CITS4404 Assignment 2

GROUP 9
Evan Willcocks(22979772)
Kuo Gao(23318508)
Sebastian Mattews(22482441)
Trung Le(23011365)

CITS4404 Assignment 2

Group name:	**Group 9**		
Project name:	Practical Project - Camouflage Worms		
Weekly meeting:	**4pm Wednesday, 4pm Sunday**		
Student Name:	Evan Willcocks	**Student Id:**	22979772
Student Name:	Kuo Gao	**Student Id:**	23318508
Student Name:	Sebastian Mattews	**Student Id:**	22482441
Student Name:	Trung Le	**Student Id:**	23011365

Contents

Abstract

In this experiment, our group used a population-based optimiser and the Camo Worm model to camouflage and sharpen OCT images. We have studied and applied the GA algorithm and its variants, and applied crossover and mutation methods in Camo Worm populations. In the cost function, we comprehensively consider the influence of factors such as colour and size and have implemented preventive mechanisms for phenomena such as overlap. Finally, the expected effect was achieved with a certain number of iterations.

Key words： Optimization algorithm, Camo Worms, cost function, visualization

1. Background

In the field of medical imaging, retinal imaging is an important component. The health status of the retina determines a series of subsequent diagnosis and treatment, such as the diagnosis of macular degeneration. The retina has a considerable level of complexity and layered structure. Optical coherence tomography (OCT) is a commonly used technique in retinal imaging. This technology is harmless to the human eye and can be applied to imaging the nerve layer behind the eye.

However, using OCT technology alone is not enough, and imaging processing is also indispensable. Clearer retinal level images have a positive impact on the discovery of potential diseases. People need to sharpen and clean images obtained using OCT technology. The Camo Worm Model is a suitable choice for this.

The Camo Worm Model is a population-based model that achieves the function of cleaning and sharpening images by camouflaging them with worms. This model is based on Kass's snake model. In the Camo Worm Model, each worm can be abstracted as a Bessel curve with radians, centre points, and radii. The effectiveness and evaluation ability of different worm models may vary greatly depending on the algorithm construction.

From the perspective of our motivation and purpose in choosing the model, in addition to the selection of the Camo Worm model itself, we also have a certain degree of innovation in the selection of algorithms. Firstly, previous worm models often focused on evaluating the overall effectiveness of the model. But in this algorithm selection, we also focused on evaluating the camouflage effect of each worm. Secondly, for each worm, we added reproduction, mutation, and other features as the number of iterations increased. We believe this will make the whole worm more like a species. In addition, in terms of selecting the cost function, we utilize the fitness function to comprehensively consider the overall fitness and individual fitness of the insect population. We also set up mechanisms to prevent the worms from gathering too much. And we also briefly explored other possible algorithms.

Our hypothesis for this experiment is that we can use Camo Worm Model to camouflage retinal OCT images, thereby achieving cleaning and sharpening effects. In the experiment, we first establish an algorithm model, then evaluate the effectiveness of the algorithm using a cost function, and finally obtain results and conclusions.

2. Optimisation Algorithm

The project specification stated that the selected optimisation algorithm should be a generic meta-heuristic population-based algorithm [1]. Our selection was a genetic algorithm (GA), a subset of evolutionary algorithms (EAs). These algorithms are inspired by biological evolution and mimic natural selection processes to optimise a given objective. GAs employ operators like crossover/recombination, mutation, and elitist selection to diversify the population and search the solution space. The decision was made to use a GA as it fit the project specification, its implementation was simple, and it had been previously studied by one of our group members, who had written a precis on a paper discussing the use of GAs in a scheduling problem [2]. While our application to Camo Worms was a stark contrast to the scheduling problem, the GA remained applicable.

We implemented two variations of GAs. The first was a rudimentary GA adapted from [3] and the second was a steady state GA (SSGA) adapted from [4] [5]. In each variation, the worms are considered to have a single chromosome, with each of the 8 worm parameters (6 Bezier curve parameters + colour and width) considered as separate genes.

Pseudocode for the rudimentary algorithm can be seen below. In each generation we initialised a random clew of worms, calculated the cost of each worm, performed elitist selection where we took the top 10% of worms based on cost, and used these elites to generate new worms via crossover and mutation until the size of the new generation/clew was equal to the size of our initial population.

