Genetic Algorithms and Image Reconstruction: how different recombination mechanisms impact algorithmic performance

Panagiotis Provias - 13191589

Supervisor: George Magoulas

MSc Computer Science

Project report

Department of Computer Science and Information Systems,

Birkbeck College, University of London

2022

This report is substantially the result of my own work, expressed in my own words, except where explicitly indicated in the text. I have read and understood the sections on plagiarism in the Programme Handbook and the College web site. I give my permission for it to be submitted to the JISC Plagiarism Detection Service.

The report may be freely copied and distributed provided the source is explicitly acknowledged.

**Abstract**

A search-based technique known as a genetic algorithm is used to address optimization issues in machine learning. The field of picture reconstruction is just one where these algorithms have been applied. To tackle the issue of image reconstruction, researchers have employed a wide variety of machine learning algorithms. Even those who have used genetic algorithms for this goal have employed various operators resulting in a wide range of outcomes. The goal of this project is to completely implement a genetic algorithm in Python for the purpose of reconstructing binary images. The effects of three different recombination, or crossover, methods –namely the single-point, double-point, and uniform crossover methods—will be analysed on algorithmic performance.

Contents

[1 Introduction 5](#_Toc113560231)

[2 Background Research 7](#_Toc113560232)

[2.1 Image reconstruction 7](#_Toc113560233)

[2.2 Machine Learning and Image Reconstruction 7](#_Toc113560234)

[2.3 Genetic Algorithms 8](#_Toc113560235)

[2.3.1 Individuals and Population 8](#_Toc113560236)

[2.3.2 Genetic Representation 9](#_Toc113560237)

[2.3.3 Fitness 9](#_Toc113560238)

[2.3.4 Genetic Algorithm Steps 9](#_Toc113560239)

[2.4 The Problem 10](#_Toc113560240)

[2.5 Aim and Objectives 11](#_Toc113560241)

[2.6 Risks and Mitigations 12](#_Toc113560242)

[3 Methods 13](#_Toc113560243)

[3.1 Objective 1: Design and Implement a Simulation Environment 13](#_Toc113560244)

[3.2 Objective 2: Algorithm Implementation 13](#_Toc113560245)

[3.3 Objective 3: Testing and Evaluation 14](#_Toc113560246)

[3.4 Objective 4: Improve Algorithm 14](#_Toc113560247)

[3.5 Samples from Literature 14](#_Toc113560248)

[4 Design 16](#_Toc113560249)

[4.1 Requirements 16](#_Toc113560250)

[4.1.1 Image Conversion 16](#_Toc113560251)

[4.1.2 Algorithm Implementation 16](#_Toc113560252)

[4.1.3 Implementation of Suggested Crossover Operators 17](#_Toc113560253)

[4.1.4 Data collection 17](#_Toc113560254)

[4.1.5 Graph/plot generation 17](#_Toc113560255)

[4.1.6 Statistical Analysis 17](#_Toc113560256)

[4.2 Development Methodology 17](#_Toc113560257)

[4.3 UML class and activity diagrams 18](#_Toc113560258)

[5 Implementation 22](#_Toc113560259)

[5.1 Tools, Libraries and Modules 22](#_Toc113560260)

[5.2 Parameter Configuration 23](#_Toc113560261)

[5.3 Operators 23](#_Toc113560262)

[5.3.1 The Genotype class 24](#_Toc113560263)

[5.3.2 The Elitism class 25](#_Toc113560264)

[5.3.3 The Fitness class 26](#_Toc113560265)

[5.3.4 The Selection class 27](#_Toc113560266)

[5.3.5 The Crossover class 28](#_Toc113560267)

[5.3.6 Additional Functionality 30](#_Toc113560268)

[5.4 Main Class 30](#_Toc113560269)

[6 Testing 32](#_Toc113560270)

[7 Experimental Study and Optimizations 33](#_Toc113560271)

[7.1 The Hypothesis 33](#_Toc113560272)

[7.2 The Experiments 33](#_Toc113560273)

[7.2.1 Generations 33](#_Toc113560274)

[7.2.2 Population 34](#_Toc113560275)

[7.2.3 Elitism 35](#_Toc113560276)

[7.2.4 Mutation 36](#_Toc113560277)

[7.3 Statistical Analysis 38](#_Toc113560278)

[7.4 Visual Results 39](#_Toc113560279)

[7.5 Optimisations 42](#_Toc113560280)

[8 Discussion and Conclusion 46](#_Toc113560281)

[9 References 46](#_Toc113560282)

[Appendix A – Source Code 46](#_Toc113560283)

[Appendix B – Test Reports 56](#_Toc113560284)

# 1 Introduction

Genetic Algorithms (GA) are adaptive search algorithms that imitate some of the concepts of Charles Darwin's theory of natural selection: heredity, fitness, selection, crossover, and mutation. These ideas from biological evolution are utilized by GAs and applied to software. Any kind of entity can benefit from applying these ideas (creatures, objects, strings, strategies etc.). These individuals have genetic sequences in common, which can range from more sophisticated sequencing—such as the human DNA with an alphabet encoding—to simpler sequencing, such as the bit sequence with binary encoding (J. Holland, 1975). GA frequently employ the latter in their calculations (Vie, Kleinnijenhuis, and Farmer, 2021).

GA are a type of machine learning (ML) technique that fall under the umbrella of reinforcement learning. They were initially created to investigate how natural habitats and ecosystems adapt (Holland, 1975). Since then, they have been changed and developed for optimization purposes (Whitley, 1994), and their applications extend across many disciplines: biology (Street and Mayo, 1999), forecasting (Ahn and Ramakrishna, 2003), data and computer science (Raymer et al., 2000, Sohail, 2021), deep learning and neural networks (Stanley et al., 2019, Chung and Shin, 2020), game theory (Vie, Kleinnijenhuis, and Farmer, 2021), image processing (Mirjalili et al., 2019), healthcare (Devarriya et al., 2020), economics (Waheeb and Ghazali, 2019), or finance (Han et al., 2019). GA are used in these fields to incrementally solve complex problems and provide solutions that improve over time.

For many years, GA have been utilized in image processing (Sheta, Braik and Aljahdali, 2012). Medical imaging (Wang et al., 2018), face recognition (Lézoray et al., 2008), retina scanning (Galbally et al., 2013), noise reduction in satellite imaging (Babb et al., 2008), reconstruction of corrupted images (Mirjalili et al., 2019), and license plate number localization are a few examples of these applications (Abo Smara and Khalefah, 2014).

Despite having several applications, GA confront significant difficulties. First, running these algorithms typically involves high computing costs. The algorithm's reliance on carefully adjusting the operating settings presents another significant difficulty. An algorithm that is not properly set may converge too slowly, not at all, or to a suboptimal solution (Katoch, Chauhan, and Kumar, 2020; Vie, Kleinnijenhuis, and Farmer, 2021). There are still many important open problems despite the successful application of GA to a variety of real-world issues. Some of these issues include:

* Selecting a suitable depiction (Ting et al., 2016).

• Determining the best algorithm to solve a particular problem or problem instance (O'Neill et al., 2010).

• How much of the natural evolution concept needs to be implemented in detail? How might the operators of an algorithm be affected by the abstraction and simplification of biological processes? (Banzhaf et al., 2006).

• GA generalization to enable widespread application by the scientific community (Naik and Dabhi, 2013).

• Globally adjusting a GA's operators to reduce complexity using a "golden ratio" (Shrestha and Mahmood, 2016).

The relationship between this field of research, and neural networks and deep learning is what makes it so intriguing. In the past few years, both fields have had tremendous success, and for good reason. Autonomous weapons, self-driving cars, Netflix show suggestions, picture reconstruction, and the recognition of images of plants and birds are just a few examples of applications for deep learning (Zendesk, 2020). The final one used ML algorithms for image reconstruction (IR). Deep learning computers continuously examine data using a logical framework similar to how people reason and make decisions. These algorithms build an artificial neural network that resembles the network of neurons in the human brain using many layers of algorithms. This kind of learning system is regarded as the foundation of artificial intelligence and is more effective than conventional ML models. Artificial intelligence is seen as the next "industrial revolution," despite its numerous critics (Wang and Siau, 2019). In this research, we will investigate GA, one of the numerous building elements that contribute to the layered algorithms of deep learning.

We outline the background research in the section that follows. The project's aim and objectives are presented in Section 3. Section 4 will present the design behind this project. Implementation details can be found in Section 5, while details on the testing techniques used can be found in Section 6. Section 7 presents the logic behind the decisions made on the tuning of the algorithm, the experiments ran, the statistical analysis of the results, plus any steps to optimize the produced results and the algorithm itself.

# 2 Background Research

## 2.1 Image reconstruction

Reconstructing an original, clear image from a corrupted one is referred to as image reconstruction or image restoration. These imperfections can be in the form of noise, motion blur, or low resolution. Due to the vast range of applications for IR, the scientific community began paying close attention to it as early as the 1950s. American scientists' attempts to create images of the Earth using astronomical imaging led to the beginning of the field's research. Image processing and reconstruction remain crucial to modern satellite imaging (Babb et al., 2008). The developed and employed IR techniques have also seen extensive use in other fields. X-ray images, angiograms, and MRI scans are just a few examples of the types of medical imaging where IR is crucial. However, these still need to be improved because of the inverse relationship between their quality and image resolution (Forghani et al., 2020). Additionally, by eliminating scratches from old movies and photographs, IR can produce images of superior quality (Yadav, and Khatak, 2017). The field of image/video coding is yet another application. The scope of this application ranges from oceanography to regular use of personal mobile phone cameras. Camera misfocusing, which can result in significant quality degradation and cannot be repaired by the most popular reconstruction techniques, is a significant problem here (Khare, and Nagwanshi, 2011; Fu, Zhang, and Li, 2021). Face recognition is another application that many people utilize daily. IR plays a vital role in a variety of applications, including access control in online banking, border control, and personal devices (such unlocking the phone's home screen). Despite the convenience these applications offer, they are vulnerable to cyberattacks that use "template invertibility" and reconstruction techniques (Mignon and Jurie, 2013; Nestor et al., 2020). It is also important to mention 3D IR. This method is used in a variety of industries, including medicine, urban planning, robotics, entertainment, computer graphics, augmented reality, artificial intelligence, and animation. Again, despite the advantages we experience in these domains, there are problems, including deformation and the requirement for sensors, that make these applications expensive to operate and prone to errors (Khilar, Chitrakala and SelvamParvathy, 2013).

## 2.2 Machine Learning and Image Reconstruction

To reconstruct images, various Machine Learning methods have been employed. Compared to conventional IR techniques, the former can produce results of superior quality (Hammernik and Knoll, 2020). Deep learning is one ML method that has gained popularity in IR as a new tool for image classification. Despite its benefits, deep learning also faces significant obstacles and constraints (Antun et al., 2020). According to Wang, Ye, and De Man's (2020) research, minute, invisible to the human eye disturbances can cause a reconstructed picture to be incorrectly classified, such as when a cat-shaped reconstruction is mistaken for a car. According to De Haan et al. (2019), deep learning IR in medical imaging can result in minor structural changes that can lead to errors in diagnosis, such as the failure to detect tiny tumors. The mathematical method of performing predictive analysis is known as linear regression, and it is another ML technique employed in IR. It has grown in prominence in IR facial recognition during the past ten years (Naseem, Togneri and Bennamoun, 2010). A powerful technique for examining the connections between various variables is linear regression, which is very simple to use and evaluate the results of. However, because it tends to oversimplify variable interactions by assuming they are linear, its application to real-world issues can ultimately be problematic (Zhu, Zhu, and Li, 2018). Another ML method that is primarily utilized in the field of medicine is clustering. When compared to other methods, it has the advantages of being simple to build, easy to understand, and faster, especially when using a K-means clustering (Zheng et al., 2018; Darma et al., 2019). GA is a reinforcement learning method that has been widely applied in IR. To find suboptimal solutions and eventually arrive at the global optimum, heuristic search strategies like GA are the best since they effectively exploit data and deal with numerous points on the problem space. As was previously indicated, despite their efficacy, GA face harsh criticism because of their intrinsic reliance on parameter adjustment and high computational cost (Piwonska and Grycuk, 2007). Compared to its 3D equivalents, 2D IR appears to produce the greatest results, whether in black-and-white or RBG photos, mostly because the former consumes more power (Abo Smara and Khalefah, 2014; Abouhawwash and Alessio, 2021).

## 2.3 Genetic Algorithms

The fundamental ideas behind GA are covered in this section. These ideas will explain how the algorithm works. We will clarify key terms like individuals and population, genetic representation, fitness, and give a quick overview of the GA steps.

### 2.3.1 Individuals and Population

Algorithms for computational search called GA use the ideas of natural selection. Individuals that make up a population are subject to the Darwinian principles of evolution (Vie, Kleinnijenhuis, and Farmer, 2021). These individuals are impacted by evolution, and succeeding iterations of the population, referred to as generations, evolve to achieve maximum fitness, i.e., the capacity to survive in a certain environment, carry out a particular activity, or maximize performance metrics. These individuals need a genetic representation for this evolution process to be possible (Katoch, Chauhan, and Kumar, 2020).

