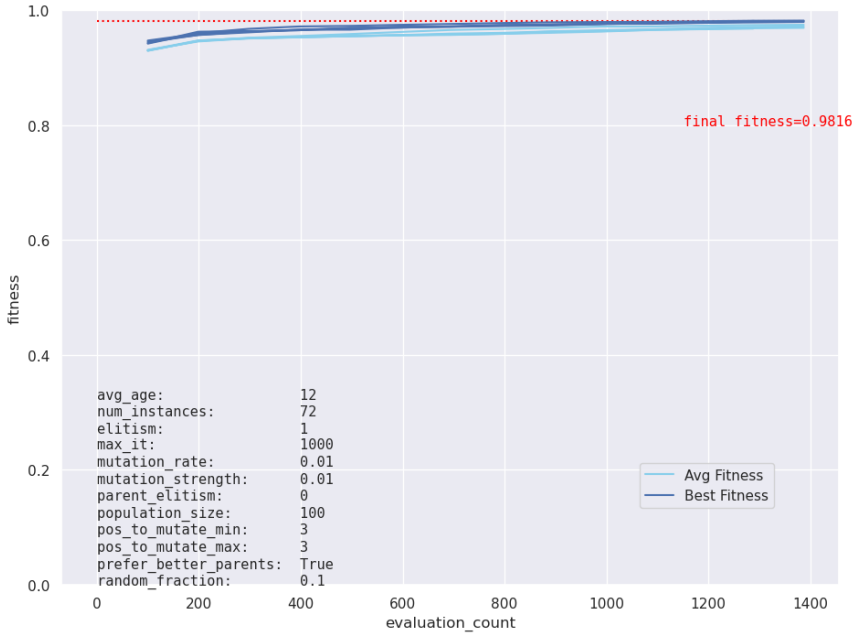


Figure 61: Sandbank: Low number of mutated positions, low mutation strength, low mutation rate

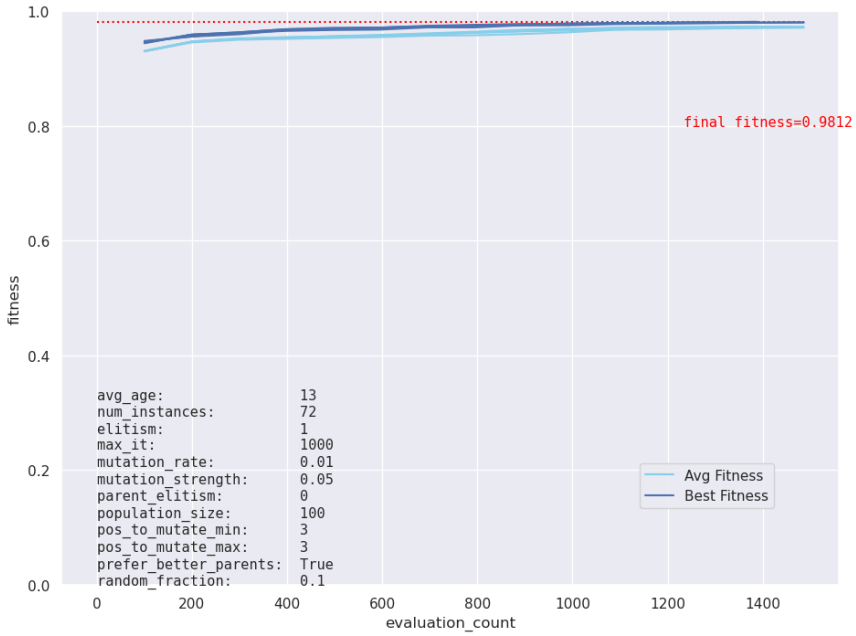


Figure 62: Sandbank: Low number of mutated positions, medium mutation strength, low mutation rate

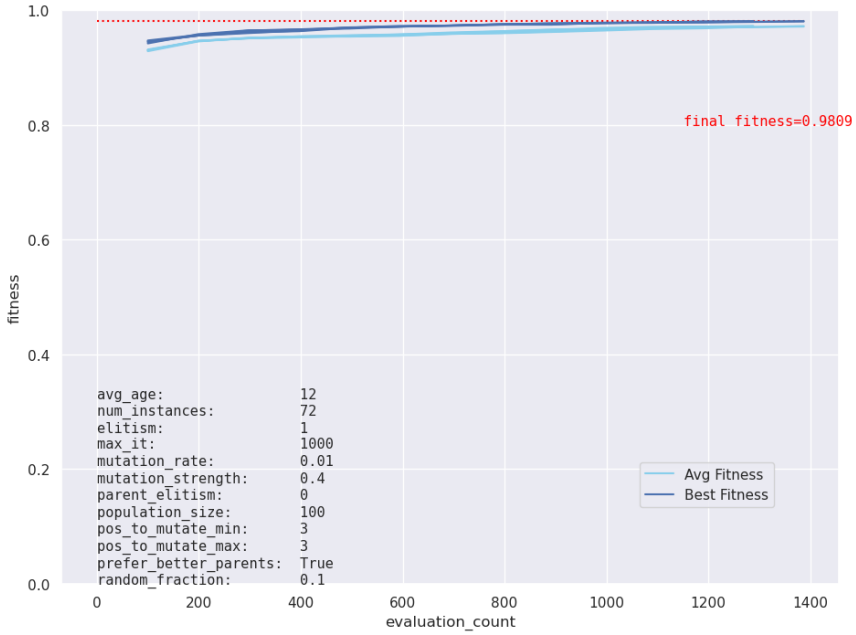


Figure 63: Sandbank: Low number of mutated positions, high mutation strength, low mutation rate

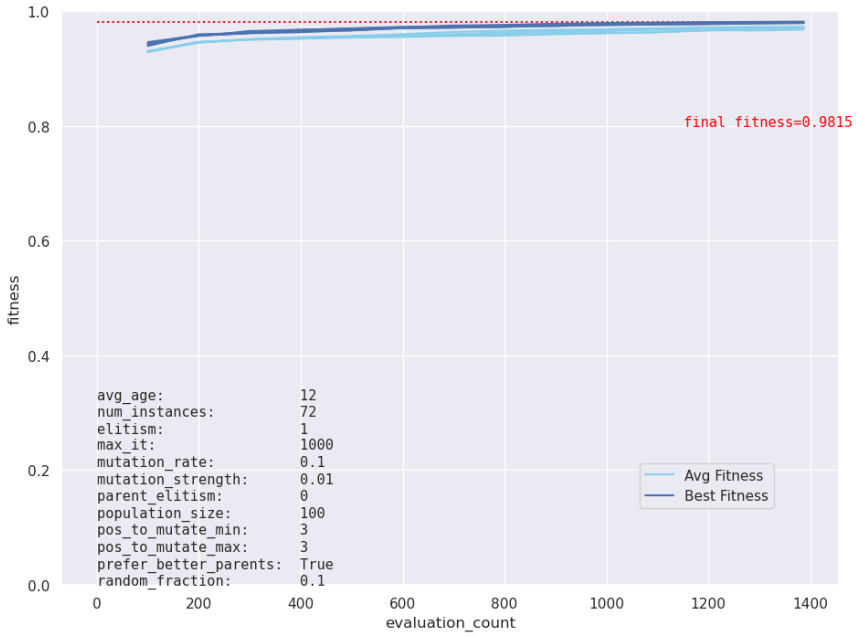


Figure 64: Sandbank: Low number of mutated positions, low mutation strength, medium mutation rate