Algorithm 1: GA Implementation

```
Initialise a clew of N random worms
Initialise elitist percentage α ∈ [0,1]
Initialise mutation rate μ ∈ [0,1]
for gen ∈ [1, ∞] do
    for worm W ∈ W_1, ... W_N do
        Compute worm fitness/cost, C(W)
    end
    Sort clew according to cost of each worm
    Select top α worms as elites
    Create new clew initially containing elites
    while Size new clew < N do
        Sample two elites as parents
        Initialise child worm
        for gene g ∈ Genes G do
            // Crossover
            if random < 0.5 then
                g_child ← g_parent1
            else
                g_child ← g_parent2
            end
            // Mutation
            if random < μ then
                g_child ← Mutated g
            end
        end
        new clew ← new clew + child worm
    end
end
```

Algorithm 1: Rudimentary Genetic Algorithm Pseudocode

The SSGA was very similar to the rudimentary version. Instead of performing an elitist selection and using these elites to rebuild the population, we selected two random parents to create a child through crossover and mutation. If the child had a better cost than the worst worm in the clew, then we removed the worst worm and inserted the new child at its correct position in the clew. This implementation performed much better computationally and converged to a better state much quicker. This was the implementation we used to produce our results. See Appendix for code implementations.

Algorithm 2: SSGA Implementation

```
Initialise a clew of N random worms
Initialise mutation rate μ ∈ [0,1]
for iter ∈ [1, ∞] do
    if iter = 1 then
        for worm W ∈ W_1, ... W_N do
            Compute worm fitness/cost, C(W)
        end
        Sort clew according to cost of each worm
    end
    Sample two random worms as parents
    Initialise child worm W_child
    for gene g ∈ Genes G do
        // Crossover
        if random < 0.5 then
            g_child ← g_parent1
        else
            g_child ← g_parent2
        end
        // Mutation
        if random < μ then
            g_child ← Mutated g
        end
    end
    // If child is better than worst worm in clew then replace
    if C(W_child) < C(W_last_in_clew) then
        Remove W_last_in_clew
        Insert W_child in correct place to maintain sort
    end
end
```

Algorithm 2: Steady State Genetic Algorithm Pseudocode

A critical component of GAs is the definition and implementation of the crossover and mutation methods. In our implementation, crossover is effectively performed 90% of the time. We tested two methods for performing crossover. The first, involved the child having a 50/50 chance of receiving the gene from either parent 1 or parent 2. In the second implementation, the child received the average of the two parent genes. This second method resulted in the clew converging to a state that had a significant reduction of genetic diversity, with the worms grouping together and looking almost identical. The first method was able to maintain diversity and not converge to suboptimal solutions, and therefore is the method used to conduct experiments on the cost function.

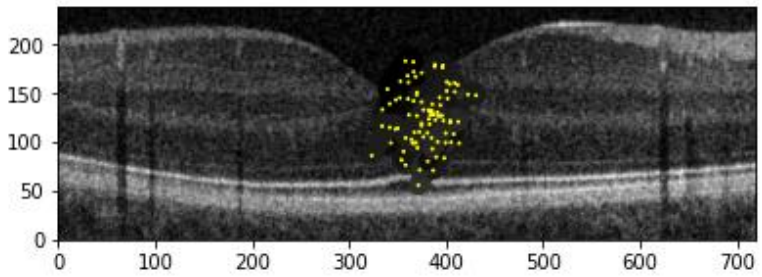

Figure 1: Example of low genetic diversity.

Mutation is performed the other 10% of the time, in which the child gene is randomised independently of the parents. The randomisation method is dependent on the gene and is performed in the same fashion that the worms are randomly initialised. We tested another method for mutation in which the child would receive a gene from a parent (crossover rate was 100%) and the gene would be mutated by adding some random number to it.

i.e. $gene = gene + rng$

Again, this resulted in convergence to a state with low genetic diversity like above and we therefore adopted the first method which introduces more diversity into the population.

3. Cost Function

This group experimented with several factors to produce the best worms. Each factor can be thought of as a solution to a particular problem. As problems were solved, and results started to get better, new problems arose.

Factor 1: Colour Matching

The first, and most obvious problem is that worms should closely match the image. Many approaches for this problem were tried, but all of them used average colour difference between image pixels and worm colour as a measure of fitness.

The initial version of this method only tested the colour difference at the intermediate points, without considering edges. This was fast, but generally gave a poor approximation near multiple layers.

Later, improved versions used a series of circles to gather all the pixels under a worm. The resulting group of pixels was named a Worm Mask. This gave slower, but precise colour results, and was used in the final results section.