### 2.3.2 Genetic Representation

Instead of using mathematical functions or abstract concepts, GA instantiate individuals and populations in a manner comparable to genetics. Since human DNA contains the bases A, C, T, and G, it can be encoded as a series of letters (human DNA), numbers (binary encoding using 0s and 1s), or both (hexadecimal encoding). Characteristics are encoded in this string of characters. The process through which the traits and actions of individuals are represented or encoded in a computer program is called encoding. The terms genotype and phenotype, which are taken from genetics, are used in GA too (Ritchie et al., 2015).

Text

Description automatically generated with medium confidence

*Figure 1. Different genetic represenations that can be used in GAs.*

### 2.3.3 Fitness

Due to an advantage, natural selection and evolution favour those individuals that are better at reproducing. Although the fitness criteria in natural evolution may be difficult to evaluate, in GA a fitness function may be utilized to gauge how well each individual performs in terms of environment adaptation or task completion. These operations could be as simple as counting the number of times a letter or number appears in the genotype of the person (Katoch, Chauhan, and Kumar, 2020).

### 2.3.4 Genetic Algorithm Steps

The majority of GA follow five key steps. The first phase is to create the original population, or the first generation, of individuals. The following four steps are then repeated iteratively: assess the fitness of each individual of the generation, choose the parents of the generation, use, and manipulate the genetic information of the parents to produce offspring, and finally introduce random mutations to the genetic data of the offspring. The next generation enters the population following these four processes. We repeat the algorithm's phases until we accomplish a certain goal or for a predetermined number of generations (Abraham, Nedjah and Mourelle, 2006).

Graphical user interface, text, application

Description automatically generated

*Figure 2. The steps of a basic GA.*

The population can explore various genotypes/phenotypes in relation to the fitness criterion given at the beginning. Now is a good time to mention another operator that is linked to crossover and mutation. A part of the top-performing genotypes can be retained and passed down to the following generation thanks to elitism operator. By doing this, the changes brought about by crossover and mutation throughout the development of the next generation are prevented from losing or damaging effective solutions. The earliest GA did not include elitism, but since GA performance has improved subsequently due to the elitism operator, researchers have begun to use it (Chang Wook and Ramakrishna, 2003; Pradhan et al., 2017).

Numerous mechanisms for the biological processes of selection, crossover/recombination, and mutation are mentioned in the literature (Mirjalili et al., 2019). Recombination is an essential component of a GA because, if the genetic information from each generation is not adequately transmitted, it can "break" the simulation. Fit solutions will therefore be lost and destroyed, never achieving a global optimum (Yu et al., 2019). The most widely employed recombination mechanisms in GA are single- and multi-point crossover (Petr Skoda and Adam, 2020). Another technique that has been widely employed in order to produce a variety of offspring is the uniform crossover (Shi, Zhou, and Chen, 2013).

## 2.4 The Problem

Many recombination methods have been used by researchers to carry out IR experiments employing GA. The majority has adopted the multi-point crossover; however, some have also attempted to implement their own methods. Even though it takes a lot of effort, this can satisfy specific algorithmic requirements.

For their tests, Pal, Bhandari, and Kundu (1994) investigated using both single- and multi-point recombination. The single-point technique, however, has produced subpar results because the algorithm was unable to achieve convergence in an acceptable period of time for the large (thousands of bits per individual) images they attempted to recreate. However, they got noticeably superior outcomes by utilizing multi-point crossover. Results from Nakata and Ogawa (1995) were comparable. For their binary IR tests, Franconi and Jennison (1997) used single-, multi-, and block crossover. They have used the block approach because their earlier studies with the other two methods had poor results and none of them had demonstrated a clear advantage over the others. But neither did their own approach outperform the other two in terms of computational efficiency. They came to their conclusions based on the challenges these algorithms present for parameter tweaking.

Single-point crossover has been shown to perform poorly in IR, even in more recent years. Single- and double-point recombination were employed by Piqonska and Grycuk (2007). They discovered that while the mutation operator had no major impact on their findings, the crossover operator did. They attempted to rebuild a 20\*20-pixel image in their rather straightforward experiment. The algorithm needs 400 generations to determine the individual with the best fitness value using the double-point method. After around 100 generations, the algorithm could not enhance the best individual using the single-point method, indicating that it would always arrive to local optima. However, the single-point crossover technique has been used successfully by other researchers (Iga and Wakahara, 2004) to rebuild an image. Using a mix of uniform and multi-point crossover called binary crossover, Abouhawwash and Alessio (2021) have had satisfactory results. They had more freedom to conduct their studies thanks to this method, although algorithmic performance was significantly impacted by parameter tuning.

The multi-point crossover method has been chosen by most researchers because it allows for a more thorough search of the problem space. Even though the uniform recombination method is common in GA and offers the most disruption, allowing for the widest exploration of the problem space in fewer generations (Whitley and Sutton, 2012), it hasn't been widely employed for IR.

## 2.5 Aim and Objectives

The objective of this project is to investigate how a GA performance in IR, specifically the reconstruction of a binary image, is affected by the uniform recombination mechanism. To compare the outcomes of each crossover mechanism, we will also use the single- and double-point crossover methods.

This project will need to meet the following objectives:

1. Design and implement a simulation environment that would allow us to set the parameters for the algorithm and input an original image that would need to be reconstructed by the GA.
2. Design and implement a GA that will iteratively try to reconstruct a binary image from a randomly generated binary image. It will also provide us with output of the reconstructed images per a predefined number of generations.
3. Conduct testing and evaluation of the results.
4. Improve algorithm.

## 2.6 Risks and Mitigations

|  |  |  |  |
| --- | --- | --- | --- |
| **Risk** | **Probability** | **Mitigation** | **Impact** |
| Unable to implement the algorithm as a whole. | Medium | Use third party, open-source libraries that support genetic algorithms in Python. | High |
| Unable to implement all three recombination mechanisms. | Low | Scale back to using the simplest mechanism (single-point crossover). | Medium |
| Unable to run simulations on local machine, due to the algorithm’s nature (computational cost). | Low | Use the department’s machines/ scale down on requirements. | High |
| Difficulty tuning parameters to run the algorithm, which would result in unreliable solutions. | Medium | Borrow parameter tuning from existing algorithms available from other researchers. | High |

*Table 1. Risks facing in this project and mitigations to tackle them.*

# 3 Methods

Python and C++ are the two programming languages that are used to develop GAs that are most frequently discussed in the literature. Although C++ would be a good contender for this project because of its computational efficiency and the fact that GAs are computationally intensive processes (Eyal Wirsansky, 2020), the idea was soon shelved. C++ is more difficult to grasp and use than Python. Additionally, Python is more effective than C++ in processing lists or strings, which are the most prevalent objects used in evolutionary algorithms to represent potential solutions (Wonjae Lee and Hak-Young Kim, 2005). Additionally, Python computations can be made faster by using numeric/accelerated libraries like NumPy (Numpy, 2009) or PyTorch (PyTorch, 2019), which can eliminate the computational gap between Python and C++ (Silva, Resende and Pardalos, 2013). There are third-party libraries for GA computations in both languages. GAlib and OpenGA are available in C++ (Mohammadi et al., 2017). Pyevolve, PyGAD, and DEAP are Python libraries that provide GA computations for Python. (Perone, 2009) (Wonjae Lee and Hak-Young Kim, 2005). However, we will create the algorithm from scratch for the purposes of this project and will not use any third-party libraries, unless it is necessary (see 2.6 Risks and Mitigations). The standard library for Python is extensive and well-documented. Python's capabilities as an object-oriented, dynamically typed programming language will make it simple to implement the algorithm in its entirety. Finally, this decision has also been influenced by Python's clean syntax, readability, and maintainability (Skorpil et al., 2019).

## 3.1 Objective 1: Design and Implement a Simulation Environment

All the methods that must be developed for this purpose will be created using Python. These comprise the algorithm's operators, and parameter tweaking, among others. The NumPy library will be used since these techniques strongly rely on random number creation. For the creation of the random binary image, we will again use the OpenCV and PIL modules. These modules are used to carry out computer vision tasks and process images. (Howse, 2013). The testing required for this objective will be the most difficult part. Before performing unit testing on the individual components, we will first check that the algorithm's logic functions as intended. The desired result would be the successful passing of all the functions and the proper parameter setup to the algorithm in Objective 2. Also, manual testing will be used for the generated images.

## 3.2 Objective 2: Algorithm Implementation

As with the previous objective, we will once more achieve this goal using Python and NumPy. As stated, Python will be used to build the method because it is simpler and faster to do so than C++. This will enable us to concentrate on reviewing the findings, adjusting, and testing out various parameter tunings. We will utilize the Python Matplotlib library for the graphs, and if that doesn't work. We must succeed in the following in order to fulfil this objective:

* The algorithm can accept a variety of parameter values, including those for the crossover method, the maximum number of generations, and the terminal condition.
* After each generation, the application must be able to show the results or graphs or at the very least, save them.

## 3.3 Objective 3: Testing and Evaluation

We will run simulations with the same parameter settings. The algorithm's performance will be assessed by considering the maximum number of generations or the amount of time required to run it until it reaches the terminal state, either by finding a solution or completing a predetermined number of generations. Also, average maximum fitness and runtime will be compared.

## 3.4 Objective 4: Improve Algorithm

After getting the preliminary results, boosting the algorithm's efficiency will be the next goal. If the outcomes are satisfactory, we hope to apply the technique to other challenging issues, like binary images of higher resolutions. If the results are noticeably worse, we will redefine the problem and adjust the parameter settings to produce noticeably better outcomes. For example, we might allow more individuals to be elites and pass their genes down to the following generation, ensuring that the best solutions are always present in the "gene pool."

## 3.5 Samples from Literature

Figure 3 demonstrates a simple illustration of how the algorithm is expected to work. This example is taken from the literature (Mirjalili et al., 2019).

Qr code

Description automatically generated

*Figure 3. Illustration of original binary image to be reconstructed, best performing individual (generation 0) and snapshots every 100 generations.*

The best performing individual produced by the algorithm is shown in Generation 0. Where the algorithm searches the problem space for candidates (solutions) with promising genotypes, there has been a noticeable improvement. According to Mirjalili, et al. (2019), the original image is thought to have 0 errors while the image at generation 0 had 81600 errors. After 500 generations, there are still 1530 errors. We used their work to describe how the algorithm should operate in our simulation and what would be the outcome because we anticipate comparable outcomes from our research.

# 4 Design

The first part of this section will present the system requirements and the development process followed to achieve the project aims. The second part will present diagrams to further establish the implementation of the algorithm developed and the dynamic between the classes and objects existing in the code.

## 4.1 Requirements

Information gathered in the previous sections of this report has brought to light the following requirements. These requirements will provide a better understanding of the core of this project, while minimizing the risk of diverting from the main objective.

### 4.1.1 Image Conversion

There are several operations that will involve image conversion/manipulation. Firstly, we will need to convert an original image to a binary image to be reconstructed. The input to the algorithm must be a binary array in order to manipulate the data using the appropriate libraries. Finally, a binary array will need to be converted back to a binary image and saved in memory.

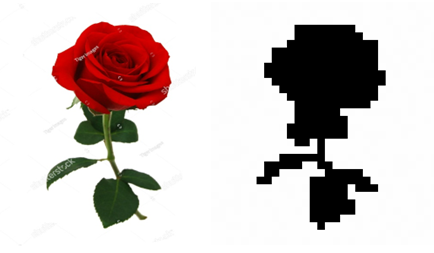


Figure 4. Left; Original image from web. Right; image after Photoshop modifications.

### 4.1.2 Algorithm Implementation

As mentioned in previous sections of this report, a genetic algorithm will be fully implemented. Even though, there are external libraries that support full functionality of genetic algorithms, it was considered that following this approach would to be an educational experience, but also, provide better understanding of the fine mechanics of these algorithms.

### 4.1.3 Implementation of Suggested Crossover Operators

Most genetic algorithms use one crossover method. Since it is the aim of this project to investigate the impact of the single-point, double-point, and uniform crossover methods on algorithmic performance, all three must be implemented.

### 4.1.4 Data collection

In order to test the hypothesis/aim of this project, data must be collected from each run of the algorithm.

### 4.1.5 Graph/plot generation

The creation of graphs and plots will assist the displaying of the results of this project. It will also help in the decision making regarding the tuning of the independent variables of the algorithm.

### 4.1.6 Statistical Analysis

The data collected must be statistically analysed to confirm whether the results are statistically significant. This will require some form of non-parametric testing.

## 4.2 Development Methodology

For this part of the Design section, inspiration was drawn from the Information Systems module as part of the MSc Computer Science, specifically from the textbook ‘System Analysis and Design with UML’ by Dennis, Wixom and Tegarden. An agile methodology will be used for the development of this project. Although the work will be carried out by a single developer, the benefits of an iterative development approach are still tangible (Dennis et al., 2015). The main reasons for choosing this approach are:

* Software is delivered early and continuously during the development process,
* Changing requirements are embraced regardless of when they occur, and
* Avoiding unnecessary work (compared to other methodologies).

For the management of the workflow and task tracking, Trello was used. The Trello board contained all the tasks that needed to be completed throughout the project in the form of tickets. Each ticket would have a label and a short description. Even though, some of these actions might have been redundant in a one-person team, it is best to adopt good design practice from an early stage. The following figures show how Trello, and its functionalities were used.