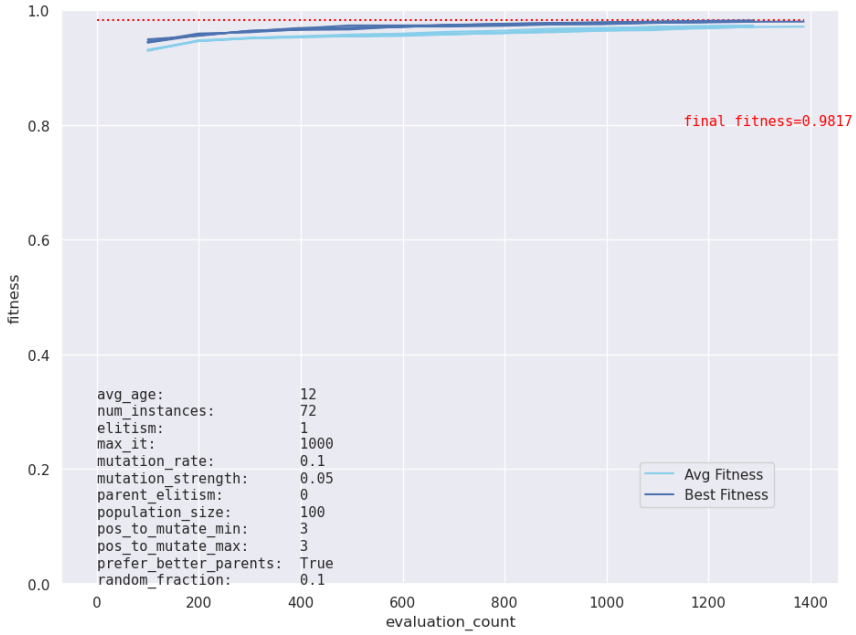


Figure 65: Sandbank: Low number of mutated positions, medium mutation strength, medium mutation rate

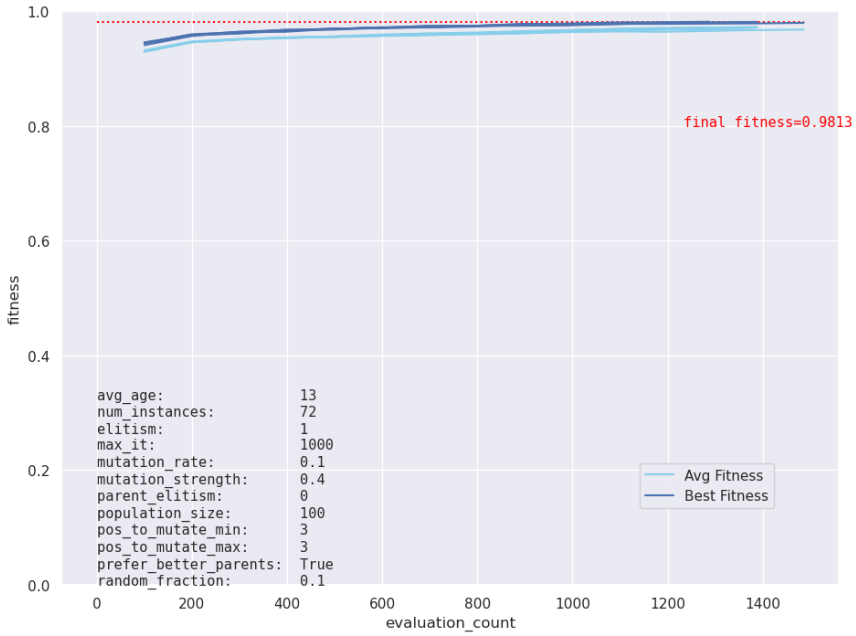


Figure 66: Sandbank: Low number of mutated positions, high mutation strength, medium mutation rate

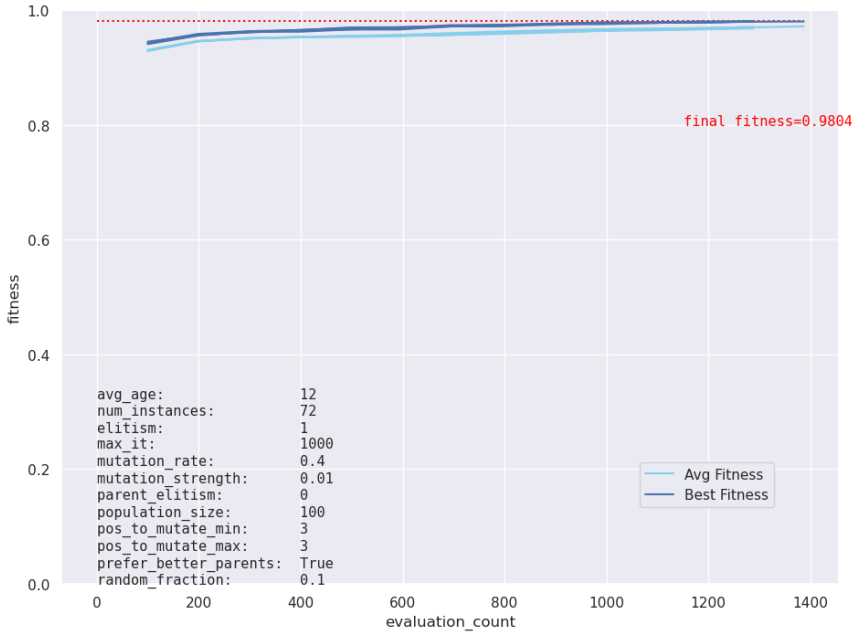


Figure 67: Sandbank: Low number of mutated positions, low mutation strength, high mutation rate

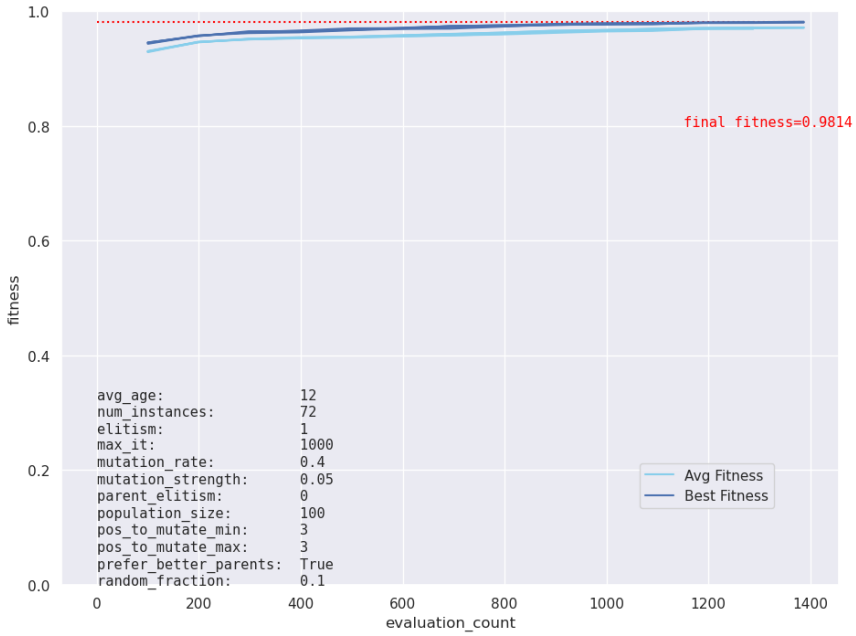


Figure 68: Sandbank: Low number of mutated positions, medium mutation strength, high mutation rate