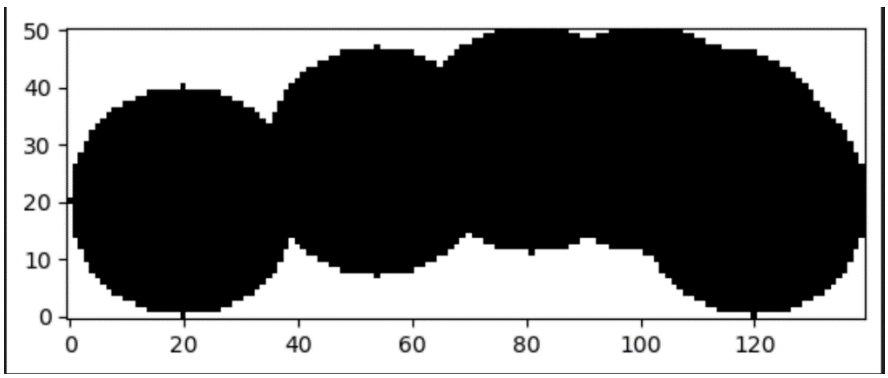

Figure 2: A series of circles used to generate a Worm Mask

Factor 2: Size

Colour matching introduced a new problem. Due to inherent noise within an image, large worms will never get a perfect colour-matching score, even when placed effectively. Conversely, extremely small worms, such as one pixel long, can get perfect scores, but are invisible to the viewer.

A few attempts were made to counteract the above-mentioned effect. The first experiment involved a size parameter, which was calculated by multiplying length by width. The idea was that this would keep small worms in check, while ineffective colour matching and noise would make huge worms not viable.

In practise, the size parameter either skewed worms to be far larger than desirable, or, if given a lower weighting, had no effect at all after many iterations.

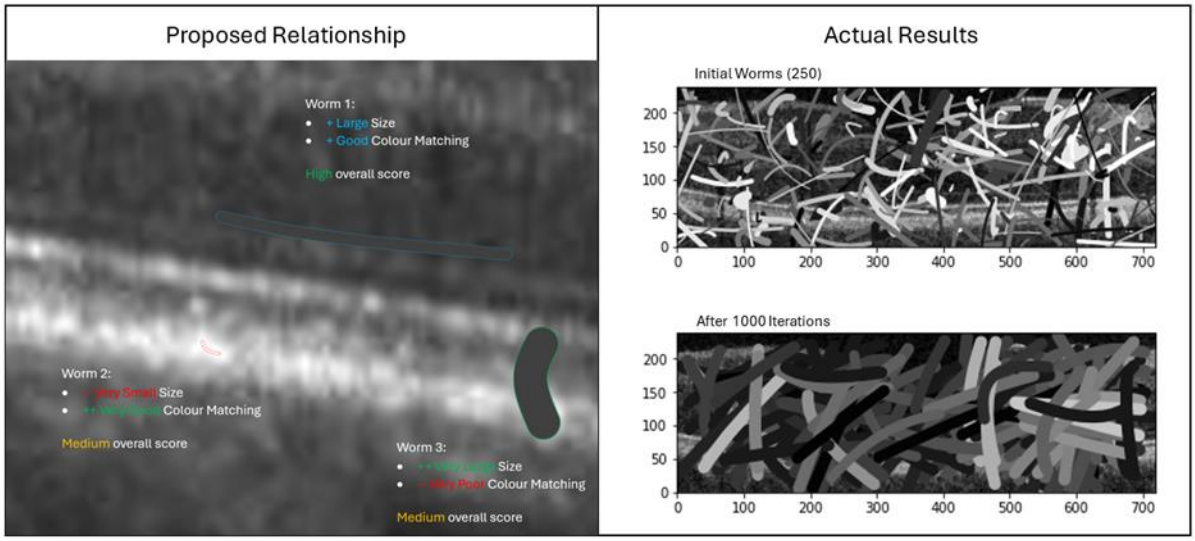

Figure 3: Theoretical vs actual relationship between Noise, Size and Colour Matching

Another version gave better results by calculating the difference in size from an "ideal value". While this did remove both tiny and huge worms, it also made all worms about the same size, even if it didn't make sense for the image.

$$abs(width_{worm} - width_{ideal})$$

A modified version of this method was eventually used in the final results. It calculates the proportional difference from a pre-set minimum value. If the worm size is larger than the minimum, it returns a negative difference, which gets limited to a perfect score of 0.

$$\max\left(0, \frac{(width_{min} - width_{worm})}{width_{min}}\right)$$

Factor 3: Contrast

After colour-matching started producing good results, the team noticed a new effect starting to emerge throughout the iterations. Large black bars were often present near the top and bottom of the image. Worms placed in this area would score highly due to low colour variance, despite doing nothing to improve image quality.