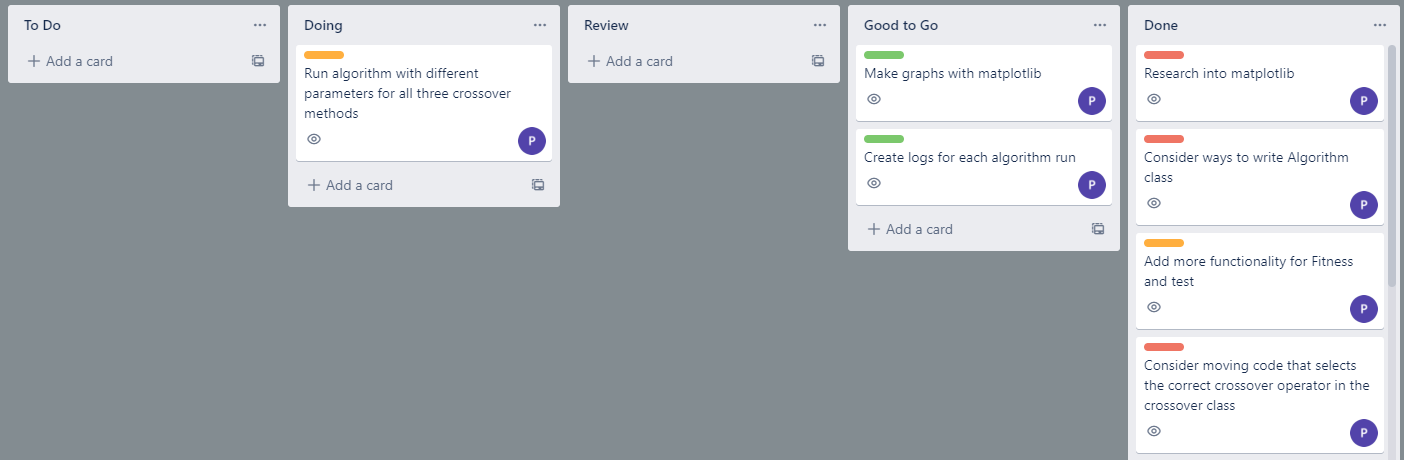


Figure 5. A snapshot of the Trello board during the development process.

A picture containing graphical user interface

Description automatically generated

Figure 6. Labels that were used for categorising tickets.

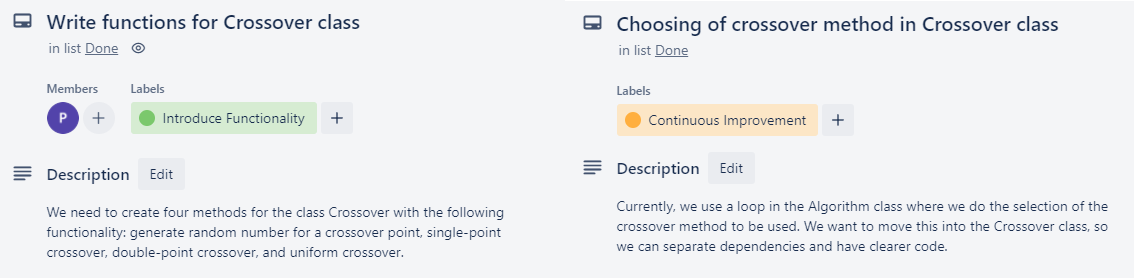


Figure 7. An example of how changing requirements were introduced in the project using the appropriate tickets.

The development process was organised in biweekly sprints with a retrospective meeting every second Thursday evening. During these meetings, notes were checked on what went right, what could have gone better (e.g., saving time), and setting the course, tasks and aim for the following sprint.

## 4.3 UML class and activity diagrams

In this section, the Unified Modeling Language (UML) will be used to depict the relationship between the classes/objects of the algorithm, as well as an activity diagram including all the steps of a single run of the algorithm. At this point it is worth mentioning that each algorithmic run consists of three sub-runs, one for each crossover operator.

As mentioned in the previous section, due to adopting an iterative (Agile) methodology for the development of this project, changes throughout the process are welcome and must be accommodated. Inevitably, many class diagrams have been sketched. In the following figures, the initial and the last class diagrams can be seen.

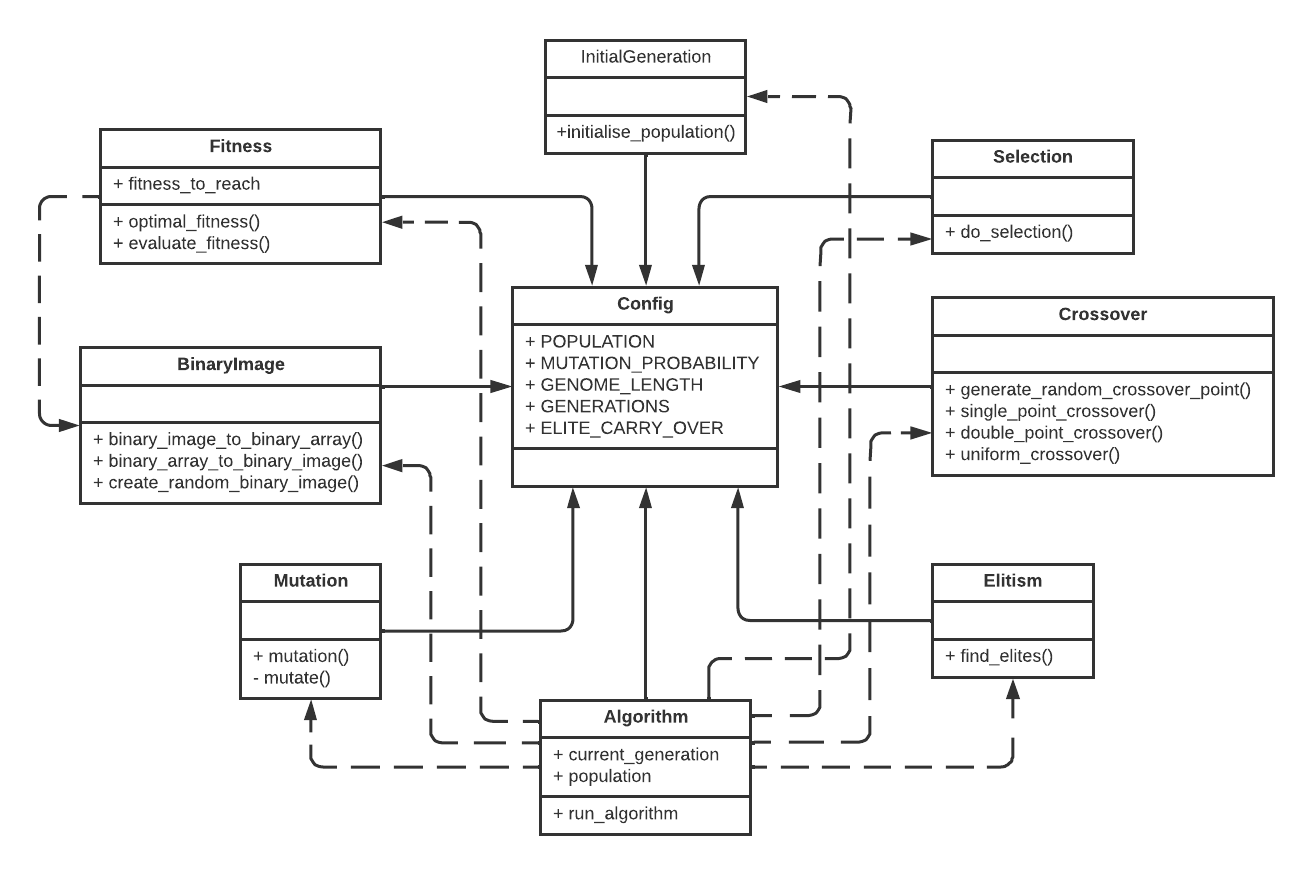


Figure 8. Initial UML class diagram.

**Diagram

Description automatically generated**

Figure 9. Final UML class diagram.

From these figures it is evident that new classes have been introduced during the development process, other classes were changed, while a few classes were completely rebranded or absorbed into existing classes. This confirms that an Agile approach compliments the completion of this project.

As genetic algorithms can be complex and include many steps, the use of an activity diagram to augment the understanding of the processes included seemed reasonable. Moreover, as in most cases, the proposed algorithm contains steps and conditionals that are not always present in a genetic algorithm. Lastly, many researchers have used them for the depiction of their implementations, therefore it seemed fit to follow along those guidelines.

Diagram

Description automatically generated

Figure 10. The activity diagram depicting the steps/logic of the suggested genetic algorithm.

# 5 Implementation

## 5.1 Tools, Libraries and Modules

As mentioned in section *3. Methods*, the programming language used for the implementation of the project has been Python. All work was carried out using PyCharm as the Integrated Development Environment (IDE). Another tool that was used extensively was Photoshop (CS6 v13.0.1 x64). Photoshop was used to reformat, reshape, and turn images from the web into black and white only images (the result can be seen in Figure 4). Also, the bit depth of the images to be reconstructed was lowered to 8. This decision lowers the runtime of the algorithm significantly because each gene would be represented as a single 0 or 1, and not a sub list, e.g., [0, 0] for 16-bit- or [0, 0, 0] for 24-bit-depth images.

Timeline

Description automatically generated with medium confidence

Figure 11. Project structure on PyCharm.

To decide which libraries would be needed to use for the implementation, the requirements were considered, as set in the previous chapter. One of the modules that was heavily used was the random module for Python, mainly for the generation of the individuals’ genes. It was also used in all aspects of the software where an element of probability was used. The NumPy library complimented this functionality, having components of its own that generates random data. Depending on the task, the most appropriate and efficient module was used.



Figure 12. The NumPy library being used (imported as np) to generate a probability matrix during uniform crossover.

NumPy was also used for array manipulation. Again, NumPy working synergistically with the OpenCV and Python Imaging Library (PIL) libraries, allowed the successful completion of one of the Design Requirements (“4.1.1 Image Conversion”). OpenCV and the PIL were used for all image manipulations, however NumPy was necessary to convert all lists into arrays before any image operations occurred. For the creation of all visualizations needed, Matplotlib was used. This library allowed the creation of the necessary graphs and plots. Lastly, for the statistical analysis, the SciPy library (SciPy, 2020) and the XlsxWriter and Statsmodels modules (PyPI, 2022) were used. SciPy and Statsmodels were used for running of any statistical test and corrections required, while XlsxWriter was used for writing the data on a MS Excel sheet for better analysis and access to the data.

## 5.2 Parameter Configuration

Parameter tuning is one of the reasons genetic algorithms underperform, or lead to inaccurate results, such as early convergence. To avoid different parameters being passed into any of the methods used by the final algorithm, the Config class was introduced. This class holds all the parameters used by any of the classes of the algorithm. In this way, we can easily tune these parameters from a single file, and we can ensure that these are being used by all the classes in the code.

Text

Description automatically generated

Figure 13. The Config class in the code.

Since these values would be used throughout every algorithmic run, the use of capital letters for the naming seemed appropriate.

## 5.3 Operators

In this section, all operators of the algorithm will be discussed. Also, helper classes and methods will be mentioned briefly.

### 5.3.1 The Genotype class

For the representation of the individual’s genotype, lists have been used for storing the data. This approach was working well up to the point where fitness was introduced to the project. Two options were considered; either create a separate list that would hold the fitness for each individual in the population (and match indexes of individuals and fitness from two separate lists) or add the fitness at the end of the list of genes, as an extra integer. Both options created several problems, mainly intricate and time-consuming list manipulations. Moreover, for all image manipulations needed, NumPy and PIL were used, and both use arrays for their operations, not lists. Which lead to the main issue of this approach; cropping the fitness value from the end of the list and reformatting the list before passing it to the appropriate helper function. Therefore, the Genotype class was introduced.

Text

Description automatically generated

Figure 14. Default constructor for the Genotype class.

This class is used to represent and hold the genes and the fitness of each individual in the population. This class has also allowed to make our code simpler by adding the Mutation class functionality into Genotype and completely delete the Mutation class. Regardless of the image to be reconstructed the length of the genome would be the sum of pixels in that image.

Text

Description automatically generated

Figure 15. Mutation methods in Genotype class.

### 5.3.2 The Elitism class

The purpose of this class is singular; to find the best performing individuals in a population, that is being passed as a parameter to the find\_elites() method. Even though it was considered putting this method in the Algorithm class, it was decided to not go ahead with this, as to keep different functions of the code separate.

Text

Description automatically generated

Figure 16. One of the methods in the Elitism class, used to find the best performing individuals in a given population.

### 5.3.3 The Fitness class

This class holds a few methods that are being used quite heavily throughout the run of the algorithm: from storing the binary genome of the image to be reconstructed, to evaluating an individual’s fitness and finding the best performing individual in a population.

In the literature, most researchers calculate fitness as a sum of all genes that are different from the optimum fitness, while the higher the fitness score of an individual, the worse it performs in relation to the task provided. This means that optimum fitness is 0. The issue with this approach is that any individuals having negative fitness needed to have their fitness converted into a positive integer, otherwise the algorithm would crash. To avoid this extra step, a reverse approach was used: all individuals are given a fitness value of 0 when they are created. Compared to the genome of the best performing individual (i.e., the image to be reconstructed), the fitness of the individual is being increased for all matching pixels.