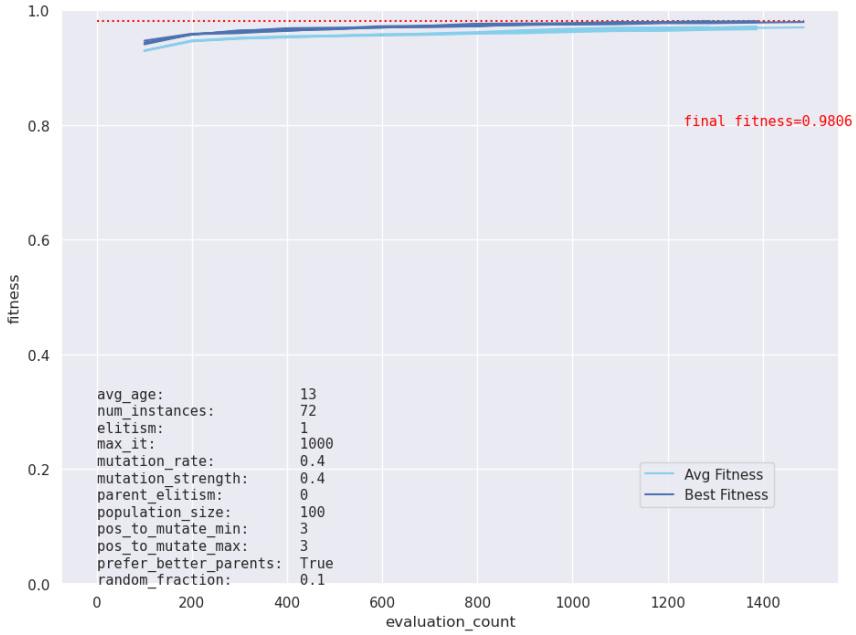


Figure 69: Sandbank: Low number of mutated positions, high mutation strength, high mutation rate

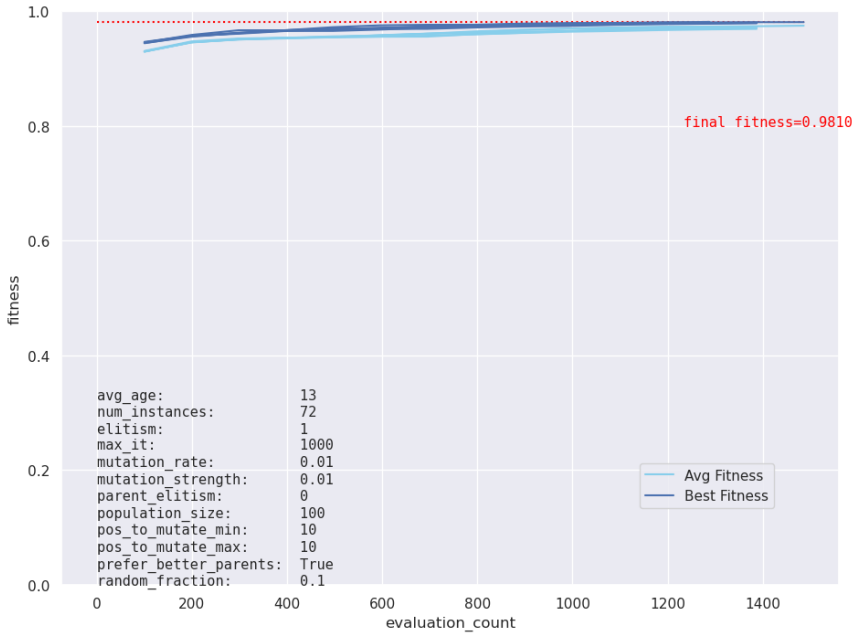


Figure 70: Sandbank: Medium number of mutated positions, low mutation strength, low mutation rate

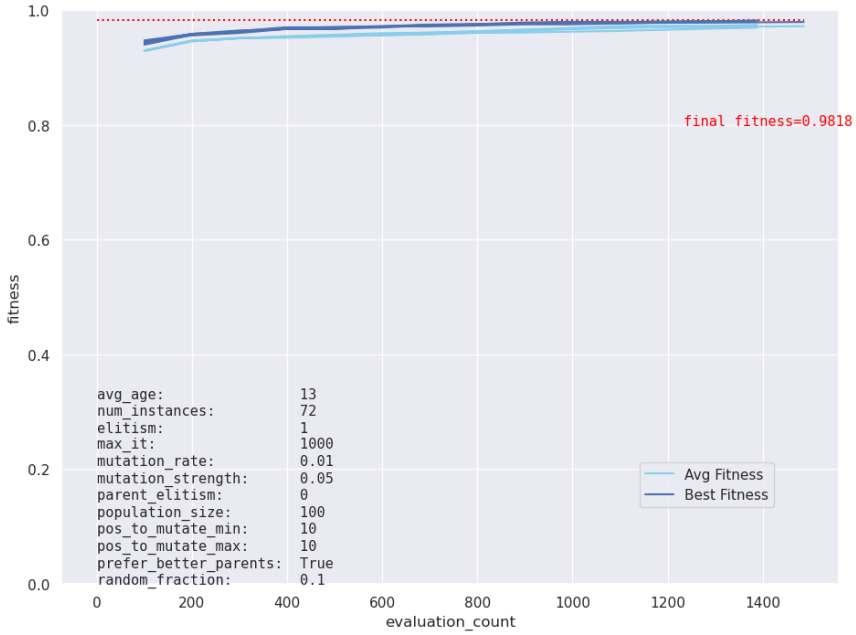


Figure 71: Sandbank: Medium number of mutated positions, medium mutation strength, low mutation rate

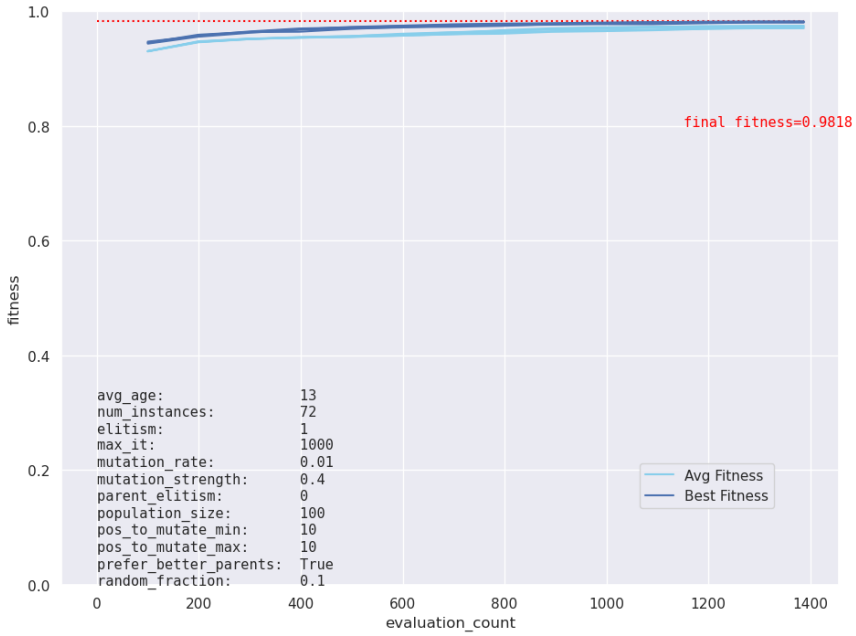


Figure 72: Sandbank: Medium number of mutated positions, high mutation strength, low mutation rate