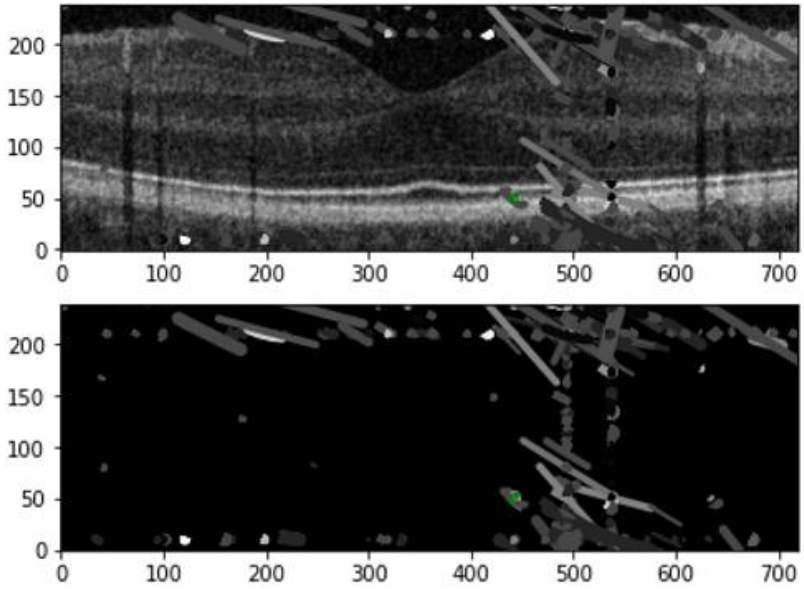

*Figure 4: An example of how worm populations will move to the black regions over time: *Top: Worms overlaid over image *Bottom: Same worms over black background*

A parameter was added to make these worms less viable, by testing for high contrast around the outside of the worm. This was designed to place worms in between layers where they would have the most effect.

This parameter went through a similar development process as the colour-matching term. The initial version used a sample of points around the outside of the worm. This fast, simple method was eventually replaced with a function which used Worm Masks to find an area around the worm. An outer mask is created which is larger than the worm, and the pixels within the worm's area are then negated from it.

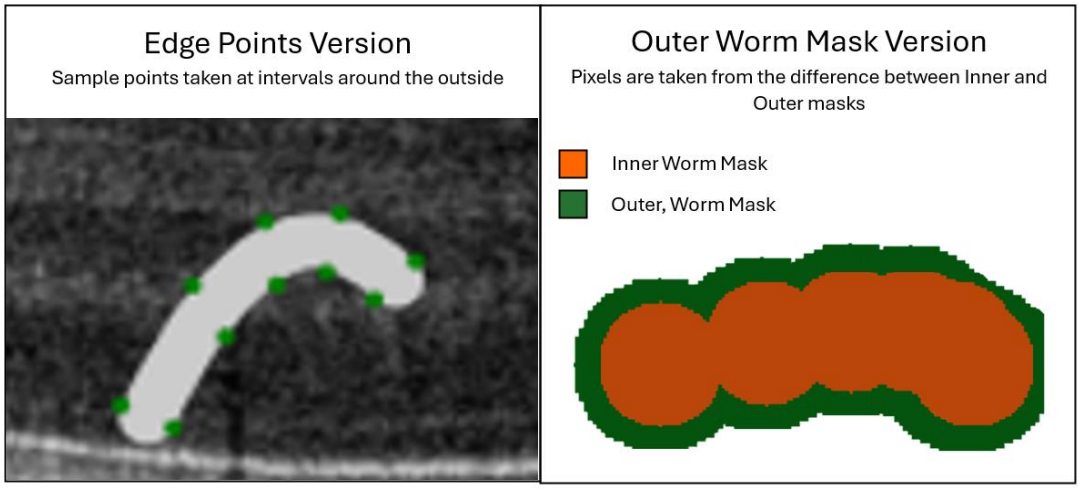

Figure 5: Comparison between the Edge Points method and Worm Mask Method

Factor 4: Overlaps

A final potential problem the team wanted to solve was that of overlaps. Overlapping worms will affect the same parts of the image, which wastes the potential smoothing each worm could provide.