Text

Description automatically generated

Figure 17. How the fitness of each individual is being assessed.

For example, for a 32x32 image, optimum fitness would be 1024. Another reason for this approach was pure logic. Most people are familiar with associating better performance with a higher score than the opposite.

### 5.3.4 The Selection class

Many approaches to the selection operator are mentioned in the literature. After thorough research, the roulette-wheel method has been used for this operator. The main benefit of the roulette-wheel operator is its simplicity to implement, and since this operator was not a part of the experiment being tested in this project, it was the obvious choice. Another reason for this decision was that it favours individuals with higher fitness while not excluding less well-performing individuals from being selected. Of course, worse performing individuals would be less likely to be selected to reproduce, similarly to natural selection in real-life habitats: better-performing individuals have a higher chance of mating and therefore passing their advantageous genes to their offspring. However, worse-performing individuals still have a probability to mate and pass their genes to the next generation, but this probability is significantly lower.

Text

Description automatically generated

Figure 18. Implementation of the selection operator.

Lastly, researchers who implemented genetic algorithms for similar problems used the same operator and seemed logical to follow their approach. However, there is a drawback to the roulette-wheel operator: as the population will, inevitably, start to converge, there is a risk of loss of selection pressure. In the next chapter we will discover ways to tackle this issue to ensure the algorithm does not lack selection pressure on the early stages of the run (Sharma, Wadhwa and Komal, 2014; Geethamani and Mayilvaganan, 2016).

### 5.3.5 The Crossover class

Extra time was dedicated for the implementation of the Crossover class, since it holds the three functions that were used for the purpose of this project. All three methods mentioned in this report have been implemented (single- and double-point, and uniform crossover). The double-point crossover method calls on the single-point crossover method twice, once for each crossover point. In this way, the double-point method returns accurate results, since the single-point method was easier to implement and thoroughly test. Plus, existing code is being reused.

Text

Description automatically generated

Figure 19. The double-point method reusing the single-point crossover method twice; once for each crossover point.

The uniform crossover method uses a probability matrix that takes values of 0 and 1. Value 0 assigns the gene to offspring A, while value 1 assigns the gene to offspring B. The probability matrix is created in random, and no parameters were used to affect this, however the code was implemented in a way that would allow such functionality.

Text

Description automatically generated

Figure 20. The uniform crossover method.

### 5.3.6 Additional Functionality

Apart from the classes that hold the main operators for the algorithm, the BinaryImage class and a file with helper functions were used. The former holds the methods that allows image and array conversion/manipulation and accessing images from memory.

Text

Description automatically generated

Figure 21. The method that converts a binary image to a binary array.

In the helper file, there are functions that allow the saving of images and arrays to file, and the generation of graphs/plots.

## 5.4 Main Class

All the classes, objects and helper functions mentioned in this chapter come together to form the algorithmic loop. The full algorithm can be seen in Appendix A – Source Code. Apart from the main method (run\_algorithm()), another method can be found in this class. initialise() is used to create the very first generation of the population.

Text

Description automatically generated

Figure 22. Initialisation of the population.

This method returns a 3D list; population > individual > chromosome. The 0s and 1s are the genes. Apart from saving images and logs for each run, messages are being printed on screen to inform the user of the state of the algorithm.

Text

Description automatically generated

Figure 23. Snapshot of an algorithm run printed on screen.

# 6 Testing

Testing took place throughout the development process. The testing methods included manual visual testing, debugging through visual feedback (print statements), and testing via unit testing. Unit tests provide a framework for testing the functionality of the code. Tests were written using the unittest framework, and specifically the TestCase class. Tables with all the unit tests of the source code can be seen in Appendix B – Test Reports. Unit testing was used to test individual methods and overall class or module behaviours. In each of the classes, the code was decomposed as far as possible into helper methods to keep the methods short and more easily testable.

Text

Description automatically generated

Figure 24. Example of a unit test.

Lastly, manual testing was used to confirm that the algorithm could (and would) eventually fully reconstruct an image. Of course, the genome of the original image and a final image could be compared to check if they matched. However, since it is not certain when the algorithm would produce such results, it was preferred to compare the two images manually. Another reason for this choice was time restrictions and unfamiliarity with the testing approach of algorithmic visual results.

# 7 Experimental Study and Optimizations

## 7.1 The Hypothesis

Since an experiment will be conducted, this is a good point to set our hypotheses.

|  |  |
| --- | --- |
| **Null Hypothesis (H0)** | **Alternative Hypothesis (HA)** |
| The uniform crossover method ***has no effect*** on algorithmic performance compared to the single-point and double-point crossover methods. | The uniform crossover method ***has an effect*** on algorithmic performance compared to the single-point and double-point crossover methods. |

*Table 2. The hypotheses set for the experiment.*

## 7.2 The Experiments

In this section the steps that have been taken to test and configure the algorithm will be discussed. This task required experimenting with various combinations of parameter configuring, while considering evidence in the literature that would assist this endeavor. For all parameters (excluding Crossover), the single-point crossover was used for the experiments.

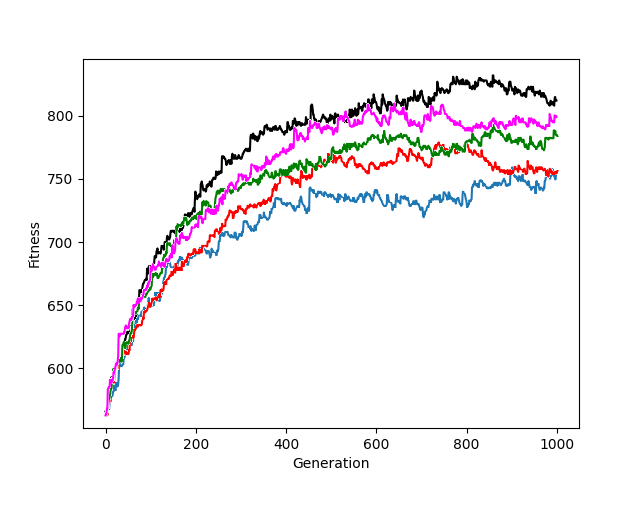
The work of other researchers has been taken into consideration to define which parameter configuration would be best for the purpose of this project. The first step was to determine the size of the population and the number of generations that the algorithm would be allowed to run for, or whether allowing the algorithm to run till it reached optimum fitness.

### 7.2.1 Generations

All runs will be limited to 1000 generations, unless explicitly specified. The reason for this choice was that this number gives the population enough time to start converging to better results, while still slowly increasing to reach its peak. Also, as it is not the purpose of this project for any algorithmic runs to reach an optimum solution, it did not seem productive to significantly increase the algorithmic run time by increasing the number of generations.

### 7.2.2 Population

The size of the population was the first parameter to be tuned. The samples selected to be tested were: 100, 200, 300, 400 and 500 individuals per generation. As expected, the higher the number of individuals the longer each algorithmic run took. The goal of this experiment was to test whether spending time on each run would benefit the algorithm in any productive way.

****

*Figure 25. Average fitness achieved per generation per population sample. Blue for 100 individuals per generation, red for 200 individuals per generation, green for 300 individuals per generation, magenta for 400 individuals per generation, and black for 500 individuals per generation.*

From the results we yielded, it was obvious that the larger the population per generation, the higher overall fitness was achieved. This is not surprising though. During our research, evidence showed that large populations achieve better results, as they can explore a wider problem space (Whitley and Sutton, 2012). However, the runtime of the algorithm greatly increases by increasing the population size, especially after 300 individuals per generation. In Table 1, the average runtime for different population sizes and the average max fitness is presented.

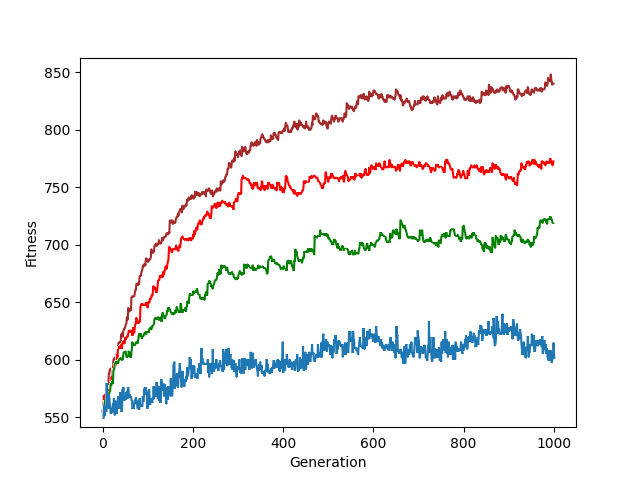
|  |  |  |
| --- | --- | --- |
| **Population size** | **Average max fitness** | **Average runtime (secs)** |
| 100 | 746 | 93.3 |
| 200 | 781 | 115.5 |
| 300 | 797 | 303.3 |
| 400 | 812 | 393.8 |
| 500 | 832 | 506.1 |

*Table 3. Average run time and average max fitness reached by different population sizes.*

Finally, a population size of 200 individuals was selected, since it provides greater fitness results, and lower runtime compared to greater population sizes.

### 7.2.3 Elitism

The next operator to configure was the number of elites carried over to the next generation. Elitism allows the algorithm to provide more normalised and stable results. Moreover, it increases the performance and the accuracy of the algorithm. This hypothesis can be confirmed in the following figure.

****

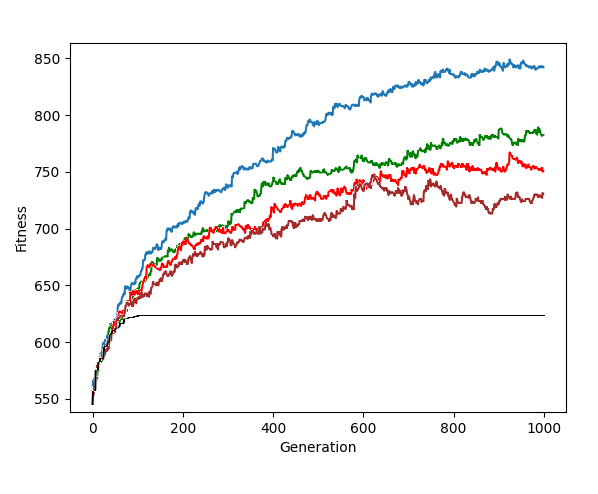
*Figure 26. Average max fitness achieved per generation given different ratios of elite carry over. Blue: 0% elites carried over, green: 1% elites carried over, red: 2% elites carried over and brown 4% elites carried over.*

As expected, the higher the elites being carried over to the next generation, the greater fitness achieved in average, as more individuals with the best performing genome are staying in the gene pool intact. Another interesting observation is the normalizing of the data. We can see that running the algorithm with 0% elites carried over, maximum fitness fluctuates throughout the run, even if, on average, it is gradually rising. When this ratio is increased, max fitness achieved per generation fluctuates less violently, as the best performing individuals start taking over the population and the gene pool becomes more homogenous, which is acceptable as it helps the algorithm avoid loss of selection pressure.

Based on these results, it was decided to use a 2% ratio of elites being carried over for the following reasons: to keep the population more diverse, not increase runtime (increase of overall fitness in each generation, and greater fitness achieved per run, does not impact our project in a constructive way), and stabilize fitness results. The last reason for our selection might not be obvious, but it will allow us to observe the algorithmic performance based on the crossover operator selected, in a more efficient way.

### 7.2.4 Mutation

The last parameter to be tested for tuning was the mutation operator. Mutation allows new, possibly beneficial, changes to be introduced to the population. To test this, mutation probability was decreased to zero; no random changes to genes would occur. The results showed that the populations always converged to a low fitness value and no changes were observed after that point. The population has reached sub-optimal convergence, as expected.



*Figure 27. Average max fitness achieved per generation based on various values of mutation probability. Black: 0 mutation probability, blue: 0.001 mutation probability, green: 0.0015 mutation probability, red: 0.002 mutation probability and brown: 0.0025 mutation probability.*

After that, different values of mutation probability found in the literature were tested as well as several values in that range. The goal of this operator was to achieve enough diversity for the population to not reach convergence in early stages, but not extreme enough to provided inaccurate results, i.e., data that was being affected negatively by causing high disruption in the population (0.0025 mutation probability - brown line). Most of the researchers used a 0.002 mutation probability which provided diverse but also stable results. Similar results were yielded with the experiments conducted. As it can be seen in Figure 27, populations with mutation probability of 0.002 (red line), yield results without extreme fluctuations (0.0025 mutation probability – brown line) nor unreliable, high increases in average max fitness per generation (0.001 mutation probability – blue line). Lastly, the mutation operator works as a control variable for the selection operator. As mentioned in Chapter 5, using the roulette-wheel selection operator, increases the risk of loss of selection pressure, i.e., only the best performing individuals will be selected to reproduce which leads to early homogeneity. Mutation exerts more pressure on the population, hence disrupting such patterns, at least on the early stages of the run.

**7.2.5 The Experimental Independent Variable**

After fine tuning all of the independent variables mentioned above, the algorithm was allowed to run several hundred times using all three crossover operators to collect sets of data to analyze and compare.

|  |  |
| --- | --- |
| **Operator** | **Value** |
| Population (P) | 200 |
| Generations (G) | 1000 |
| Mutation Probability (M) | 0.002 |
| Elite Carry Over (E) | 2% |

*Table 4. Independent variables values for algorithm iterations.*

Since the data obtained from each run of the algorithm is random, the number of data collected will be expanded, in order to increase the reliability and accuracy of the results. Initially, the algorithm will run for 200 iterations for each crossover operator. The initial set of data collected will be analyzed and checked whether these results are statistically significant.