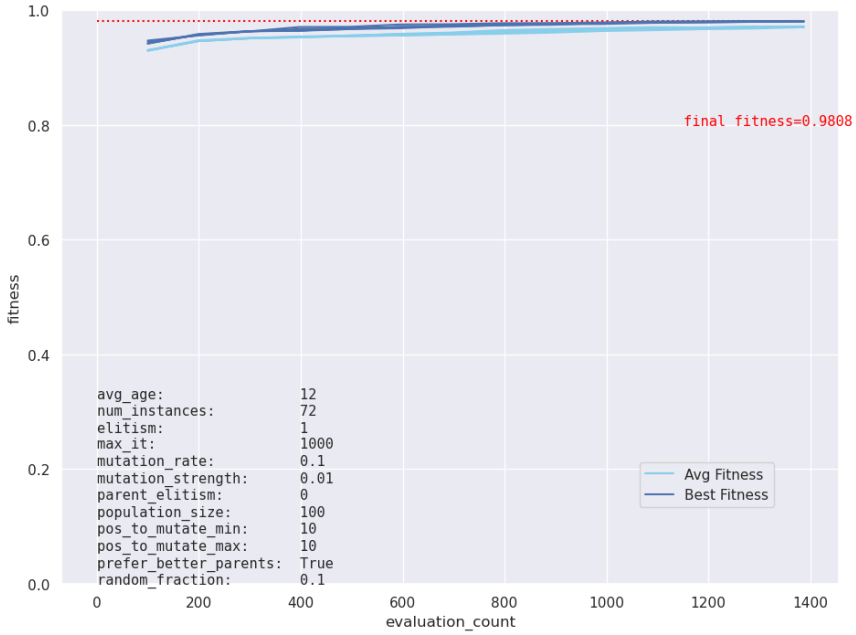


Figure 73: Sandbank: Medium number of mutated positions, low mutation strength, medium mutation rate

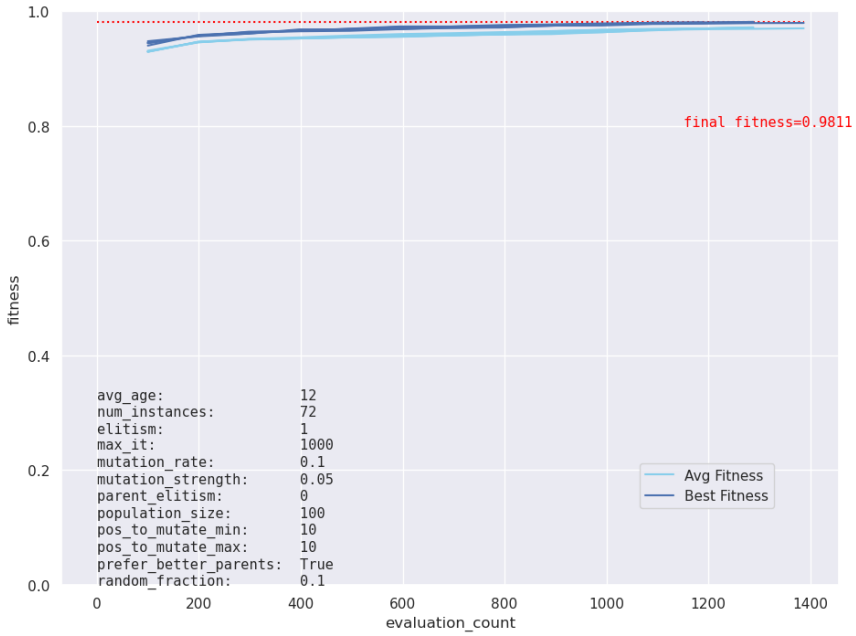


Figure 74: Sandbank: Medium number of mutated positions, medium mutation strength, medium mutation rate

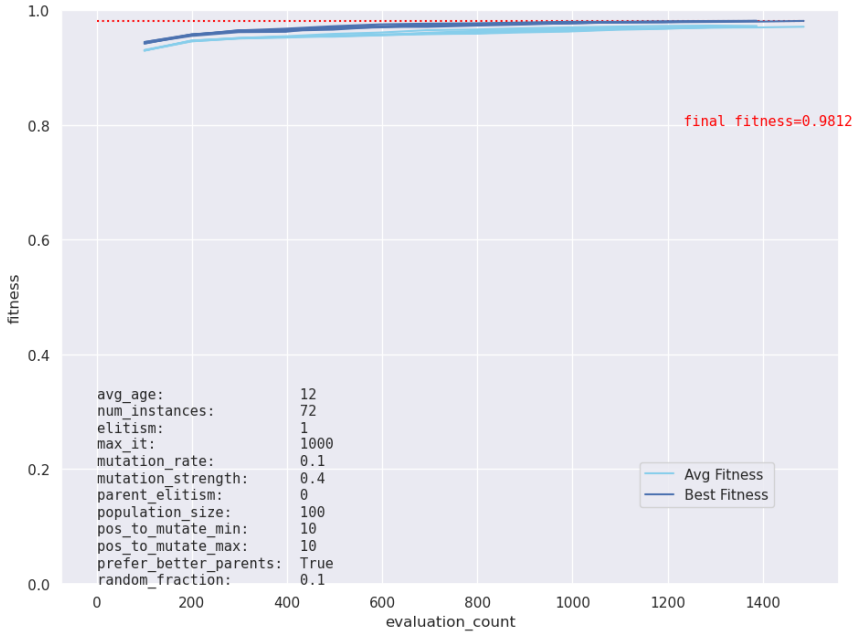


Figure 75: Sandbank: Medium number of mutated positions, high mutation strength, medium mutation rate