Different variations of this parameter were attempted. The initial version found the average distance to every other worm. However, since there was no maximum to this term, it led to worms pushing to the edges of the image unintentionally.

$$\frac{(\sum dist)}{total}$$

A better version involved simply counting the number of times a worm overlaps other worms, and negating that from the total score.

$$\frac{overlaps}{worms_{total}}$$

Final Function

$$W_1 \times Size + W_2 \times Overlaps + W_3 \times AverageColourDif + W_4 \times AverageContrast$$

Table 1: Cost function parameters detail:

Term	Category	Description
W_1, W_2, W_3, W_4	Weight	The tuning weights for each of the 4 parameters
Size	Internal Knowledge	A Parameter to fix worm size. Best version used difference from minimum value$\max\left(0, \frac{(width_{min} - width_{worm})}{width_{min}}\right)$
Overlaps	Group Knowledge	Total number of overlaps with other worms, as a proportion of total worms: $\frac{worms_{overlap}}{worms_{tot}}$
AverageColourDif	Environmental Knowledge	The difference in colour between the worm and the average background colour
AverageContrast	Environmental Knowledge	The difference in colour of the image at the outside of the worm vs the inside

Different implementations used different combinations and weightings for these terms.

4. Results

This section will explain the results produced after applying the Steady State Genetic Algorithm with the given cost function explained in the previous section. It will also cover the progress along the way of optimizing the cost function.

The first experiments are only testing a few sample points around the worms. These results can be observed in Figure 7. The middle parts of the image have been covered almost entirely by the worms, with exception of a gap around the white space. The cost over iterations is shown in Figure 6 to converge around the 2000th iteration, with only small changes and improvements over the last 8000 iterations. This means that even if more iterations added, the results are potentially the same as below.

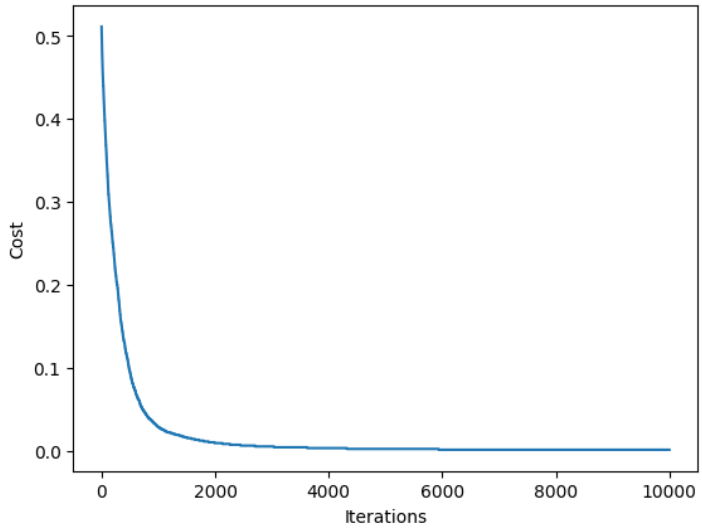

Figure 6: Cost vs Iterations (SSGA) with mask under colours (w1, w2, w3 = 1) (250 worms. 10000 iterations)

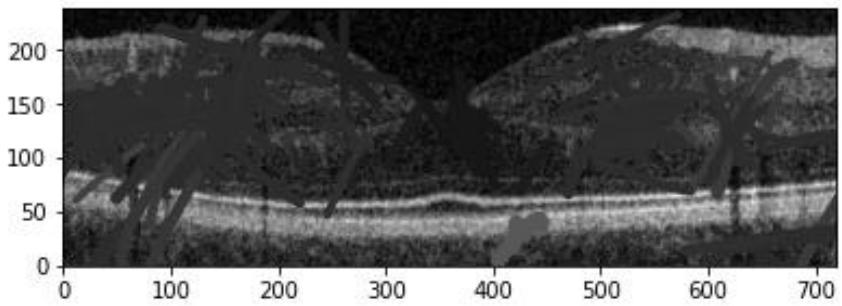

Figure 7: Result of the above SSGA with mask (10000 iterations)

The next set of experiments introduces the use of the Worm Masks class, which generates all the points covered by the worm and tests them for Colour Matching. The first version of the test has every weight set to 1. Figure 9 shows the filtered image after 2000 iterations and after 10000 iterations.