## 7.3 Statistical Analysis

After the collection of the data, the Wilcoxon signed ranked test was used to check whether the results were statistically significant, while the Bonferroni correction was applied to the *p*-values obtained. The sums R+ and R- were used and calculated as suggested in the literature. R+ is the sum of ranks for the problems in which the first method outperformed the second, and R- is the sum of ranks for the opposite. Ranks of difference d = 0, were split evenly among the two sums, with one being ignored if the number of 0s is odd.

The reasons the Wilcoxon signed ranked test was chosen for the analysis of the data were:

* Comparing multiple samples in pairs of two.
* The populations are evenly distributed.
* Exceptionally good/bad performances of the problem have less effect on the Wilcoxon test compared to other methods, like the t-test (Derrac et al., 2011).

All computations were carried out using the SciPy library and the Statsmodels module.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Comparison | Runtime | | | Fitness | | |
| R+ | R- | *p*-value | R+ | R- | *p*-value |
| Uniform vs Single-point | 92.8 | 21.2 | < 0.05 | 308 | 1847 | < 0.05 |
| Uniform vs Double-point | 70.9 | 15.9 | < 0.05 | 425 | 1515 | < 0.05 |

*Table 5. Wilcoxon signed ranks test results with Bonferroni correction applied.*

Table 5 shows the R+, R- , and *p*-values computed for all the comparisons concerning runtime and fitness for the uniform crossover method. As the table states, the uniform crossover method shows a significant improvement in runtime compared to the other methods, with a significance level α = 0.05. However, the uniform crossover method shows a significant deterioration in fitness compared to the other two crossover methods, with α = 0.05.

Based on these results, the **null hypothesis** **H0 is rejected**, while there is evidence that the **alternative hypothesis HA is supported**.

## 7.4 Visual Results

At this point, it is prudent to present the visual results produced by the algorithm, i.e., the images produced for all three operators at various stages of the runs. First, the original image to be reconstructed is presented, followed by the generated images.



Figure 28. Original image to be reconstructed.

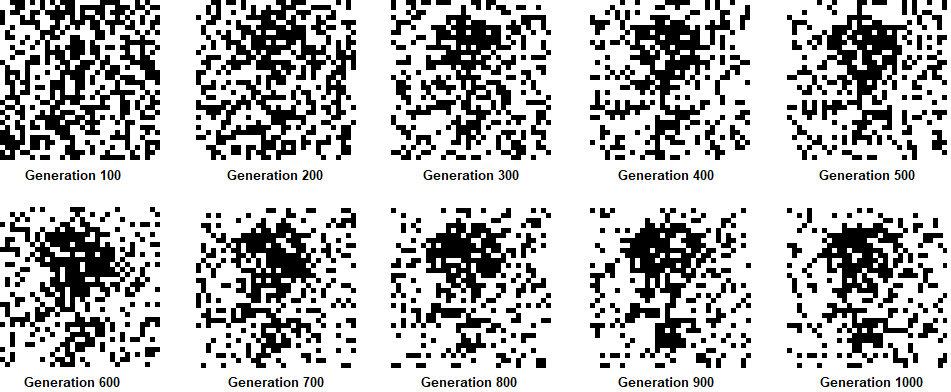


Figure 29. Images generated per 100 generations using the single-point crossover method.

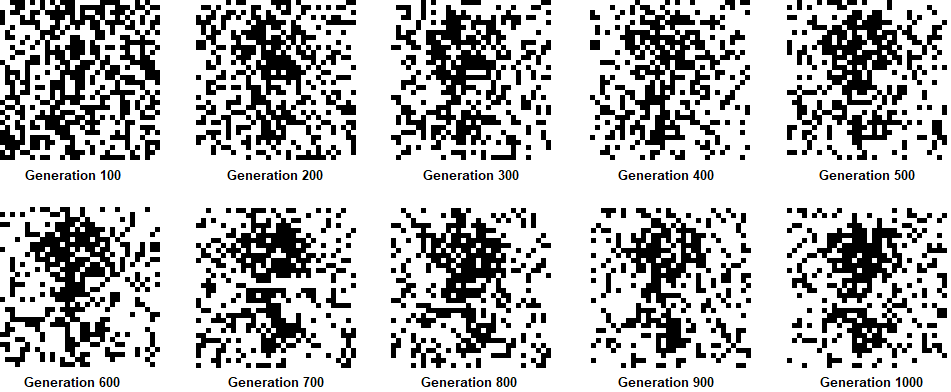


Figure 30. Images generated per 100 generations using the double-point crossover method.

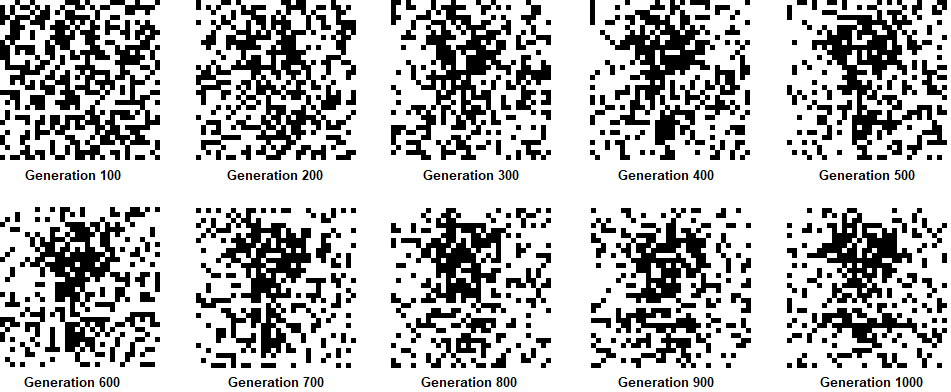


Figure 31. Images generated per 100 generations using the uniform crossover method.

The set of images used above were randomly selected from a big set of images that have been produced by the algorithm throughout the 200 runs. On average, from all the crossover methods used by the algorithm, the results produced had 247 errors, i.e., pixels that did not match the original image. Also, it is evident that during the first 500 generations the quality of the image significantly improves, without much difference observed after that point, till completion of the runs, which confirms the statement made earlier; after the initial spike in maximum fitness, the results start converging with a potential small increase in overall fitness. To further establish these results, the algorithm was allowed to run for 10000 generations for each crossover method. The results are presented below.

Chart, scatter chart

Description automatically generated

Chart, scatter chart

Description automatically generated

Chart, scatter chart

Description automatically generated

Figure 32. Algorithmic runs using parameters in Table 2 for 10000 generations. Top; single-point crossover, middle: double-point crossover, bottom; uniform crossover.

It is evident that after a few hundred generations no noticeable improvement has been made. The fluctuations depicted are due to mutations which disrupt the population and insert new genes to the gene pool. However, not all mutations are beneficial. In any case, maximum fitness stagnates within the range 735 – 800. This means that the algorithm has reached local optima in all cases.

## 7.5 Optimisations

Since the initial image results collected (as examples shown in Figures 29, 30, and 31) are satisfactory but do not have great quality (average 247/1024 errors, or 24% of the total image), further experimentation with the parameters was required. In Table 6, the new set of parameters used for testing is presented.

|  |  |
| --- | --- |
| **Operator** | **Value** |
| Population (P) | 300 |
| Generations (G) | 1000 |
| Mutation Probability (M) | 0.0015 |
| Elite Carry Over (E) | 4% |

*Table 6. Independent variables values for optimised algorithm iterations.*

The goal with these changes were:

* Population: explore a wider area of the problem.
* Mutation: slightly reduce disruption to well performing individuals without risking early convergence (a bigger population sample also helps with that).
* Elite carry over: increase the number of well performing individuals being retained in the population unchanged.

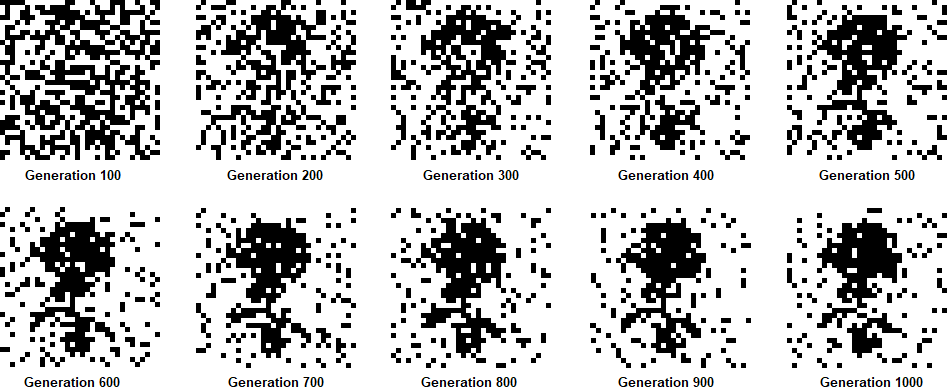


Figure 33. Images generated per 100 generations using the single-point crossover method.

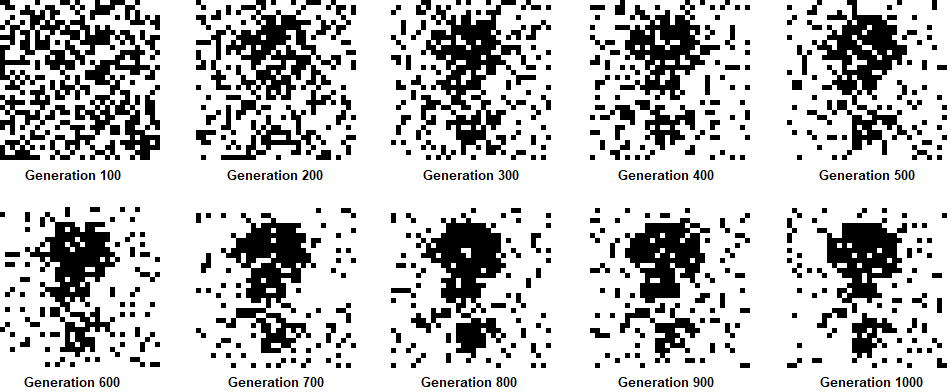


Figure 34. Images generated per 100 generations using the double-point crossover method.

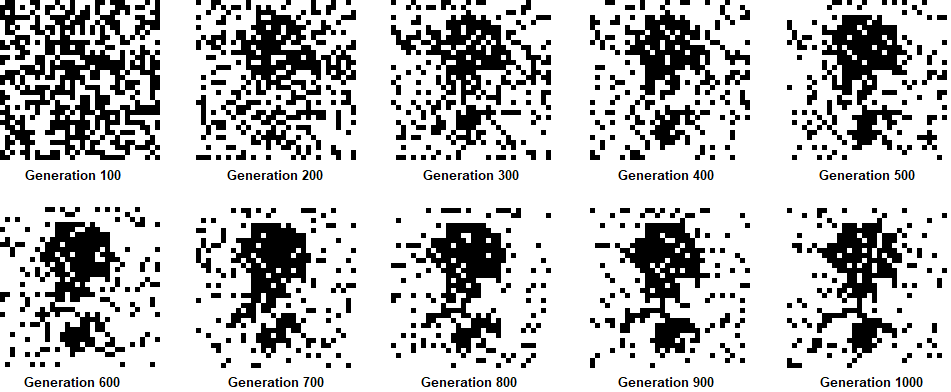


Figure 35. Images generated per 100 generations using the uniform crossover method.

From these images, it is obvious that the results collected are superior to the initial ones. Furthermore, the pixels that were mismatched for this set of runs was lower: on average 117/1024 errors, or 11.4% of the total image. Carrying out the same analysis as with the initial results, the following data was obtained:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Comparison | Runtime | | | Fitness | | |
| R+ | R- | *p*-value | R+ | R- | *p*-value |
| Uniform vs Single-point | 83.5 | 1.9 | < 0.05 | 956 | 187 | < 0.05 |
| Uniform vs Double-point | 54.0 | 1.5 | < 0.05 | 651 | 326 | < 0.05 |

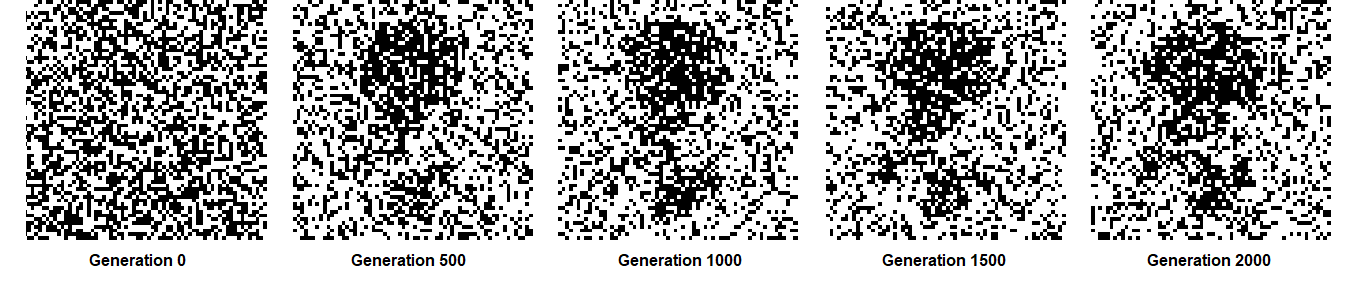
*Table 7. Wilcoxon signed ranks test results with Bonferroni correction applied.*

It is worth mentioning that for this set of parameters the algorithm ran for 150 loops, due to the increased runtime required plus, time limitations to deliver the report in time. For this set of results, the alternative hypothesis HA is still supported. However, it is interesting to notice that using this set of parameters, the uniform method outperformed both methods in all variables tested, with a significance level α = 0.05.