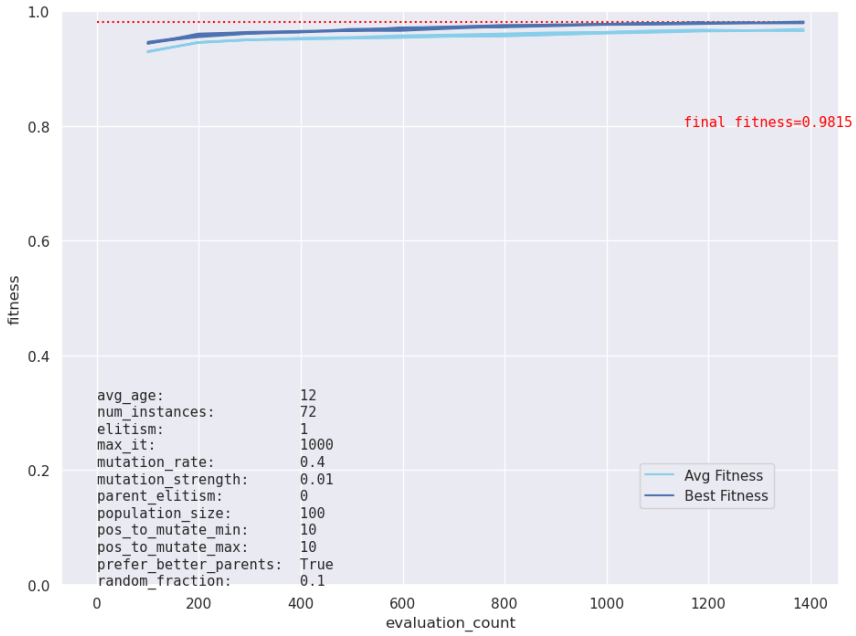


Figure 76: Sandbank: Medium number of mutated positions, low mutation strength, high mutation rate

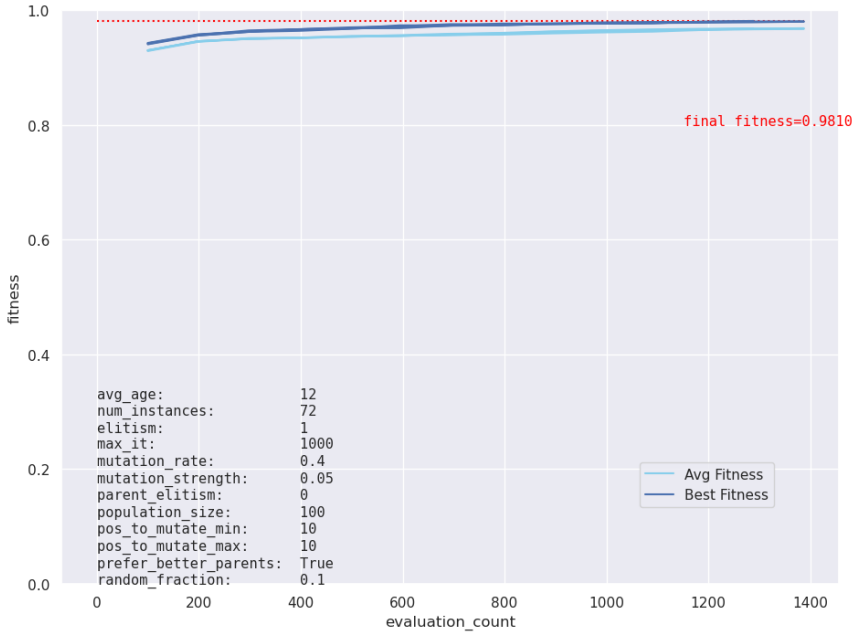


Figure 77: Sandbank: Medium number of mutated positions, medium mutation strength, high mutation rate

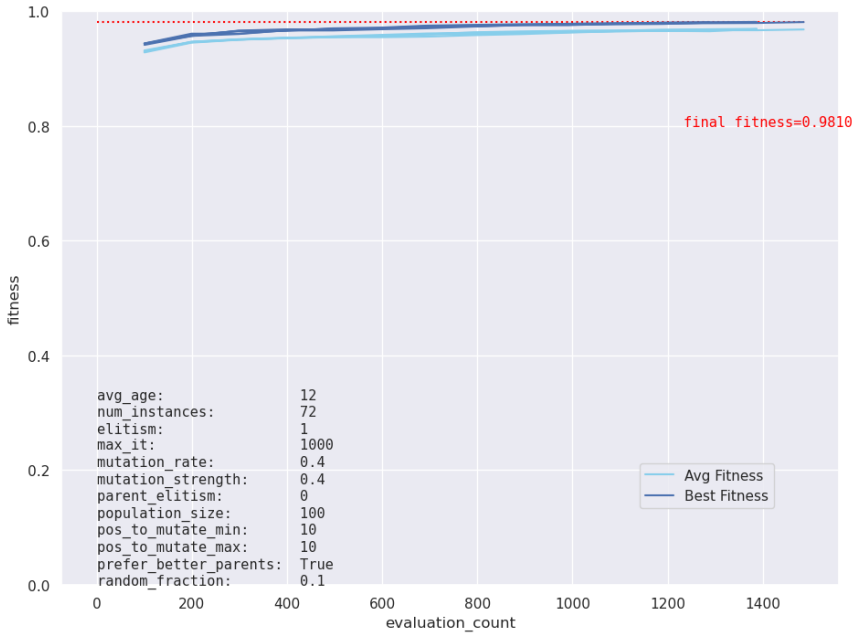


Figure 78: Sandbank: Medium number of mutated positions, high mutation strength, high mutation rate

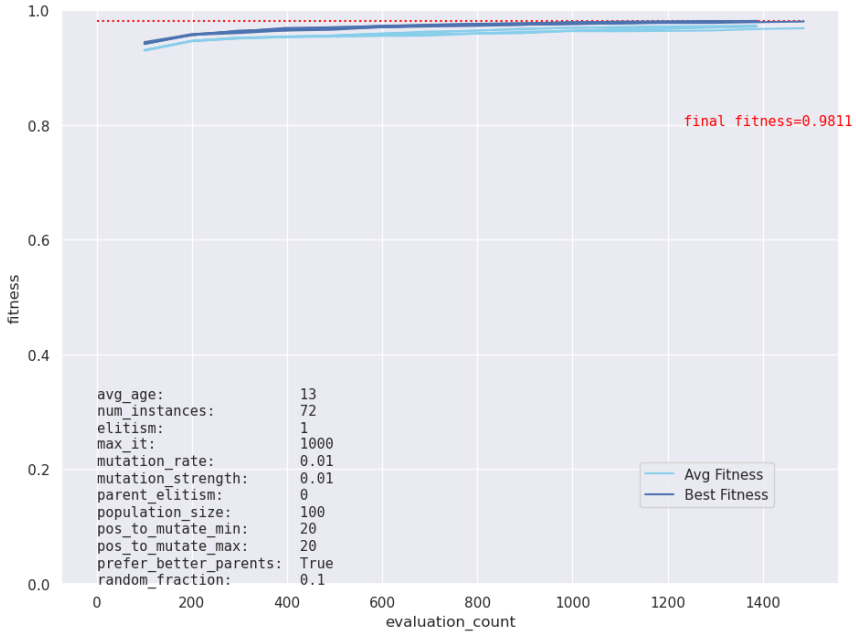


Figure 79: Sandbank: High number of mutated positions, low mutation strength, low mutation rate