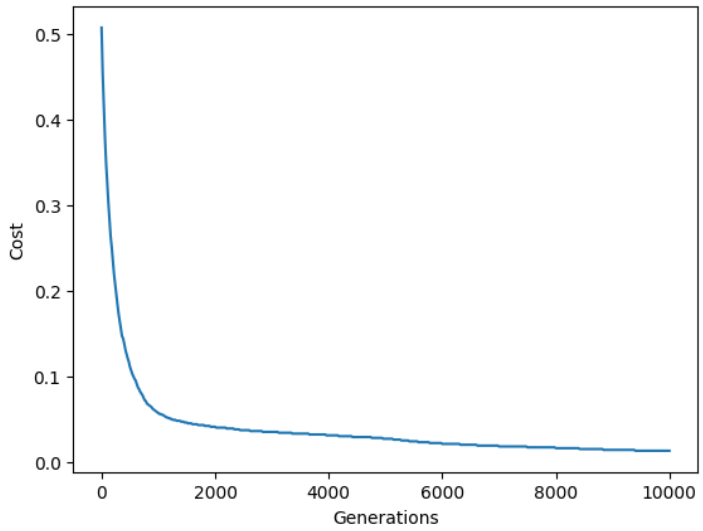

Figure 8: Cost vs Iterations (SSGA) with mask under colours (w1, w2, w3 = 1) (250 worms, 10000 iterations)

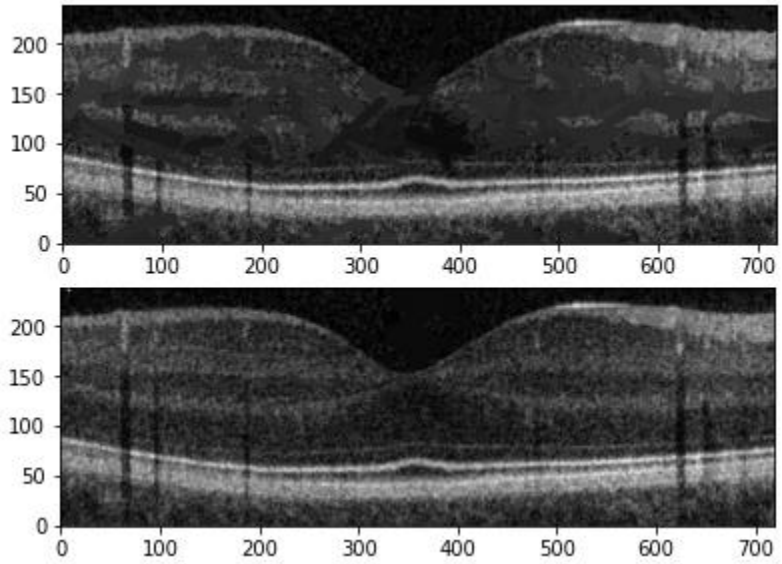

*Figure 9: Result of SSGA with detailed mask (250 worms): *Top: at 2000th iter; *Bottom: at 10000th iter;*

The top image presents that the worms were able to partially smoothen out the middle part of the image. Based on this result at the 2000th iteration, it could be expected that after another 8000 more, the worms should be able to filter the entire image. Instead, in the bottom image, the worms are hardly seen, and the image looks almost identical to the original image. Figure 8 indicates that the algorithm has not been converged yet, which means potentially the worms can keep get smaller.

This is where the size of the worm needs to be limited. The results after introducing the minimum size parameter are presented in the Figure 10:

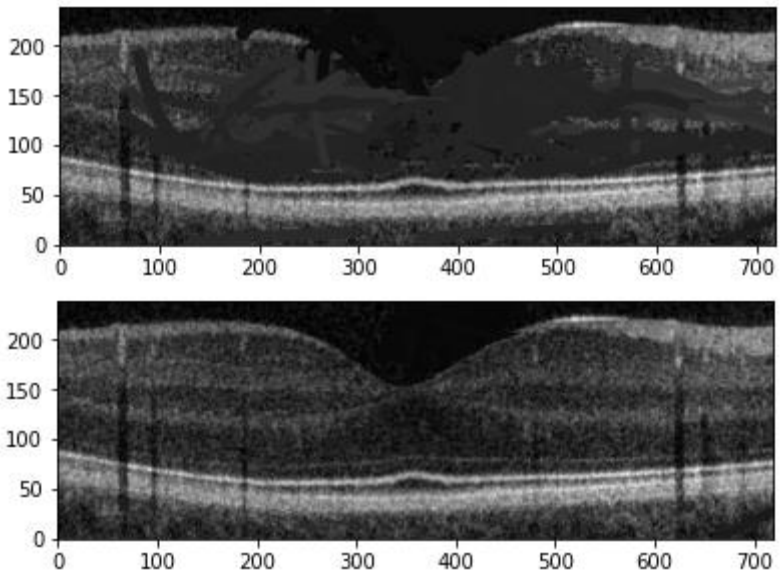

*Figure 10: Result of SSGA with detailed mask and limited worm size (250 worms): *Top" at 4000th iter; *Bottom: at 10000th iter*