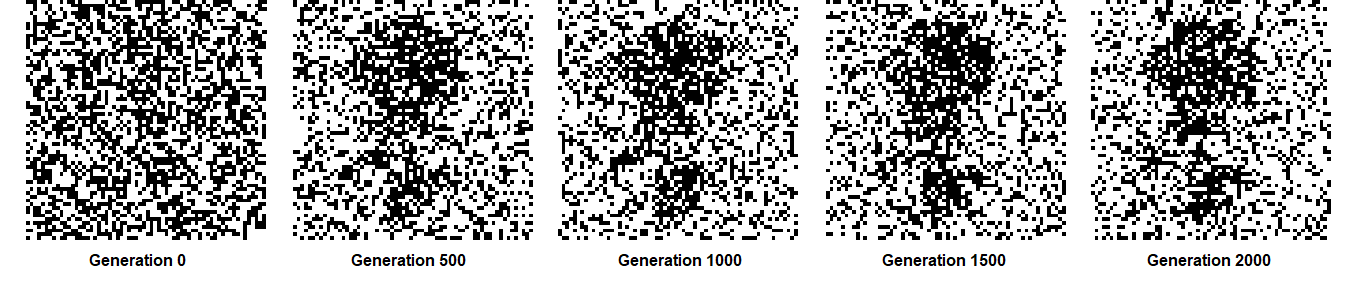
Following the suggested optimisations, as this new set of parameters provides better results, it will be used for the reconstruction of images of higher resolutions. The same image of the rose was used, after altering the dimensions to 64x64 pixels.



*Figure 36. Images generated per 100 generations using the single-point crossover method.*



*Figure 37. Images generated per 100 generations using the double-point crossover method.*



*Figure 38. Images generated per 100 generations using the uniform crossover method.*

The results were not as encouraging as expected, even if the generations were increased to 2000, to provide the algorithm with more time to reconstruct an image of larger dimensions.

# 8 Discussion and Conclusion

From the outset, the goal was to fully implement

This has been one of the shortcomings of the implementation. It was evident that such a functionality would be useful, however, it was too late to make the changes needed to accommodate this, as it would require refactoring most of the code, a task both difficult and time-consuming. One way to combat this issue is to increase the number of generations to a large number, e.g., 10000 generations, so that the population has enough time to reach its peak.

The project has been a highly educational and enjoyable experience. Some challenging cases emerged where taught skills from the MSc Computer Science course were revises and applied. The project has provided a better understanding of the software development cycle and confidence in experimenting with new technologies and ideas has been acquired.

Working within an agile context further served to underpin the creative process during development.

Considered as a whole, the project has resulted in a software library that could be deployed in its current form for the analysis of certain classification problems. Further work could expand the scope of the package by increasing the range of supported algorithms.

Ultimately the question is whether the project achieved its set goals. My assertion is that it has done so in the following ways:

# 9 References

‌Abo Smara, G. and Khalefah, F. (2014). Localization of License Plate Number Using Dynamic Image Processing Techniques and Genetic Algorithms. IEEE Transactions on Evolutionary Computation, 18(2), pp.244–257.

Abouhawwash, M. and Alessio, A.M. (2021). Multi-Objective Evolutionary Algorithm for PET Image Reconstruction: Concept. IEEE Transactions on Medical Imaging, 40(8), pp.2142–2151.

Abraham, A., Nedjah, N. and Mourelle, L. de M. (2006). Evolutionary Computation: from Genetic Algorithms to Genetic Programming. Studies in Computational Intelligence, pp.1–20.

Ahn, C., & Ramakrishna, R. S. (2003). Elitism-based compact genetic algorithms. IEEE Transactions on Evolutionary Computation, 7 (4), 367-385.

Antun, V., Renna, F., Poon, C., Adcock, B. and Hansen, A.C. (2020). On instabilities of deep learning in image reconstruction and the potential costs of AI. Proceedings of the National Academy of Sciences, 117(48), pp.30088–30095.

‌Babb, B., Moore, F., Peterson, M. and Lamont, G. (2008). Improved satellite image compression and reconstruction via genetic algorithms. SPIE Proceedings.

Banzhaf, W., Beslon, G., Christensen, S., Foster, J.A., Képès, F., Lefort, V., Miller, J.F., Radman, M. and Ramsden, J.J. (2006). From artificial evolution to computational evolution: a research agenda. Nature Reviews Genetics, 7(9), pp.729–735.

Bao, Z. and Watanabe, T. (2010). Circuit Design Optimization Using Genetic Algorithm with Parameterized Uniform Crossover. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, E93-A(1), pp.281–290.

‌Chang Wook and Ramakrishna, R.S. (2003). Elitism-based compact genetic algorithms. IEEE Transactions on Evolutionary Computation, 7(4), pp.367–385.

Chung, H., & Shin, K.-s. (2020). Genetic algorithm-optimized multi-channel convolutional neural network for stock market prediction. Neural Computing and Applications, 32 (12), 7897-7914.

Darma, P.N., Baidillah, M.R., Sifuna, M.W. and Takei, M. (2019). Improvement of image reconstruction in electrical capacitance tomography (ECT) by sectorial sensitivity matrix using a K-means clustering algorithm. Measurement Science and Technology, 30(7), p.075402.

de Haan, B.K., Rivenson, Y., Wu, Y. and Ozcan, A. (2019). Deep-Learning-Based Image Reconstruction and Enhancement in Optical Microscopy. Proceedings of the IEEE, pp.1–21.

Devarriya, D., Gulati, C., Mansharamani, V., Sakalle, A., & Bhardwaj, A. (2020). Unbalanced breast cancer data classification using novel fitness functions in genetic programming. Expert Systems with Applications, 140, 112866.

Dennis, A., Barbara Haley Wixom, David Paul Tegarden and Seeman, E. (2015). System analysis & design : an object-oriented approach with UML. 5th ed. Hoboken, Nj: Wiley.

‌Eyal Wirsansky (2020). Hands-on genetic algorithms with Python : applying genetic algorithms to solve real-world deep learning and artificial intelligence problems / Eyal Wirsansky. Birmingham, Uk: Packt Publishing Ltd.

Forghani, F., Mahl, A., Patton, T.J., Jones, B.L., Borden, M.A., Westerly, D.C., Altunbas, C., Miften, M. and Thomas, D.H. (2020). Simulation of x‐ray‐induced acoustic imaging for absolute dosimetry: Accuracy of image reconstruction methods. Medical Physics, 47(3), pp.1280–1290.

Franconi, L. and Jennison, C. (1997). Statistics and Computing, 7(3), pp.193–207.

Fu, Q., Zhang, X. and Li, H. (2021). Convolutional neural network-based restoration method of basketball contour image. Microprocessors and Microsystems, 83, p.104004.

‌Galbally, J., Ross, A., Gomez-Barrero, M., Fierrez, J. and Ortega-Garcia, J. (2013). Iris image reconstruction from binary templates: An efficient probabilistic approach based on genetic algorithms. Computer Vision and Image Understanding, 117(10), pp.1512–1525.

Geethamani, G.S. and Mayilvaganan, D.M. (2016). Analysis Of Genetic Crossover Techniques Based on Roulette Wheel Selection Algorithm and Steady State Selection Algorithm. International Journal of Engineering and Computer Science. doi:10.18535/ijecs/v5i1.5.

Hammernik, K. and Knoll, F. (2020). Machine learning for image reconstruction. Handbook of Medical Image Computing and Computer Assisted Intervention, pp.25–64.

Han, J.-B., Kim, S.-H., Jang, M.-H., & Ri, K.-S. (2019). Using genetic algorithm and narx neural network to forecast daily bitcoin price. Computational Economics, 1-17.

Holland, J. (1975). Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. MIT press.

Howse, J. (2013). OpenCV computer vision with Python : learn to capture videos, manipulate images, and track objects with Python using the OpenCV library. Birmingham, England: Packt Publishing.

Iga, C. and Wakahara, T. (2004). Character Image Reconstruction from a Feature Space Using Shape Morphing and Genetic Algorithms. Ninth International Workshop on Frontiers in Handwriting Recognition.

Katoch, S., Chauhan, S.S. and Kumar, V. (2020). A review on genetic algorithm: past, present, and future. Multimedia Tools and Applications.

Khare, C., and Nagwanshi, K. K. (2011). Image Restoration Technique with Non-Linear Filters. International Journal of Engineering Trends and Technology.

Khilar, R., Chitrakala, S. and SelvamParvathy, S. (2013). 3D image reconstruction: Techniques, applications, and challenges. 2013 International Conference on Optical Imaging Sensor and Security (ICOSS).

Kramer, O. (2017). Genetic Algorithms. Genetic Algorithm Essentials, pp.11–19.

Kora, P. and Yadlapalli, P. (2017). Crossover Operators in Genetic Algorithms: A Review. International Journal of Computer Applications, 162(10), pp.34–36.

Lézoray, O., Charrier, C., Cardot, H. and Lefèvre, S. (2008). Machine Learning in Image Processing. EURASIP Journal on Advances in Signal Processing, 2008(1).‌

Mignon, A. and Jurie, F. (2013). Reconstructing faces from their signatures using RBF regression. Procedings of the British Machine Vision Conference 2013.

Miranda Lakshmi, T., Martin, A. and Prasanna Venkatesan, V. (2013). An Analysis of Students Performance Using Genetic Algorithm. Journal of Computer Sciences and Applications, 1(4), pp.75–79.

Mirjalili, S., Song Dong, J., Sadiq, A.S. and Faris, H. (2019). Genetic Algorithm: Theory, Literature Review, and Application in Image Reconstruction. Nature-Inspired Optimizers, pp.69–85.

‌Mohammadi, A., Asadi, H., Mohamed, S., Nelson, K. and Nahavandi, S. (2017). OpenGA, a C++ Genetic Algorithm Library. 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC).

Nakata, J. and Ogawa, K. (1995). Image reconstruction from projections using a genetic algorithm. Proceedings of 1994 IEEE Nuclear Science Symposium - NSS’94.

Naseem, I., Togneri, R. and Bennamoun, M. (2010). Linear Regression for Face Recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence, [online] 32(11), pp.2106–2112.

Nestor, A., Lee, A.C.H., Plaut, D.C. and Behrmann, M. (2020). The Face of Image Reconstruction: Progress, Pitfalls, Prospects. Trends in Cognitive Sciences, 24(9), pp.747–759.

O’Neill, M., Vanneschi, L., Gustafson, S. and Banzhaf, W. (2010). Open issues in genetic programming. Genetic Programming and Evolvable Machines, 11(3-4), pp.339–363.

Naik, T.R. and Dabhi, V.K. (2013). Improving Generalization Ability of Genetic Programming: Comparative Study. Journal of Bioinformatics and Intelligent Control, 2(4), pp.243–252.

‌Numpy (2009). NumPy — NumPy. [online] Numpy.org. Available at: <https://numpy.org/>.

Pal, S.K., Bhandari, D. and Kundu, M.K. (1994). Genetic algorithms for optimal image enhancement. Pattern Recognition Letters, 15(3), pp.261–271.

‌Perone, C.S. (2009). Pyevolve. ACM SIGEVOlution, 4(1), pp.12–20.

PyPI. (2022). XlsxWriter. [online] Available at: https://pypi.org/project/XlsxWriter/ [Accessed 1 Sep. 2022].

SciPy (2020). SciPy.org — SciPy.org. [online] Scipy.org. Available at: https://scipy.org/.

Shrestha, A. and Mahmood, A. (2019). Review of Deep Learning Algorithms and Architectures. IEEE Access, 7, pp.53040–53065.

‌Skoda, P. and Adam, F. (2020). Knowledge discovery in big data from astronomy and earth observation: astrogeoinformatics. Amsterdam: Elsevier.

Piwonska, A. and Grycuk, U. (2007). Images Reconstruction with Use of a Genetic Algorithm. 6th International Conference on Computer Information Systems and Industrial Management Applications (CISIM’07).

Pradhan, D., Wang, S., Ali, S., Yue, T. and Liaaen, M. (2017). CBGA-ES: A Cluster-Based Genetic Algorithm with Elitist Selection for Supporting Multi-Objective Test Optimization. 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST).

Pradhan, D., Wang, S., Ali, S., Yue, T. and Liaaen, M. (2017). CBGA-ES: A Cluster-Based Genetic Algorithm with Elitist Selection for Supporting Multi-Objective Test Optimization. 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST).

‌PyTorch (2019). PyTorch. [online] Pytorch.org. Available at: <https://pytorch.org/>.

Raymer, M. L., Punch, W. F., Goodman, E. D., Kuhn, L. A., & Jain, A. K. (2000). Dimensionality reduction using genetic algorithms. IEEE transactions on evolutionary computation, 4 (2), 164-171.