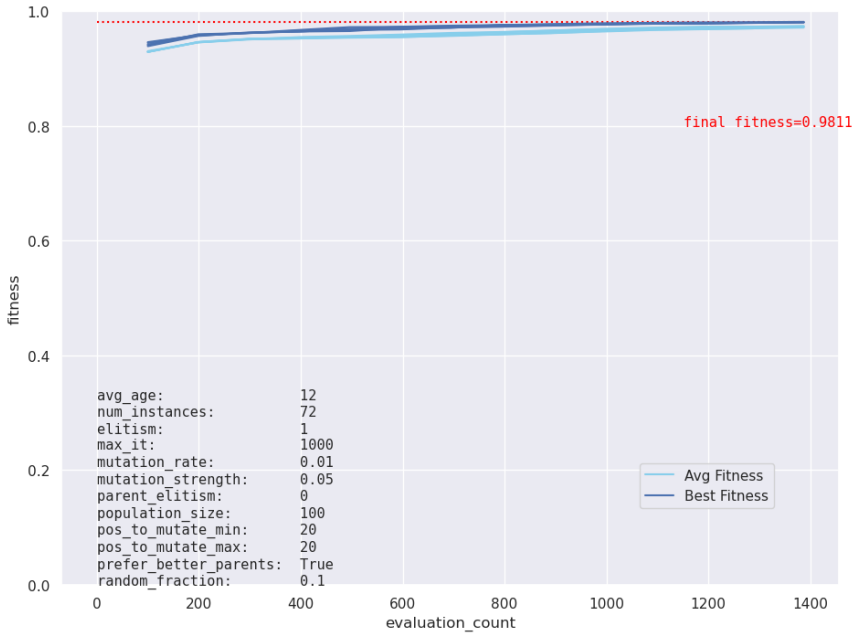


Figure 80: Sandbank: High number of mutated positions, medium mutation strength, low mutation rate

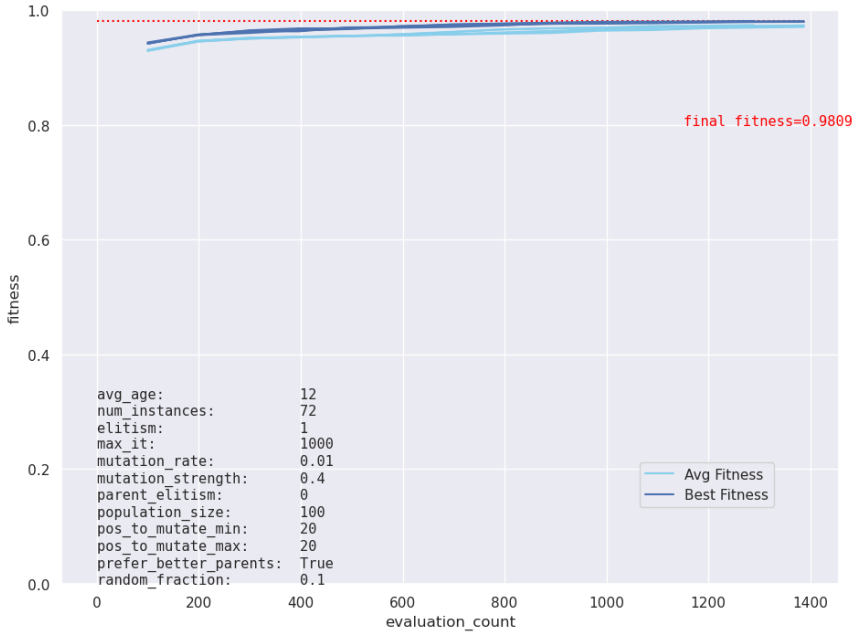


Figure 81: Sandbank: High number of mutated positions, high mutation strength, low mutation rate

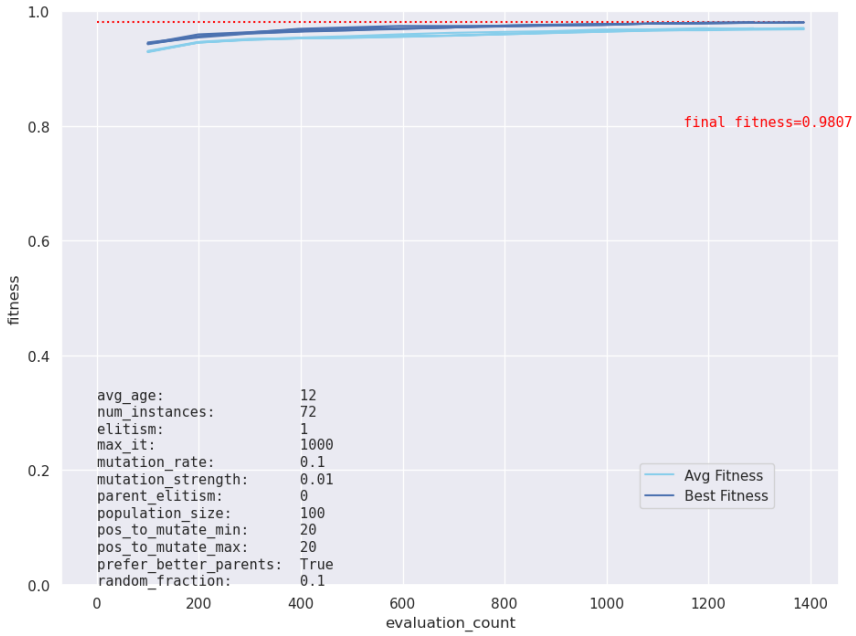


Figure 82: Sandbank: High number of mutated positions, low mutation strength, medium mutation rate

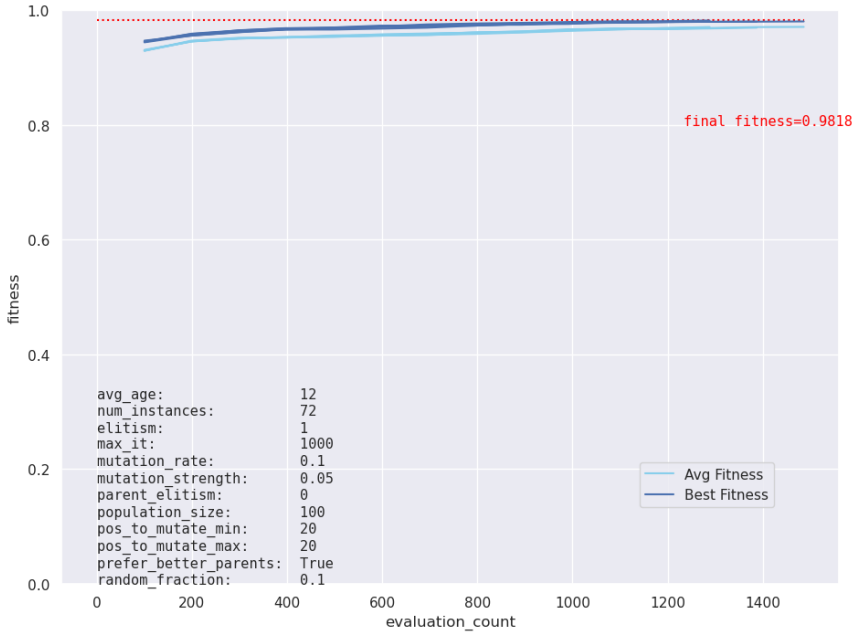


Figure 83: Sandbank: High number of mutated positions, medium mutation strength, medium mutation rate