Compared to the previous figure, the worms after 4000 iterations are now much bigger, but after 10000 iterations, they still start to disappear. The minimum size parameter already punished the worms smaller than 50 pixels in total, so this cannot be due to 1-pixel worms. An alternate theory for the worms disappearance is that the worms moved to the top of image, which is consistently black, and can potentially give the worms 0 cost for colour parameter. Figure 11 bottom image proves this theory.

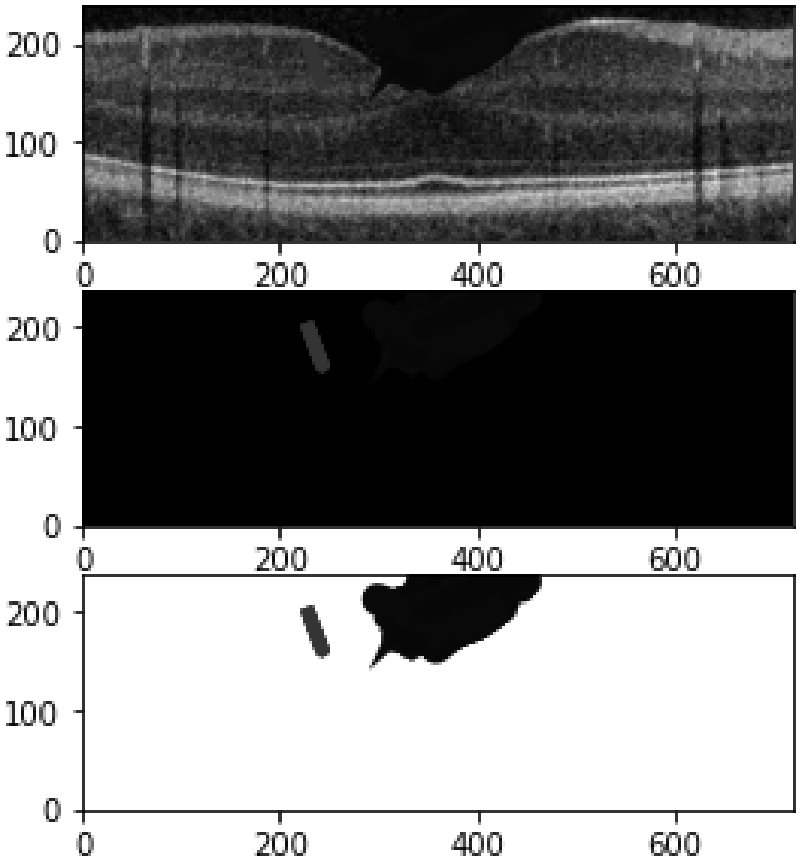

Figure 11:Worm placement of Figure 10 at 10000th iter (250 worms) (w1, w2, w3 = 1)