Derrac, J., García, S., Molina, D. and Herrera, F. (2011). A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. Swarm and Evolutionary Computation, 1(1), pp.3–18. doi:10.1016/j.swevo.2011.02.002.

Sharma, P., Wadhwa, A. and Komal, K. (2014). Analysis of Selection Schemes for Solving an Optimization Problem in Genetic Algorithm. International Journal of Computer Applications, 93(11), pp.1–3. doi:10.5120/16256-5714.

Sheta, A., Braik, M.S. and Aljahdali, S. (2012). Genetic Algorithms: A tool for image segmentation. 2012 International Conference on Multimedia Computing and Systems.

Shi, M., Zhou, B. and Chen, M. (2013). Application of improved genetic algorithm based on uniform design sampling crossover operator in regression model. Journal of Computer Applications, 32(11), pp.3050–3053.

Shrestha, A. and Mahmood, A. (2016). Improving Genetic Algorithm with Fine-Tuned Crossover and Scaled Architecture. Journal of Mathematics, 2016, pp.1–10.

‌Silva, R.M.A., Resende, M.G.C. and Pardalos, P.M. (2013). A Python/C++ library for bound-constrained global optimization using a biased random-key genetic algorithm. Journal of Combinatorial Optimization, 30(3), pp.710–728.

‌Skorpil, V., Oujezsky, V., Cika, P. and Tuleja, M. (2019). Parallel Processing of Genetic Algorithms in Python Language. 2019 PhotonIcs & Electromagnetics Research Symposium - Spring (PIERS-Spring).

Sohail, A. (2021). Genetic Algorithms in the Fields of Artificial Intelligence and Data Sciences. Annals of Data Science.

Srinivas, M., and Patnaik, L.M. (1994). Genetic algorithms: a survey. Computer, 27(6), pp.17–26.

Stanley, K. O., Clune, J., Lehman, J., & Miikkulainen, R. (2019). Designing neural networks through neuroevolution. Nature Machine Intelligence, 1 (1), 24-35.

Street, A. G., & Mayo, S. L. (1999). Computational protein design. Structure, 7 (5), R105-R109.

Ting, C.-K., Wang, T.-C., Liaw, R.-T. and Hong, T.-P. (2016). Genetic algorithm with a structure-based representation for genetic-fuzzy data mining. Soft Computing, 21(11), pp.2871–2882.

Vie, A., Kleinnijenhuis, A. M., and Farmer D. J. (2021). Qualities, challenges, and future of genetic

algorithms: a literature review. SSRN. <http://dx.doi.org/10.2139/ssrn.3726035>.

Waheeb, W., & Ghazali, R. (2019). A new genetically optimized tensor product functional link neural network: An application to the daily exchange rate forecasting. Evolutionary Intelligence, 12 (4), 593-608.

Wang, G., Ye, J.C. and De Man, B. (2020). Deep learning for tomographic image reconstruction. Nature Machine Intelligence, [online] 2(12), pp.737–748

Wang, G., Ye, J.C., Mueller, K. and Fessler, J.A. (2018). Image Reconstruction is a New Frontier of Machine Learning. IEEE Transactions on Medical Imaging, [online] 37(6), pp.1289–1296.

Wang, W. and Siau, K. (2019). Artificial Intelligence, Machine Learning, Automation, Robotics, Future of Work and Future of Humanity. Journal of Database Management, 30(1), pp.61–79.‌

Whitley, D. (1994). A genetic algorithm tutorial. Statistics and computing, 4 (2), 65-85.

Whitley, D. and Sutton, A.M. (2012). Genetic Algorithms — A Survey of Models and Methods. Handbook of Natural Computing, pp.637–671.

‌Wonjae Lee and Hak-Young Kim (2005). Genetic algorithm implementation in Python. Fourth Annual ACIS International Conference on Computer and Information Science (ICIS’05).

Yadav, A., and Khatak, A. (2017). Survey on Image Restoration Techniques. International Journal For Technological Research In Engineering, 5(4) 3031-35.

Yu, Y., Gao, S., Wang, Y. and Todo, Y. (2019). Global optimum-based search differential evolution. IEEE/CAA Journal of Automatica Sinica, 6(2), pp.379–394.

Zendesk. (2020). Deep learning vs. machine learning: What’s the difference? [online] Available at: https://www.zendesk.co.uk/blog/machine-learning-and-deep-learning/#georedirect [Accessed 10 Apr. 2022].

Zheng, X., Ravishankar, S., Long, Y. and Fessler, J.A. (2018). PWLS-ULTRA: An Efficient Clustering and Learning-Based Approach for Low-Dose 3D CT Image Reconstruction. IEEE Transactions on Medical Imaging, 37(6), pp.1498–1510.

Zhu, Y., Zhu, C. and Li, X. (2018). Improved principal component analysis and linear regression classification for face recognition. Signal Processing, 145, pp.175–182.

# Appendix A – Source Code

Class Algorithm

class Algorithm(Config):  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 # ======================= Initialise the first generation =====================================================  
 self.population = self.initialise()  
 self.current\_generation = []  
  
 def initialise(self):  
 *"""  
 Method to create the initial population for the algorithm.* ***:return****: a 3D list of Genotype objects.  
 """* population = [[0] \* self.GENOME\_LENGTH for i in [1] \* self.POPULATION]  
 for individual in range(self.POPULATION):  
 population[individual] = Genotype()  
 return population  
  
 def run\_algorithm(self):  
 *"""  
 The method that runs the algorithm.* ***:return****: prints results in terminal, and saves images and graphs for each run.  
 """* # ======================= Algorithm loop ======================================================================  
 crossover\_operator\_validation = ["S", "D", "U"]  
 for crossover\_operator in crossover\_operator\_validation:  
 self.population = self.initialise()  
 self.current\_generation = []  
  
 imaging = BinaryImage()  
 fitness = Fitness()  
 selection = Selection()  
 crossover = Crossover()  
 elitism = Elitism()  
  
 # ======================= Optimal individual ==============================================================  
 optimal\_genotype = fitness.optimal\_fitness\_genotype()  
  
 start\_time = time.time()  
  
 f = open("images/algorithm\_log.txt", "a")  
  
 # Variables and objects that will help keep track of numbers for logging.  
 maximum\_algorithmic\_fitness = 0  
 max\_fitness\_in\_generation = 0  
 generation\_list = []  
 fitness\_list = []  
  
 for generation in range(self.GENERATIONS):  
 generation\_list.append(generation)  
 print(f"Running generation: {generation + 1}")  
 self.current\_generation = self.population  
 next\_generation = []  
  
 # ======================= Evaluate fitness for each individual ========================================  
 for individual in self.current\_generation:  
 fitness.evaluate\_fitness(optimal\_genotype, individual)  
  
 # ======================= Seek elites in population ===================================================  
 elites = elitism.find\_elites(self.current\_generation)  
 for elite in elites:  
 next\_generation.append(copy.deepcopy(elite))  
  
 # ======================= Selection ===================================================================  
 # The size of the population needs to stay the same throughout the run of the algorithm.  
 # At this stage we already have the elites in the next generation.  
 # Therefore, the size of the next population will be (population - elites)  
 # Lastly, we divide this number by 2, as the following operations will be done in pairs of individuals;  
 # parent\_one and parent\_two, which will produce offspring\_one and offspring\_two  
  
 for i in range(int((self.POPULATION - self.ELITE\_CARRY\_OVER) / 2)):  
 parent\_one = selection.roulette\_selection(self.current\_generation)  
 parent\_two = selection.roulette\_selection(self.current\_generation)  
  
 # ======================= Crossover ===============================================================  
 children = crossover.do\_crossover(parent\_one, parent\_two, crossover\_operator)  
  
 for child in children:  
 next\_generation.append(copy.deepcopy(child))  
  
 # ======================= Mutation ====================================================================  
 for individual in next\_generation:  
 individual.mutation()  
  
 # ======================= Population fitness evaluation ===============================================  
 # Update maximum fitness for the run of the algorithm.  
 max\_fitness\_in\_generation = fitness.get\_max\_fitness(next\_generation)  
 if max\_fitness\_in\_generation > maximum\_algorithmic\_fitness:  
 maximum\_algorithmic\_fitness = max\_fitness\_in\_generation  
  
 # Finish if algorithm has reached optimum fitness.  
 if fitness.fitness\_to\_reach == max\_fitness\_in\_generation:  
 print(f"Maximum fitness has been reached after {generation + 1} generations!")  
 top\_individual = fitness.get\_max\_fitness\_genotype(next\_generation)  
 image = imaging.binary\_array\_to\_binary\_image(top\_individual.genes)  
 helper.save\_binary\_image(image, generation, crossover\_operator)  
 # Add this to the fitness\_list otherwise trying to create the graph breaks cause x and y axes must  
 # have same first dimension  
 fitness\_list.append(max\_fitness\_in\_generation)  
 break  
 else:  
 # Display maximum fitness for the generation and save an image every 100 generations.  
 print(f"Max fitness in current population is {max\_fitness\_in\_generation}\n")  
 if generation % 100 == 0:  
 top\_individual = fitness.get\_max\_fitness\_genotype(next\_generation)  
 image = imaging.binary\_array\_to\_binary\_image(top\_individual.genes)  
 helper.save\_binary\_image(image, generation, crossover\_operator)  
 fitness\_list.append(max\_fitness\_in\_generation)  
  
 # ======================= Prepare for next iteration ==================================================  
 self.population = next\_generation  
 self.current\_generation = []  
  
 # Save an image for the last iteration of the algorithm.  
 top\_individual = fitness.get\_max\_fitness\_genotype(next\_generation)  
 image = imaging.binary\_array\_to\_binary\_image(top\_individual.genes)  
 helper.save\_binary\_image(image, generation, crossover\_operator)  
 print(f"Maximum fitness for the run: {maximum\_algorithmic\_fitness}")  
  
 # Create the plot.  
 helper.create\_plot(generation\_list, fitness\_list, crossover\_operator)  
  
 # ======================= Update algorithm log ============================================================  
 end\_time = time.time()  
 total\_runtime = "%.2f" % (end\_time - start\_time)  
 time\_of\_run = time.strftime("%d%m%Y-%H%M%S")  
 reach\_global\_optimum = "No"  
 if fitness.fitness\_to\_reach == max\_fitness\_in\_generation:  
 reach\_global\_optimum = 'Yes'  
 f.write(f"{time\_of\_run}\n")  
 f.write(f"Did the algorithm reach maximum fitness? : {reach\_global\_optimum}\n")  
 f.write(f"Crossover method used: {str(crossover\_operator)}\n")  
 f.write(f"Population size: {self.POPULATION}\n")  
 f.write(f"Number of generations: {self.GENERATIONS}\n")  
 f.write(f"Mutation probability: {self.MUTATION\_PROBABILITY}\n")  
 f.write(f"Number of elites in each generation: {self.ELITE\_CARRY\_OVER}\n")  
 f.write(f"Maximum fitness reached: {maximum\_algorithmic\_fitness}/{fitness.fitness\_to\_reach}\n")  
 f.write(f"Total runtime: {total\_runtime} seconds\n")  
 f.write("\n")  
 f.close()

Class BinaryImage

class BinaryImage(Config):  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
  
 def binary\_image\_to\_binary\_array(self, img):  
 *"""  
 An image of black and white pixels is used to generate a numpy array of 0s and 1s.* ***:param*** *img: binary image (\*.png).* ***:return****: a 2D binary array.  
 """* # Convert black and white pixels of the image to an array of binary arrays, each line is a new array.  
 # Any pixels that have colour of greater than 0 (not black), are set to 1  
 np\_img = np.array(img)  
 np\_img[np\_img > 0] = 1  
  
 # Convert the image to a binary array.  
 binary\_array = np\_img.astype(int)  
 return np.ndarray.tolist(binary\_array)  
  
 def binary\_array\_to\_binary\_image(self, binary\_array):  
 *"""  
 Generate a binary image from a binary numpy array.* ***:param*** *binary\_array: 2d binary array.* ***:return****: a binary image.  
 """* # Convert the array to a sequence of numbers  
 img = np.array(im.fromarray(np.asarray(binary\_array) \* 255))  
 return img  
  
 def open\_original\_image(self):  
 *"""  
 This method is used only to convert the original image to be reconstructed to a binary array.* ***:return****: a Genotype object.  
 """* img = im.open("original\_image.png")  
 optimal\_array = self.binary\_image\_to\_binary\_array(img)  
 optimal\_individual = Genotype()  
 optimal\_individual.genes = optimal\_array  
 return optimal\_individual  
  
 def create\_random\_binary\_image(self):  
 *"""  
 A random binary image is generated with dimensions size \* size and saved to file.* ***:return****: a binary image.  
 """* # Generate black image (sequence)  
 img = np.zeros((self.GENOME\_LENGTH \* self.GENOME\_LENGTH, 1), np.uint8)  
  
 # Determine number of white pixels, and set (=255 is white). We arbitrarily set the ratio of white to black  
 # pixels to be 50:50. At this point we have an image that is half black and half white.  
 img[0:int(self.WHITE\_RATIO \* self.GENOME\_LENGTH \* self.GENOME\_LENGTH)] = 255  
  