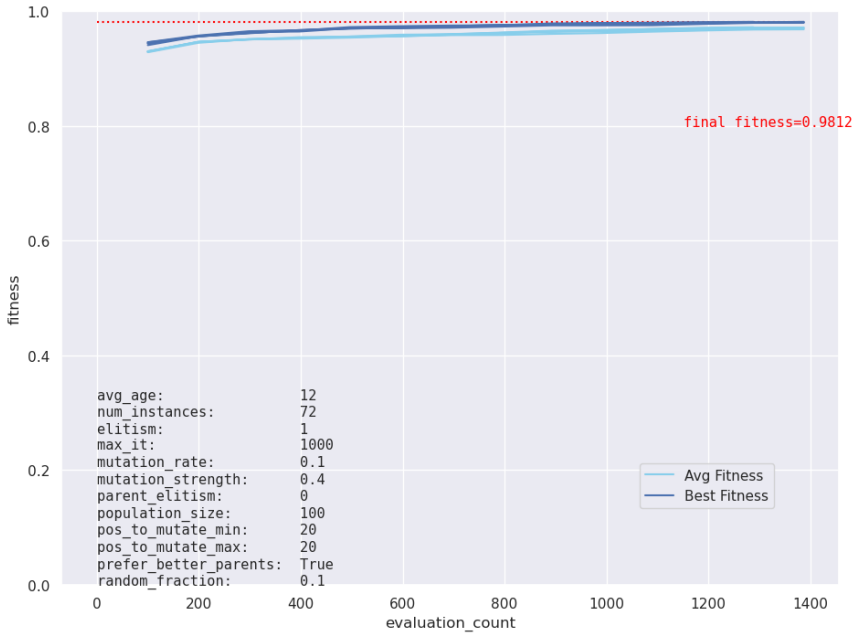


Figure 84: Sandbank: High number of mutated positions, high mutation strength, medium mutation rate

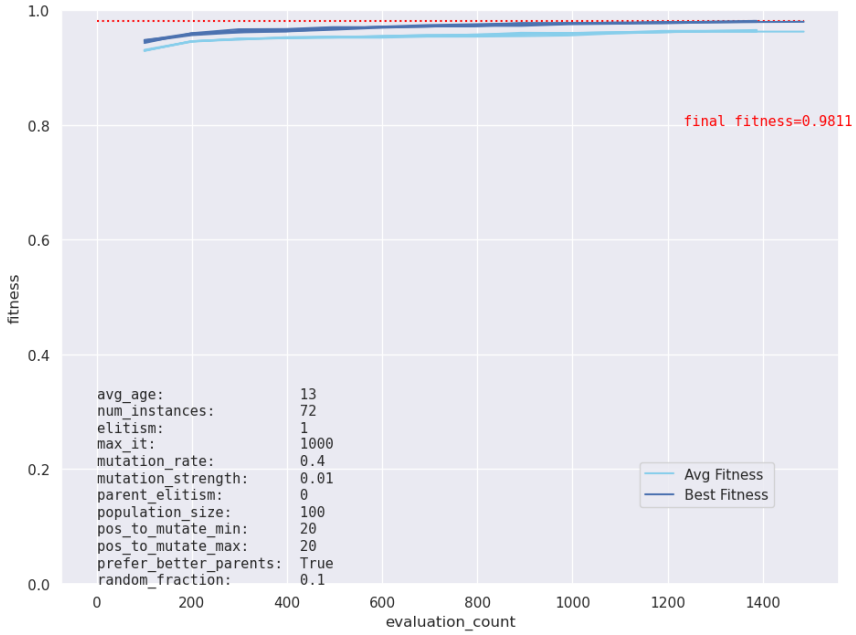


Figure 85: Sandbank: High number of mutated positions, low mutation strength, high mutation rate

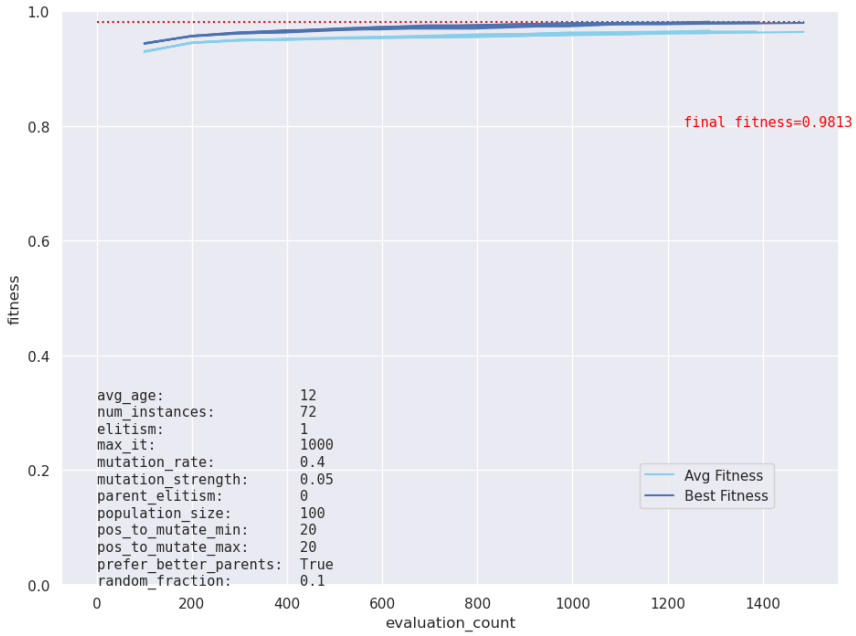


Figure 86: Sandbank: High number of mutated positions, medium mutation strength, high mutation rate

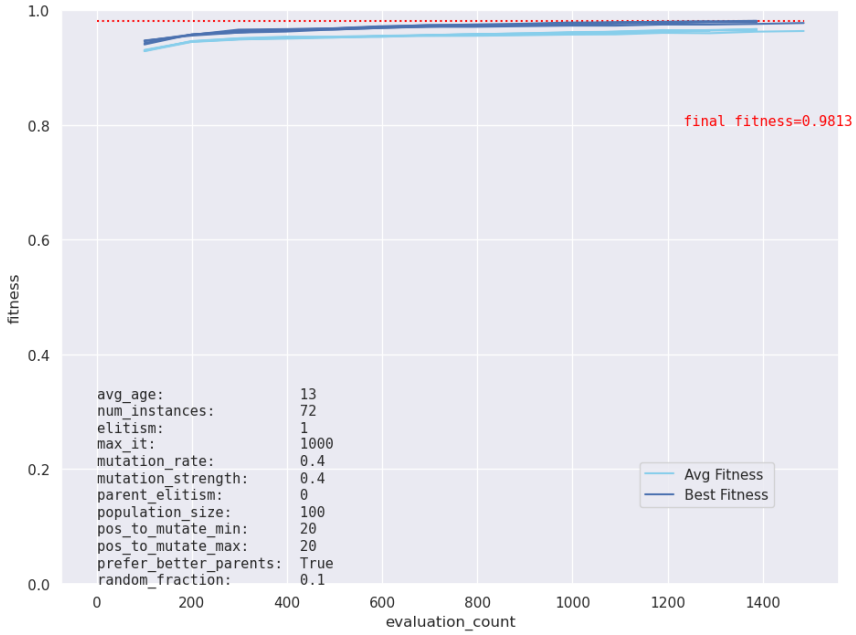


Figure 87: Sandbank: High number of mutated positions, high mutation strength, high mutation rate