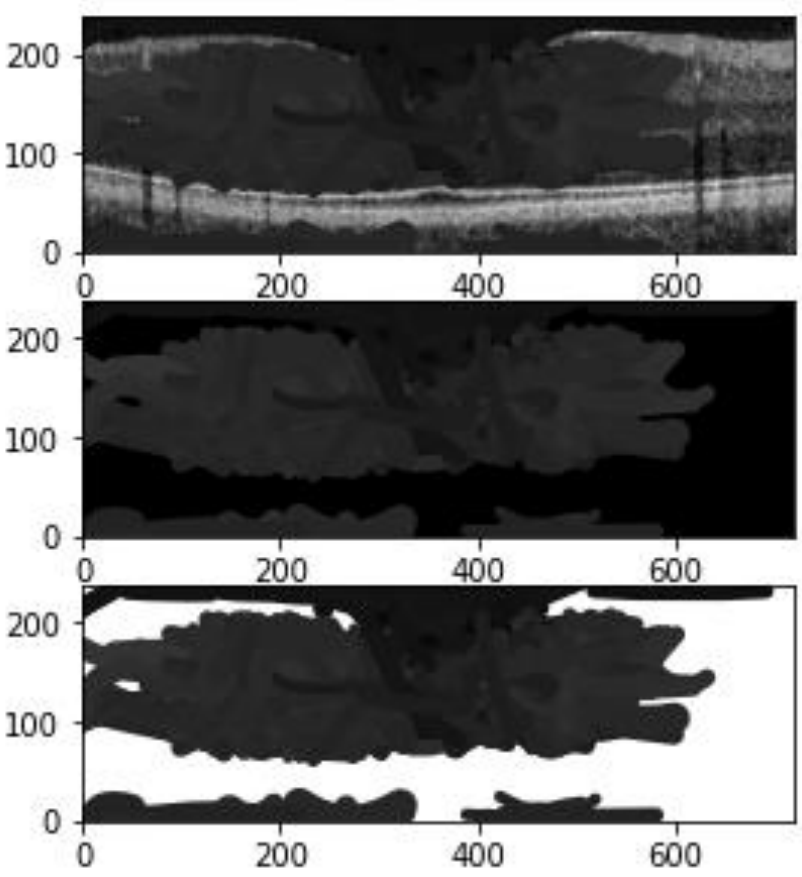

Figure 12: Worm placement with 1000 worms and 10000th iter (w1, w2, w3 = 1, 2, 1)

The final experiment involved increasing the weight of the distance and overlaps, which improved the results at the 10000th iteration, as seen in Figure 12. The minimum size of the worms was also increased to be 500 instead of 50, and the number of worms was increased to 1000. Finally, a penalty was applied to the cost function to penalize the worms with nearly 0 cost. This tried to ensure worms were not smoothing parts of the image that were already low in noise. The bottom image shows that not only did the middle part smooth, but the top part was also completely filled in.

After this set of experiments, the team noticed that the white space was not covered by any of the worms. This is because in this white space, the noise is dark in colour, which can cause the average differences between the worm and the image to be increased. After a set number of iterations, worms in this space will eventually be removed due to their uncompetitive score. One of the solutions is to find the most frequently appearing colour in the image with a Fourier transform, and then calculating the difference in terms of colour on the worm. The distance parameter could also be tailored to force worms to spread out into the whiter space.

The next experiment is to see the difference in time taken between implementations using intermediate points and the implementation using the Worm Mask class which represent every single point of the worm. The test is conducted using the time indicated by the Jupyter Notebook, as per described by Table 2 below:

Table 2: Time taken (Using 500 iterations)

Worms	250	300	350	400	450	500
Samples	24.2	29.7	41.7	53.9	55.4	63.1
Mask Class	26.9	32.3	36.9	48.6	54.9	64

The result of the time taken is unexpected, as the samples should have the run time much faster than the Mask Class, given less points of the worm covered. This could be because the algorithm itself accounts for most of the time, and changes to the cost function produces negligible differences.

5. Conclusion

With the Steady State Genetic Algorithm and the final cost function, the team was able to propose a solid solution to the Camo Worm problem. It can generate up to 1000 worms within a reasonable time and was able to smooth most of the darkened part. The result was not perfect, and there are still problems left to be unsolved, such as the brighter part of the image being left uncovered and perfecting the overlapping parameter. These could be improved in the future by using Fourier transforms rather than raw colour values. Overall, this method has proven to be effective in some situations, especially darker images.

6. References

[1] C. MacNish, *CITS4404 Assignment 2 Practical Project - Camouflage Worms,* UWA, 2024.

[2] D. Whitley, O. Quevedo, M. Roberts, V. Shetty, and Piyabutra Jampathom, "Scheduling Multi-Resource Satellites using Genetic Algorithms and Permutation Based Representations," *Proceedings of the Genetic and Evolutionary Computation Conference*, Jul. 2023, doi: https://doi.org/10.1145/3583131.3590387.

[3] "Genetic Algorithms," Geeks For Geeks, 2024. [Online]. Available: https://www.geeksforgeeks.org/genetic-algorithms/. [Accessed April 2024].

[4] "Steady State Genetic Algorithm (SSGA)," Geeks For Geeks, 2021. [Online]. Available: https://www.geeksforgeeks.org/steady-state-genetic-algorithm-ssga/. [Accessed April 2024].

[5] "Python program to insert an element into sorted list," Geeks For Geeks, 2023. [Online]. Available: https://www.geeksforgeeks.org/python-program-to-insert-an-element-into-sorted-list/. [Accessed April 2024].

APPENDIX

Code Repository: https://github.com/evnw/CITS4404-Assignment2