 # Shuffle pixels, and reshape image  
 np.random.shuffle(img)  
 img = np.reshape(img, (self.GENOME\_LENGTH, self.GENOME\_LENGTH))  
 return img

Class Config

class Config:  
 def \_\_init\_\_(self):  
 self.GENERATIONS = 1000  
 self.POPULATION = 300  
 self.GENOME\_LENGTH = 32  
 self.MUTATION\_PROBABILITY = 0.0015  
 self.ELITE\_CARRY\_OVER = int(self.POPULATION \* 0.04)  
 self.UNIFORM\_PROBABILITY = 0.5

Class Crossover

class Crossover(Config):  
 def \_\_init\_\_(self):  
 Config.\_\_init\_\_(self)  
  
 def generate\_random\_crossover\_point(self):  
 *"""  
 Generate an integer number that will be used for the single point crossover method.* ***:return****: int  
 """* return random.randint(1, self.GENOME\_LENGTH - 1)  
  
 def single\_point\_crossover(self, genotype\_one, genotype\_two, crossover\_point):  
 *"""  
 With this method, we select one crossover point. Each parental array is split into two parts. Combining a part  
 of one parent with that one of the other parent, we generate two children.* ***:param*** *genotype\_one: Genotype object.* ***:param*** *genotype\_two: Genotype object.* ***:param*** *crossover\_point: int.* ***:return****: an array consisting of two Genotype objects (i.e., the two offspring).  
 """* genotype\_one.genes, genotype\_two.genes = genotype\_one.genes[:crossover\_point] + \  
 genotype\_two.genes[crossover\_point:], \  
 genotype\_two.genes[:crossover\_point] + \  
 genotype\_one.genes[crossover\_point:]  
 return [genotype\_one, genotype\_two]  
  
 def double\_point\_crossover(self, genotype\_one, genotype\_two, crossover\_point\_one, crossover\_point\_two):  
 *"""  
 With this method, we randomly select two crossover points. We run a loop for those points using the single-point  
 crossover method for each of the points.* ***:param*** *genotype\_one: Genotype object.* ***:param*** *genotype\_two: Genotype object.* ***:param*** *crossover\_point\_one: int.* ***:param*** *crossover\_point\_two: int.* ***:return****: an array consisting of two Genotype objects (i.e., the two offspring).  
 """* points = [crossover\_point\_one, crossover\_point\_two]  
  
 # In case the random numbers generated are the same, rerun.  
 while crossover\_point\_one == crossover\_point\_two:  
 crossover\_point\_one = self.generate\_random\_crossover\_point()  
 crossover\_point\_two = self.generate\_random\_crossover\_point()  
 self.double\_point\_crossover(genotype\_one, genotype\_two, crossover\_point\_one, crossover\_point\_two)  
  
 for index in points:  
 genotype\_one, genotype\_two = self.single\_point\_crossover(genotype\_one, genotype\_two, index)  
 return [genotype\_one, genotype\_two]  
  
 def uniform\_crossover(self, genotype\_one, genotype\_two):  
 *"""  
 With this method, each bit is chosen in random from each parent using a probability matrix. In the end, we  
 generate two children with genomes that can be widely different from the parental ones.* ***:param*** *genotype\_one: Genotype object.* ***:param*** *genotype\_two: Genotype object.* ***:return****: an array consisting of two Genotype objects (i.e., the two offspring).  
 """* # Generate the probability matrix  
 probability\_matrix = np.random.rand(self.GENOME\_LENGTH)  
 for index in range(len(probability\_matrix)):  
 # Values less or greater than 0.5 can be considered here.  
 if probability\_matrix[index] < self.UNIFORM\_PROBABILITY:  
 genotype\_one.genes[index], genotype\_two.genes[index] = \  
 genotype\_two.genes[index], genotype\_one.genes[index]  
 return [genotype\_one, genotype\_two]  
  
 def do\_crossover(self, parent\_one, parent\_two, crossover\_operator):  
 *"""  
 Method that returns the children after crossover of two parents selected for reproduction.* ***:param*** *parent\_two: a Genotype object.* ***:param*** *parent\_one: a Genotype object.* ***:param*** *crossover\_operator: int.* ***:return****: a list of two Genotype objects.  
 """* if crossover\_operator == "S":  
 crossover\_point = self.generate\_random\_crossover\_point()  
 children = self.single\_point\_crossover(parent\_one, parent\_two, crossover\_point)  
 elif crossover\_operator == "D":  
 crossover\_point\_one = self.generate\_random\_crossover\_point()  
 crossover\_point\_two = self.generate\_random\_crossover\_point()  
 children = self.double\_point\_crossover(parent\_one, parent\_two,  
 crossover\_point\_one, crossover\_point\_two)  
 else:  
 children = self.uniform\_crossover(parent\_one, parent\_two)  
 return children

Class Elitism

class Elitism(Config):  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
  
 def find\_elites(self, population):  
 *"""  
 This method sorts the fitness of all individuals in order to get the indices of the individuals with the highest  
 fitness. Then we grab those individuals and store them in a new list. In case there are individuals with the  
 same fitness on the cut-off point, we select the individual with the higher index.* ***:param*** *population: list that holds the genetic information of all individuals in the current generation.* ***:return****: list of individuals to be included in the next generation without any crossover/mutation applied.  
 """* elites = []  
 fitness = []  
 for individual in population:  
 fitness.append(individual.fitness)  
 elites\_indices = sorted(range(len(fitness)),  
 key=lambda i: fitness[i])[-self.ELITE\_CARRY\_OVER:]  
  
 for index in elites\_indices:  
 elites.append(population[index])  
 return elites

Class Fitness

class Fitness(Config):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.fitness\_to\_reach = self.GENOME\_LENGTH \* self.GENOME\_LENGTH  
  
 def optimal\_fitness\_genotype(self):  
 *"""  
 This method is used to get the array of the image to be reconstructed. This is the array that will be used to  
 compare each individual's array, therefore its fitness, while the algorithm is running.* ***:return****: a Genotype object.  
 """* optimal\_array = BinaryImage().open\_original\_image()  
 return optimal\_array  
  
 def evaluate\_fitness(self, optimal\_genotype, individual\_genotype):  
 *"""  
 This method is used to calculate the fitness level of an individual in the population, compared to the perfect  
 individual, i.e., the array of the image to be reconstructed. It stores the fitness of the individual in a  
 list. This will be needed during selection and elitism.* ***:param*** *optimal\_genotype: Genotype object.* ***:param*** *individual\_genotype: Genotype object.* ***:return****: change an individual's fitness (void method).  
 """* individual\_genotype.fitness = self.fitness\_to\_reach  
 for chromosome in range(self.GENOME\_LENGTH):  
 for gene in range(self.GENOME\_LENGTH):  
 if optimal\_genotype.genes[chromosome][gene] == individual\_genotype.genes[chromosome][gene]:  
 continue  
 else:  
 individual\_genotype.fitness -= 1  
  
 def get\_max\_fitness(self, population):  
 *"""  
 A method that returns the maximum fitness in a given population.* ***:param*** *population: an array of Genotype objects.* ***:return****: int.  
 """* return max(individual.fitness for individual in population)  
  
 def get\_max\_fitness\_genotype(self, population):  
 *"""  
 A method that returns the individual with the maximum fitness in a given population.* ***:param*** *population: an array of Genotype objects.* ***:return****: a Genotype object.  
 """* population.sort(key=lambda x: x.fitness, reverse=True)  
 return population[0]

Class Genotype

class Genotype(Config):  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.genes = [[random.randint(0, 1) for i in range(0, self.GENOME\_LENGTH)]  
 for i in range(0, self.GENOME\_LENGTH)]  
 self.fitness = 0  
  
 def mutation(self):  
 *"""  
 Depending on config.MUTATION\_PROBABILITY, we produce a random number between 0 and 1 to decide whether a  
 mutation for the specific gene should take place or not. Mutation is the reversal of 0 to 1, and 1 to 0.* ***:return****: void - method that only flips genes that should be mutated.  
 """* for chromosome in range(self.GENOME\_LENGTH):  
 for gene in range(self.GENOME\_LENGTH):  
 if random.uniform(0, 1) <= self.MUTATION\_PROBABILITY:  
 self.genes[chromosome][gene] = self.\_\_mutate\_\_(self.genes[chromosome][gene])  
  
 def \_\_mutate\_\_(self, gene):  
 *"""  
 Private method that handles the flipping of genes (i.e., mutation).* ***:param*** *gene: int.* ***:return****: int.  
 """* if gene == 0:  
 return 1  
 return 0

Class Selection

class Selection(Config):  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
  
 def roulette\_selection(self, population):  
 *"""  
 For the selection operator we are using the roulette-wheel operator (read README for more information). This  
 method is selecting an individual from the population that will be used to create offspring for the next  
 generation.* ***:return****: Genotype object.  
 """* cumulative\_fitness = sum(individual.fitness for individual in population)  
 selection\_probability = [individual.fitness/cumulative\_fitness for individual in population]  
 return population[npr.choice(len(population), p=selection\_probability)]

Helper functions

def save\_binary\_array(binary\_array, index):  
 *"""  
 Helper function that saves a binary array in a txt file.* ***:param*** *index: int. Used to number an array that's being saved.* ***:param*** *binary\_array: a 2d array.* ***:return****: txt file.  
 """* np.savetxt(f"binary\_array\_{str(index)}.txt", binary\_array, fmt='%d')  
  
  
def save\_binary\_image(img, index, crossover\_operator):  
 *"""  
 Helper function that saves a binary image as a png file.* ***:param*** *crossover\_operator: character to be added to the front of the filename.* ***:param*** *index: int. Used to number an image that's being saved.* ***:param*** *img: a binary image.* ***:return****: png file.  
 """* path = 'images'  
 cv2.imwrite(os.path.join(path, f"{crossover\_operator}\_binary\_image\_{str(index)}.png"), img)  
  
  
def create\_plot(generation\_list, fitness\_list, crossover\_operator):  
 *"""  
 Helper function that creates and saves a plot.* ***:param*** *crossover\_operator: character to be added to the front of the filename.* ***:param*** *generation\_list: a list holding numbers of all generations.* ***:param*** *fitness\_list: a list holding the maximum fitness of each generation.* ***:return****: a plot.  
 """* plt.plot(generation\_list, fitness\_list)  
 plt.xlabel("Generation")  
 plt.ylabel("Fitness")  
 path = f"images/{crossover\_operator}\_plot.png"  
 plt.savefig(path)  
 plt.show()

# Appendix B – Test Reports

Class TestBinaryImage

|  |  |  |
| --- | --- | --- |
| **Case ID** | **Test Case Details** | **Pass/Fail** |
| create\_random\_binary\_image1 | Test that the image generated has the specific ratio of black and white pixels (50% white, 50% black). | Pass |
| create\_random\_binary\_image2 | Test that the image generated has the specific ratio of black and white pixels (70% white, 30% black). | Pass |
| binary\_image\_to\_binary\_array1 | Test the length of the array returned is of length 32. | Pass |
| binary\_image\_to\_binary\_array2 | Test the length of the array returned is of length 64. | Pass |
| binary\_array\_to\_binary\_image1 | Test that the image produced from a predefined 2d binary array, has the number of white pixels and the number of black pixels as specified (8 white pixels and 8 black pixels). | Pass |

A screenshot of a computer

Description automatically generated with medium confidence

Class TestCrossover

|  |  |  |
| --- | --- | --- |
| **Case ID** | **Test Case Details** | **Pass/Fail** |
| single\_point\_crossover1 | Test that the length of the genome and chromosomes of the offspring produced is correct (length of 4). | Pass |
| single\_point\_crossover2 | Test that the genome of the offspring is the expected when random crossover point is predefined. | Pass |
| double\_point\_crossover1 | Test that the length of the genome and chromosomes of the offspring produced is correct (length of 6). | Pass |
| double\_point\_crossover2 | Test that the genome of the offspring is the expected when random crossover point is predefined. | Pass |
| uniform\_crossover1 | Test that the length of the genome and chromosomes of the offspring produced is correct (length of 4). | Pass |

Text

Description automatically generated

Text

Description automatically generated

Class TestAlgorithm

|  |  |  |
| --- | --- | --- |
| **Case ID** | **Test Case Details** | **Pass/Fail** |
| initialise1 | Test that the length of the population generated and the chromosomes and genes of the individuals are as specified (200, 32 and 32 respectively). | Pass |

Text

Description automatically generated

Class TestElitism

|  |  |  |
| --- | --- | --- |
| **Case ID** | **Test Case Details** | **Pass/Fail** |
| find\_elites1 | Test that the genome of the returned 1 individual is the one of the best performing one. | Pass |
| find\_elites2 | Test that the genome of the returned 2 individuals is the one of the two best performing ones. | Pass |

Text

Description automatically generated

Class TestFitness

|  |  |  |
| --- | --- | --- |
| **Case ID** | **Test Case Details** | **Pass/Fail** |
| evaluate\_fitness1 | Test that the returned fitness of the individual is equal to 5, as set by the programmer. | Pass |
| get\_max\_fitness1 | Test that the returned value of fitness is the maximum one in a population of individuals with different values of fitness. | Pass |
| get\_max\_fitness\_genotype1 | Test that the genome of the individual returned is the one of the best performing individual. | Pass |

Text

Description automatically generated

Text

Description automatically generated