A.2.8. Required Amount of Elitism

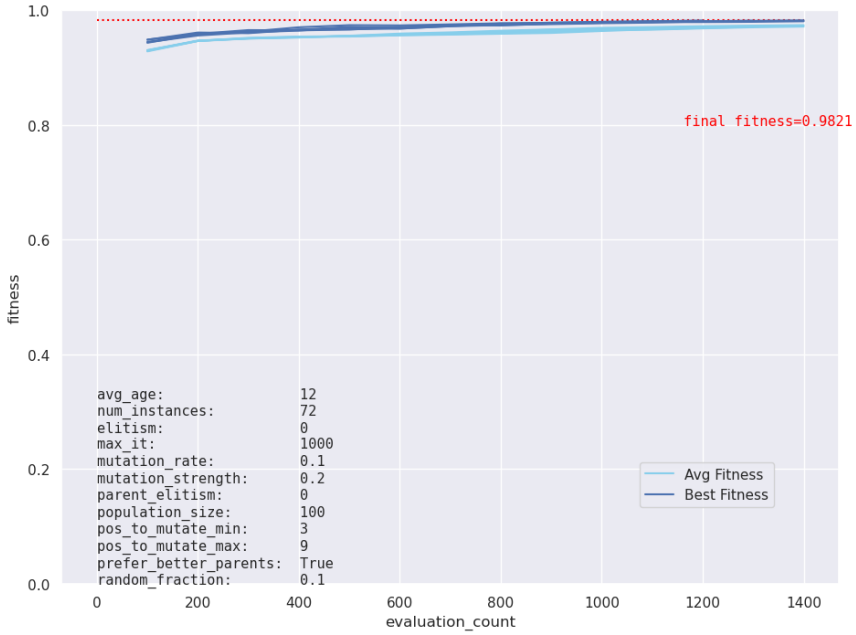


Figure 88: Sandbank: No elitism

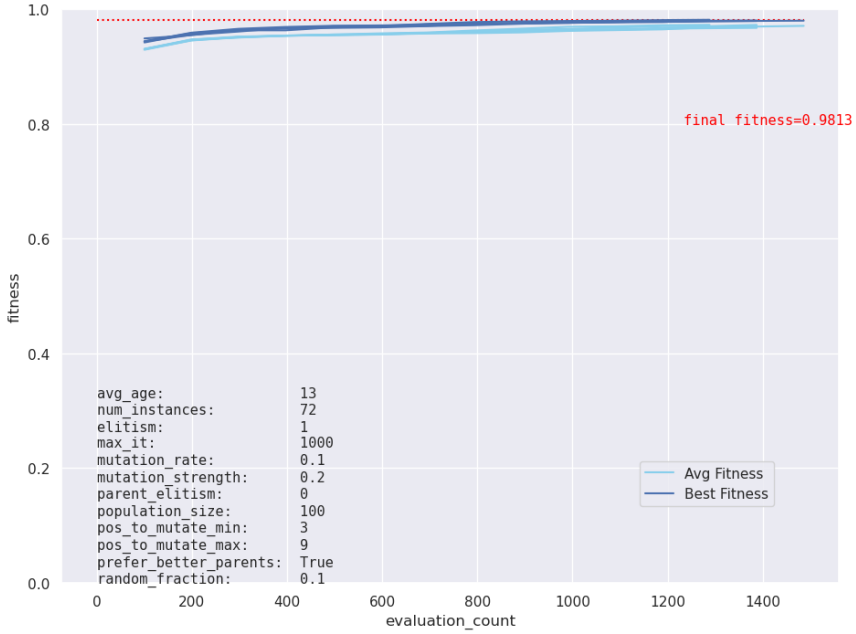


Figure 89: Sandbank: One elite

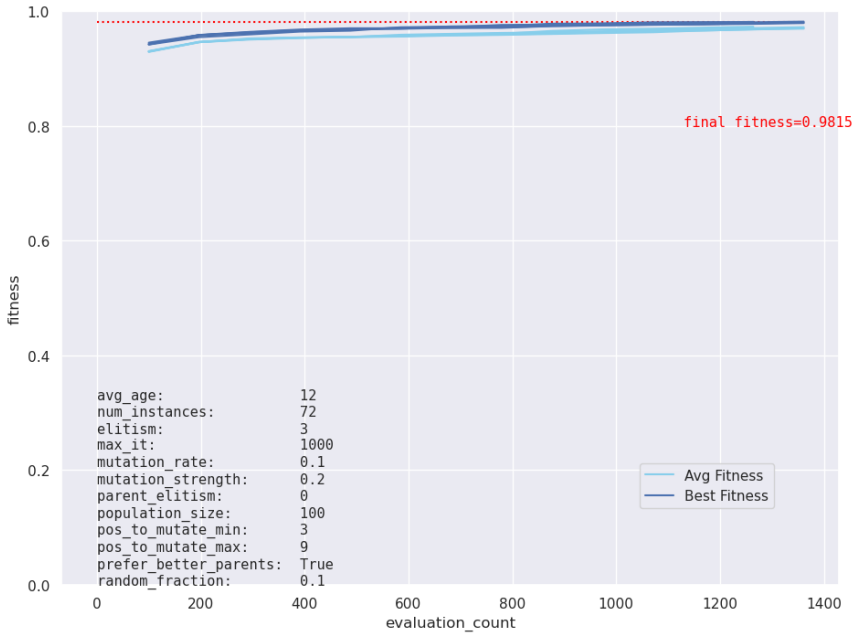


Figure 90: Sandbank: Three elites

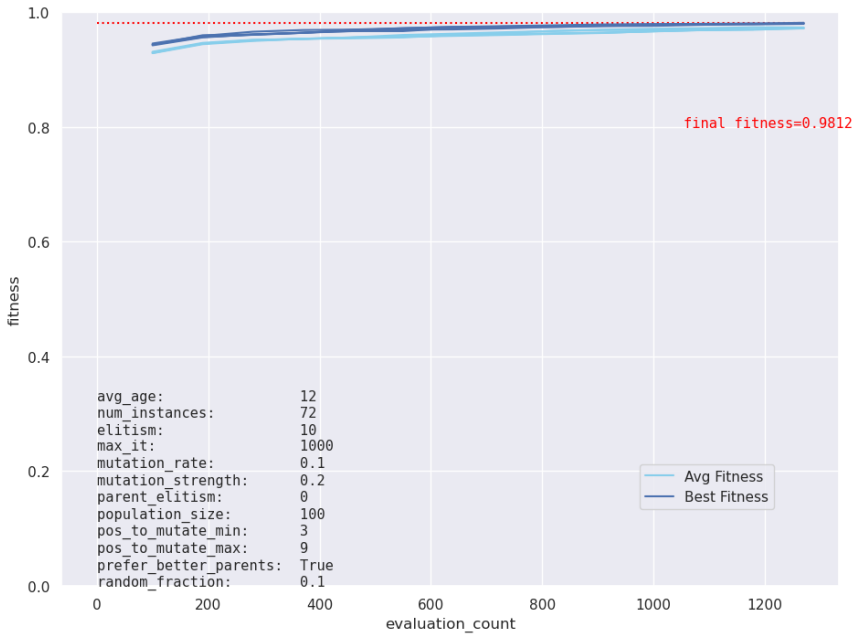


Figure 91: Sandbank: Ten elites